Sorbonne Université – UFR 919 Ingéniérie

L3 - Bases de données
UE LU3IN009

http://www-bd.lip6.fr/wiki/doku.php?id=site:enseignement:licence:3i009:start

Support de cours

Stéphane Gançarski

Stephane.Gancarski@lip6.fr

2019

LU3IN009: Systèmes de Gestion de Bases de Données

http://www-bd.lip6.fr/wiki/doku.php?id=site:enseignement:licence:3i009:start

Description du cours

Ce module s'inscrit dans la suite du module 2I009 de L2. Après un bref rappel des concepts vus en L2, le cours présente comment l'algèbre relationnelle utilisée en interne par les SGBD pour évaluer les requêtes. Ensuite, le contrôle de la concurrence puis l'optimisation de schéma sont abordées Enfin les outils classiques de bases de données (triggers instead of et vues) viennent compléter le module.

Toutes les séances sont **TD** en première partie et **TME** en seconde partie.

Actualités et informations pratiques

Lire les instructions de connexion oracle
Lire les instructions d'utilisation de H2
Récuperer Raeval et les fichiers foofle RaevalFoofle
Notes de cours S. Gançarski 2003 (un peu ancien, quelques bugs, mais utile) polylicbd2003.pdf

Contrôle de connaissance

Le contrôle de connaissances est composé d'un examen final (60%) et d'un contrôle continu (40%) sous la forme de 4 interros (10% chacune). Les interros auront lieu pendant les séances de TD, les semaines 4, 7, 9 et 12 (voir planning)
Tous les documents (déjà lus) sont autorisés à l'examen. Pour les interros, uniquement une feuille A4 recto/verso manuscrite. Les 3 première sinterros sont des QCM

Bases de Données : rappels et problèmes à traiter

- **Fichiers et Bases de Données**
- Systèmes de Gestion de Bases de Données (SGBD)
- Langages et modèles de données
- Modèle Entité-Association
- Modèle Relationnel
- Calcul relationnel et SQL

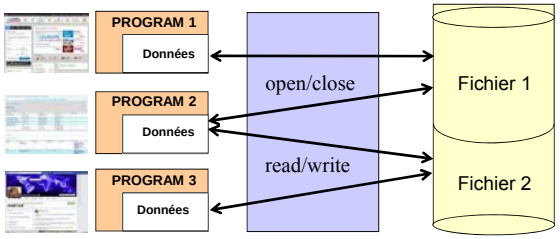
Qu'est-ce qu'une Base de Données (BD) ?

- Une **base de données** (BD) est une *collection de données structurées* sur des entités (objets, individus) et des relations dans un contexte (applicatif) particulier.
- Un **système de gestion de base de données** (SGBD) est *un (ensemble de) logiciel(s)* qui facilite la création et l'utilisation de bases de données.
- Les données sont définies, administrées et gérées en utilisant des langages fondés sur des modèles de données (2I et 3I, modèle *relationnel*).

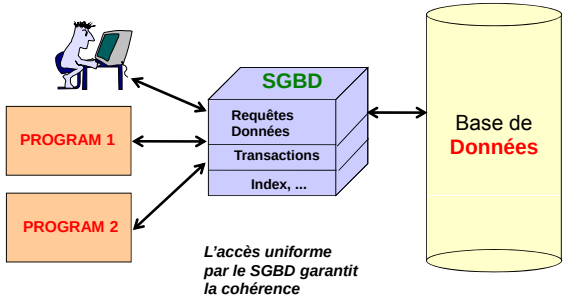
Fichier ≠ Bases de Données

Fichiers :

- opérations simples : ouvrir/fermer, lire/écrire
- différentes méthodes d'accès (séquentiel, indexé, haché, etc.)
- utilisation par plusieurs programmes difficile (format ?, concurrence)
- La gestion de données structurées dans des fichiers représente une partie importante du **coût de développement et de maintenance** d'applications.



Approche « Base de Données »



Problèmes avec les fichiers résolus avec une base de données

Fichier :

- Faible structuration des données
- Dépendance entre programmes et fichiers
- Redondance des données
- Absence de contrôle de cohérence globale des données
- Accès aux nuplets un par un, par programme

Base de Données :

- Structuration des données à travers un schéma de données
- **Indépendance entre programmes et gestion de données**
- Données partagées
- Contrôle de la cohérence logique et physique (schémas, transactions)
- Accès ensembliste et déclaratif

Objectifs du cours 3I009

- Étudier les « bases de données » du point de vue :
  - *de l'utilisateur : accès par appli ou SQL (expert)*
  - ***du développeur : définir schéma (EA,rel.) et implémenter applis***
  - *de l'administrateur :* schéma physique et performances (tuning), sécurité et fiabilité
- Comprendre les **principes** des Systèmes de Gestion de Bases de Données (SGBD) Relationnels
- Apprendre à **construire des applications** sur un SGBD
- Étudier les **mécanismes internes** des SGBD

Plan général du cours

- Rappel SGBD, modèle et langages relationnel
- Evaluation et optimisation des requêtes : du stockage disque au résultat — *Performance d'exécution*
- Transactions et tolérance aux pannes
- Contrôle de concurrence — *Cohérence d'exécution multiutilisateur*
- Dépendances fonctionnelles
- Normalisation de schémas relationnels — *Théorie de la conception*
- SQL : Triggers et vues, JDBC — *Outillage pour le développement*

Bibliographie

Livres en français (aussi disponibles à la bibliothèque MathInfo Enseignement) :

- C. Chrisment et.al., Bases de données relationnelles, Hermes - Lavoisier
- G. Gardarin. Bases de données - objet et relationnel. Eyrolles.
- J.L. Hainaut, Bases de données : concepts, utilisation et développement, Dunod
- S. Abiteboul, R. Hull, V. Vianu, Les fondements des bases de données, Vuibert

Livres en anglais :

- R. Ramakrishnan and J. Gehrke. Database Management Systems. 3e édition, McGraw Hill, 2002 - http://pages.cs.wisc.edu/~dbbook/
- H.G. Molina, J.D. Ullman, J. Widom. Database Systems: The Complete Book, Goal Series - http://infolab.stanford.edu/~ullman/dscb.html
- C.J. Date, Introduction aux bases de données, 7e édition, Vuibert
- M.T. Özsu, P. Valduriez. Principles of Distributed Database Systems. 2nd edition, Prentice Hall, 1999

Notes de cours

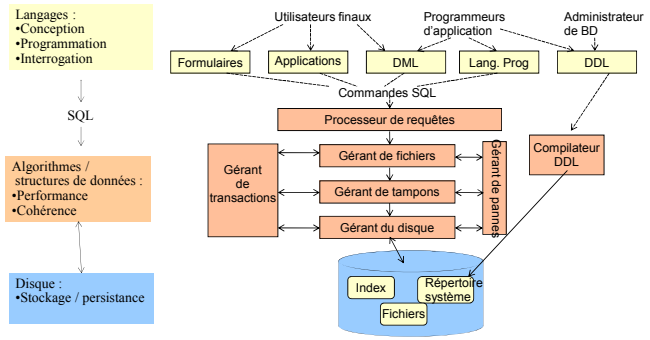
- S. Gancarski. Introduction aux bases de données. UPMC, Paris 6, janvier 2003 – lien sur le site web

Système de Gestion de Bases de Données (SGBD) Fonctions

- Représentation et structuration de l'information :
  - « Modéliser le monde réel et ses règles de fonctionnement »
  - Description de la *structure des données* : Employés(nom, age, salaire)
  - Description de *contraintes logiques sur les données* : 0 ≤ age ≤ 150, ...
  - *Vue : réorganisation (virtuelle)* de données pour des besoins spécifiques
- Gestion de l'intégrité des données :
  - Vérification des contraintes spécifiées dans le schéma
  - Exécution *transactionnelle* des requêtes (mises-à-jours)
  - Gestion de la concurrence multi-utilisateur et des pannes
- Traitement et optimisation de requêtes :
  - La performance est un problème géré par l'administrateur du SGBD et non pas par le développeur d'application (indépendance physique)

Objectif : faciliter le partage de grands volumes de données entre différents utilisateurs / applications

Architecture d'un SGBD



Langages et interfaces d'un SGBD

- **Langages de *conception*** : Entité-Association, UML
  - Utilisation : conception *haut-niveau* d'applications (données et traitements)
- **Langage base de données :** SQL, calcul relationnel, algèbre
  - langages *déclaratifs* : spécifier « *quoi* » (SQL) et non « *comment* » (Algèbre, généré par le SGBD)
  - puissance d'expression limitée par rapport à un langage de programmation comme C ou Java
  - utilisation : *définition schémas, interrogation et mises-à-jour, administration (SQL)*
- **Langages de *programmation*** : PL/SQL, Java, PHP, …
  - langages *impératifs* avec une interface SQL (ex. JDBC)
  - langages expressifs (« complet » au sens d'Alan Turing)
  - utilisation : *programmation d'applications (avec SQL pour accéder aux données)*
- **Langages de *bas niveau*:** C,…
  - Mettre en œuvre le SGBD, accès aux couches physiques, implémentation des opérateurs, ….

Langages BD (SQL)

Langage de **Définition de Données** (LDD)

- pour définir les schémas externes (vues), logiques et physiques
- les définitions sont stockées dans le répertoire système (dictionnaire)

Langage de **Manipulation de Données** (LMD)

- langage *déclaratif* pour interroger (**langage de requêtes**) et mettre à jour les données
- peut être autonome (par ex. SQL seul) ou intégré dans un langage de programmation (à travers une API comme JDBC)

Histoire des bases de données

Années 1960:

- début 1960: Charles Bachmann développe le premier SGBD, IDS, chez Honeywell
  - modèle réseau: les associations entre les données sont représentées par un graphe
- fin1960: IBM lance le SGBD IMS
  - modèle hiérarchique: les associations entre les données sont représentées par un arbre
- fin 1960: standardisation du modèle réseau Conference On DAta Systems Languages (CODASYL)

Histoire des bases de données

- 1970: Ted Codd définit le modèle relationnel au IBM San Jose Laboratory (aujourd'hui IBM Almaden)
  - 2 projets de recherche majeurs
    - INGRES, University of California, Berkeley
      - devint le produit INGRES, suivi par POSTGRES, logiciel libre, qui devint le produit ILLUSTRA, racheté par INFORMIX
    - System R, IBM San Jose Laboratory
      - devint DB2, inspira ORACLE
  - 1976: Peter Chen définit le modèle Entité-Association (Entity-Relationship)

Histoire des bases de données

Années 1980

- maturation de la technologie relationnelle
- standardisation de SQL

Années1990

- amélioration constante de la technologie relationnelle
- support de la distribution et du parallélisme
- modèle objet, ODMG, BD objets
- fin 1990 : le relationnel-objet, SQL3
- nouveaux domaines d'application: entrepôts de données et décisionnel, Web, multimédia, mobiles, etc.

Années 2000

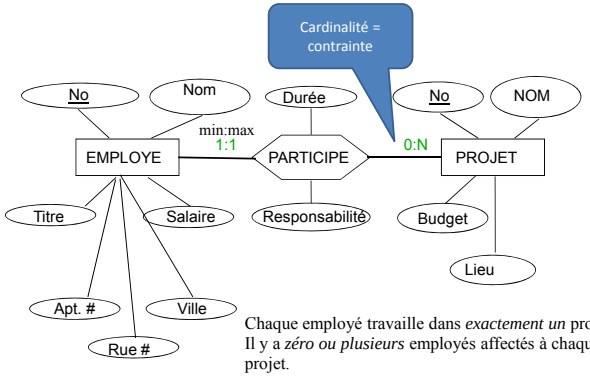
- apparition de XML et de nouvelles architectures (eg. P2P)
- NoSQL, MapReduce, RDF (Web sémantique), …

Le modèle Entité-Association (E/A)

« E/R (entity-relationship) model » en anglais

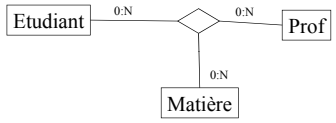
- Modèle / langage de *conception* :
  - Définition de schémas conceptuels
- Modélisation *graphique* des entités, de leurs attributs et des associations entre entités.
- Objectif :
  - détection d'erreurs de conception *avant* le développement
  - traduction « automatique » dans un modèle logique.
- Supporté par les outils CASE pour BD
- ~ partie "modélisation de données" dans UML

Schéma entité-association

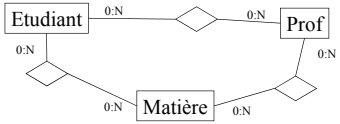


Chaque employé travaille dans *exactement un* projet.
Il y a *zéro ou plusieurs* employés affectés à chaque projet.

Association ternaire



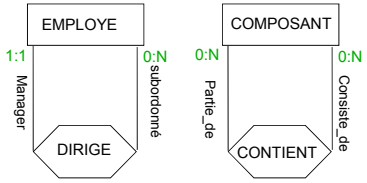
est différent de



Pourquoi ?

Association réflexive

Association réflexive : une entité d'une classe C est associé à une ou plusieurs entités de la *même* classe C.

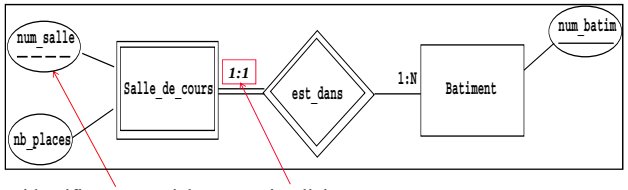


Entités fortes et faibles

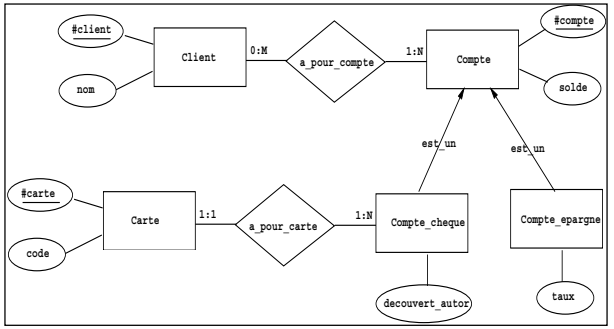
Entité forte: entièrement identifiable par ses attributs

Entité faible: ne peut être identifiée que par rapport à une autre entité, dite dominante, à laquelle elle se réfère. Son identificateur est :

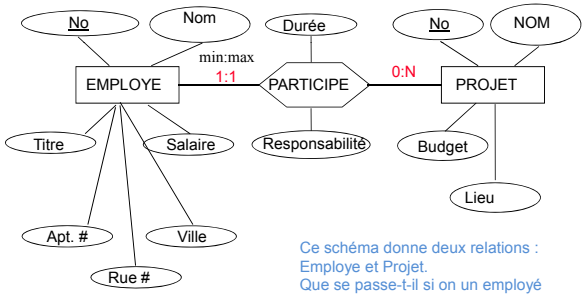
identificateur partiel + identificateur de l'entité dominante.



Spécialisation



Modèle relationnel et E/A



Ce schéma donne deux relations :
Employe et Projet.
Que se passe-t-il si on un employé peut être dans plusieurs projets ?
Dans aucun ?

Une base de données relationnelle

schéma

instance

n-uplet

EMP

ENO	ENOM	EMPLOI
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

PARTICIPE

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E7	P5	Engineer	23
E8	P3	Manager	40

PROJ

PNO	PNOM	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000
P5	CAD/CAM	500000

Monde ouvert
vs. Monde fermé

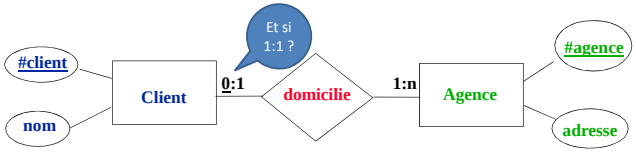
Modèle relationnel

Fondements mathématiques solides :
- théorie des ensembles
- logique du premier ordre (calcul relationnel)

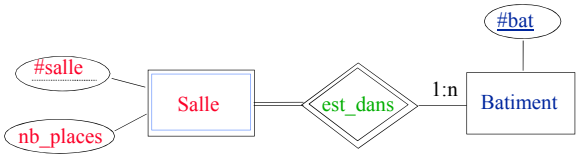
Langages de requêtes *simples, puissants et efficaces*

*Mais…*un schéma relationnel peut contenir des centaines de tables avec des milliers d'attributs.

- Problème: comment éviter des erreurs de conception?
- Deux solutions (complémentaires):
  - Génération (automatique) à partir d'un schéma E/A
  - Théorie des dépendances et normalisation (on verra plus tard)

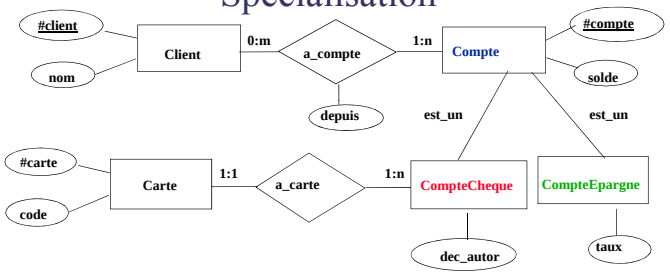


Client(#client, nom) Agence(#agence, adresse)
Domicilie(#client, #agence)



La clé de l'entité dominante fait partie de la clé de l'entité faible
Salle(#salle, #bat, nbplaces) Batiment(#bat, …)

Spécialisation



Compte(#compte, solde) CompteCh(#compte, dec-aut) CompteEp(#compte, taux)

Si Compte est un type d'entité « abstrait » (sans instances) :
CompteCh(solde, dec-aut, #compte) CompteEp(solde, taux, #compte)

Calcul relationnel de n-uplets

Une requête exprimée dans le ***calcul n-uplet*** a la forme

$$Q(x_1, x_2, \ldots, x_n) = \{ x_1.A_j,\ldots,x_2.A_k, \ldots, x_n.A_m \,/\, F(x_1, x_2, \ldots, x_n) \}$$

- F est une *formule logique*,
- $x_1, \ldots, x_n$ sont des ***variables n-uplet***,
- $x_i.A_j$ désigne l'attribut A_j d'une instance (n-uplet) de la variable x_i.

Formules logiques : syntaxe

Une formule logique F est une expression composée de

- **atomes** : R(x), T(x,y), x<3, x=y, ...
- **opérateurs booléens** :
  - $\wedge$ (conjonction)
  - $\vee$ (disjonction)
  - $\neg$ (négation)
- **quantificateurs :**
  - $\exists$ (quantificateur existentiel)
  - $\forall$ (quantificateur universels)
- virgules et parenthèses

$\Rightarrow$ F = formule de la *logique du premier ordre sans fonctions*

Exemples de requêtes

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

- Q(x)={ x.Ename | Emp(x) }
- Q(x)={ x.Pname, x.Budget | Project(x) }
- Q(x)={ x.Title | Emp(x) }
- Q(x)={ x.Ename | Emp(x) $\wedge$ x.City = 'Paris' }
- Q(x)={ x.City | Emp(x) $\vee$ Project(x) }
- Q(x)={ x.City | Project(x) $\wedge \neg\exists y$ (Emp(y) $\wedge$ x.City = y.City)}

- Traduction en SQL ?

Exemples de requêtes

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

- Noms de tous les employés?
  Q(x)={ x.Ename | Emp(x) }

la variable libre x est liée à tous les n-uplets de la table Emp

- Noms de tous les projets avec leurs budgets?
  Q(x)={ x.Pname, x.Budget | Project(x) } :

la variable libre x est liée à tous les n-uplets de la table Project

- Titres (d'emploi) pour lequel il y a au moins un employé?
  Q(x)={ x.Title | Emp(x) }

Exemples de requêtes

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

- Employés qui travaillent à Paris?
  Q(x)={x.Ename | Emp(x) $\wedge$ x.City = 'Paris' }

x est liée à tous les n-uplets t de Emp où t.City = 'Paris'

- Villes où il y a un employé ou un projet?
  Q(x)={ x.City | Emp(x) $\vee$ Project(x) }

x est liée à tous les n-uplets t de Emp et à tous les n-uplets t' de Project

- Villes où il y a des projets mais pas d'employés?
  Q(x)={ x.City | Project(x) $\wedge \neg\exists y$(Emp(y) $\wedge$ x.City = y.City) }

Exemple de requêtes

Emp (Eno, Ename, Title, City)
Pay(Title, Salary)

Project(Pno, Pname, Budget, City)
Works(Eno, Pno, Resp, Dur)

- Noms des projets de budget > 225?

Q(x)={ x.Pname | Project(x) ∧ x.Budget > 225 }

- Noms et budgets des projets où travaille l'employé E1?

Q(x)={ x.Pname, x.Budget | Project(x) ∧
∃y (Works(y) ∧ x.Pno=y.Pno ∧ y.Eno='E1') }

Sûreté des requêtes

Problème:

- La taille de Q(x)={ *x.A* | *F*(*x*) } doit être finie

- Exemple: le résultat de Q(x)={ x.A | ¬R(x) } *est infini : x est lié à tous les n-uplets qui ne sont pas dans la table R*

Sûreté:

- Une requête est sûre si, pour toute BD conforme au schéma, le résultat de la requête peut être calculé en utilisant seulement les constantes apparaissant dans la BD et la requête.
- Puisque la BD est finie, l'ensemble de ses constantes est fini de même que les constantes de la requête; donc, *le résultat de la requête est fini.*

Sûreté des requêtes (suite)

La *caractérisation syntaxique* de requêtes sûres est difficile.

Quelques conseils pour construire des requêtes sûres :

- Toujours commencer une requête par {x.A | R(x) …}
- Une quantification ∃x ou ¬∃x doit toujours être suivie d'un atome R(x) : x est bornée aux n-uplets de la table R
- Éviter d'utiliser ∀ et le remplacer par ∃ grâce à l'équivalence (∀x R(x)) ⇔ ¬∃ x ¬R(x) (on revient sur le cas d'avant)

- La syntaxe SQL garantit la formulation de requêtes sûres (il n'y a pas de ∀ générique)

Requêtes d'interrogation SQL

Structure de base d'une requête SQL simples :

```
SELECT [DISTINCT]   var_i.A_ik, …     attributs
FROM     R_i1 var_1, R_i2 var_2…      tables
WHERE    P                            prédicat/condition
```

où:

- var_i désigne la table R_{ij}
- Les variables dans la clause SELECT et dans la clause WHERE doivent être *liées* dans la clause FROM
- Le mot clé DISTINCT (optionnel) permet d'éliminer des doublons.

Simplifications :

- Si var_j n'est pas spécifiée, alors la variable s'appelle par défaut R_{ij}.
- Si une seule table/variable *var* possède l'attribut A, on peut écrire plus simplement A au lieu de *var*.A.

Exemples de requêtes

•Q(x)={ x.Ename | Emp(x) }

select Ename from Emp

•Q(x)={ x.Ename | Emp(x) ∧ x.City = 'Paris' }

select Ename from Emp where City = Paris

•Q(x)={ x.Pname, x.Budget | Project(x) ∧
∃y (Works(y) ∧ x.Pno=y.Pno ∧ y.Eno='E1') }

select x.Pname, x.Budget from Project x, Works y
where x.Pno=y.Pno and y.Eno='E1'

•Q(x)={ x.City | Emp(x) ∨ Project(x) }

(select City from Emp) union (select City from Project)

•Q(x)={ *x*.City | Project(*x*) ∧ ¬∃*y* (Emp(*y*) ∧ *x*.City = *y*.City)}

select City from Project
where not exists (select * from Emp
where Project.City = Emp.City)

Requêtes et valeurs NULL

Les valeurs d'attributs peuvent être inconnues : NULL

- une *opération* avec un attribut de valeur NULL retourne NULL
- une *comparaison* avec un attribut de valeur NULL retourne **UNKNOWN**
- UNKNOWN introduit une logique *à trois valeurs* :
- Vrai = 1, UNKNOWN = 0.5, Faux = 0
- x AND y = min(x,y), x OR y = max(x,y), not(x) = 1-x
- Attention : NULL n'est pas une constante :
  - « NAME = NULL » ou « NULL + 30 » sont incorrects.
  - Pour vérifier si la valeur d'un attribut est inconnue, on utilise IS NULL :

```
SELECT Pname
FROM Proj WHERE City IS NULL
```


Requête avec NULL

Project

Pno	Pname	Budget	City
1	Développement	NULL	Paris
2	Conception	20000	Lyon

```
SELECT Pno FROM Project
WHERE Budget = 1000
```

→ **Réponses ???**

```
SELECT P1.Pno FROM Project P1, Project P2
WHERE P1.Budget = P2.Budget AND P1.Pno <> P2.Pno
```


Insertion de tuples

INSERT INTO ***table*** **[(** ***column*** **[, ...])]**
{ VALUES ({ *expression* | DEFAULT } [, ...]) | *query* }

•en spécifiant des valeurs différentes pour tous les attributs *dans l'ordre* utilisé dans CREATE TABLE :

INSERT INTO *R* **VALUES** (value(A_1), …, value(A_n))

•en spécifiant les noms d'attributs (indépendant de l'ordre) :

INSERT INTO *R* (A_i, …, A_k) **VALUES** (value(A_i), …, value(A_k))

•insertion du résultat d'une requête (copie) :

INSERT INTO *R* <*requête_SQL*>

Suppression de tuples

DELETE FROM *table* **[WHERE** *condition* **]**

Supprimer tous les employés qui ont travaillé dans le projet P3 pendant moins de 3 mois :

```
DELETE FROM Emp
WHERE Eno IN ( SELECT Eno
                 FROM Works
                WHERE PNO='P3' AND Dur < 3)
```

Attention : il faut aussi effacer les n-uplets correspondants dans la table Works (cohérence des données).

Modification de tuples

UPDATE *table*
SET *column* = { *expression* | **DEFAULT** } **[, ...]**
[WHERE *condition* **]**

UPDATE *R*
SET A_i=value, …, A_k=value
WHERE *P*

Commandes DDL

Création de schémas :

CREATE SCHEMA *nom_schema* **AUTHORIZATION** *nom_utilisateur*

Création de tables :

```
CREATE TABLE nom_table
   (Attribute_1 <Type>[DEFAULT <value>],
    Attribute_2 <Type>[DEFAULT <value>],
    …
    Attribute_n <Type>[DEFAULT <value>]
   [<Constraints>])
```


Type DATE

- Format par défaut : YYYY-MM-DD
- Fonctions :
  - SYSDATE : date / heure actuelle
  - TO_DATE('98-DEC-25:17:30','YY-MON-DD:HH24:MI')

SELECT * **FROM** my_table
WHERE datecol = TO_DATE('04-OCT-2010','DD-MON-YYYY');

- Arithmétique :
  Date +/- N jours
  SYSDATE + 1 = demain

On peut utiliser la fonction Extract (date|year|…) from expression_date

Autres commandes DDL

DROP SCHEMA *nom_schema* [,…] [CASCADE | RESTRICT]

- supprime les schémas indiqués
- CASCADE : toutes les tables du schéma
- RESTRICT : seulement les tables vides (par défaut)

DROP TABLE *nom_table* [,…] [CASCADE | RESTRICT]

- RESTRICT : supprime la table seulement si elle n'est référencée par aucune contrainte (clé étrangère) ou vue (par défaut)
- CASCADE : supprime aussi toutes les tables qui « dépendent » de *nom_table*

ALTER TABLE *nom_table* OPERATION

- *modifie* la définition de la table
- opérations:
- Ajouter (ADD), effacer (DROP), changer (MODIFY) attributs et contraintes
- changer propriétaire, …

SQL : Requêtes imbriquées

Requête imbriquée dans la clause WHERE d'une requête externe:

```
SELECT …
FROM   …
WHERE  [Opérande] Opérateur (SELECT …
                             FROM …
                             WHERE …)
```

Opérateurs ensemblistes :

- **(A1,…An) IN <sous-req>** : appartenance ensembliste
- **EXISTS <sous-req>** : test d'existence
- **(A1,…An) <comp> [ALL|ANY] <sous-req>** : comparaison avec quantificateur (ANY par défaut)

Expression « IN »

```
SELECT …
FROM …
WHERE (A1,…,An) [NOT] IN (SELECT B1,…,Bn
                           FROM …
                           WHERE …)
```

Sémantique : la condition est vraie si le n-uplet désigné par $(A_1, \ldots, A_n)$ de la requête *externe* appartient (n'appartient pas) au résultat de la requête *interne*.

ALL/ANY

```
SELECT …
FROM …
WHERE (A1,…,An) θ  ALL/ANY (SELECT B1,…,Bn
                            FROM …
                            WHERE …)
```

- On peut utiliser une comparaison $\theta \in \{=, <, <=, >, >=, <>\}$ et ALL ($\forall$) ou ANY ($\exists$): La condition est alors vraie si la comparaison est vraie pour tous les n-uplets /au moins un n-uplet de la requête interne.

Comment peut-on exprimer « IN » avec ALL/ANY?

Expression "EXISTS"

Q

```
SELECT …
FROM …
WHERE
```

Q'

```
[NOT] EXISTS (SELECT *
              FROM …
              WHERE P)
```

- *Sémantique procédurale :*
  - pour chaque n-uplet *x* de la requête externe Q, exécuter la requête interne Q'; s'il existe au moins un n-uplet y dans le résultat de la requête interne, alors sélectionner *x*.
- *Sémantique logique :*
  - $\{ x\ldots \mid Q(x) \wedge [\neg]\exists y\ (Q'(y)) \}$
- Les deux requêtes sont généralement *corrélées* : la condition *P* dans la requête interne Q' exprime une jointure entre les tables de Q' et les tables de la requête externe Q.

SQL : Fonctions d'agrégation

Pour calculer une valeur numérique à partir d'une relation on applique des fonctions d'agrégation Agg(A) ou A est un nom d'attribut (ou *) et Agg est une fonction parmi :

- COUNT(A) ou COUNT(*) : nombre de valeurs ou n-uplets,
- SUM(A) : somme des valeurs,
- MAX(A) : valeur maximale,
- MIN(A) : valeur minimale,
- AVG(A) : moyenne des valeurs

dans *l'ensemble des valeurs* désignées par A

```
SELECT Agg1(Ai), …, Agg(Aj)
FROM   R1, ..., Rm
WHERE  P
```


Requêtes de groupement : GROUP BY

Pour *partitionner* les n-uplets résultats en fonction des valeurs de certains attributs :

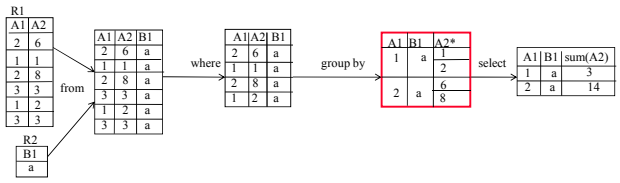
```
SELECT Ai, …, An, aggr1, aggr2, …
FROM   R1, …, Rm
WHERE  P
GROUP BY Aj …, Ak
```

Règle: **tous** les attributs projetés (A_i, …, A_n) dans la clause SELECT

- *n'apparaissent pas dans une opération d'agrégation* et
- *sont inclus* dans l'ensemble des attributs (A_j …, A_k) de la clause GROUP BY (qui peut avoir d'autres attributs en plus)

GROUP BY

```
SELECT A1, B1, sum(A2)
FROM   R1, R2
WHERE  A1 < 3
GROUP BY A1, B1
```



Predicats sur des groupes

Pour *garder (éliminer) les groupes (partitions)* qui satisfont (ne satisfont) pas une certaine condition :

```
SELECT A1, …, An
FROM   R1, …, Rm
WHERE  P
GROUP BY Aj …, Ak
HAVING Q
```

Règle : La condition *Q* porte sur des valeurs atomiques retournées par un *opérateur d'agrégation* sur les attributs qui *n'apparaissent pas* dans le GROUP BY

Calcul relationnel et SQL

Convertir en SQL :

« Quels employés travaillent dans tous les projets »

La requête SQL correspondante est:

```
SELECT e.Eno
FROM   Emp e
WHERE NOT EXISTS
  (SELECT *
   FROM   Project p
   WHERE NOT EXISTS
     (SELECT *
      FROM   Works w
      WHERE  p.Pno=w.Pno
        AND  e.Eno = w.Eno))
```

```
{ e.Eno |
Emp(e) ∧
    ¬∃p(
        Project(p) ∧
        ¬ ∃ w(
            Works(w) ∧
            p.Pno = w.Pno
            ∧ w.Eno=e.Eno ))}
```


Couplage SQL–langage de programmation (Embedded SQL)

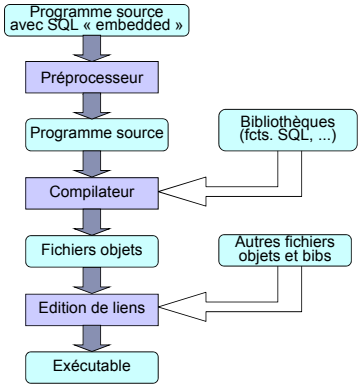
Comment accéder une BD depuis un programme ?

- SQL n'est pas suffisant pour écrire des applications (SQL n'est pas « Turing complet »)

SQL a des liaisons (bindings) pour différents langages de programmation

- C, C++, Java, PHP, etc.
- les liaisons décrivent la façon dont des applications écrites dans ces langages hôtes peuvent interagir avec un SGBD relationnel

Développement d'application



Utilisation de Embedded SQL

Interface = commandes « **EXEC SQL** »

Variables partagées entre SQL et le langage hôte (C, PHP, Java, ..) pour :

- passer des paramètres aux requêtes avant leur évaluation (par exemple nom d'une personne lu par le programme, ...)
- accéder au résultats des requêtes dans le programme (par exemple afficher le résultat)

Exemple de curseur

Pour chaque projet employant plus de 2 programmeurs, donner le numéro de projet et la durée moyenne d'affectation des programmeurs

```
…
EXEC SQL BEGIN DECLARE SECTION;
  char pno[3]; /* project number */
  real avg-dur; /* average duration */
EXEC SQL END DECLARE SECTION;
…
EXEC SQL DECLARE duration CURSOR FOR
    SELECT Pno, AVG(Dur)
    FROM   Works
    WHERE  Resp = 'Programmer'
    GROUP BY Pno
    HAVING COUNT(*) > 2;
…
EXEC SQL OPEN duration;
…
while(1) {
  EXEC SQL FETCH FROM duration INTO :pno, :avg-dur
  if(strcmp(SQLSTATE, "02000") then break
  else < print the info >
}
EXEC SQL CLOSE duration
…
```

Déclaration du curseur

Exécution de la requête

Lecture n-uplets

Fermeture curseur

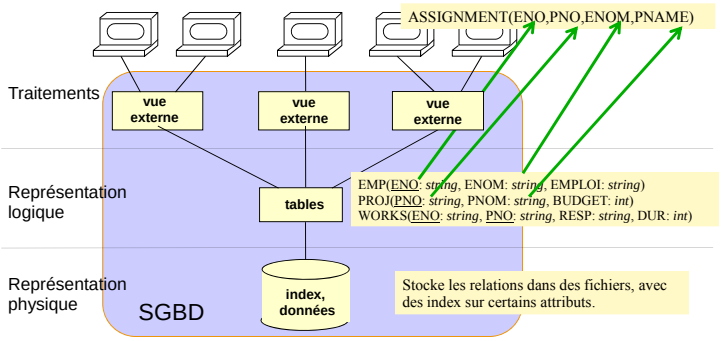
Mise-à-jour en Embedded SQL

Exemple: transfert d'un montant entre deux budgets de projets

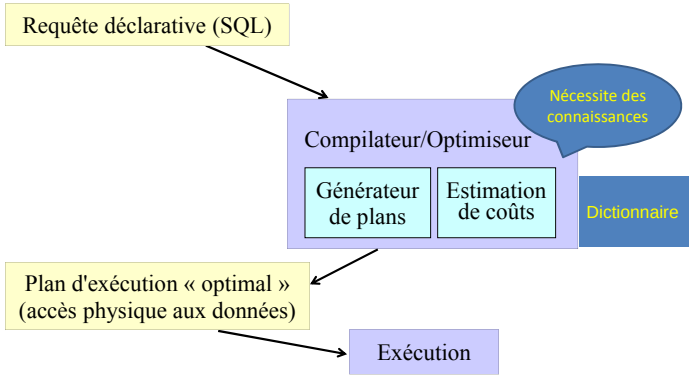
```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main() {
     EXEC SQL WHENEVER SQLERROR GOTO error:
     EXEC SQL CONNECT TO Company;
     EXEC SQL BEGIN DECLARE SECTION;
       int pno1, pno2; /* 2 numéros de projet */
       int amount; /* montant du transfert */
     EXEC SQL END DECLARE SECTION;
     /* Code (omis) pour lire pno1, pno2 et amount */
     EXEC SQL UPDATE Project
       SET Budget = Budget + :amount
       WHERE Pno = :pno2;
     EXEC SQL UPDATE Project
       SET Budget = Budget - :amount
       WHERE Pno = :pno1;
     EXEC SQL COMMIT;
     return(0);
 error:
     printf("update failed, sqlcode = %ld\n", SQLCODE);
     EXEC SQL ROLLBACK;
     return(-1);
 }
```

Cours 2 : Algèbre relationnelle

Architecture ANSI/SPARC



Traitement de requêtes



Problème général

Comment évaluer une requête SQL efficacement ?

Le nom et le titre des employés qui travaillent dans des projets avec un budget > 250 et plus que 2 employés:

```
SELECT DISTINCT Ename, Title
  FROM Emp, Project, Works
 WHERE Budget > 250
   AND Emp.Eno=Works.Eno
   AND Project.Pno=Works.Pno
   AND Project.Pno IN (SELECT Pno
                       FROM Works
                    GROUP BY Pno
                      HAVING COUNT(*) > 2)
```

Problème

- Requête SQL = expression *déclarative*
- Plan d'exécution = programme *impératif*
  - Boucles, tests, …
  - Opérations sur des tables et des index : ***algèbre***
  - Génération de tables et index temporaires
- Problème : Comment trouver un plan d'exécution
  - Correct : il fait ce que la requête "dit"
  - **Efficace : il le fait vite!**

Problème complexe car de nombreux plans possibles (encore plus si la BD est répartie)

Évaluation et optimisation de requêtes

- Traitement de requêtes SQL
- Algèbre relationnelle
- Optimisation de requêtes

Langages d'Interrogation Relationnels

- SQL :
  - langage « pratique » pour la programmation
- Calcul relationnel :
  - formules logiques qui décrivent le « sens » formel d'une requête SQL (sauf group by, order).
- Algèbre :
  - composition d'opérations qui décrit une exécution possible d'une requête SQL / calcul
  - Expression relationnelle (ex. la relation R) : retourne un ensemble de n-uplets (ex. le contenu de R)
  - « algèbre » : on reste dans l'espace des expr. rel.

Algèbres relationnelles

- Algèbre ensembliste : opérateurs sur des *ensembles* de n-uplets (relations)
- Algèbre physique : opérateurs *implantés* dans un SGBD
- Pour un opérateur logique il existe généralement plusieurs opérateurs physiques qui l'implantent (choix d'opérateurs)
- Certains opérateurs physiques n'ont pas d'opérateur équivalent au niveau des ensembles (tri, group-by, ...) ou alors il faut considérer une *algèbre étendue*
  - Algèbre ⇔ calcul algèbre étendue ⇔ SQL

Algèbre relationnelle étendue

- Inclus opérateurs pour group by, select avec doublon et order by
- Group by : opérateurs complexe à modéliser, surtout si clause having
- Select avec doublons : nécessite de changer de modèle (on ne travaille plus sur des ensembles, mais sur des « ensembles avec doublon »)
- Order by : nécessite de changer le modèle (on ne travaille plus sur des ensembles de n-uplets, mais sur des listes ordonnées)

⇒ Pour simplifier on étudiera principalement des requêtes sans group by, sans doublons et sans order by

Algèbre relationnelle (ensembliste)

Opérateurs unaires :

<Opérateur>$_{<parametres>}$ *<Opérande>* → *<Résultat>*

Opérateurs binaires :

<Opérande> *<Opérateur>*$_{<parametres>}$ *<Opérande>* → *<Résultat>*

Langage *fermé* : les opérandes et les résultats sont *toujours* des relations (ensembles de n-uplets) → composition d'opérations

Opérateurs de l'algèbre relationnelle

Opérateurs de base :

- sélection
- projection
- produit cartésien
- opérations ensemblistes: union, différence
- *renommage*

Opérateurs dérivés :

- intersection
- jointure
- division

Sélection

Sélection d'un sous-ensemble de la relation opérande :

$$\sigma_F(R)$$

- *R* est une expression relationnelle
- *F* est une *formule logique sans quantificateur* composée de
  - opérandes: constantes et attributs
  - opérateurs de comparaison :<, >, =, ≠, ≤, ≥
  - opérateurs logiques : ∧, ∨ , ¬

Résultat : *sous-ensemble* des n-uplets de *R* qui satisfont la formule F

En SQL ?

Exemple de sélection

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

$\sigma_{\text{TITLE='Elect. Eng.'}}(\text{EMP})$

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E6	L. Chu	Elect. Eng.

Projection

Projection sur un ensemble d'attributs d'une relation

$$\pi_{A_1,\ldots,A_n}(R)$$

- R est une expr. relationnelle
- $\{A_1,\ldots, A_n\}$ est un sous-ensemble des attributs de R

Résultat : ensemble de n-uplets de R sans les attributs (colonnes) qui ne se trouvent pas dans $\{A_1,\ldots, A_n\}$

En SQL ?

Exemple de projection

PROJ

PNO	PNAME	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	*CAD/CAM*	250000
P4	Maintenance	310000
P5	*CAD/CAM*	500000

$\Pi_{\text{PNO,BUDGET}}(\text{PROJ})$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000
P5	500000

$\Pi_{\text{PNAME}}(\text{PROJ})$

PNAME
Instrumentation
Database Develop.
CAD/CAM
Maintenance

Deux sémantiques :
- Ensembliste : élimination des *n-uplets doublons*
- SQL : avec doublons => distinct.

Pourquoi ?

Produit cartésien

Produit cartésien entre deux tables :

$$R \times S$$

- R est une table de degré k_1, cardinalité n_1
- S est une table de degré k_2, cardinalité n_2

Résultat : relation de *degré* $(k_1 + k_2)$ et contient $(n_1 * n_2)$ n-uplets, où chaque n-uplet est la *concaténation* d'un n-uplet de R avec un n-uplet de S.

En SQL
Select … from R, S
where <pas de lien entre R et S>
(« Monsieur c'est bloqué ! ») ?

Exemple de produit cartésien

Clé ?

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

8 n-uplets

PAY

TITLE	SALARY
Elect. Eng.	55000
Syst. Anal.	70000
Mech. Eng.	45000
Programmer	60000

4 n-uplets

EMP × PAY

ENO	ENAME	EMP.TITLE	PAY.TITLE	SALARY
E1	J. Doe	Elect. Eng.	Elect. Eng.	55000
E1	J. Doe	Elect. Eng.	Syst. Anal.	70000
E1	J. Doe	Elect. Eng.	Mech. Eng.	45000
E1	J. Doe	Elect. Eng.	Programmer	60000
E2	M. Smith	Syst. Anal.	Elect. Eng.	55000
E2	M. Smith	Syst. Anal.	Syst. Anal.	70000
E2	M. Smith	Syst. Anal.	Mech. Eng.	45000
E2	M. Smith	Syst. Anal.	Programmer	60000
E3	A. Lee	Mech. Eng.	Elect. Eng.	55000
E3	A. Lee	Mech. Eng.	Syst. Anal.	70000
E3	A. Lee	Mech. Eng.	Mech. Eng.	45000
E3	A. Lee	Mech. Eng.	Programmer	60000
...				
E8	J. Jones	Syst. Anal.	Elect. Eng.	55000
E8	J. Jones	Syst. Anal.	Syst. Anal.	70000
E8	J. Jones	Syst. Anal.	Mech. Eng.	45000
E8	J. Jones	Syst. Anal.	Programmer	60000

32 n-uplets

Union

Union *ensembliste* entre deux tables :

$R \cup S$

- *R* et S sont des relations *compatibles pour l'union* (même arité et domaines d'attributs)

Résultat : n-uplets qui sont dans *R* ou dans *S*

PAY1

TITLE	SALARY
Elect. Eng.	55000
Syst. Anal.	70000
Mech. Eng.	*45000*

∪

PAY2

TITLE	SALARY
Syst. Anal.	73100
Mech. Eng.	*45000*
Programmer	60000

=

PAY1 ∪ PAY2

TITLE	SALARY
Elect. Eng.	55000
Syst. Anal.	70000
Mech. Eng.	*45000*
Programmer	60000
Syst. Anal.	73100

Différence

Différence *ensembliste* entre deux tables :

$R - S$

- *R* et S sont des relations *compatibles* pour l'union.

Résultat : n-uplets qui sont dans *R*, mais pas dans *S*

PAY1

TITLE	SALARY
Elect. Eng.	55000
Syst. Anal.	70000
Mech. Eng.	*45000*

–

PAY2

TITLE	SALARY
Syst. Anal.	73100
Mech. Eng.	*45000*
Programmer	60000

=

PAY1 – PAY2

TITLE	SALARY
Elect. Eng.	55000
Syst. Anal.	70000

Renommage

Renommage de plusieurs attributs d'une table :

$\rho_{A_1,\ldots,A_n \rightarrow B_1,\ldots,B_n}(R)$

R est une relation

$\{A_1,\ldots, A_n\}$ est un sous-ensemble des attributs de *R*

$\{B_1,\ldots, B_n\}$ est un ensemble d'attributs

Résultat : une relation avec les mêmes n-uplets (le même contenu) où chaque attribut A_i a été renommé en B_i

On note aussi $R_{A1 \rightarrow B1,\ A2 \rightarrow B2\ldots}$

On peut aussi renommer la relation R en S, noté R_S

Exemple de renommage

- EMP(ENO, TITLE, ENAME)
- $\rho_{TITLE,ENAME \rightarrow JOB, NOM}(EMP)$ change le schéma de la relation EMP en EMP (ENO, NOM, JOB)
- ne change rien au contenu

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

$\rho_{TITLE,ENAME \rightarrow JOB, NOM}$

EMP

ENO	NOM	JOB
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

Opérateurs de l'algèbre relationnelle

Opérateurs de base :
- sélection
- projection
- produit cartésien
- opérations ensemblistes: union, différence
- *renommage*

Opérateurs dérivés :
- intersection
- jointure
- division

Intersection

Intersection de deux tables:

$$R \cap S = R - (R - S)$$

R, S sont deux tables compatibles (attributs deux à deux de même domaine)

Résultat : ensemble de n-uplets qui se trouvent à la fois dans R *et* dans S

PAY1

TITLE	SALARY
Elect. Eng.	55000
Syst. Anal.	70000
Mech. Eng.	*45000*

$\cap$

PAY2

TITLE	SALARY
Syst. Anal.	73100
Mech. Eng.	*45000*
Programmer	60000

=

PAY1 $\cap$ PAY2

TITLE	SALARY
Mech. Eng.	*45000*

Jointure

Jointure entre deux tables R et S :

$$R \bowtie_F S = \sigma_F(R \times S)$$

R et S sont des relations (sans attributs en commun)

F est une formule logique composée d'au moins un atome de la forme $A_i\ \theta\ B_j$ où

$\theta \in \{<, >, =, \neq, \leq, \geq\}$, A_i est un attribut de R, B_j est un attribut de S

Résultat : sous-ensemble des n-uplets dans le produit cartésien *R* $\times$ *S qui satisfont la formule F*

Types de jointure

θ-jointure (théta-jointure)

la formule *F* utilise les comparaisons <, >, ≠, ≤, ≥

Equi-jointure

la formule *F* n'utilise que l'égalité : =

$R \bowtie_{R.A=S.B} S$

Jointure naturelle : R(X,Y), S(X, Y')

Equi-jointure où on élimine les attributs en communs

$R \bowtie S = \Pi_{R.X,\, R.Y,\, S.Y'} \sigma_F (R \times S) = \Pi_{S.X,\, R.Y,\, S.Y'} \sigma_F (R \times S)$

la condition de jointure F est R.X = S.X (X représente tous les attributs en commun entre R et S)

Exemple de jointure naturelle

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

PAY

TITLE	SALARY
Elect. Eng.	55000
Syst. Anal.	70000
Mech. Eng.	45000
Programmer	60000

EMP ⋈ PAY

ENO	ENAME	TITLE	SALARY
E1	J. Doe	Elect. Eng.	55000
E2	M. Smith	Analyst	70000
E3	A. Lee	Mech. Eng.	45000
E4	J. Miller	Programmer	60000
E5	B. Casey	Syst. Anal.	70000
E6	L. Chu	Elect. Eng.	55000
E7	R. Davis	Mech. Eng.	45000
E8	J. Jones	Syst. Anal.	70000

Clé ?

Exemple de θ–jointure

EMP

ENO	ENAME	TITLE	CONTR
E1	J. Doe	Elect. Eng	12
E2	M. Smith	Syst. Anal.	12
E3	A. Lee	Mech. Eng.	12
E4	J. Miller	Programmer	24
E5	B. Casey	Syst. Anal.	24
E6	L. Chu	Elect. Eng.	36
E7	R. Davis	Mech. Eng.	36
E8	J. Jones	Syst. Anal.	12

WORKS

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E7	P5	Engineer	23
E8	P3	Manager	40

$EMP \bowtie_{EMP.ENO=WORKS.ENO \wedge CONTR<DUR} WORKS$

EMP. ENO.	ENAME	TITLE	WORKS. ENO	PNO	RESP	DUR	CONTR
E2	M. Smith	Syst. Anal.	E2	P1	Manager	24	12
E3	A. Lee	Mech. Eng.	E3	P4	Engineer	48	12
E6	L. Chu	Elect. Eng.	E6	P4	Manager	48	36
E8	J. Jones	Syst. Anal.	E8	P3	Manager	40	12

Résultat (en français) ?

Contrats et leurs employés embauchés sur une durée inférieure à la durée de leur participation au projet.

Division

Soient les relations R et S tels que *le schéma de S contient tous les attributs de R*

$R(A_1,\ldots,A_k,A_{k+1},\ldots,A_{k+n})$ est de degré $k+n$ et

$S(A_1,\ldots,A_k)$ de degré k.

La division de R par S :

$$T(A_{k+1},\ldots,A_{k+n}) = R \div S$$

est la « plus grande » relation de degré *n telle que*

$$T \times S \subseteq R.$$

$R \div S$ contient les tuples de schéma $(A_{k+1},\ldots,A_{k+n})$ qui sont associés, dans R, à tous les tuples de S.

Exemple de division

EMP

ENO	PNO	PNAME	BUDGET
E1	P1	Instrumentation	150000
E2	P1	Instrumentation	150000
E2	P2	Database Develop.	135000
E3	P1	Instrumentation	150000
E3	P4	Maintenance	310000
E4	P2	Instrumentation	150000
E5	P2	Instrumentation	150000
E6	P4	Maintenance	310000
E7	P3	CAD/CAM	250000
E8	P3	CAD/CAM	250000
E3	P2	Database Develop.	135000
E3	P3	CAD/CAM	250000

PROJ

PNO	PNAME	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000

EMP÷PROJ

ENO
E3

Expression de la division

R

X	Y
x1	y1
x2	y1
x3	y1
x4	y1
x1	y2
x3	y2
x2	y3
x3	y3
x4	y3
x1	y4
x2	y4
x3	y4

S

X
x1
x2
x3

T

Y
y1
y4

Exprimons T en fonction de R et S…

Remarques sur l'algèbre non étendue

Permet de retrouver toutes les valeurs contenues dans la BD (n-uplets, n-uplets tronqués, n-uplets concaténés)

Et rien d'autre…

Certains opérateurs sont commutatifs ($R \bowtie \sigma_F(S)) = \sigma_F(R \bowtie S)$

D'autres ne le sont pas $\Pi_X(R-S) != \Pi_X(R) - \Pi_X(S)$

Requêtes algébriques

Emp (Eno, Ename, Title, City)
Pay(Title, Salary)
Project(Pno, Pname, Budget, City)
Works(Eno, Pno, Resp, Dur)

Villes où il y a des employés ou des projets?

- $\Pi_{City}(Emp) \cup \Pi_{City}(Project)$

Villes où il y a des projets mais pas d'employés?

- $\Pi_{City}(Project) - \Pi_{City}(Emp)$

Requêtes algébriques

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

Noms des projets de budget > 225?

- $\Pi_{Pname}(\sigma_{Budget>225}(Project))$

Noms et budgets des projets où travaille l'employé E1?

- $\Pi_{Pname, Budget}(Project \bowtie (\sigma_{Eno='E1'}(Works)))$
- $\Pi_{Pname, Budget}(\sigma_{Project.Pno=Works.Pno}(Project \times \sigma_{Eno='E1'}(Works)))$

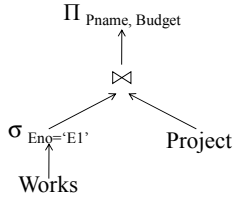
Employés qui travaillent dans chaque projet?

- $\Pi_{Eno, Pno}(Works) \div \Pi_{Pno}(Project)$

Arbre algébrique

Comme toute expression algébrique, peut se représenter sous forme d'arbre.
Utile pour manipuler les requêtes (optimisation, vues)
Exemple

$\Pi_{Pname, Budget}(Project \bowtie \sigma_{Eno='E1'}(Works)))$



Conclusion : algèbre relationnelle

- L'algèbre relationnelle définit un ensemble d'opérations pour interroger une BD relationnelle
- Les opérations peuvent être composées pour former des requêtes complexes
  - Avantage : facilite l'implantation d'un moteur de requêtes
  - Inconvénient : sémantique "opérationnelle"

L'algèbre « cache » la sémantique formelle (ensembliste) du modèle relationnel → calcul relationnel

Calcul de n-uplets et algèbre

- **Théorème:** Le calcul relationnel sûr et l'algèbre relationnelle ont une *puissance d'expression équivalente* (complétude relationnelle)
- *Autrement* : toutes les requêtes qu'on peut exprimer en utilisant l'algèbre relationnelle (sélection, projection, jointure, ...) peuvent être exprimées dans le calcul relationnel sûr et vice-versa.

Traduction: sélection, projection, jointure

Deux tables : R(A,B,C) S(C,D)

- $\Pi_{A,B}$ (R) ≡ { t.A, t.B | R(t) }
- $\Pi_B(\sigma_{A=3}(R))$ ≡ { t.B | R(t) ∧ t.A = 3 }
- R ⋈ S ≡ { t.A, t.B, t.C, u.D | R(t) ∧ S(u) ∧ t.C = u.C }
- Π_C (R) - Π_C (S) ≡ { t.C | R(t) ∧ ¬∃u (S(u) ∧ t.C = u.C)

Traduction de la division

R(A,B,C,D) ÷ S(C,D) s'exprime par la requête suivante :

R÷S = { x.A, x.B | R(x) ∧
∀u (S(u) → ∃v (R(v) ∧
v.A = x.A ∧ v.B = x.B ∧
v.C = u.C ∧ v.D = u.D)) }

Remarque : F → G est équivalent à ¬ F ∨ G

Requêtes algébriques (suite)

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

1. Projets ayant au moins deux employés?
2. Projets ayant exactement deux employés ?
3. Couples d'employé (même nom, même ville) ?
4. Quel grade (title) est le mieux payé ?
5. Quels sont les projets où tous les grades sont représentés ?
6. Quels employés n'habitent pas la(es) ville(s) où ils travaillent ?

Requêtes algébriques (TME)

- **Sponsorise(NSp, NJo, Somme),**
- **Joueur(NJo, Eq, Taille, Age),**
- **Equipe(NEq, Ville, Couleur, StP)**
- **Match(Eq1, Eq2, Date, St),**
- **Distance(St1, St2, NbKm)**

1. Quelles équipes ont déjà joué au stade préféré de l'équipe des Piépla ?
2. Quels sont les joueurs qui ne sont pas sponsorisés par Adadis ?
3. Quel est le(s) plus grand(s) joueur(s) sponsorisé par Adadis ?
4. A quelle date a eu lieu un match entre deux équipes sponsorisées par le même sponsor ?

LU3IN009 Licence d'informatique

Cours 3 : Traitement et optimisation de requêtes

EMP(ENO, ENAME, TITLE)
PROJECT(PNO, PNAME, BUDGET)
WORKS(ENO, PNO, RESP, DUR)

•Problème

Soit la requête

pour chaque projet de budget > 250 qui emploie plus de 2 employés, donner le nom et le titre des employés

Comment l'exprimer en SQL ?

- **Avec count : traduire en algèbre ?**
- **Sans count : plus complexe (variables supplémentaires)**

Avec count :

```
SELECT  DISTINCT Ename, Title
        FROM    Emp, Project, Works
        WHERE   Budget > 250
        AND     Emp.Eno=Works.Eno
        AND     Project.Pno=Works.Pno
        AND     Project.Pno IN
                (SELECT Pno
                 FROM   Works
                 GROUP BY Pno
                 HAVING COUNT(*) > 2);
```

Sans count :

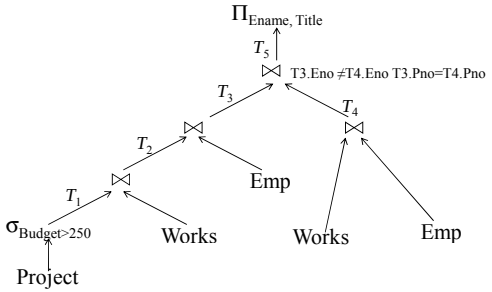
```
SELECT  DISTINCT E1.Ename, E1.Title
        FROM    Emp E1, Project, Works W1, Emp E2, Works W2
        WHERE   Budget > 250
        AND     E1.Eno=W1.Eno AND E2.Eno = W2.Eno
        AND     Project.Pno=W1.Pno AND W2.Pno = W1.Pno
        AND     E1.Eno <> E2.Eno;
```


Un plan d'exécution possible

```
SELECT  DISTINCT E1.Ename, E1.Title
        FROM    Emp E1, Project, Works W1, Emp E2, Works W2
        WHERE   Budget > 250
        AND     E1.Eno=W1.Eno AND E2.Eno = W2.Eno
        AND     Project.Pno=W1.Pno AND W2.Pno = W1.Pno
        AND     E1.Eno <> E2.Eno
        )
```

$T_1 \leftarrow$ *Lire* la table `Project` et *sélectionner* les tuples de `Budget` > 250
$T_2 \leftarrow$ *Joindre* T_1 avec la relation `Works`
$T_3 \leftarrow$ *Joindre* T_2 avec la relation `Emp`
$T_4 \leftarrow$ *Joindre* `Works` avec la relation `Emp`
$T_5 \leftarrow$ *Joindre* T_3 avec T_4 sur Pno enn vérifiant T3.Eno ≠T4.Eno
$T_6 \leftarrow$ *Projeter* T5 sur `Ename`, `Title`
Résultat ← *Éliminer doublons* dans T_6

Représentation algébrique



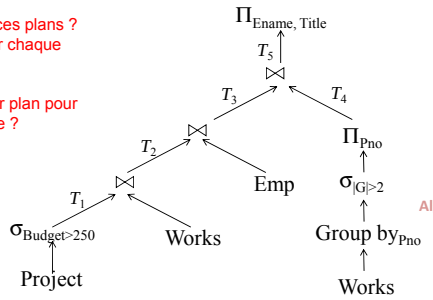
Un plan d'exécution possible (algèbre étendue)

```
SELECT   DISTINCT Ename, Title
FROM     Emp, Project, Works
WHERE    Budget > 250
AND      Emp.Eno=Works.Eno
AND      Project.Pno=Works.Pno
AND      Project.Pno IN
         (SELECT  Pno
          FROM    Works
          GROUP BY Pno
          HAVING  COUNT(*) > 2)
```

T_1 ← *Lire* la table `Project` et *sélectionner* les tuples de `Budget` > 250
T_2 ← *Joindre* T_1 avec la relation `Works`
T_3 ← *Joindre* T_2 avec la relation `Emp`
T_4 ← *Grouper* les tuples de `Works` sur `Pno` et pour les groupes qui ont plus de 2 tuples, *projeter* sur `Pno`
T_5 ← *Joindre* T_3 avec T_4
T_6 ← *Projeter* $T5$ sur `Ename, Title`
Résultat ← *Éliminer doublons* dans T_6

Représentation algébrique

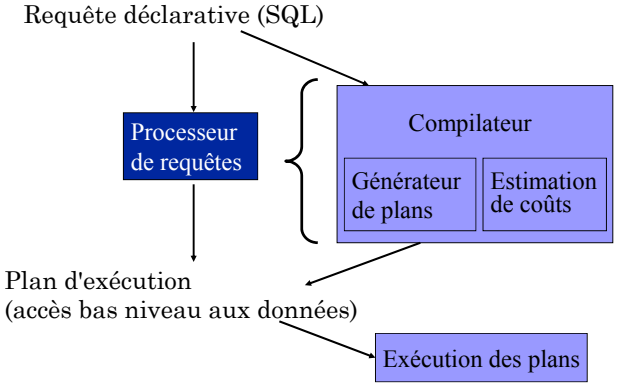
- Comment obtenir ces plans ?
- Comment exécuter chaque nœud / sous-arbre
- Quel est le coût ?
- Quel est le meilleur plan pour une requête donnée ?



Algèbre étendue

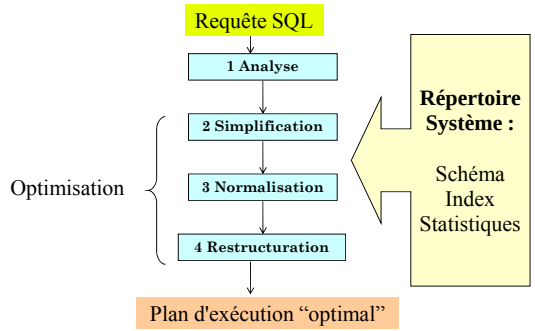
$\Pi_{Ename, Title}(\Pi_{Pno}(\sigma_{|G|>2}\ Group_{Pno}\ (Works)) \bowtie (Emp \bowtie ((\sigma_{Budget>250000}\ Project) \bowtie Works)))$

•Traitement des requêtes



8

Étapes du traitement d'une requête



•Normalisation de requête

- Analyse lexicale et syntaxique
  - vérification de la validité de la requête
  - vérification des attributs et relations
  - vérification du typage de la qualification
- Mise de la requête en forme normale
  - forme normale conjonctive
    $(p_{11} \vee p_{12} \vee \ldots \vee p_{1n}) \wedge \ldots \wedge (p_{m1} \vee p_{m2} \vee \ldots \vee p_{mn})$
  - forme normale disjonctive
    $(p_{11} \wedge p_{12} \wedge \ldots \wedge p_{1n}) \vee \ldots \vee (p_{m1} \wedge p_{m2} \wedge \ldots \wedge p_{mn})$
  - OR devient union
  - AND devient jointure ou sélection

•Simplification

- Pourquoi simplifier?
  - plus une requête est simple, plus son exécution peut être efficace
- Comment? en appliquant des transformations
  - élimination de la redondance
    - règles d'idempotence
      $p_1 \wedge \neg(p_1) \equiv$ faux
      $p_1 \wedge (p_1 \vee p_2) \equiv p_1$
      $p_1 \vee$ faux $\equiv p_1$
      …
  - application de la transitivité (att1=att2 ,att2=att3)
- Éliminer des opérations redondantes :
  - ex. : pas besoin de distinct après une projection sur une clé
- utilisation des règles d'intégrité
  - CI : att1 <100 Q: … where att1 > 1000…

élagage

Exemple de simplification

```
SELECT    Title
FROM      Emp
WHERE     Ename = 'J. Doe'          P1
OR        (NOT(Title = 'Programmer')    ¬P2
AND       (Title = 'Programmer'     P2
OR        Title = 'Elect. Eng.')    P3
AND       NOT(Title = 'Elect. Eng.'))   ¬P3
```

⇓ P1 ∨ (¬P2 ∧(P2 ∨ P3) ∧ ¬P3)

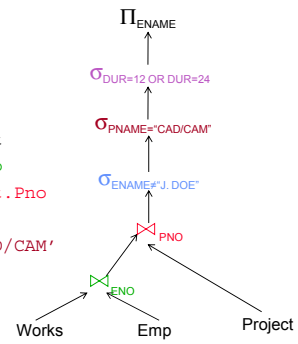
```
SELECT    Title
FROM      Emp
WHERE     Ename = 'J. Doe'
```

Traduction en algèbre

Conversion en arbre algébrique

Exemple (ordre de la clause where) :

```
SELECT   Ename
FROM     Emp, Works, Project
WHERE    Emp.Eno = Works.Eno
AND      Works.Pno = Project.Pno
AND      Emp.Ename <>'J.Doe'
AND      Project.name = 'CAD/CAM'
AND      (Works.Dur=12 OR
          Works.Dur=24)
```



Alternatives de traduction

```
SELECT   Ename
FROM     Emp e,Works w
WHERE    e.Eno = w.Eno
  AND    w.Dur > 37
```

Stratégie 1:

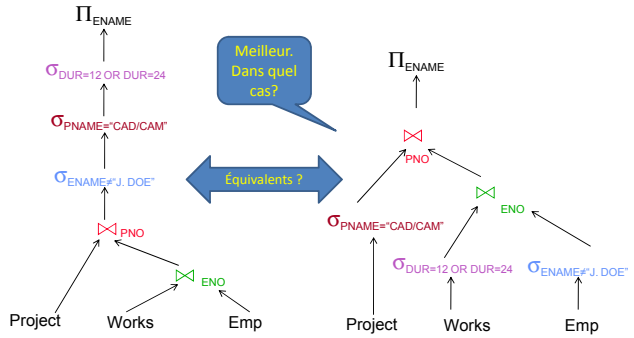
$$\Pi_{ENAME}(\sigma_{DUR>37 \wedge EMP.ENO=WORKS.ENO}(Emp \times Works))$$

Stratégie 2:

$$\Pi_{ENAME}(Emp \bowtie_{ENO} (\sigma_{DUR>37}(Works)))$$

- La stratégie 2 semble "meilleure" car elle évite un produit cartésien et sélectionne un sous-ensemble de Works avant la jointure
- Problème : Comment mesurer la qualité d'une stratégie ?

Alternatives de traduction



Optimisation de requête

Objectif : trouver le plan d'exécution le moins « *coûteux* »

Fonction de coût : donne une *estimation* du coût total réel d'un plan d'exécution

coût total = **coût I/O** (entrées/sorties) + coût CPU

- coût(I/O) ~ *1000* · coût(CPU) : on peut *négliger le coût CPU*
- I/O se calcule en *nombre de page* (même coût de transférer une page vide ou pleine)

Problème 1 : Définition d'une bonne **fonction de coût**

Solution : statistiques (à maintenir !) et fonctions d'estimations

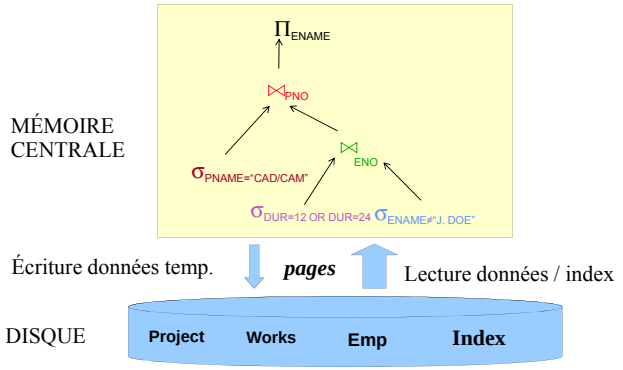
Problème 2 : Taille de l'**espace de recherche**

Espace de recherche = ensemble des expressions algébriques équivalentes pour une même requête.

Peut-être très grand. Optimisation en temps borné si non-compilé

Solution : *recherche non-exhaustive* d'une bonne solution (pas forcément la meilleure) en limitant l'espace de recherche, ou utilisation d'*heuristiques*

Coût d'un plan d'exécution



Estimer le coût d'un plan

La fonction de coût donne une estimation des temps I/O et CPU

nombre instructions et accès disques (écriture/lecture en nb pages)

1. Estimation du ***nombre d'accès disque*** pendant l'évaluation de chaque nœud de l'arbre algébrique

   Dépend entre autres de la place mémoire disponible/taille des opérandes (principalement pour jointure) et de l'algo. utilisé pour mettre en œuvre l'opérateur (cf. semaine prochaine)

2. Estimation de la ***taille du résultat*** de chaque nœud par rapport à ses entrées :

   sélectivité des opérations – « facteur de réduction »

   influe sur la taille du résultat = opérande du prochain opérateur

   basé sur les statistiques maintenues par le SGBD

Estimer le coût d'un plan

Deux hypothèses (fortes) :

1. Hypothèse d' ***uniformité :*** les différentes valeurs d'un attribut ont la même probabilité.

   Hypothèse plausible dans certains cas, pas dans d'autres (ex. âge)

2. Hypothèse d'***indépendance*** des attributs : la probabilité d'un attribut ne dépend pas de la proba. d'un autre.

   plausible : taille et couleur des cheveux

   peu plausible : taille et âge

Ces hypothèses sont trop forte mais

- On ne sait pas faire mieux sinon il faut stocker des histogrammes …
  - Coûteux
  - Compromis lecture / écriture
- L'erreur n'est pas fatale : au pire on choisit une solution un peu lente

Tailles des relations intermédiaires

Sélection :

$taille(R) = card(R) * largeur(R)$

$card(\sigma_F(R)) = SF_\sigma(F) * card(R)$

où SF_σ est une *estimation* de la **sélectivité du prédicat** , dont la forme générale est "taille des sélectionnés / taille des possibles (domaine)"

Si A continue. Sinon, remplacer par Card

$SF_\sigma(A = valeur) = \frac{1}{card(\prod_A(R))}$	
$SF_\sigma(A > valeur) = \frac{max(A) - valeur}{max(A) - min(A)}$	$SF_\sigma(A < valeur) = \frac{valeur - min(A)}{max(A) - min(A)}$
$SF_\sigma(p(A_i) \wedge p(A_j)) = SF_\sigma(p(A_i)) \bullet SF_\sigma(p(A_j))$	
$SF_\sigma(p(A_i) \vee p(A_j)) = SF_\sigma(p(A_i)) + SF_\sigma(p(A_j)) - (SF_\sigma(p(A_i)) \bullet SF_\sigma(p(A_j)))$	
$SF_\sigma(A \in \text{ens\_valeurs}) = SF_\sigma(A=valeur) \bullet card(\text{ens\_valeurs})$	

Tailles des relations intermédiaires

Projection

$card(\Pi_A(R)) <= card(R)$ (égalité si A est *unique*)

Produit cartésien

$card(R \times S) = card(R) \cdot card(S)$

Union

borne sup. : $card(R \cup S) = card(R) + card(S)$

borne inf. : $card(R \cup S) = max\{card(R), card(S)\}$

Différence

borne sup. : $card(R–S) = card(R)$ /* $R \cap S = \varnothing$

borne inf. : 0 /* $R \subset S$

Tailles des relations intermédiaires

Jointure :

- cas particulier: *A* est clé de *R* et *B est clé étrangère dans S vers R :*

$$card(R \bowtie_{A=B} S) = card(S)$$

- plus généralement

$$card(R \bowtie S) = SF_J \cdot card(R) \cdot card(S)$$

Comment l'obtenir ? Il faut des infos supplémentaire (SFj peut être stocké)

Règles de transformation

- Commutativité des opérations binaires
  - $R \times S \equiv S \times R$
  - $R \bowtie S \equiv S \bowtie R$
  - $R \cup S \equiv S \cup R$
- Associativité des opérations binaires
  - $(R \times S) \times T \equiv R \times (S \times T)$
  - $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
- Idempotence des opérations unaires
  - $\Pi_{A'}(\Pi_{A''}(R)) \equiv \Pi_{A'}(R)$
  - $\sigma_{p_1(A_1)}(\sigma_{p_2(A_2)}(R)) \equiv \sigma_{p_1(A_1) \wedge p_2(A_2)}(R)$
    - où $R[A]$ et $A' \subseteq A, A'' \subseteq A$ et $A' \subseteq A''$

Règles de transformation

- Commutativité de la sélection et de la projection (si proj. des attr. sél.)
- Commutativité de la sélection avec les opérations binaires

$\sigma_{p(A)}(R \times S) \equiv (\sigma_{p(A)}(R)) \times S$

$\sigma_{p(A_i)}(R \bowtie_{(A_j,B_k)} S) \equiv (\sigma_{p(A_i)}(R)) \bowtie_{(A_j,B_k)} S$

$\sigma_{p(A_i)}(R \cup T) \equiv \sigma_{p(A_i)}(R) \cup \sigma_{p(A_i)}(T)$

où A_i appartient à R et T

- Commutativité de la projection avec les opérations binaires

$\Pi_C(R \times S) \equiv \Pi_{A'}(R) \times \Pi_{B'}(S)$

$\Pi_C(R \bowtie_{(A_j,B_k)} S) \equiv \Pi_{A'}(R) \bowtie_{(A_j,B_k)} \Pi_{B'}(S)$

$\Pi_C(R \cup S) \equiv \Pi_C(R) \cup \Pi_C(S)$

où $R[A]$ et $S[B]$; $C = A' \cup B'$ où $A' \subseteq A$, $B' \subseteq B$, $A_j \subseteq A'$, $B_k \subseteq B'$

Règles de transformation (suite)

- Distributivité de le jointure par rapport à l'union

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

Ces règles de transformation permettent de passer d'une expression à une autre expression équivalente. Ceci permet d'explorer l'espace des plan d'exécution possibles pour une requête donnée

Heuristiques

Observation : opérations plus ou moins *coûteuses* et plus ou moins *sélectives*

Idée : réordonner les opérations :

faire les opérateurs les moins coûteux (projection, sélection) et les plus sélectives en premier, de manière à réduire la taille des données d'entrée pour les opérateurs les plus coûteux (jointure). La place en mémoire est un facteur primordial pour l'efficacité d'une jointure (cf. dans 2 semaines)

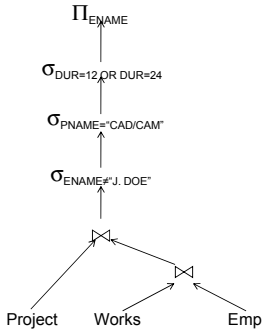
Méthode *heuristique* :

descendre les sélections, puis les projections au maximum grâce aux règles de transformation.

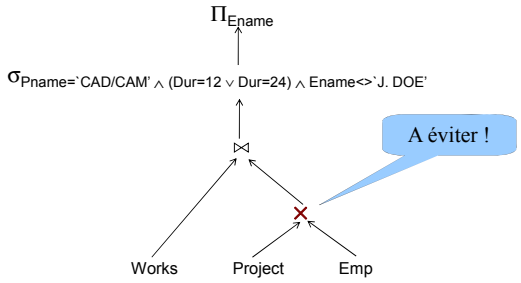
N'est pas toujours meilleur, car dépend de la présence d'index, de la nécessité d'écrire des relations temporaires…

Exemple

```
SELECT Ename
FROM Project p, Works w,
     Emp e
WHERE w.Eno=e.Eno
AND w.Pno=p.Pno
AND Ename<>`J. Doe'
AND p.Pname=`CAD/CAM'
AND (Dur=12 OR Dur=24)
```



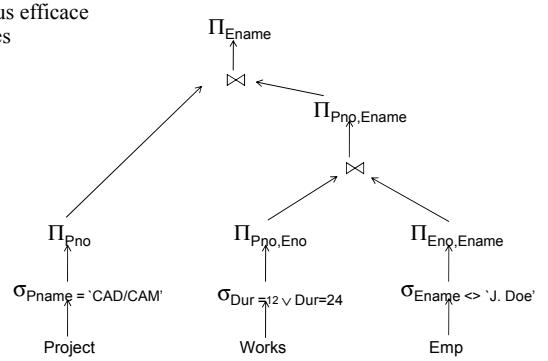
Requête équivalente



Appliquons l'heuristique décrite précédemment…

Autre requête équivalente

En principe plus efficace que les requêtes précédentes.



Conclusion

- Un SGBD doit transformer une requête déclarative en un programme impératif :
  - Plan d'exécution
  - Algèbre
- Calculer les tailles des résultats intermédiaire donne une idée du coût d'un plan mais..
  - Comment mettre en œuvre les opérateurs ?
  - Comment accéder aux données ?
  - Comment enchaîner les opérateurs ?
  - Comment trouver le meilleur plan en fonction de ce qui précède

Réponses dans les deux prochains cours

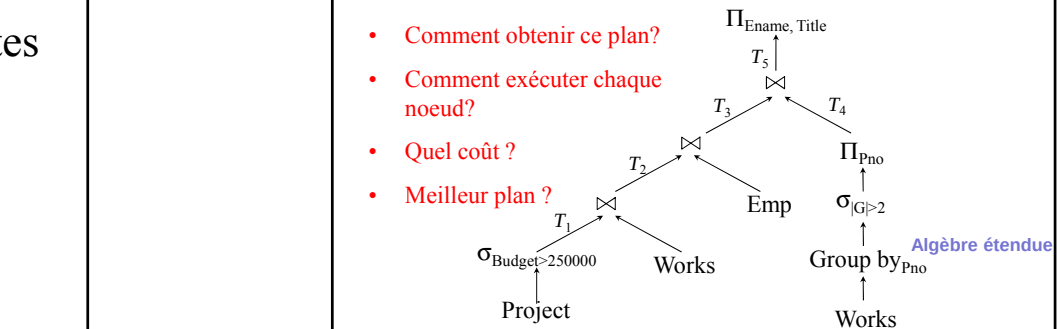
LU3IN009 Licence d'informatique

Cours 4 – Optimisation de requêtes

Stephane.gancarski@lip6.fr

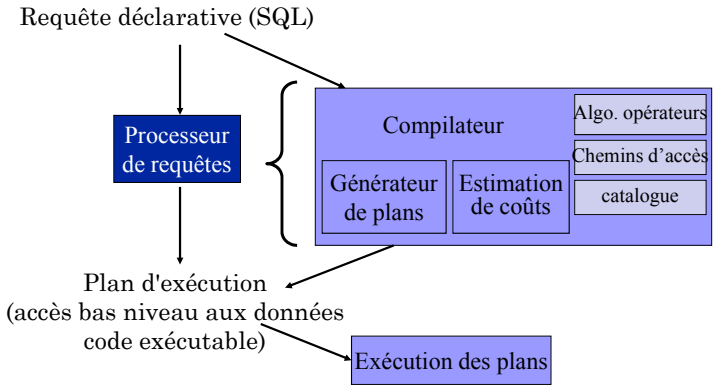
Plan d'exécution

- Comment obtenir ce plan?
- Comment exécuter chaque noeud?
- Quel coût ?
- Meilleur plan ?



$\Pi_{Ename, Title}(\Pi_{Pno}(\sigma_{|G|>2}\ Group_{Pno}\ (Works))$
$\bowtie$ (Emp $\bowtie$ (($\sigma_{Budget>250000}$ Project) $\bowtie$ Works)))

Traitement des requêtes



Cours 4 - Optimisation de requêtes

1. Organisation des données, chemins d'accès
2. Implémentation des opérateurs relationnels
3. Restructuration de la requête
4. Coût des opérations
5. Optimisation du coût
6. Espace de recherche
7. Stratégie de recherche

Stockage des données

- Les données sont stockées en mémoire non volatile
  - Disque magnétique, flash (carte SD, disque SSD), bande magnétique,
- Gestion de l'espace disque
  - L'unité de stockage est : la page
  - La taille d'1 page est fixe pour un SGBD (souvent 8Ko, parfois plus)
- 2 opérations élémentaires pour accéder au données stockées:
  - lire une page, écrire une page
- Le coût d'une opération SQL dépend principalement du nombre de pages lues et/ou écrites.
  - Coût E/S >> Coût calcul en mémoire
  - Coût dépend donc fortement de la façon dont les données sont organisées sur le disque → modèle de coût complexe.
- Gestion de l'espace en mémoire centrale
  - Réalisée par le SGBD (gestionnaire de tampon)
  - Gestion dédiée plus efficace qu'un OS généraliste.

Organisation des données

- Un enregistrement représente une donnée pouvant être stockée.
  - ex. une ligne ou une colonne d'une table
- Les enregistrements sont stockées dans les pages d'un fichier
- Un enregistrement a un identificateur unique servant d'adresse pour le localiser
  - (idFichier + idpage + offset ⇔ rowid dans Oracle).
  - Le gestionnaire de fichier peut accéder directement à la page sur laquelle se trouve un enregistrement grâce son adresse.
- La façon d'organiser les enregistrements dans un fichier a un impact important sur les performances.
  - Elle dépend du type de requêtes. Ex. OLTP (ligne) vs. OLAP (colonne).
  - Elle dépend aussi du type de mémoire. Ex. Flash très lent écriture.
  - Ce cours : stockage sur disque, OLTP
- Un SGBD offre en général plusieurs méthodes d'accès.
  - L'administrateur de la base détermine la méthode d'accès la plus adéquate

Organisation séquentielle

- Non trié :
  - Très facile à maintenir en mise à jour
  - Parcourir toutes les pages quelque soit la requête
- Trié :
  - Un peu plus difficile à maintenir
  - Parcours raccourci car on peut s'arrêter dès qu'on a les données cherchées

En BD, il y a presque toujours un compromis à faire entre lecture et écriture

Organisations Indexées

- Objectifs
  - Accès rapide à partir d'une clé de recherche
  - Accès séquentiel trié ou non
- Moyens
  - Utilisation d'index permettant la recherche de l'adresse de l'enregistrement à partir d'une clé de recherche
- Exemple
  - Dans une bibliothèque, rechercher des ouvrages par thème, par auteur ou par titre.
  - Dans un livre, rechercher les paragraphes contenant tel mot.

Entrée d'un index

- On appelle une entrée la structure qui associe une clé de recherche avec l'adresse des enregistrement concernés
  - Adresse : localisation d'un enregistrement
- Trois alternatives pour la structure d'une entrée
  1. Entrée d'index contient les données
     - localisation directe: on n'utilise pas l'adresse
  2. Entrée d'index contient (k, ptr)
     - Pas plus d'un enregistrement par valeur
  3. Entrée d'index contient (k, liste de ptr)
     - on peut avoir plusieurs enregistrements par valeur

Index non plaçant

- Les index **non plaçants** sont dit secondaires
- Index = structure auxiliaire en plus des données
- Permet d'indexer des données quelle que soit la façon dont elle sont stockées
  - Donnés stockées sans être triées
  - Données triées selon un attribut **autre** que celui indexé
- Définir un index non plaçant en SQL
  - create index *NOM* on *TABLE*(*ATTRIBUTS*);
    - create index IndexAge on Personne(âge)

Index plaçant

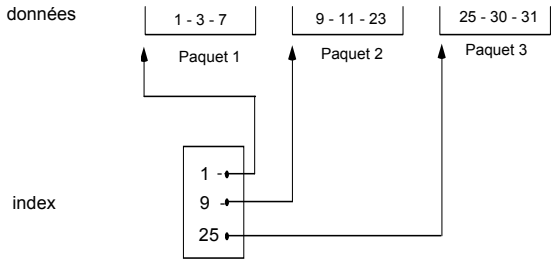
- Le stockage des données est organisé par l'index.
- Les enregistrements ayant la même valeur de clé sont juxtaposés
  - Stockage contigus dans un paquet et dans les paquets contigus suivants si nécessaire
- Définir l'organisation des données lors de la création de la table.
  - create table...organization index : données **triées** selon la clé primaire
  - create cluster... : données regroupées par valeur d'un attribut
- Un entrée contient les données (*cf.* alternative 1)
- Evidemment, pas plus d'un index plaçant par table
  - Appelé index principal

Index plaçant non dense

Un index contenant **toutes** les clés est dit **dense**

- Index non dense concerne seulement les index plaçants
  - Les données doivent être stockées triées
- Objectif: obtenir un index occupant moins de place
- Méthode: enlever des entrées. Ne garder que les entrées nécessaires pour atteindre le bloc (i.e. la page) de données contenant les enregistrements recherchés
  - Garder l'entrée ayant la plus petite (ou la plus grande) clé de chaque page.
  - Ne pas indexer 2 fois la même clé dans 2 pages consécutives
- Inconvénient: toutes les valeurs de l'attribut indexé ne sont pas dans l'index. Cf diapo (index couvrant une requête)

Exemple d'index plaçant non dense



Index unique

- Un index est dit unique si l'attribut indexé satisfait une contrainte d'unicité
  - Contrainte d'intégrité:
    - attribut (ou liste d'attributs) déclaré(e) comme étant : unique ou primary key
  - Une entrée a la forme (clé, ptr)
  - *cf.* alternative 2
- Sinon : cas général d'un index **non** unique
  - Une entrée a la forme (clé, **liste** de ptr)
  - *cf.* alternative 3

Accès aux données par un index

- Sert pour évaluer une sélection
  - une égalité: prénom = 'Alice'
  - l'accès est dit 'ciblé' si l'attribut est unique
  - un intervalle : age between 7 and 77
  - une inégalité : age > 18 <, >, ≤, ≥
  - une comparaison de préfixe : prénom like 'Ch%'
    - Rmq : un index ne permet **pas** d'évaluer une comparaison de suffixe. Exple prénom like '%ne'
  - Rmq: si les entrées de l'index ne sont pas triées (cas d'une table de hachage), seule l'égalité est possible de manière efficace

Index couvrant une requête

- Un index (non plaçant) couvre une requête s'il est possible d' « évaluer la requête » sans lire les données
- Tous les attributs mentionnés dans la requête doivent être indexés
- Index couvrant une sélection
  - Pour chaque prédicat *p* de la clause *where*, il faut un index capable d'évaluer *p*.
- Index couvrant une projection
  - Pour chaque attribut de la clause *select*, il faut un index **dense** (i.e, contenant toutes les valeurs de l'attribut projeté)
- Avantage
  - Evite de lire les données, évaluation plus rapide d'une requête
- Concerne seulement les index non plaçants
  - Un index plaçant contenant les données, elles sont forcément lues.

Index composé

- Clé composée considérée comme une clé simple formée de la concaténation des attributs
- Sélection par préfixe de la clé composée
  - Clé composée (a1, a2, a3, ..., an)
  - Il existe n préfixes : (a1), (a1,a2) , ..., (a1,a2, ...,an)
    - Rmq: (a2,a3) n'est pas un préfixe
- Index composé utilisable pour une requête
  - On appelle (p1, p2, ...p_m) les attributs mentionnés dans le prédicat de sélection
  - (p1, p2, ...p_m) doit être un préfixe de la clé composée
  - Prédicat d'**égalité** pour tous les attributs p1 à p_{m-1}
  - Egalité, inégalité ou comparaison de préfixe pour le dernier attribut p_m

Index composé: exemple

- Exemple : create index I1 on Personne(âge, ville)
- Utilisable pour les requêtes : Select * from Personne ...
  - Where âge > 18
  - Where âge =18 and ville = 'Paris'
  - Where âge =18 and ville like 'M%'
- Inutilisable pour :
  - Where ville= 'Paris'
  - Where âge > 18 and ville = 'Paris'

Choix entre un accès séquentiel ou un accès par index

- Définir un ou plusieurs index
- Poser des requêtes. Le SGBD utilise les index existants
  - s'il estime que c'est plus rapide que le parcours séquentiel des données.
  - Décision basée sur des règles heuristiques ou sur une estimation de la durée de la requête (voir TME)
- L'utilisateur peut forcer/interdire le choix d'un index
  - **Select** *
  - From Personne
  - Where age < 18
  - Devient
    - **Select /*+ index(personne IndexAge) */ ***
    - From Personne
    - Where age < 18
  - Syntaxe d'une directive:
    - index(*TABLE INDEX*)
    - no_index(*TABLE INDEX*)

Index hiérarchisé

- Lorsque le nombre d'entrées de l'index est très grand
- L'ensemble des entrées d'un index peuvent, à leur tour, être indexées. Cela forme un index hiérarchisé en plusieurs niveaux
  - Le niveau le plus bas est l'index des données
  - Le niveau n est l'index du niveau n+1
  - Intéressant pour gérer efficacement de gros fichiers
  - Le plus connu : arbre B+ (+ de détail au prochain cours)

Organisation par hachage

- Les fichiers sont placés dans des paquets en fonction d'une clé
- On applique une *fonction de hachage* sur la clé d'un n-uplet, ce qui détermine l'adresse du paquet où stocker le n-uplet
- On peut rajouter une indirection *: table de hachage*
- Efficace pour des accès par égalité, pas adapté aux requêtes par intervalle (données non triées)
- La fonction de hachage doit bien répartir les données dans les paquets
- Hachage statique vs hachage dynamique (+ de détail au prochain cours)

Implémentation des opérateurs

Rappel: accès disque >> accès mémoire (négligeable)
Coûts n'incluent pas écriture temporaire éventuelle
R contient *n* pages disques

- Sélection sur égalité, sur intervalle
- Projection
- Jointure

Sélection

- Sélection sur égalité
  - parcours séquentiel (scan)
    - le nombre d'accès disques est en O(*n*)
  - Parcours (scan) avec index
    - index B^+ : $O(\log_k(n))$ /* hauteur de l'arbre
      + un accès par nuplet (cf TME)
    - hachage : O(1) /* statique en supposant une bonne répartition. O(2) hachage dynamique
- Sélection sur intervalle
  - parcours séquentiel (scan) : idem
  - Parcours (scan) avec index
    - index B^+ : $O(\log_k(n) + M)$) + un accès par nuplet
      M nombre de pages contenant des clés correspondant à l'intervalle
    - hachage : O(X) où X est le nombre de valeur dans l'intervalle

Implémentation des opérateurs

- Projection
  - sans élimination des doubles – O(*n*)
  - avec élimination des doubles
    - en triant – $O(2n\log_k(n))$
    - en hachant – O(*n*+2*t*) où *t* est le nombre de pages du fichier haché après proj. et avant élimination
      - La fonction de hachage doit être choisie pour que, à chaque entrée de la table de hachage corresponde un nombre de pages assez petit pour tenir en mémoire
      - *t* vaut *n* si uniquement élimination des doubles, t < n si projection et élimination en même temps (les nuplets sont plus petits)

Implémentation des opérateurs

- Jointure (R sur n pages, S sur m pages)
  - boucle imbriquée (nested loop): $T = R \bowtie S$

```
foreach tuple r∈R do   /* foreach page de R
  foreach tuple s∈S do
    if r==s then T = T + <r,s>
```

    - O(*n*+*n***m*)
  - amélioration possible pour réduire les accès disques
    - boucles imbriquées par pages ou blocs : permet de joindre chaque n-uplet (page) de S avec non plus un seul n-uplet (page) de R, mais avec tous (on suppose p) ceux qui tiennent en MC (p+1)
    - O(n+m*n/p)
    - Cas particulier si R tient en mémoire, n/p = 1, O(n+m)

Implémentation des opérateurs

- Jointure
  - boucle imbriquée et index sur attribut de jointure de S (cas typique : jointure sur clé étrangère)

```
foreach tuple r∈R do
  accès aux tuples s∈S par index (ou hachage)
  foreach tuple s do
    T = T + <r,s>  /* O(n+M),M=card(R)*k(coût index)
```

  - tri-fusion
    - trier *R* et *S* sur l'attribut de jointure : tri externe $O(2nLog_k(n))$
    - fusionner les relations triées : O(n+m)
    - Peut être optimisé en commençant la fusion avant la fin complète du tri
  - hachage
    - hacher *R* et *S* avec la même fonction de hachage : O(n+m) L + O(n+m) E
    - pour chaque paquet i de *R* et i de *S*, trouver les tuples où *r*=*s* : O(n+m) L

Tri externe

- Algo
  - Trier des paquets de k pages tenant en mémoire disponible

    n/k paquets, 2n E/S
  - Charger les premières pages de chaque paquet et trier
    - Dès qu'une page est vide, charger la suivante du même paquet
    - On obtient des paquets de k^2 pages triés

    n/k^2 paquets, 2n E/S
  - Continuer jusqu'à obtenir un paquet de k^s >= n pages
- Coût total
  - A chaque étape on lit et écrit toutes les données : **2n E/S**
  - Nombre d'étape s, tel que k^s=n : **$s = \log_k(n)$**
  - Soit en tout $2n\log_k(n)$
  - Nb : si tri pour fusion, pas besoin de faire la dernière étape, on fait la fusion directement avec l'autre relation

Optimisation

- Elaborer des plans
  - arbre algébrique, restructuration, ordre d'évaluation
- Estimer leurs coûts
  - fonctions de coût
    - en terme de temps d'exécution
    - coût I/O + coût CPU
    - poids très différents
      - par ex. coût I/O = 1000 * coût CPU
- Choisir le meilleur plan
  - Espace de recherche : ensemble des expressions algébriques équivalentes pour une même requête
  - algorithmes de recherche:
    - parcourir l'espace de recherche
    - algorithmes d'optimisation combinatoire

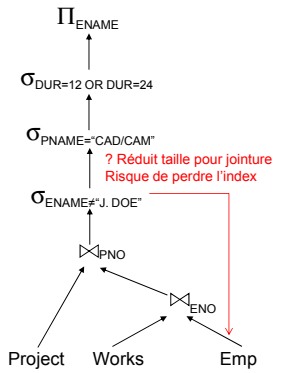
Restructuration

- Objectif : choisir l'ordre d'exécution des opérations algébriques (élaboration du plan logique).
- Conversion en arbre algébrique
- Transformation de l'arbre (optimisation)
  - règles de transformation (équivalence algébriques),
  - estimation du coût des opérations en fonction de la taille
  - Estimation du résultat intermédiaire (taille et ordre?)
  - En déduire l'ordre des jointures

Restructuration

- Conversion en arbre algébrique
- Exemple

```
SELECT  Ename
FROM  Emp, Works, Project
WHERE Emp.Eno=Works.Eno
AND Works.Pno=Project.Pno
AND   Ename NOT='J.Doe'
AND   Pname = 'CAD/CAM'
AND   (Dur=12 OR Dur=24)
```



Calcul du coût d'un plan

- La fonction de coût donne les temps I/O et CPU
  - nombre d'instructions et d'accès disques
- Estimation de la taille du résultat de chaque noeud
  - Permet d'estimer le coût de l'opération suivante
  - sélectivité des opérations – "facteur de réduction"
  - propagation d'erreur possible
- Estimation du coût d'exécution de chaque noeud de l'arbre algébrique
  - utilisation de pipelines ou de relations temporaires importante
    - Pipeline : les tuples sont passés directement à l'opérateur suivant.
      - Pas de relations intermédiaires (petites mémoires, ex. carte à puce).
      - Permet de paralléliser (BD réparties, parallèle)
      - Intéressant même pour cas simples : $\sigma_{F \wedge F'}(R)$, index sur F' $\rightarrow \sigma_{F}(\sigma_{F'}(R))$
    - Relation temporaire : permet de trier mais coût de l'écriture

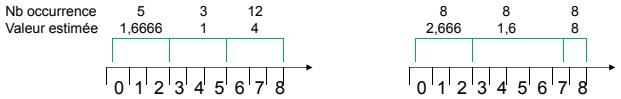
Statistiques

- Relation
  - cardinalité : card(R)
  - taille d'un tuple : largeur de R
  - fraction de tuples participant une jointure / attribut
  - …
- Attribut
  - cardinalité du domaine
  - nombre de valeurs distinctes distinct(A,R) = $\Pi_A(R)$
  - Valeur max, valeur min
- Hypothèses
  - indépendance entre différentes valeurs d'attributs
  - distribution uniforme des valeurs d'attribut dans leur domaine
  - Sinon, il faut maintenir des histogrammes
    - Equilarge : plages de valeurs de même taille
    - Equiprofond: plages de valeurs contenant le même nombre d'occurrence
    - Equiprofond meilleur pour les valeurs fréquentes (plus précis) voir transparent suivant
- Stockage :
  - Les statistiques sont des métadonnées, stockées sous forme relationnelle (cf. TME)
  - Rafraîchies périodiquement, pas à chaque fois.

Le fameux compromis L/E

Histogramme

Select * from R where A = 8



Equilarge
Valeur estimée 4
Valeur réelle dans [0,12]
(faire l'hypothèse d'uniformité dans l'intervalle)

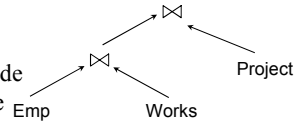
Equiprofond
Valeur estimée 8
Valeur réelle 8

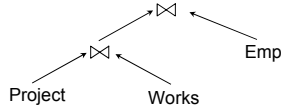
Espace de recherche

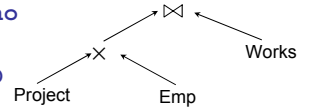
- Caractérisé par les plans "équivalents" pour une même requête
  - ceux qui donnent le même résultat
  - générés en appliquant les règles de transformation vues précédement
- Le coût de chaque plan est en général différent
- L'ordre des jointures est important

Arbres de jointures

- Avec *N* relations, il y a O(*N*!) arbres de jointures équivalents qui peuvent être obtenus en appliquant les règles de *commutativité* et d'*associativité*



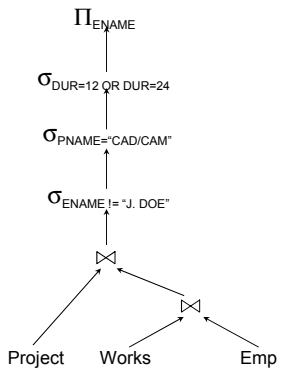




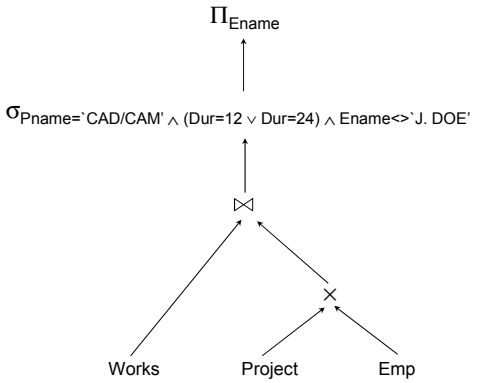

```
SELECT   Ename,Resp
 FROM    Emp, Works,Project
 WHERE   Emp.Eno=Works.Eno
 AND
    Works.PNO=Project.PNO
```


Exemple

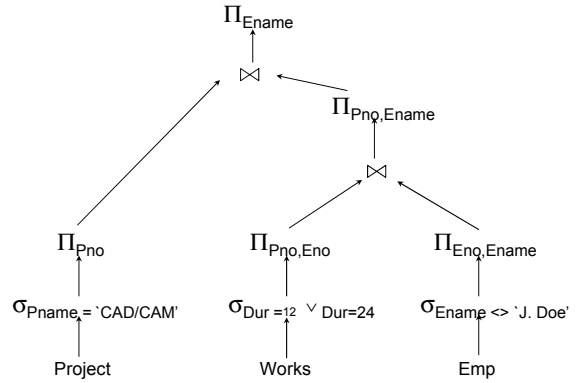
```
SELECT Ename
FROM Project p, Works w,
     Emp e
WHERE w.Eno=e.Eno
AND   w.Pno=p.Pno
AND   Ename<>`J. Doe'
AND   p.Pname=`CAD/CAM'
AND   (Dur=12 OR Dur=24)
```



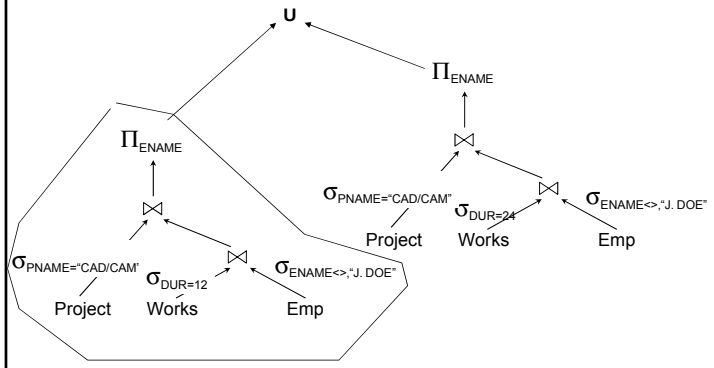
Requête équivalente



Autre requête équivalente



Encore une requête équivalente

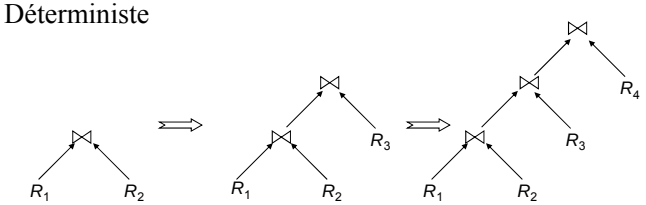


Stratégie de recherche

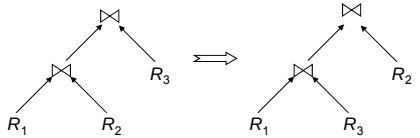
- Il est en général trop coûteux de faire une recherche exhaustive
- Déterministe
  - part des relations de base et construit les plans en ajoutant une relation à chaque étape
  - programmation dynamique: largeur-d'abord
  - excellent jusqu'à 5-6 relations
- Aléatoire
  - recherche l'optimalité autour d'un point de départ particulier
  - réduit le temps d'optimisation (au profit du temps d'exécution)
  - meilleur avec > 5-6 relations
    - **recuit simulé (simulated annealing)**
    - **amélioration itérative (iterative improvement)**

Stratégies de recherche

- Déterministe



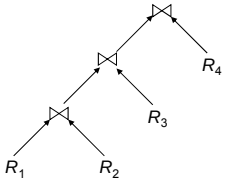
- Aléatoire



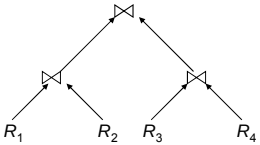
Algorithmes de recherche

- Limiter l'espace de recherche
  - heuristiques
    - par ex. appliquer les opérations unaires avant les autres
    - Ne marche pas toujours (perte d'index, d'ordre)
  - limiter la forme des arbres

Arbre linéaire



Arbre touffu



Génération de plan physique

- Sélection :
  - Commencer par les conditions d'égalité avec un index sur l'attribut
  - Filtrer sur cet ensemble de n-uplets ceux qui correspondent aux autres conditions
- Jointure
  - Utilisation des index, des relations déjà triées sur l'attribut de jointure, présence de plusieurs jointures sur le même attribut
- Pipelines ou matérialisation

Conclusion

- Point fondamental dans les SGBD
- Importance des métadonnées, des statistiques sur les relations et les index, du choix des structures d'accès.
- L'administrateur de bases de données peut améliorer les performances en créant de nouveaux index, en réglant certains paramètres de l'optimiseur de requêtes (voir TME et cours M1)

Cours 5 : méthodes d'accès

- Fonctions et structure des SGBD
- Structures physiques
  - Stockage des données
  - Organisation de fichiers et indexation
    - index
    - arbres B+
    - hachage

Objectifs des SGBD (rappel)

- Contrôle intégré des données
  - Cohérence (transaction) et intégrité (CI)
  - partage
  - performances d'accès
  - sécurité
- Indépendance des données
  - logique : cache les détails de l'organisation conceptuelle des données (définir des vues)
  - physique : cache les détails du stockage physique des données (accès relationnel vs chemins d'accès physiques)

Arbre B+

- Les arbres B+ sont des index hiérarchiques
- Ils améliorent l'efficacité des recherches
  - L'arbre est peu profond.
    - Accès rapide à un enregistrement : chemin court de la racine vers une feuille
    - Rmq: l'arbre peut être très large, sans inconvénient
  - L'arbre est toujours équilibré
    - *Balanced* tree en anglais
    - Tous les chemins de la racine aux feuilles ont la même longueur
  - L'arbre est suffisamment compact
    - Peut souvent tenir en mémoire
    - Un noeud est au moins à moitié rempli

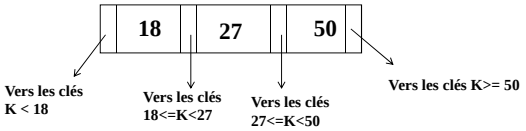
Arbre B+ : coût d'accès

- Le coût d'accès est
  - proportionnel à la longueur d'un chemin
  - Identique quelle que soit la feuille atteinte

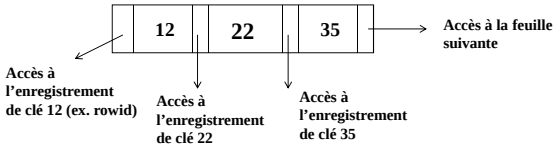
  → **coût d'accès prévisible**
- Avantage :
  - permet d'estimer le coût d'accès, a priori, pour décider d'utiliser ou non un index
- Mesure du coût:
  - Nombre de nœud lus / écrits
  - Nombre de pages de données lues / écrites

Arbre B+

- Les nœuds internes servent à atteindre une feuille



- Les feuilles donnent accès aux enregistrements



5

Arbre B+: 3 types de nœuds (cas non plaçant)

- Racine
  - point d'entrée pour une recherche
- Nœud intermédiaire
  - Peut contenir une valeur pour laquelle il n'existe aucun enregistrement
- Feuille
  - Les feuilles contiennent toutes les clés pour lesquelles il existe un enregistrement
  - Les feuilles contiennent uniquement des clés de la BD

6

Ordre d'un arbre, degré d'un nœud

- Soit n le nombre de clés contenues dans un nœud
- Un arbre-B+ est d'**ordre d** si
  - Pour un nœud intermédiaire et une feuille : $d \leq n \leq 2d$
  - Pour la racine: $1 \leq n \leq 2d$
  - Tout nœud doit tenir dans une page :
    - 2d correspond à une page complètement remplie
- Degré sortant d'un nœud
  - Un nœud intermédiaire (et la racine) ayant n valeurs de clés a **n+1** pointeurs vers ses fils
  - Une feuille n'a pas de fils
- À part la racine, toute page est entre moitié pleine et complètement pleine (compacité, limite la hauteur de l'arbre)

7

Hauteur min et max d'un arbre B+

- Hauteur minimum quand l'arbre est plein : $h = \log_{2d+1}(N) + 1$
  - 2d clés par nœud
  - N feuilles (il y a donc N*2d clés dans la table correspondante)
- Hauteur maximum quand les nœuds sont le plus vides possible : $h = \log_{d+1}(N) + 2$
  - d clés par nœud, 1 pour la racine
  - 2N feuilles (2N*d clés dans la table correspondante)
- En pratique d est assez grand, donc h reste raisonnable (4 pour une relation « normale »)

8

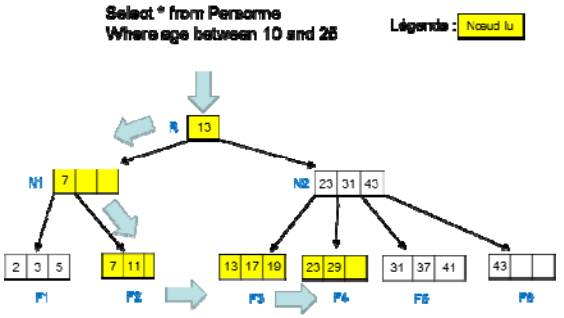
Arbre-B+ : chainage des feuilles

- But: supporter les requêtes d'intervalle
  - Exple de requête: ... where age between 18 and 25
  - Traverser l'index pour atteindre une borne de l'intervalle, puis parcours séquentiel des feuilles
- Chainage double pour supporter les requêtes avec une inégalité « < »
  - Ex: ... where age < 6

9

Parcours du chainage

- Avantage :
  - lire un seul chemin (moins de lectures)



10

Insertion

- Rechercher la feuille où insérer la nouvelle valeur.
- Insérer la valeur dans la feuille s'il y a de la place.
  - Maintenir les valeurs triées dans la feuille
- Si la feuille est pleine (2d valeurs avant insertion), il y a éclatement. Il faut créer un nouveau nœud :
  - Insérer les d+1 premières valeurs dans le nœud original, et les d autres dans le nouveau nœud (à droite du premier).
  - La plus petite valeur du nouveau nœud est insérée dans le nœud parent, ainsi qu'un pointeur vers ce nouveau nœud.
  - Remarque : les deux feuilles ont bien un nombre correct de valeurs (elles sont au moins à moitié pleines)

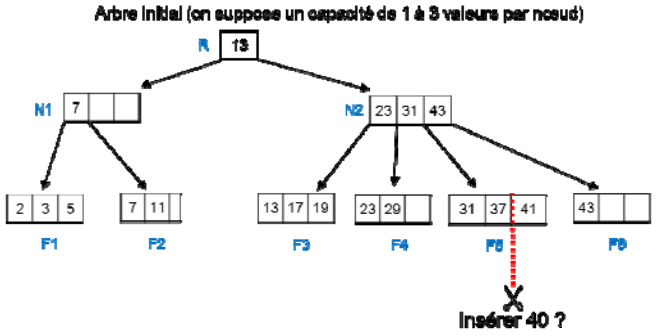
11

Insertion (cont.)

- S'il y a éclatement dans le parent, il faut créer un nouveau nœud frère M, à droite du premier
  - Les d premières valeurs restent dans le nœud N, les d dernières vont dans le nouveau nœud M.
  - La valeur restante est insérée dans parent de N et M pour atteindre M.
  - Rmq: M et N ont bien chacun d+1 fils
- Les éclatements peuvent se propager jusqu'à la racine et créer un nouveau niveau pour l'arbre.

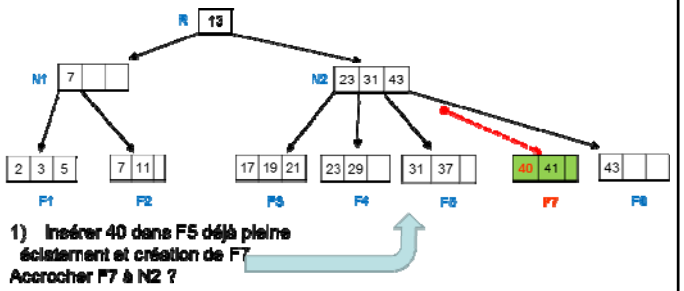
12

Insertion
Exemple 1 : état initial



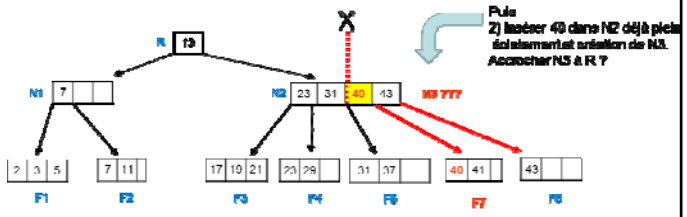
13

Exemple 1 (suite)



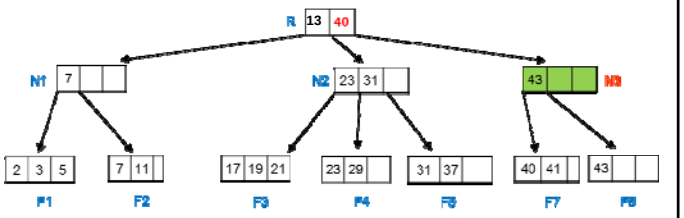
14

Exemple 1 (suite)



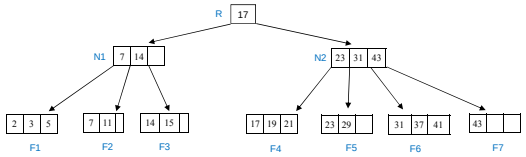
15

Exemple 1 (fin)



16

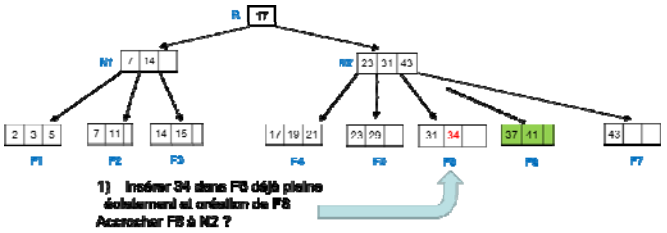
Insertion
Exemple 2 : état initial



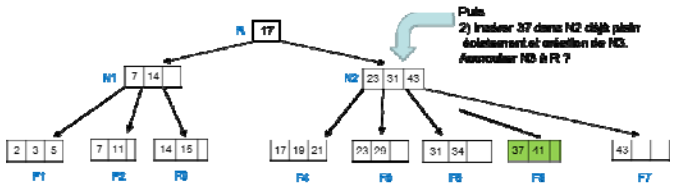
Insérer 34 ?

17

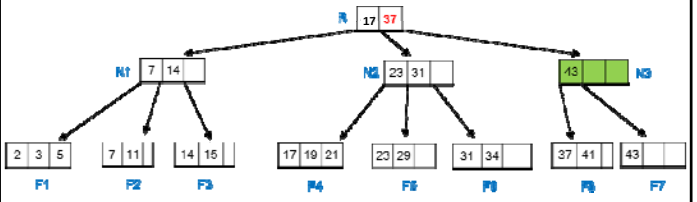
Exemple 2 (suite)



Exemple 2 (suite)



Exemple 2 (fin)



Suppression

- Supprimer la valeur (et le pointeur vers l'enregistrement) de la feuille où elle se trouve.
- Si la feuille est encore suffisamment pleine, il n'y a rien d'autre à faire.
- Sinon, redistribuer les valeurs avec une feuille **ayant le même parent**, afin que toutes les feuilles aient le nombre minimum de valeurs requis.
  - conséquence : ajuster le contenu du nœud père.
- Si la redistribution est **impossible**, il faut fusionner 2 feuilles
  - conséquence: supprimer une valeur dans le nœud père.
- Si le parent n'est pas suffisamment plein, appliquer récursivement l'algorithme de suppression.
  - Remarque1 : la propagation récursive peut entraîner la perte d'un niveau.

21

Résumé des opérations

- Insertion
  - simple
  - éclatement
    - d'une feuille
    - d'une feuille puis éclatement d'ancêtres
- Suppression
  - simple
  - redistribution
    - entre 2 feuilles
  - fusion
    - entre 2 feuilles
    - entre 2 feuilles puis redistribution ou fusion d'ancêtres
- Rmq
  - Toujours insérer/supprimer une clé au niveau des feuilles
  - Jamais de redistribution lors d'une insertion. L'éclatement est préférable pour faciliter les prochaines insertions.

22

Avantages et Inconvénients

- Avantages des organisations indexées par arbre b (b+) :
  - Régularité = pas de réorganisation du fichier nécessaires après de multiples mises à jour.
  - Lecture séquentielle rapide: possibilité de séquentiel physique et logique (trié)
  - Accès rapide en 3 E/S pour des fichiers de 1 M d'articles
- Inconvénients :
  - Les suppressions génèrent des trous difficiles à récupérer
  - Avec un index non plaçant, l'accès à plusieurs enregistrements (intervalle ou valeur non unique) aboutit à lire plusieurs enregistrements non contigus. Lire de nombreuses pages non contiguës dure longtemps
  - Taille de l'index pouvant être importante.

23

Exercice Arbre B+

- Un arbre B+ a 3 niveaux. Chaque nœud contient 1 ou 2 clés.
- Les feuilles ont les clés 1,4, 9,16, 25, 36, ~~49~~, 54, 61, 70, 81, 84, 87, 88, 95, ~~99~~
- Les nœuds intermédiaires ont les clés 9, 54, 70, 88
- La racine contient 2 clés, les plus petites possibles parmi celles des feuilles
- Représenter l'arbre, puis insérer la clé 32

24

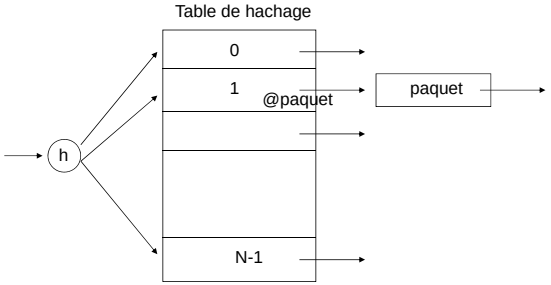
Organisations par Hachage

- Fichier haché statique (Static hashed file)
  - Fichier de taille fixe dans lequel les articles sont placés dans des paquets dont l'adresse est calculée à l'aide d'une fonction de hachage fixe appliquée à la clé.
  - On peut rajouter une indirection : table de hachage.
    - H(k) donne la position d'une cellule dans la table.
    - Cellule contient adresse paquet
    - Souplesse (ex. suppression d'un paquet)
- Différents types de fonctions :
  - Conversion en nb entier
  - Modulo P
  - Pliage de la clé (combinaison de bits de la clé)
  - Peuvent être composées

Défi : Obtenir une distribution uniforme pour éviter les collisions (saturation)

25

Hachage statique



26

Hachage statique

- Très efficace pour la recherche (condition d'égalité) : on retrouve le bon paquet en une lecture de bloc.
- Bonne méthode quand il y a peu d'évolution
- Choix de la fonction de hachage :
  - Mauvaise fonction de hachage ==> Saturation locale et perte de place
  - Solution : autoriser les débordements

27

Techniques de débordement

- l'adressage ouvert
  - place l'article qui devrait aller dans un paquet plein dans le premier paquet suivant ayant de la place libre; il faut alors mémoriser tous les paquets dans lequel un paquet plein a débordé.
- le chaînage
  - constitue un paquet logique par chaînage d'un paquet de débordement à un paquet plein.
- le rehachage
  - applique une deuxième fonction de hachage lorsqu'un paquet est plein, puis une troisième, etc..., toujours dans le même ordre.

Le chaînage est la solution la plus souvent utilisée. Mais si trop de débordement, on perd tout l'intérêt du hachage (séquentiel)

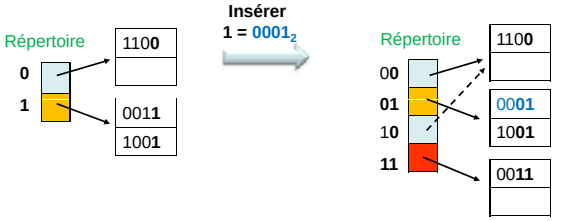
28

Hachage dynamique

- Hachage dynamique :
  - techniques permettant de faire grandir progressivement un fichier haché saturé en distribuant les enregistrements dans de nouvelles régions allouées au fichier.
- Deux techniques principales
  - Hachage extensible
  - Hachage linéaire

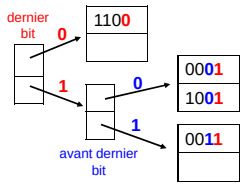
Hachage extensible

- Ajout d'un niveau d'indirection vers les paquets (tableau de pointeurs), qui peut grandir (considérer + de bits) : répertoire
- Jamais de débordement
  - Accès direct à tout paquet via le répertoire (i.e, une seule indirection)



Hachage extensible

- Répertoire similaire à un arbre préfixe (*trie*)
  - Si on considère les bits en commençant par le dernier (i.e, celui de poids faible)



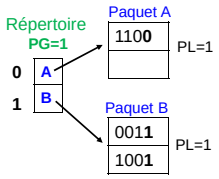
➲ Suffixe utilisé pour l'indexation = profondeur

Hachage extensible : notations

- Le répertoire est noté **R[**P_0, P_1, P_2,..., P_k **] PG=***pg* avec
  - P_i ... les noms d'un paquet,
  - *pg* la profondeur globale.
- Rmq : le répertoire contient k cases avec k = 2^{pg}
- Un paquet est noté $P_i(v_j, ..., ...)$ **PL=***pl* avec
  - P_i le nom du paquet, par exemple A,B, ... ,
  - V_j les valeurs que contient le paquet,
  - *pl* la profondeur locale.
- On peut aussi préciser le contenu d'une case particulière du répertoire avec
  - R[i]= L (avec R[0] étant la 1ère case)
- La valeur v est dans le paquet référencé dans la case R[v modulo 2^{pg}]

Hachage extensible
Création du répertoire

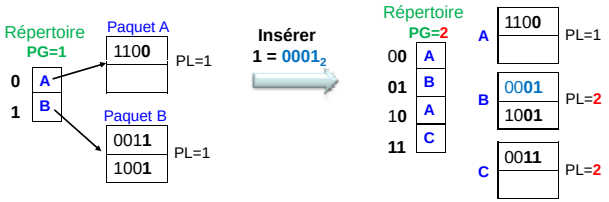
- Etat initial : N valeurs à indexer dans des paquets pouvant contenir p valeurs.
  - Il faut au moins N/p paquets
  - La taille initiale du répertoire est $k=2^{PG}$ tq $2^{PG} \geq N/p$
  - On a k paquets donc PL =PG pour tous les paquets initiaux



33

Hachage extensible: Insertion

- Insertion de v dans le paquet Pi
- Cas 1) Pi est n'est pas plein, insertion immédiate dans Pi
- Cas 2) Pi est plein et PLi < PG alors éclater Pi
  - Créer un nouveau paquet Pj
  - Incrémenter les profondeurs locales de Pi et Pj (PL = PL+1)
  - Répartir les valeurs de Pi et v entre Pi et Pj
    - Si Pi est encore plein, réappliquer l'algo d'insertion : cas 2) ou 3)
- Cas 3) Pi est plein et PLi = PG alors doubler le répertoire
  - Recopier le contenu des k premières cases dans les k nouvelles cases suivantes
  - PG = PG+1 puis on retombe sur le cas 2)



34

Hachage extensible
Suppression et fusion

- Lors d'une suppression, si un paquet Pi devient vide et si PLi=PG alors
- on tente de fusionner Pi avec le paquet Pj référencé dans la case ayant le même suffixe que celle qui référence Pi
  - Suffixe commun (en base 2) de longueur PG -1
  - Exple si PG=3 et PLi=3, les cases ayant le même suffixe (de longueur 2) sont :
    - R[0] et R[4]
    - R[1] et R[5]
    - ...
    - R[3] et R[7]
- ~~Si Pi = Pj alors pas de fusion et Pi reste vide~~
- ~~Sinon~~ supprimer Pi et décrémenter la profondeur locale de Pj et mettre le répertoire à jour (le pointeur de Pi doit maintenant pointer Pj)

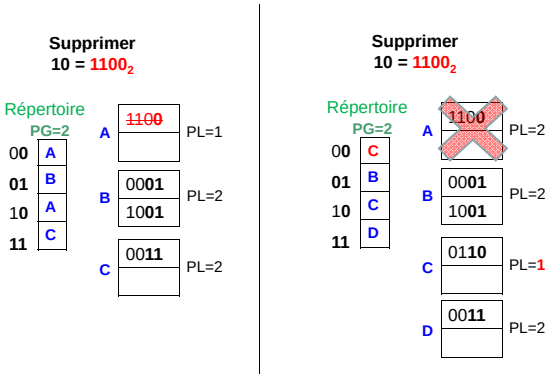
- Si pour tous les paquets restants on a PL < PG alors on peut diviser le répertoire par 2 et décrémenter PG (les deux moitiés du répertoire sont identiques). En pratique, pas toujours fait

- Rmq: aucune fusion si PLi <= PG – 1 (il n'y a pas de paquet ayant un suffixe commun de longueur PG-1). Le paquet Pi reste vide

35

Hachage extensible
Exemples de suppression

-



36

Hachage extensible (suite)

- Avantage : accès à un seul bloc (si le répertoire tient en mémoire)
- Profondeur locale/globale
- Inconvénient :
  - interruption de service lors du doublement du répertoire.
  - Peut ne plus tenir en mémoire.
  - Si peu d'enregistrement par page, le répertoire peut être inutilement gros

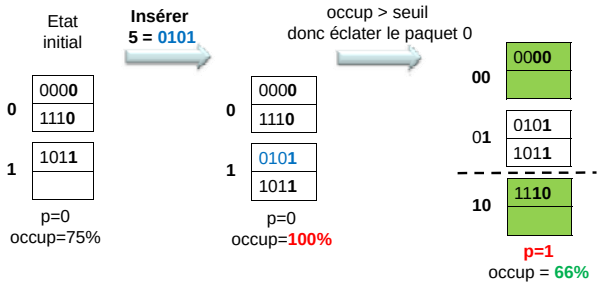
37

Hachage linéaire (1)

- Garantit que le nombre moyen d'enregistrements par paquet ne dépasse pas un certain seuil (ex. taux d'occupation moyen d'un paquet < 80%)
- Ajouter les nouveaux paquets au fur et à mesure, en éclatant chaque paquet dans l'ordre, **un par un** du premier au Nième paquet.
- Avantage par rapport au hachage extensible : pas besoin de répertoire
  - Plus rapide si les données sont uniformément réparties dans les paquets
- Inconvénient: débordement quand le seuil n'est pas atteint et le paquet est plein.
- Il faut une suite de fonction de hachage qui double le nombre de paquets à chaque fois :
  - Ex : N paquets initialement, h(x) fonction de hachage initiale
  - $h_i(x) = h(x) \bmod (2^i N)$
- Il faut marquer quel est le prochain paquet à éclater (noté p)
- Quand les N paquets ont éclaté, on recommence avec N' = 2N

Hachage linéaire (2)

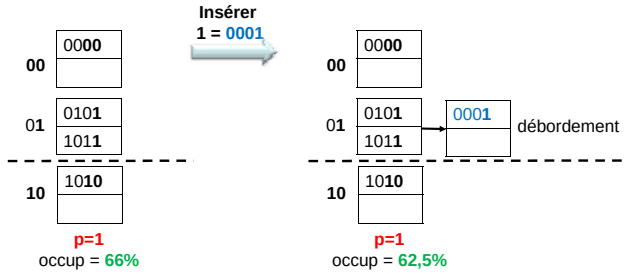
- Exemple avec 2 paquets initiaux. N=2, seuil = 80%
- $h_0(x) = h(x) \bmod 2$, $h_1(x) = h(x) \bmod 4$
- p=0 : le prochain paquet à éclater est le paquet 0



39

Hachage linéaire (3)

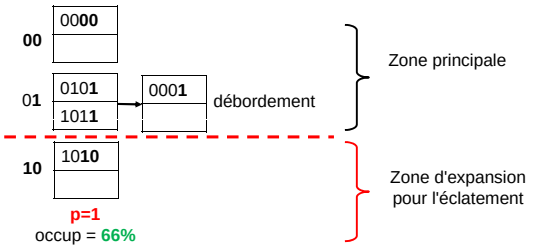
- Insérer 1 dans le paquet 1
- Débordement du paquet (pas d'éclatement car le taux d'occupation reste inférieur au seuil)



40

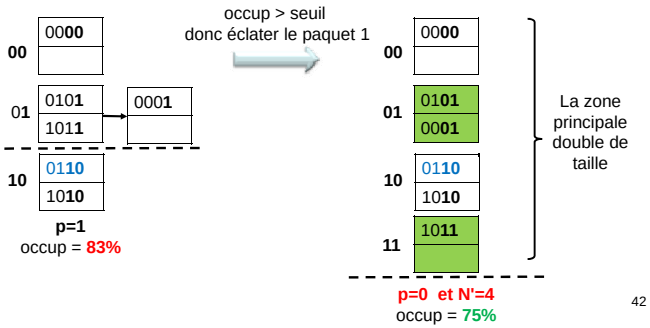
Hachage linéaire (4)

- Accès à l'enregistrement de clé c ?
- Soit i = h_0(c) et i'= h_1(c)
- Si i >= p alors lire le paquet i sinon lire le paquet i'



Hachage linéaire (5)

- Insérer 6 = 0110 dans le paquet 10_2
- Le dernier paquet de la zone principale éclate
- donc on repart à 0 après avoir agrandi la zone principale



Exercice Hachage extensible

- Chaque paquet **contient au plus 2 valeurs.**
- **Question 1.** On considère un répertoire R de profondeur globale PG=1. Avec 2 paquets P0 et P1 R={PO, P1}. Initialement les deux paquets contiennent:
- **P0**(4,8) **P1**(1,3)
- Insérer la valeur 12.
- Quelle est la profondeur globale après insertion ?
- Détailler le contenu du répertoire et des paquets modifiés ou créés, et leur profondeur locale (PL).

Conclusion

- Les fichiers séquentiels sont efficaces pour le parcours rapide, l'insertion et la suppression, mais lents pour la recherche.
- Les fichiers triés sont assez rapides pour les recherches (très bons pour certaines sélections), mais lents pour l'insertion et la suppression.
- Les fichiers hachés sont efficaces pour les insertions et les suppressions, très rapides pour les sélections avec égalité, peu efficaces pour les sélections ordonnées.
- Les index permettent d'améliorer certaines opérations sur un fichier. Les Arbres-B+ sont les plus efficaces.

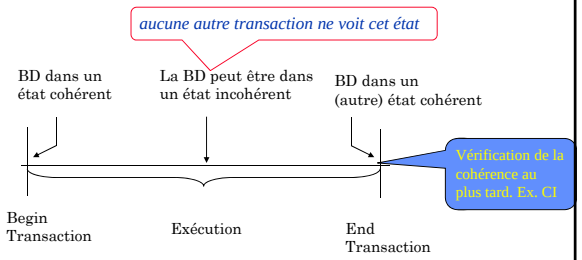
UFR 919 Ingénierie – module LU3IN009
cours 6 : Gestion de transactions

Définition
Exemples
Propriétés des transactions
Fiabilité et tolérance aux pannes
Journaux
Protocoles de journalisation
Points de reprise
Transactions avancées

Transaction

Ensemble d'actions qui réalisent des transformations cohérentes de la BD

- opérations de lecture ou d'écriture de données, appelées *granules* (tuples, pages, etc.)



Syntaxe

Une transaction est délimitée par Begin_transaction et End_transaction et comporte :

- des opérations de lecture ou d'écriture de la BD (select, insert, delete, update)
- des opérations de manipulation (calculs, tests, etc.)
- des opérations transactionnelles: commit, abort, etc.

On ne s'intéresse pas aux opérations de manipulation (logique interne de la transaction) car leur analyse serait trop coûteuse et pas toujours possible (ex. client jdbc, communique uniquement les opérations de lecture/écriture et transactionnelles, les opérations de manipulations ne sont pas vues par le système transactionnel)

Exemple de transaction simple

```
Begin_transaction Budget-update
begin
  EXEC SQL UPDATE Project
      SET Budget = Budget * 1.1
      WHERE Pname = `CAD/CAM';
end
```

Granules lus :
- Pname de tous les n-uplets de Project (sauf si index)
- Budget du n-uplet de Pname 'CAD/CAM'

Granules écrits :
- Budget du n-uplet de Pname 'CAD/CAM'

BD exemple

Considérons un système de réservation d'une compagnie aérienne avec les relations:

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL)
FC(FNO, DATE, CNAME)

Exemple de transaction de réservation

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL) FC(FNO, DATE, CNAME)

```
Begin_transaction Reservation
begin
   input(flight_no, date, customer_name);
   EXEC SQL  UPDATE     FLIGHT
             SET        STSOLD = STSOLD + 1
             WHERE      FNO = flight_no AND DATE = date; /*
        1 place vendue
   EXEC SQL  INSERT
             INTO       FC(FNO, DATE, CNAME);
             VALUES     (flight_no, date, customer_name,);
        /* 1 résa en plus
   output("reservation completed")
end . {Reservation}
```

Problème : s'il n'y a plus de place dans l'avion ?
- **Surbooking ?**
- **Contrainte d'intégrité (STSOLD <= CAP) ? Message d'erreur…**

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL) FC(FNO, DATE, CNAME)

Terminaison de transaction

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL    SELECT      STSOLD,CAP
                INTO        temp1,temp2
                FROM        FLIGHT
                WHERE       FNO = flight_no AND DATE =  date;
    if temp1 = temp2 then
        output("no free seats");
        Abort
    else
        EXEC SQL  UPDATE  FLIGHT
                  SET     STSOLD = STSOLD + 1
                  WHERE   FNO = flight_no AND DATE = date;
        EXEC SQL  INSERT
                  INTO    FC(FNO, DATE, CNAME);
                  VALUES  (flight_no, date, customer_name);
      Commit
      output("reservation completed")
  endif
end . {Reservation}
```


Propriétés des transactions

ATOMICITE : Les opérations entre le début et la fin d'une transaction forment une *unité d'exécution. Tout (commit) ou rien (abort)*

DURABILITE: Les mises-à-jour des transactions **validées** *persistent.*

Gestion des **pannes**
- Cache
- Journalisation

COHERENCE : Chaque transaction accède et retourne une base de données dans un état cohérent (pas de violation de contrainte d'intégrité).

ISOLATION : Le résultat d'un ensemble de *transactions concurrentes* et validées correspond au résultat d'une exécution *successive* des mêmes transactions.

Gestion de la **cohérence**
- Sérialisibilité
- Algorithmes de contrôle de concurrence

Les transactions délimitent

Les responsabilités respectives du programmeur et du système.

Programmeur :

- Écrire des transactions implantant la logique de l'appli

Système doit garantir :

- l'exécution *atomique* et *fiable* en présence de pannes
- l'exécution *correcte* en présence d'utilisateurs concurrents

Fiabilité

Problème:

Comment maintenir

atomicité

durabilité

des transactions

Types de pannes

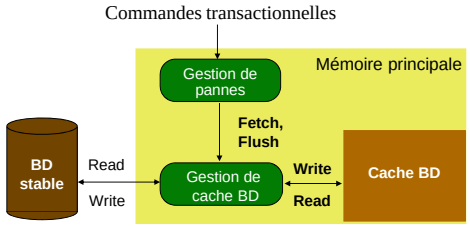
Panne de transaction
- abandon (normal –if- ou dû à un interblocage)
- en moyenne 3% des transactions abandonnent anormalement

Panne système
- panne de processeur, mémoire, alimentation, …
- le contenu de la mémoire principale est perdu mais disk ok

Panne disque
- panne de tête de lecture ou du contrôleur disque
- les données de la BD sur disque sont perdues

gravité

Architecture pour la gestion de pannes



Stratégies de mise-à-jour

Mise-à-jour en place

- chaque mise-à-jour cause la modification de données dans des pages dans le cache BD
- l'ancienne valeur est écrasée par la nouvelle

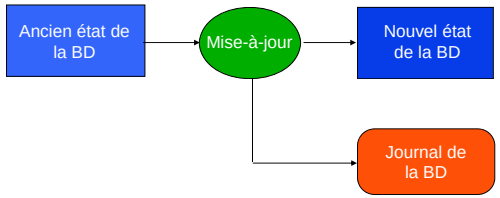
Mise-à-jour hors-place

- les nouvelles valeurs de données sont écrites séparément des anciennes dans des pages ombres
- mises-à-jour des index compliquée
- peu utilisé en pratique car très cher (sauf pour transactions avancées)

Pb : si on fit la mise-à-jour en place, comment réaliser l'abandon d'une transaction ? Journal….

Journal de la BD

Chaque action d'une transaction doit réaliser l'action, ainsi qu'écrire un enregistrement dans le journal (fichier en ajout seulement avec purge de temps à autre)



Journalisation

Le journal contient les informations nécessaires à la restauration d'un état cohérent de la BD

- identifiant de transaction
- type d'opération (action)
- granules accédés par la transaction pour réaliser l'action
- ancienne valeur de granule (image avant)
- nouvelle valeur de granule (image après)
- …

Structure du journal

Structure d'un enregistrement :

- N° transaction (Trid)
- Type enregistrement {début, update, insert, commit, abort}
- TupleId (rowid sous Oracle)
- [Attribut modifié, Ancienne valeur, Nouvelle valeur] ...

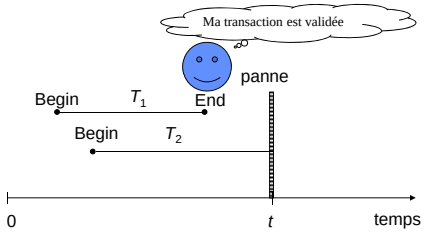
Problème de taille

- on tourne sur N fichiers de taille fixe
- possibilité d'utiliser un fichier haché sur Trid/Tid

Exemple de journal

Début du journal → T_1, begin
T_1, x, 99, 100
T_2, begin
T_2, y, 199, 200
T_3, begin
T_3, z, 51, 50
T_2, w, 1000, 10
T_2, commit
T_4, begin
T_3, abort
T_4, y, 200, 50
T_5, begin
T_5, w, 10, 100
Fin du journal → T_4, commit

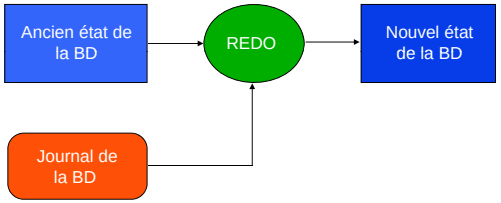
Pourquoi journaliser?



Lors de la reprise

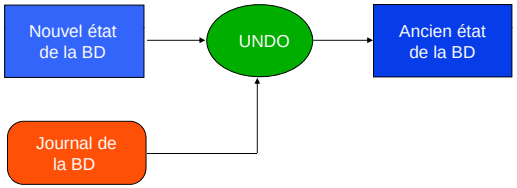
- toutes les mises-à-jour de T_1 doivent être faites dans la BD (REDO)
- aucune mise-à-jour de T_2 ne doit être faite dans la BD (UNDO)

Protocole REDO



L'opération REDO utilise l'information du journal pour refaire les actions qui ont été exécutées ou interrompues
Elle génère la nouvelle image

Protocole UNDO



L'opération UNDO utilise l'information du journal pour restaurer l'image avant du granule. Faite en principe avant le REDO

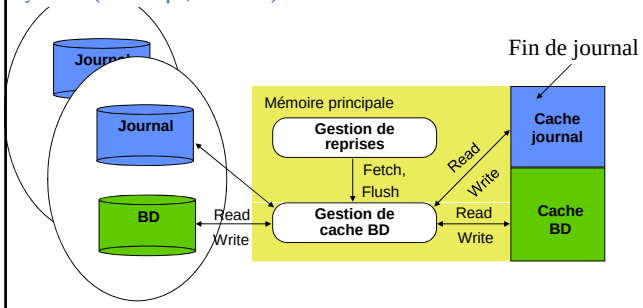
UNDO: parcours vers l'arrière, REDO : parcours vers l'avant

Début du journal → T_1, begin
T_1, *x*, 99, 100
T_2, begin
T_2, *y*, 199, 200
T_3, begin
T_3, *z*, 51, 50
T_2, *w*, 1000, 10
Fin du journal → T_2, commit

UNDO : T_2 rien (marquée pour Redo), z:=51, x:=99
REDO : y:=200, w:=10

Interface du journal

Réplication
asynchrone (crash disque, incendie…)



Gestion du cache BD

Le cache améliore les performances du système, mais a des répercussions sur la reprise (dépend de la politique de migration sur le disque).

Pour simplifier le travail de reconstruction, on peut

- empêcher des migrations cache->disque
  - Fix : ne peut migrer *pendant* la transaction
- forcer la migration en fin de transaction
  - Flush : doit migrer à chaque commit

Fix et flush facilite le recouvrement mais contraignent la gestion du cache

Gestion du cache BD

Impact sur la reprise :

- No-fix/no-flush : UNDO/REDO

Undo nécessaire car les écritures de transactions non validées ont peut être été écrites sur disque et donc rechargées à la reprise.

Redo nécessaire car les écritures de transactions validées n'ont peut être pas été écrites sur disque

- Fix/no-flush : REDO
- No-fix/flush : UNDO

Abandons en cascade, recouvrabilité (1/2)

Soient deux transactions, T0 et T1, exécutant l'une après l'autre les instructions suivantes :

1. *variable1 := Lire (A);*
2. *variable1 := variable1 – 2 ;*
3. *Ecrire (A, variable1);*
4. *variable2 := Lire (B);*
5. *variable2 := variable2 / variable1 ;*
6. *Ecrire (B, variable2);*

Le système, sur lequel elles s'exécutent, tient à jour un journal susceptible de contenir les enregistrements suivants:

<*No de Transaction*, **start** | **commit** | **abort** >
<*No de Transaction, identification de granule, ancienne valeur, nouvelle valeur*>

Les valeurs initiales de A et B étant respectivement 4 et 14, quel est le contenu du journal lorsque la seconde transaction (T1) se termine ?

Comment restaurer la base en mode nofix ? En mode fix ?

Abandons en cascade, recouvrabilité (2/2)

Soient deux transactions, T0 et T1, exécutant l'une après l'autre les instructions suivantes :

1. *variable1 := Lire (A);*
2. *variable1 := variable1 – 2 ;*
3. *Ecrire (A, variable1);*
4. *variable2 := Lire (B);*
5. *variable2 := variable2 / variable1 ;*
6. *Ecrire (B, variable2);*

On suppose maintenant qu'une transaction T2 effectue le morceau de code suivant :

variable := Lire(A);
Ecrire(A, variable + 2);

entre l'exécution des instructions (3) et (4) de T1, sur un système qui fait les *écritures en mode immédiat (noFix)*.
Comment pourra-t-on restaurer une base cohérente à la terminaison de T1 sur erreur dans chacun des cas suivant : (a) T2 a encore d'autres instructions à exécuter, et (b) T2 ayant terminé son code avec l'exécution de ses 2 instructions, l'enregistrement < *T2*, ***commit*** > figure dans le journal ?

Ecriture du journal sur disque

Synchrone (forcée): à chaque ajout d'un enregistrement

- ralentit la transaction
- facilite le recouvrement

Asynchrone : périodique ou quand le buffer est plein ou…

- Au plus tard quand la transaction valide

Quand écrire le journal sur disque?

Supposons une transaction *T* qui modifie la page *P*

Cas chanceux

- le système écrit *P* dans la BD sur disque
- le système écrit le journal sur disque pour cette opération
- PANNE!... (avant la validation de *T*)

Nous pouvons reprendre (undo) en restaurant *P* à son ancien état grâce au journal

Cas malchanceux

- le système écrit *P* dans la BD sur disque
- PANNE!... (avant l'écriture du journal)

Nous ne pouvons pas récupérer car il n'y a pas d'enregistrement avec l'ancienne valeur dans le journal

Solution: le protocole Write-Ahead Log (WAL)

Protocole WAL

Observation:

- si la panne précède la validation de transaction, alors toutes ses opérations doivent être défaites, en restaurant les images avant (*partie undo* du journal)
- dès qu'une transaction a été validée, certaines de ses actions doivent pouvoir être refaites, en utilisant les images après (*partie redo* du journal)

Protocole WAL:

- avant d'écrire dans la BD sur disque, la partie *undo* du journal doit être écrite sur disque
- lors de la validation de transaction, la partie *redo* du journal doit être écrite sur disque avant la mise-à-jour de la BD sur disque

Points de reprise

Réduit la quantité de travail à refaire ou défaire lors d'une panne

Un point de reprise enregistre une liste de transactions actives

Pose d'un point de reprise:

- écrire un enregistrement begin_checkpoint dans le journal
- écrire les buffers du journal et de la BD sur disque
- écrire un enregistrement end_checkpoint dans le journal

Remarque :

- Procédure similaire pour rafraichissement des sauvegardes

Procédures de reprise

Reprise à chaud

- perte de données en mémoire, mais pas sur disque
- à partir du dernier point de reprise, déterminer les transactions
  - validées : REDO
  - non validées : UNDO
- Variante *ARIES (IBM DB2, MS SQL Server)* : refaire *toutes* les transactions et défaire les transactions non terminées au moment du crash

Reprise à froid

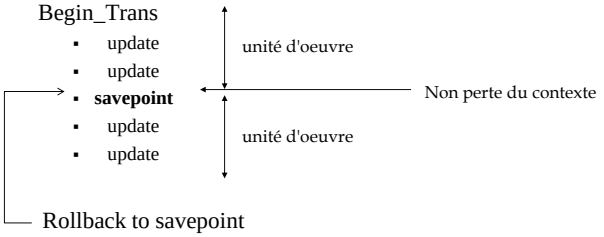
- perte de données sur disque
- à partir de la dernière sauvegarde (ne contient que des opérations validées) et du dernier point de reprise, faire REDO des transactions validées
- UNDO inutile

Il peut y avoir des pannes pendant la procédure de reprise….

Points de Sauvegardes

Introduction de points de sauvegarde intermédiaires

- (savepoint, commitpoint)



Conclusion

Assurer l’atomicité et la durabilité n’est pas simple

Journalisation

Interdire les exécutions non recouvrables

Eviter les abandons en cascade

UFR 919 Ingénierie – module LU3IN009
cours 7 : Transactions et concurrence d'accès

- Définition, exemples et propriétés des transactions
- Fiabilité et tolérance aux pannes
- Contrôle de concurrence

Contrôle de concurrence

Les problèmes de concurrence
Degrés d'isolation dans SQL
Exécutions et sérialisabilité
Contrôle de concurrence
Améliorations

Contrôle de concurrence

Objectif : *synchroniser* les *transactions concurrentes* afin de maintenir la cohérence de la BD, tout en maximisant le degré de concurrence

Principes:

- Exécution simultanée des transactions pour des raisons de performance
  par ex., exécuter les opérations d'une autre transaction quand la première commence à faire des accès disques
- Les résultats doivent être équivalents à des exécutions non simultanées (isolation)
  besoin de *raisonner sur l'ordre d'exécution* des transactions

Problèmes de concurrence

Perte d'écritures : on perd *une partie* des écritures de T1 et *une partie* des écritures de T2 sur des données partagées
[T1: Write a1 → A; T2 : Write b2 →B; T1: Write b1 → B; T2 : Write a2 →A;]
A=a2, B=b1 : on perd une écriture de T1 et une écriture de T2

Non reproductibilité des lectures : une transaction écrit une donnée entre deux lectures d'une autre transaction
[T1 : Read A; T2 : Write b2→A; T1: Read A;]
Si b2 est différente de la valeur initiale de A, alors T1 lit deux valeurs différentes.

Conséquence : *introduction d'incohérence*
Exemple avec une contrainte (A = B) et deux transactions T1 et T2
Avant : A=B
[T1 : A*2→A; T2 : A+1→A; T2 : B+1 → B; T1 : B*2 → B;]
Après : A = 2*A+1, B=2*B+2 — A <> B

Degrés d'isolation SQL-92

Lecture sale (lecture d'une maj. non validées):

T1: Write(A); T2: Read(A); T1: abort (abandon en cascade)

- T_1 modifie A qui est lu ensuite par T_2 avant la fin (validation, annulation) de T_1
- Si T_2 annule, T_1 a lu des données qui n'existent pas dans la base de données

Lecture non-répétable (maj. intercalée) :

T1: Read(A); T2: Write(A); T2: commit; T1: Read(A);

- T_1 lit A; T_2 modifie ou détruit A et valide
- Si T_1 lit A à nouveau et obtient *résultat différent*

Fantômes (requête + insertion) :

T1: Select where R.A=…; T2: Insert Into R(A) Values (…);

- T_1 exécute une requête Q avec un prédicat tandis que T_2 insère de nouveaux n-uplets (fantômes) qui satisfont le prédicat.
- Les *insertions* ne sont pas détectées comme concurrentes pendant l'évaluation de la requête (résultat incohérent possible).

Degrés d'isolation SQL-92

haut ↑ degré de concurrence ↓ bas

bas ↓ degré d'isolation ↑ haut

Degré	Lecture sale	Lectures non répétable	Fantômes
READ_UNCOMMITTED	possible	possible	possible
READ_COMMITTED	impossible	possible	possible
REPEATABLE_READ	impossible	impossible	possible
SERIALIZABLE	impossible	impossible	impossible

Ne devrait pas s'appeler comme ça

Exécution (où Histoire)

•Exécution : ordonnancement des opérations d'un ensemble de transactions

•Ordre : *total* (séquence = transactions plates)
ou *partiel* (arbre, modèles avancés))

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

H_1= $W_2(x)$ $R_1(x)$ $R_3(x)$ $W_1(x)$ C_1 $W_2(y)$ $R_3(y)$ $R_2(z)$ C_2 $R_3(z)$ C_3

Exécution en série

•**Exécution en série :** histoire où il n'y a pas d'entrelacement des opérations de transactions

•**Hypothèse** : chaque transaction est *localement* cohérente

•Si la BD est cohérente avant l'exécution des transactions, alors elle sera également cohérente après leur exécution en série.

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

H_s= $W_2(x)$ $W_2(y)$ $R_2(z)$ C_2 $R_1(x)$ $W_1(x)$ C_1 $R_3(x)$ $R_3(y)$ $R_3(z)$ C_3

$T_2 \rightarrow T_1 \rightarrow T_3$

Exécution sérialisable

Opérations conflictuelles : deux opérations de deux transactions différentes sont en *conflit* si elles accèdent le même granule et *une des deux opérations est une écriture*.

Exécutions équivalentes : deux exécutions H1 et H2 d'un ensemble de transactions sont équivalentes (de conflit) si

- l'ordre des opérations de chaque transaction et
- l'ordre des opérations conflictuelles (validées)

sont identiques dans H1 et H2.

La bonne définition

Exécution sérialisable : exécution où il existe *au moins une* exécution en série (ou *sérielle)* équivalente (de conflit).

Exécutions équivalentes et sérialisables

T_1:	T_2:	$T3$:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

Les exécutions suivantes ne sont pas équivalentes :

H_1= $W_2(x)\ R_1(x)\ R_3(x)\ W_1(x)\ C_1\ R_3(y)\ W_2(y)\ R_2(z)\ C_2\ R_3(z)\ C_3$

H_2= $W_2(x)\ R_1(x)\ W_1(x)\ C_1\ R_3(x)\ W_2(y)\ R_3(y)\ R_2(z)\ C_2\ R_3(z)\ C_3$

H_2 est équivalente à H_s, qui est sérielle $\Rightarrow H_2$ est *sérialisable :*

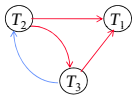
H_s= $W_2(x)\ W_2(y)\ R_2(z)\ C_2\ R_1(x)\ W_1(x)\ C_1\ R_3(x)\ R_3(y)\ R_3(z)\ C_3$

Est ce que H1 est sérialisable ?

Graphe de précédence (GP)

Graphe de précédence $GP_H=\{V,P\}$ pour l'exécution H:

- $V=\{T_i \mid T_i$ est une transaction *validée* dans $H\}$
- $P=\{T_i \rightarrow T_k$ si $o_{ij} \in T_i$ et $o_{kl} \in T_k$ sont en conflit et $o_{ij} <_H o_{kl}\}$

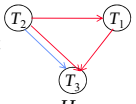


H_1

H_1= $W_2(x)\ R_1(x)\ R_3(x)\ W_1(x)\ C_1\ R_3(y)\ R_2(z)\ W_2(y)\ C_2\ R_3(z)\ C_3$

H_2= $W_2(x)\ R_1(x)\ W_1(x)\ C_1\ R_3(x)\ W_2(y)\ R_3(y)\ R_2(z)\ C_2\ R_3(z)\ C_3$

Théorème: l'exécution H est sérialisable ssi GP_H ne contient pas de cycle (facile à prouver, ordre partiel).



H_2

H_2 correspond à l'exécution en série : T_2-T_1-T_3

Méthodes de contrôle de concurrence

- Méthode optimiste
- Verrouillage à deux-phases (2PL)
- Multiversion (snapshot)
- Estampillage

Algorithmes de verrouillage

Les transactions font des demandes de verrous à un *gérant de verrous :*

- verrous en lecture (v*l*), appelés aussi verrous partagés
- verrous en écriture (*ve*), appelés aussi verrous exclusifs

Compatibilité (de verrous sur le même granule et deux transactions différentes) :

	vl	*ve*
vl	**Oui**	Non
ve	Non	Non

Algorithme Lock

```
Bool Function Lock (Transaction t, Granule G, Verrou V) {
  /* retourne vrai si t peut poser le verrou V sur le granule G et faux sinon
     (t doit attendre) */
   Cverrous := {};
   Pour chaque transaction t' ≠ t ayant verrouillé le granule G faire {
   Cverrous = Cverrous ∪ t'.verrous(G) } ; // cumuler les verrous sur G
   }
   si Compatible(V, Cverrous) alors {
      t.verrous(G) = t.verrous(G) ∪ { V };  // marquer l'objet verrouillé
        return true ;
   } sinon {
       /* insérer le couple (t, V) dans la liste d'attente de G  */
       G.attente = G.attente ∪ { t } ;
       bloquer la transaction t ;
       return false ;
   }
}
```


Algorithme Unlock

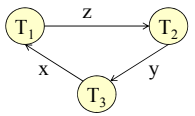
```
Procedure Unlock(Transaction t, Granule G) {
   /* t libère tous les verrous sur G et redémarre les transactions en
   attente (si possible) */
    t.verrou(G) := {};
    Pour chaque couple (t', V) dans G.attente faire {
      si Lock(t', G, V) alors {
          G.attente = G.attente - {(t', V)};
          débloquer la transaction t';
      }
    }
}
```


Graphe d'Attente (GA)

$vl_1(x)$, $ve_2(z)$, $ve_3(y)$, $vl_1(z)$, $ve_3(x)$, $vl_2(y)$…

Granule	ve	vl	Attente ve	Attente vl
x		T_1	T_3	
y	T_3			T_2
z	T_2			T_1

Graphe d'attente :



Cycle → Interblocage

Ne pas confondre avec le G*raphe de Précédence (GP, sérialisibilité)*

Résolution des interblocages

Prévention

- Définir des critères de priorité de sorte à ce que le problème ne se pose pas
- Ne pas accepter toutes les attentes (abandon)

Détection

- Gérer le Graphe d'Attente (GA)
- Lancer un algorithme de détection de circuits dès qu'une transaction attend trop longtemps
- Choisir une victime qui brise le circuit

Prévention des interblocages

Algorithme : Les transactions sont numérotées par ordre d'arrivée et on suppose que T_i *désire un verrou détenu par* T_j:

- Choix préemptif (« seuls plus jeunes attendent »):
  - j > i : T_i prend le verrou et T_j est *abandonnée*.
  - j < i : T_i attend.
- Choix non-préemptif (« seuls plus vieux attendent »):
  - j > i : T_i attend.
  - j < i : T_i est *abandonnée*.

Théorème : Il ne peut pas avoir d'interblocages, si une transaction abandonnée est toujours relancée avec *le même numéro*.

Verrouillage et sérialisabilité

Est-ce que le verrouillage permet de garantir la sérialisabilité ???

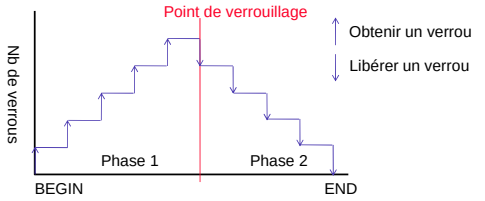
$H_1 = W_2(x)\ R_1(x)\ R_3(x)\ W_1(x)\ C_1\ R_3(y)\ R_2(z)\ W_2(y)\ C_2\ R_3(z)\ C_3$

On a vu que H1 n'est pas sérialisable, pourtant…

=> Il ne faut pas libérer les verrous trop tôt

Verrouillage à deux phases

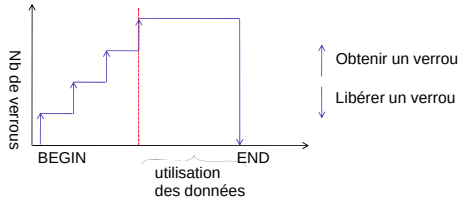
- Chaque transaction verrouille l'objet avant de l'utiliser.
- Quand une demande de verrou est en conflit avec un verrou posé par une autre transaction *en cours*, la transaction qui demande doit attendre.
- **Quand une transaction libère son premier verrou, elle ne peut plus demander d'autres verrous.**



Verrouillage à deux phases strict

On tient les verrous jusqu'à la fin (commit, abort).

Evite les abandons en cascade



Verrouillage à deux phases (2PL): Conclusion

Théorème : Le protocole de verrouillage à deux phases génère des *historiques sérialisables en conflit* (mais n'évite pas les fantômes).

Autres versions de 2PL (Oracle, snapshot isolation):

- Relâchement des verrous en lecture après l'opération :
  - non garantie de la reproductibilité des lectures (READ_COMMITTED)
  + verrous conservés moins longtemps : plus de parallélisme
- Accès à la version précédente lors d'une lecture bloquante :
  - nécessité de conserver une version (journaux)
  + une lecture n'est jamais bloquante

Transactions dans Oracle

Une transaction démarre lorsqu'on exécute une instruction SQL qui modifie la base ou le catalogue (DML et DDL).

- Ex : UPDATE, INSERT, CREATE TABLE…

Une transaction se termine dans les cas suivants :

- L'utilisateur valide la transaction (COMMIT)
- L'utilisateur annule la transaction (ROLLBACK sans SAVEPOINT)
- L'utilisateur se déconnecte (la transaction est validée)
- Le processus se termine anormalement (la transaction est défaite)

Amélioration des performances dans Oracle :

- Niveaux d'isolation
- Contrôle de concurrence multiversion

Verrouillage

Commandes transactionnelles

COMMIT

- Termine la transaction courante et écrit les modifications dans la base.
- Efface les points de sauvegarde (SAVEPOINT) de la transaction et relâche les verrous.

ROLLBACK

- Défait les opérations déjà effectuées d'une transaction

SAVEPOINT

- Identifie un point dans la transaction indiquant jusqu'où la transaction doit être défaite en cas de rollback.
- Les points de sauvegarde sont indiqués par une étiquette (les différents points de sauvegarde d'une même transaction doivent avoir des étiquettes différentes).

SET TRANSACTION

SET TRANSACTION
- Spécifie le comportement de la transaction :
  - Lectures seules ou écritures (READ ONLY ou READ WRITE)
  - Établit son niveau d'isolation (ISOLATION LEVEL)
  - Permet de nommer une transaction (NAME)

Cette instruction est facultative. Si elle est utilisée, elle doit être la première instruction de la transaction, et n'affecte que la transaction courante.

Niveaux d'isolation

Oracle propose deux niveaux d'isolation, pour spécifier comment gérer les mises à jour dans les transactions
- **SERIALIZABLE**
- **READ COMMITTED**

Définition
- Pour une transaction :
  - **SET TRANSATION ISOLATION LEVEL SERIALIZABLE**
  - **SET TRANSATION ISOLATION LEVEL READ COMMITTED**
- Pour toutes les transactions à venir (dans une session)
  - **ALTER SESSION SET ISOLATION LEVEL = SERIALIZABLE;**
  - **ALTER SESSION SET ISOLATION LEVEL = READ COMMITTED;**

Différence read-committed serializable pour lectures

en mode RC, on lit la dernière valeur validée depuis le début de la commande SQL Select

en mode SR, on lit la dernière valeur validée depuis le début de la transaction (multiversion).

En d'autre terme, RC permet des lectures non reproductibles alors que SR ne le permet pas

Différence read committed / serializable pour écriture

• Dans les deux modes, une transaction attend que le verrou X se libère pour prendre le verrou et faire la mise à jour

• En mode SR, une mise-à-jour de T sera refusée s'il y a eu une modif. faite et validée par une autre transaction depuis le démarrage de T.

•La valeur que T « écrase » doit avoir été validée *avant le début* de T

Mais : E1(a), E2(b), L1(b), L2(a), V1, V2 sera accepté en mode SR n'est ni équivalent à T1, T2, ni équivalent à T2, T1 car les deux lectures vont lire l'état initial (avant le début de l'exécution)

Sérialisation par estampillage : Idée principale

- Hypothèse :
  - Une « vieille » transaction ne génère pas de conflit avec une plus « jeune ».
- Une transaction t peut
  - *lire* une granule g si la dernière transaction qui a *écrit* sur g est plus âgée que t
  - écrire sur une granule g si les dernières transactions qui on lu ou écrit sur g sont plus âgées que t
- Sinon on annule t en entier et on la redémarre (elle renaît)

Sérialisation par estampillage (1/3)

On affecte

- **à chaque transaction** t une estampille unique TS(t) dans un domaine ordonné.
- **à chaque granule g**
- une étiquette de lecture EL(g) et
- une étiquette d'écriture EE(g)

qui contient *l'estampille de la dernière transaction qui a lu, respectivement, écrit g*.

Sérialisation par estampillage (2/3)

La transaction t veut lire g :

•Si TS(t) ≥ EE(g), la lecture est acceptée et

EL(g) := max(EL(g), TS(t)).

•Sinon la lecture est refusée et t est relancée (abort) avec une nouvelle estampille (*plus grande que toutes les autres)*.

La transaction t veut écrire g :

•Si TS(t) ≥ max(EE(g), EL(g)), l'écriture est acceptée et

EE(g) := TS(t).

•Sinon l'écriture est refusée et t est relancée avec une nouvelle estampille (*plus grande que toutes les autres).*

Sérialisation par estampillage (3/3)

$L_1(b), L_2(b), E_2(b), L_1(a), L_2(a), E_2(a), \mathbf{E_1(b)?}$

•$TS(t_1) < TS(t_2)$: T_1 sera relancée avec une nouvelle estampille $TS'(t_1) > TS(t_2)$:

$L_2(b), E_2(b), \quad L_2(a), E_2(a), L_3(b), L_3(a), \mathbf{E_3(b)}$

Remarque :

•Il existe des historiques qui ne peuvent pas être *produits* par 2PL mais sont admis par estampillage et vice-versa.

Règle de Thomas

$L_1(b), L_1(a), E_2(b,v), L_2(a), E_2(a), \mathbf{E_1(b,v')?}$
$L_1(b), L_1(a), \mathbf{E_1(b,v')}, E_2(b,v), L_2(a), E_2(a)$

Observation : Aucune transaction avec TS(t) > TS(t_1) a lu b :
L'abandon de T_1 n'est pas nécessaire car ***v'*** n'aurait jamais été lue, si l'écriture s'était passée plus tôt.

Nouvelle règle pour l'écriture:

- Si TS(t) ≥ max(EE(g), EL(g)), l'écriture est acceptée et EE(g) := TS(t).
- **Si TS(t) < EE(g) et TS(t) ≥ EL(g), l'écriture est *ignorée.***
- Sinon l'écriture est refusée et T est relancée avec une nouvelle estampille.

Le problème des annulations

H= $W_2(x)\ R_1(x)\ W_1(x)\ \mathbf{C_1}\ R_3(x)\ W_2(y)\ R_3(y)\ R_2(z)\ R_3(z)\ \mathbf{A_2}$

Quand une transaction est abandonnée (A_2) il faut annuler

- les écritures de T_2
- **et** les transactions T_1 et T_3 qui utilisent ces écritures

Problèmes :

- annulation de transactions *en cascade* : possible mais coûteux
- annulation de transactions déjà validées (T_1) : impossible !
  Exécution non recouvrable

Recouvrabilité

- Exécution *recouvrable* : pas d'annulation de transactions validées
- Solution : **retardement du *commit***
- *Si T « lit de » T', alors T doit valider après T'*

H= $W_2(x)\ R_1(x)\ W_1(x)\ C_1\ R_3(x)\ W_2(y)\ R_3(y)\ R_2(z)\ R_3(z)\ A_2$

Éviter annulation en cascade

H= $W_2(x)\ R_1(x)\ W_1(x)\ R_3(x)\ W_2(y)\ R_3(y)\ R_2(z)\ R_3(z)\ A_2$

- T2 annule ⇒T1 et T3 doivent annuler
- Solution : **retardement des *lectures***
- *Une transaction ne doit lire que des transactions validées*

H= $W_2(x)\ R_1(x)\ W_1(x)\ R_3(x)\ W_2(y)\ R_3(y)\ R_2(z)\ R_3(z)\ A_2$

Exécutions strictes

$H=W_2(x)\ W_1(x)\ ....\ A_2\ ...\ A_1$

T2 annule ⇒ comment restaurer la valeur de *x* ?

A_2 restaure *x* initial, A_1 restaure *x* écrit par T_2

(il est possible de changer le comportement, mais complexe)

Solution : **retardement des *lectures et écritures***

Une transaction ne doit écrire que sur des données validées

$H= W_2(x)\ W_1(x)\ ...\ A_2$

2PL strict garantit des exécutions strictes

Modèles étendus

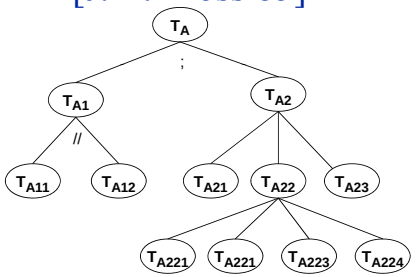
Applications longues composées de plusieurs transactions coopérantes

Seules les mises-à-jour sont journalisées

Si nécessité de défaire une suite de transactions:

- contexte ad-hoc dans une table temporaire
- nécessité d'exécuter des compensations

Transactions Imbriquées T.I. (1) [J. E. Moss 85]



TI = Ensemble de transactions qui peuvent être elles mêmes imbriquées (on dit aussi « emboitées »)

Transactions Imbriquées (2)

Sous-transaction : *unité d'exécution*

- Une sous-transaction démarre après et finit avant sa mère. Des sous-transactions au même niveau peuvent être exécutées en concurrence (sur différents sites)
- Chaque sous-transaction est exécutée de manière indépendante; elle peut décider soit de valider soit d'abandonner

Sous transaction : *unité de reprise*

- Si une sous-transaction valide, la mise à jour de la BD a lieu seulement lorsque la transaction racine valide.
- Si une sous-transaction abandonne, ses descendants abandonnent.

Transactions Imbriquées (3)

Une sous-transaction peut-être:

- **Obligatoire**: si elle abandonne, son père doit abandonner
- **Optionnelle**: si elle abandonne, son père peut continuer (abandon partiel)
- **Contingente** : si elle abandonne, une autre peut être exécutée à sa place

Si d'abandon d'une sous-transaction, le père soit :

- Abandonne aussi (et donc son sous-arbre abandonne)
- Continue sans les résultats de la sous-transaction abandonnée
- Relance la sous-transaction ou une alternative

Transactions Imbriquées (4)
Contrôle de la concurrence

(R1) Seules les transactions feuilles accèdent aux objets

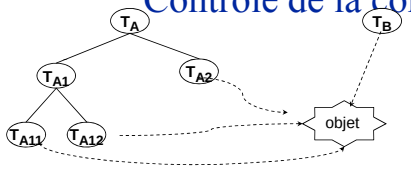
(R2) Quand une sous-transaction valide, ses verrous sont hérités par sa mère

(R3) Quand une sous-transaction abandonne, ses verrous sont relâchés

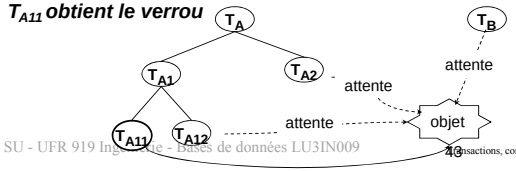
(R4) Une sous-transaction ne peut accéder à un verrou que si il est libre ou détenu par un ancêtre

⇒ *R1 à R4 garantissent l'isolation*

Transactions Imbriquées (5)
Contrôle de la concurrence

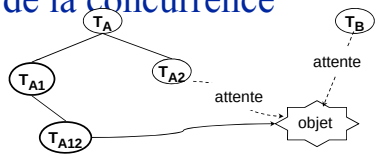


T_{A11} obtient le verrou

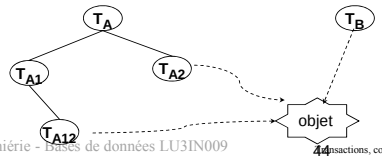


Transactions Imbriquées (6)
Contrôle de la concurrence

Si T_{A11} valide



Si T_{A11} abandonne



Transactions Imbriquées (7)

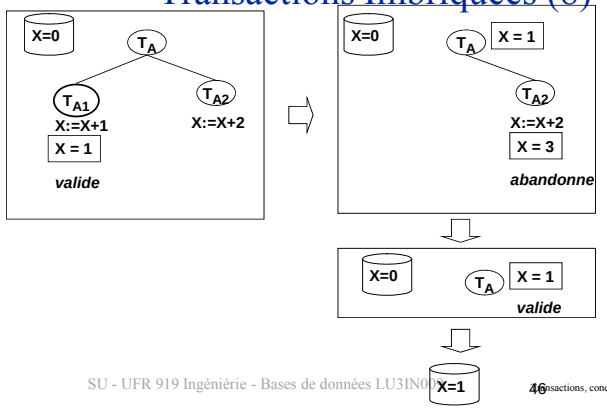
(R'1) Quand une sous-transaction valide, ses effets sont hérités par sa mère

(R'2) Quand une sous-transaction abandonne, ses effets sont abandonnés.

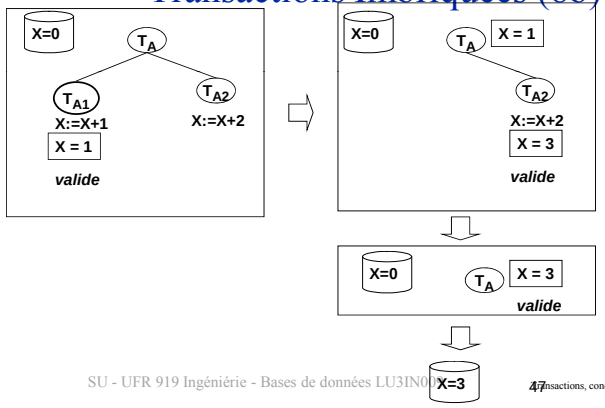
(R'3) Quand la transaction racine valide, ses effets sont écrits dans la base

R'1 à R'3 garantissent atomicité et durabilité

Transactions Imbriquées (8)



Transactions Imbriquées (8b)



Sagas

Groupe de transactions avec transactions compensatrices

En cas de panne du groupe, on exécute les compensations

Seules les sous-transactions sont isolées…. Ce n'est plus vraiment un modèle transactionnel

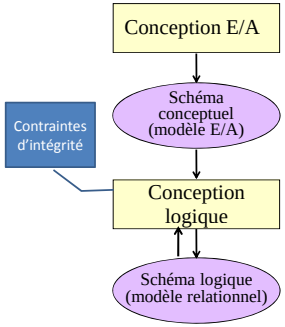
Conclusion

- La gestion des transactions est une tâche importante dans un SGBD :
  - Gestion de pannes
  - Contrôle de concurrence
  - ACID
  - garantir la cohérence sans trop bloquer les ressources

UFR 919 Ingénierie – module LU3IN009
cours 8 - Conception de bases de données

- Conception logique
- Dépendances fonctionnelles
- Décomposition de schémas et normalisation
- Formes normales
- Algorithme de décomposition

Étapes de conception



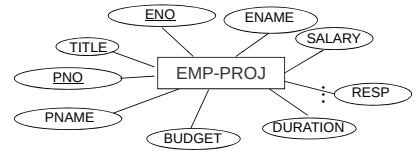
Conception "graphique" et conversion E/A-relationnel

Conception logique pour la validation formelle et l'amélioration du schéma relationnel

Objectifs de la conception logique

- Éviter des **incohérences** dans les données :
  Une personne n'a qu'une date de naissance, le prix d'un produit est unique, …. Coûteux à vérifier
- Éviter la **redondance** d'information :
  La même information est stockée plusieurs fois
  Anomalies: insertion, suppression, modification
- Éviter les **valeurs nulles** :
  Difficiles à interpréter : inconnu, connu mais non disponible, inapplicable, Rend les jointures difficiles à spécifier
- Éviter les **jointures inutiles** :
  Améliorer les performances : la jointure est coûteuse

Pourquoi pas une seule relation ? (le cas extrême)



EMP-PROJ

ENO	ENAME	TITLE	SALARY	PNO	PNAME	BUDGET	DURATION	RESP
E1	J. Doe	Elect. Eng.	40000	P1	Instrumentation	150000	12	Manager
E2	M. Smith	Analyst	34000	P1	Instrumentation	150000	24	Analyst
E2	M. Smith	Analyst	34000	P2	Database Develop.	135000	6	Analyst
E3	A. Lee	Mech. Eng.	27000	P3	CAD/CAM	250000	10	Consultant
E3	A. Lee	Mech. Eng.	27000	P4	Maintenance	310000	48	Engineer
E4	J. Miller	Programmer	24000	P2	Database Develop.	135000	18	Programmer
E5	B. Casey	Syst. Anal.	34000	P2	Database Develop.	135000	24	Manager
E6	L. Chu	Elect. Eng.	40000	P4	Maintenance	310000	48	Manager
E7	R. Davis	Mech. Eng.	27000	P3	CAD/CAM	250000	36	Engineer
E8	J. Jones	Syst. Anal.	34000	P3	CAD/CAM	250000	40	Manager

Problèmes

Problème principal : Redondance d'information

les attributs TITLE, SALARY, BUDGET, ENAME… sont répétés pour chaque projet dans lequel travaille l'employé.

Trop d'attributs pour une seule relation

Conséquence : Anomalies

Anomalie d'insertion : il est difficile/impossible d'insérer un nouveau projet tant qu'il n'a pas d'employé affecté (valeurs nulles)

Anomalie de suppression : si le *dernier employé* d'un projet est supprimé, le projet est automatiquement supprimé aussi. Pour éviter cela, il faut prévoir un traitement spécifique à l'effacement du dernier employé.

Anomalie de modification : si un attribut *d'un projet*, par ex. son budget, est modifié, *tous les n-uplets des employés du projet* doivent être modifiés (pas de on update cascade oracle => trigger)

Solution : ***couper la relation en plusieurs. Comment ? Ce cours et le suivant***

Plan

Comment « décrire » plus précisément et formellement les données stockées dans une base de données ?

⇨Dépendances fonctionnelles

Comment passer d'un schéma relationnel vers un autre schéma « équivalent » par rapport à cette description ?

⇨ Décomposition de schémas

Comment caractériser un « bon schéma » ?

⇨ Formes normales

Comment passer d'un schéma quelconque (universel) vers un « bon » schéma équivalent ?

⇨ Algorithme de décomposition

Conception de bases de données

- Conception logique
- Dépendances fonctionnelles
- Décomposition de schémas et normalisation
- Formes normales
- Algorithme de décomposition

Dépendances fonctionnelles

Préliminaire important

Une dépendance fonctionnelle est une contrainte d'intégrité. Par conséquent :

- **Elle est observée dans le monde réel**
- **Elle doit être maintenue par le système (donc de manière efficace)**
- **Elle fait partie du schéma de la base**
- **Elle ne peut pas être déduite des valeurs d'une instance particulière (ce n'est pas parce qu'une DF est satisfaite par une instance qu'elle existe dans le monde réel)**

Dépendances fonctionnelles

R(Etudiant, Prof, Cours, Salle, Heure)

On sait (on l'a observé sur le monde réel)

qu'un étudiant ne peut pas être dans deux salles en même temps

ou

si on connaît le nom de l'étudiant et l'heure on peut identifier la salle (si elle existe)

ou

si t est un n-uplet dans R, il n'y a pas un autre n-uplet t' où t.Etudiant=t'.Etudiant et t.Heure=t'.Heure et t.Salle <> t'.Salle

On note alors

Etudiant,Heure → Salle

Dépendances fonctionnelles

Les *dépendances fonctionnelles* sont un moyen pour exprimer des ***contraintes sémantiques*** sur les données :

R(Etudiant, Prof, Cours, Salle, Heure)

Etudiant, Heure → Salle : un étudiant ne peut pas être dans deux salles différentes à la même heure

Salle,Heure → Cours : il ne peut pas avoir deux cours différents en même temps dans la même salle

Cours → Prof, Salle : il y a exactement un prof et une salle pour chaque cours

Inférence : On peut montrer à partir des dépendances ci-dessus

- qu'un étudiant ne peut pas suivre deux cours en même temps : **Etudiant,Heure → Cours**
- que (Salle,Heure, Etudiant) est une clé de R.

Notations

Attributs : A, B, C, …
Ensembles d'attributs : X, Y, Z
U = univers des attributs; XY correspond à X ∪ Y

Schémas de relations R, R' :
avec leurs attributs: R(X), R'(XY), …
Relations : r, r'
Relations avec leurs attributs: r(X), r'(XY), …

N-uplets : t, t'
Valeurs d'attributs de n-uplets: t.A, t.X

Dépendances fonctionnelles (DF): f, g, h
Ensembles de DF: F, G, H

Dépendances fonctionnelles

- Une dépendance fonctionnelle f est une expression

$$f: X \rightarrow Y$$

où X et Y sont des ensembles d'attributs (on dit que f est définie sur XY).

- Une *relation r(Z)* satisfait *la dépendance* fonctionnelle $f: X \rightarrow Y$ si

1. $XY \subseteq Z$ (elle contient tous les attributs dans XY) et
2. tous les n-uplets t et t' dans r(Z) qui ont les mêmes valeurs pour les attributs dans X, partagent également les mêmes valeurs pour les attributs Y :

$$\forall t,t' \in r: t.X = t'.X \Rightarrow t.Y = t'.Y.$$

Inférence de DFs

Soit F un *ensemble de dépendancesfonctionnelles* défini sur un ensemble d'attributs U :

- On dit que F implique (logiquement) la DF $X \rightarrow Y$ (avec $XY \subseteq U$) si *toutes les relations r(U)* qui satisfont toutes les dépendances fonctionnelles dans F, satisfont également $X \rightarrow Y$.
- On note alors :

$$F \models X \rightarrow Y$$ (parfois $F \Rightarrow X \rightarrow Y$)

Exemple

- Est-ce que $F \models A \rightarrow C$ si $F = \{A \rightarrow B, B \rightarrow C\}$?
- Problème d'inférence : Il faut montrer que toutes les relations r(A,B,C) qui satisfont F, satisfont également $A \rightarrow C$.
- Preuve logique :

On sait que r satisfait F ssi

$\forall t,t' \in r: t.A = t'.A \Rightarrow t.B = t'.B$ et

$t.B = t'.B \Rightarrow t.C = t'.C$

On peut conclure que (modus ponens)

$\forall t,t' \in r: t.A = t'.A \Rightarrow t.C = t'.C$

Surclés et clés

- Soit R(Z) un schéma de relation, F un ensemble de dépendances fonctionnelles sur Z, et X un sous-ensemble des attributs de R, $X \subseteq Z$:
- X est une surclé de R avec F si $F \models X \rightarrow Z$.
- une surclé X est appelé un clé de R, s'*il n'existe pas* de sous-ensemble stricte $Y \subset X$ *de* X qui est aussi une surclé.

Une clé de R est un ensemble minimal (-au sens de l'inclusion) qui détermine tous les autres

Remarques :

- Une *clé primaire* est une surclé (généralement une clé) qui peut être utilisée pour organiser *physiquement* les données (*organizing index* dans *create table*).
- On peut trouver toutes les surclés et clés d'une relation à partir d'un ensemble de DF par *inférence.*
- *Chaque clé est une surclé mais pas inversement.*
- *L'ensemble Z est une surclé triviale de R(Z).*
- *Si un attribut A est une surclé, il est forcément une clé.*
- *Quand est-ce que Z est une clé de R(Z) ?*

Problème d'inférence

- **Question :** Est-ce que A est une clé de R(ABC) avec $F = \{A \rightarrow B, B \rightarrow C\}$?
- Il faut montrer que **$F \models A \rightarrow ABC$ (inférence)**.
- Preuve logique :
- On sait que r satisfait F ssi

$\forall t,t' \in r: t.A = t'.A \Rightarrow t.B = t'.B$ *et*

$t.B = t'.B \Rightarrow t.C = t'.C$

- On peut conclure

$\forall t,t' \in r: t.A = t'.A \Rightarrow t.A = t'.A \wedge t.B = t'.B \wedge t.C = t'.C$

Exemple

- **Question :**
  - Est-ce que A est une clé de R(AC) avec F = {A → B, B → C} ?
  - Ou : Est-ce que **F |= A → C ?**
- Attention : il *ne suffit pas* de regarder les dépendances dans F !
- Explication :
- Comme R ne contient pas l'attribut B, on pourrait conclure que A n'est pas une clé.
  - Mais : on peut montrer que **F |= A → C (inférence)** et ainsi que l'attribut A est une clé de R(AC) avec F = {A → B, B → C}

Il nous faut un moyen pratique de calculer les inférences :
•Fermeture d'un ensemble d'attribut
•Axiomes d'Armstrong

Inférence par le calcul de fermeture transitive d'un ensemble de DF

- La fermeture transitive (ou clôture) d'un ensemble de dépendances fonctionnelles F est l'ensemble de toutes les dépendances fonctionnelles *qu'on peut déduire de* F:

$$F^+=\{ X \rightarrow Y \mid F \models X \rightarrow Y \}$$

- Une relation r(Z) satisfait un ensemble de dépendances fonctionnelles F si elle satisfait toutes les dépendances fonctionnelles X → Y où
- $XY \subseteq Z$ et
- X → Y est *dans la fermeture transitive de F*.

Fermeture transitive d'un ensemble de dépendances fonctionnelles

- F = {A → B, B → C}
- F^+ = {A→B, B→C, A→C, *A→A, A→AB, A→AC, A→ABC, AB→A, AB→AB, AB→AC, AB→ABC, ABC→A, ABC→AB, ABC→AC, AB→ABC, … C→C, BC→C*}

- La fermeture contient beaucoup de *solutions triviales* et peut être très longue à calculer. Mais pas nécessaire car ce dont on a besoin de savoir c'est si une DF f ∈ F^+

Inférence par le calcul de la fermeture d'un ensemble d'attributs

La *fermeture d'un ensemble d'attributs* X contient tous les attributs qui dépendent des attributs dans X:

$$[X]^+_F = \{ A \mid F \models X \rightarrow A \}$$

Exemple avec F = {A → B, B → C}, R(AC)
$[A]^+_F$ = { A, B, C } : on peut conclure que
- A → C est dans la fermeture de F et
- A est un surclé (et clé) de R

On peut montrer facilement que

$$F \models X \rightarrow A \Leftrightarrow A \in [X]^+_F$$

Fermeture d'un ensemble d'attributs

function *ComputeX*$^+$(*X*, *F*)
begin
$X^+ := X$
while exists $(Y \rightarrow Z) \in F$ **tel que** $Y \subseteq X^+$ et $Z \not\subseteq X^+$
$X^+ := X^+ \cup Z$
$F := F - \{Y \rightarrow Z\}$
return(X^+)
end

Exemple de fermeture d'attributs

Soit *F* l'ensemble de DF sur R(A,B,C,D,E,G,H)

- A → B
- C → DE
- EG → H

Est-ce que CG est une (sur)clé de R ?

ComputeX$^+$({C, G}, *F*)

- Initialisation : X^+ = X = {C, G}
- Itération 1(C → DE): X^+ = {C, G, D, E}
- Itération 2 (EG → H): X^+ = {C, G, D, E, H}
- Iteration 3 : on n'a plus de DF à appliquer

Conclusion ?

Calcul de clés : Exemple

R(Etudiant, Prof, Cours, Salle, Heure)

- Etudiant Heure → Salle
- Salle Heure → Cours
- Cours → Prof Salle
- Etudiant Salle → Heure

Les attributs de R qui ne sont jamais en partie droite d'une DF font partie de toutes les clés (ne peuvent être « obtenus » à partir des autres)

Comment trouver toutes les clés par rapport aux DF ?

- Quels attributs n'apparaissent jamais à droite d'une DF ? : Etudiant
- Est-ce que Etudiant est une clé ? : Non : Etudiant+ = Etudiant
- Quels attributs on essaie en premier à ajouter ? : Heure ou Salle
- Pourquoi ?
- Est-ce que (Etudiant,Heure) est une clé ?
  Oui : (Etudiant, Heure)+ = Etudiant Heure Salle Cours Prof
- Est-ce que (Etudiant ,Salle) est une clé ?
  Oui : (Etudiant ,Salle)+ = Etudiant Salle Heure Cours Prof
- On peut s'arrêter là ? Non : On peut aussi essayer Etudiant Cours ? ...

Équivalence et couverture minimale (1/2)

- Deux ensembles de DF différents peuvent exprimer les même contraintes:

F ={AB → D, D →C, C →D, AB →C}
G ={AB → C, D →C, C →D}

- F et G sont *équivalents* (expriment les mêmes contraintes).
On note : F ≡ G
- G est plus « compacte »

Équivalence et couverture minimale (2/2)

F est équivalent à G (F ≡ G) si $F^+ = G^+$
pas de panique, on n'a pas besoin de calculer F^+ni G^+ car on peut montrer que
F ≡ G ssi ∀f∈ F, G |= f et ∀g ∈ G, F |= g

F est un *ensemble minimal* de DF ssi
1. Toute DF de F est sous la forme X → A (un seul attribut à droite, forme canonique)
2. F *ne contient pas* de DF X → A qui
   - *Est redondante* : $F^+ = (F - \{ X \rightarrow A \})^+$ *ou*
   - *Contient des attributs en trop à gauche* : X'⊂ X et F |= X' →A ∉F^+

F est une *couverture minimale* de G si
F ≡ G et F est un *ensemble minimal* de DF.

Exemple de couverture minimale

F={AB→CD, ACE→B, D→C, C→D, CD→BE}
Forme canonique (un seul attribut sur le côté droit) :
F_1={AB→C, AB→D, ACE→B, D→C, C→D, CD→B, CD→E}
Test de redondance : on peut enlever
- ACE→B
  - C→D, CD→B |= C→ B : on peut enlever ACE→B
- et AB→C ou AB→D :
  - AB→D, D→C |= AB→C : on peut enlever AB→C ou
  - AB→C, C→D |= AB→D : on peut enlever AB→D

Exemple de couverture minimale

F={AB→CD, ACE→B, D→C, C→D, CD→BE}
Forme canonique (un seul attribut sur le côté droit) :
•F_1={AB→C, AB→D, ACE→B, D→C, C→D, CD→B, CD→E}
Test de redondance : on peut enlever ACE→B et (AB→C ou AB→D) (deux solutions):
1.F_2={AB→C, D→C, C→D, CD→B, CD→E}
Test de redondance des attributs à gauche :
▪$[C]^+_{F2}$ = {C, D, B, E} et $[D]^+_{F2}$ = {C, D, B, E}
▪On peut enlever C ou D dans CD → B et CD → E (4 solutions)
1.F_3={AB→D, D→C, C→D, CD→B, CD→E}
Test de redondance des attributs à gauche (idem; 4 solutions)

8 couvertures minimales

1.$F_{2.1}$={AB→C, D→C, C→D, C→B, C→E}
2.$F_{2.2}$={AB→C, D→C, C→D, C→B, D→E}
3.$F_{2.3}$={AB→C, D→C, C→D, D→B, C→E}
4.$F_{2.4}$={AB→C, D→C, C→D, D→B, D→E}
5.$F_{3.1}$={AB→D, D→C, C→D, C→B, C→E}
6.$F_{3.2}$={AB→D, D→C, C→D, C→B, D→E}
7.$F_{3.3}$={AB→D, D→C, C→D, D→B, C→E}
8.$F_{3.4}$={AB→D, D→C, C→D, D→B, D→E}

Inférence avec les axiomes d'Armstrong

Il est possible de montrer qu'un ensemble de dépendances fonctionnelles F implique une DF X → A, grâce aux axiomes d'Armstrong, qui sont

- sains : si F|— X → A alors F|= X → A et
- complets: si F|= X → A alors F|— X → A

Remarques :

- F|— X → A signifie que la DF X → A peut être déduite à partir de F et les axiomes d'Armstrong.
- F^+ = ensemble de DF qu'on peut déduire de F en appliquant (récursivement) les axiomes d'Armstrong

Axiomes d'Armstrong

Soient *X*, *Y* et *Z* des ensembles d'attributs du schéma de relation *R.*

Axiomes d'Armstrong :

- Augmentation: $\{X \rightarrow Y\} \Rightarrow \{XZ \rightarrow YZ\}$
- Transitivité: $\{X \rightarrow Y, Y \rightarrow Z\} \Rightarrow \{X \rightarrow Z\}$
- Réflexivité: $W \subseteq X \Rightarrow \{X \rightarrow W\}$

Règles additionnelles :

- Union: $\{X \rightarrow Y, X \rightarrow Z\} \Rightarrow (X \rightarrow YZ)$
- Décomposition: $\{X \rightarrow YZ\} \Rightarrow \{X \rightarrow Y, X \rightarrow Z\}$
- Pseudotransitivité: $\{X \rightarrow Y, WY \rightarrow Z\} \Rightarrow \{XW \rightarrow Z\}$

Axiomes d'Armstrong : exemple

- F = {A → C, B → D}
- On peut montrer que AB est une surclé de R=ABCD :

1. $AB \rightarrow ABC$ (A → C + augmentation)
2. $ABC \rightarrow ABCD$ (B → D + augmentation)
3. AB → ABCD (1.+2. + transitivité)

UFR 919 Ingénierie – module LU3IN009 cours 9 Conception de bases de données (suite)

- Conception logique
- Dépendances fonctionnelles
- Décomposition de schémas et normalisation
- Formes normales
- Algorithme de décomposition

Normalisation

Normalisation : Méthodologie de conception pour produire un « *bon schéma* » par *décomposition* (descendante) d'un schéma d'origine ou par *génération* (ascendante)

Le schéma produit doit

- *éviter les anomalies de mises-à-jour : forme normale*
- *préserver la sémantique du schéma d'origine : sans perte d'informations et de dépendances*

Idée :

- On part d'un *schéma de relation R* et d'un *ensemble de dépendances fonctionnelles F* définies sur R (contraintes sémantiques)
- On applique un ensemble de transformations logiques de R en respectant les contraintes définies par F

La redondance vient des DF, il est normal que les DF soient utilisées pour normaliser

Raffinement par décomposition

EMP-PROJ

ENO	ENAME	TITLE	SALARY	⋈	PNO	PNAME	BUDGET	DURATION	RESP
E1	J. Doe	Elect. Eng.	40000		P1	Instrumentation	150000	12	Manager
E2	M. Smith	Analyst	34000		P1	Instrumentation	150000	24	Analyst
E2	M. Smith	Analyst	34000		P2	Database Develop.	135000	6	Analyst
E3	A. Lee	Mech. Eng.	27000		P3	CAD/CAM	250000	10	Consultant
E3	A. Lee	Mech. Eng.	27000		P4	Maintenance	310000	48	Engineer
E4	J. Miller	Programmer	24000		P2	Database Develop.	135000	18	Programmer
E5	B. Casey	Syst. Anal.	34000		P2	Database Develop.	135000	24	Manager
E6	L. Chu	Elect. Eng.	40000		P4	Maintenance	310000	48	Manager
E7	R. Davis	Mech. Eng.	27000		P3	CAD/CAM	250000	36	Engineer
E8	J. Jones	Syst. Anal.	34000		P3	CAD/CAM	250000	40	Manager

ENO→ENAME, TITLE, SALARY
PNO →PNAME, BUDGET *ENO, PNO →DURATION, RESP*
Décomposé en EMP(ENO,ENAME, TITLE, SALARY)
PROJECT(PNO, PNAME, BUDGET)
EMP-PROJbis(ENO, PNO, DURATION, RESP)

- Qu'est-ce qu'une bonne décomposition ?
- Est-ce que une décomposition donnée est bonne ?
- Comment obtenir une bonne décomposition ?

Problèmes de la normalisation

- **Problème** : Décomposer le schéma (S,F) en plusieurs relations sans "perdre" des informations et/ou des dépendances dans F :
  - Décomposition sans perte d'informations : pour *chaque base de données (BD)* du schéma S qui satisfait F, il doit être possible de la reconstruire (par des jointures) à partir des tables obtenues après la décomposition (par projection) sans inventer de nouveaux n-uplet
  - Décomposition avec préservation des dépendances : il doit être possible de vérifier toutes les contraintes définies par F sans faire de jointures (efficacité).
- Impact sur les requêtes ?
  - Le temps d'exécution peut augmenter à cause des jointures nouvelles

Décomposition de relations

- Un schéma $\{R_1, R_2, \ldots, R_n\}$ est une décomposition du schéma de relation R si $R = R_1 \cup R_2 \cup \ldots \cup R_n$
  et il y a suffisamment d'attributs commun pour joindre tous les R_i. Par exemple, {R1(AB), R2(CD)} n'est pas une décomposition de R(ABCD)
- Pourquoi la décomposition de R(ABC) en R1(AB), R2(BC) est «mauvaise» (s'il n'y a pas de DF) ?
- *Réponse* : Il existe des instances de R où la jointure de R1 et R2 « invente » des n-uplets nouveaux (exemple ci-dessous).
- Est-ce que la décomposition est encore mauvaise quand on sait que R satisfait $B \rightarrow C$?
- *Réponse* : pas sûr (la relation R ci-dessous ne satisfait pas cette DF)

R(A,B,C)
1,2,3
4,2,5

décomp.

R1(A,B)	R2(B,C)
1,2	2,3
4,2	2,5

jointure

(R1 ⋈ R2)(A,B,C)
1,2,3
1,2,5
4,2,3
4,2,5

Décomposition SPI

Décomposition Sans Perte d'Information (SPI) :

Pour toute instance *r* de *R* décomposée en instances r_i de R_1, …, R_n (par projection), on doit pouvoir reconstruire *r* à partir des r_i (par jointure) sans nuplets supplémentaires

Ci-dessous R satisfait $B \rightarrow C$, la décomposition est peut-être SPI. Il faut que ça marche pour toute instance.

R(A,B,C)
1,2,3
4,3,5

décomp.

jointure

R1(A,B)	R2(B,C)
1,2	2,3
4,3	3,5

Décomposition sans perte d'information (SPI)

Une décomposition de R en $\{R_1, R_2, \ldots, R_n\}$ est sans perte d'information (SPI) par rapport à un ensemble de DF F ssi:
$\forall r(R)$: si r satisfait F, alors $r = \Pi_{R_i}(r) \bowtie \ldots \bowtie \Pi_{R_n}(r)$

Comment vérifier SPI seulement en regardant les DF:
Algorithme de poursuite (« chase »)

- On a toujours $r \subseteq \Pi_{R_i}(r) \bowtie \ldots \bowtie \Pi_{R_n}(r)$
  (SPI = pas de nuplets en trop)

- Il suffit de prouver l'inclusion dans l'autre sens : on montre que tous les n-uplets t générés par la jointure entre les n-uplets $t_i \in \Pi_{R_i}(r)$ étaient dans r à cause des DF dans F

Tableaux (1/2)

Soit donnée une décompo. de R en $\{R_1, R_2, \ldots, R_n\}$ avec l'ensemble de DF F.

Tableau (représente une instance de R) :

- On veut montrer que $r \supseteq \Pi_{R_i}(r) \bowtie \ldots \bowtie \Pi_{R_n}(r) = r_i \bowtie \ldots \bowtie r_n$
- On prend un n-uplet quelconques dans la jointure et on montre qu'il est dans r
- On définit pour chaque attribut A une constante c_A.
- On crée un tableau (relation) qui contient tous les attributs de R et un nuplet t_i pour chaque schéma R_i tel que
- $t_i.A = c_A$ si l'attribut A fait partie de R_i t_i est défini sur A
- $t_i.A$ est une nouvelle valeur (différente de toutes les autres) si l'attribut A ne fait pas partie de R_i t_i n'est pas défini sur A
- t_i représente le n-uplet de $\Pi_{R_i}(r) = r_i$ qui a donné t dans la jointure

Tableaux (2/2)

FournisseurProduit(NomF, Adr, NomP, Prix)
est décomposé en
Fournisseur(NomF, Adr) Produit(NomP, NomF, Prix)

Tableau (FournisseurProduit):

	NomF	Adr	NomP	Prix
t1	nom	adr	*b31*	*b41*
t2	nom	*b22*	prod	prix

- t1 est défini sur NomF et Adr
- t2 est défini sir NomF, NomP et Prix

Algorithme de poursuite (1/2)

Soit donnés un tableau T et un ensemble de DF F.

Algorithme de poursuite :

do

a)Choisir une DF **X** → Y et trouver toutes les lignes T_X de T qui sont *identiques pour tous les attributs dans* **X**.

b) Unification sur Y : *remplacer* dans toutes les lignes T_X et pour tous les attributs A dans Y la valeur $t_i.A$ par

- la constante c_A s'il existe au moins un n-uplet t_j T_X avec $t_j = c_A$
- une valeur (différente de c_A) sinon.

until il existe au moins une ligne complètement défini (succès = décomposition SPI) ou il n'y a plus de remplacement possible (échec).

En pratique, considérer les DF dans un ordre donné, et recommencer à la première lorsqu'on a passé la dernière et pas de ligne complètement définie

Algorithme de poursuite (2/3)

FournisseurProduit(NomF, Adr, NomP, Prix)
est décomposé en
Fournisseur(NomF, Adr) Produit(NomP, NomF, Prix)
F = {NomF → Adr, (NomF, NomP) → Prix}
Tableau de poursuite:

	NomF	Adr	NomP	Prix
t1	nom	adr	*b31*	*b41*
t2	nom	adr	prod	prix

NomF → Adr ⇒ b22 = adr ⇒ t1 ⋈ t2 ∈ r ⇒ décomp. est SPI

Algorithme de poursuite (3/3)

FournisseurProduit(NomF, Adr, NomP, Prix)
est décomposé en
Rel1(NomF, Adr, NomP) Rel2(NomF, Prix)
F = {NomF → Adr, (NomF, NomP) → Prix}
Tableau de poursuite:

	NomF	Adr	NomP	Prix
t1	nom	adr	prod	*b41*
t2	nom	adr	*b32*	prix

NomF → Adr ⇒ b22 = adr ⇒ echec ⇒ décomp. *n'est pas* SPI

Décomposition en 2 et plusieurs relations

Théorème: Soit $\{R_1,R_2\}$ une décomposition de R et F l'ensemble des DF qui s'appliquent à R, alors *la décomposition de R en deux relations $\{R_1,R_2\}$ est SPI ssi l'intersection de R_1 et R_2 est une surclé d'au moins une des deux relations:*

$$\{R_1,R_2\} \text{ SPI} \equiv (R_1 \cap R_2) \rightarrow R_1 - R_2 \text{ } ou \text{ } (R_1 \cap R_2) \rightarrow R_2 - R_1$$

Preuve: il suffit de faire le tableau de poursuite

Théorème: Si $\{\mathbf{R_1}, R_2, \ldots, R_n\}$ est SPI de R et $\{S_1, S_2, \ldots, S_m\}$ est SPI de $\mathbf{R_1}$, *alors* $\{S_1, S_2, \ldots, S_m, R_2, \ldots, R_n\}$ est SPI de R.

Décomposition SPD

Décomposition Avec Préservation des DF (SPD) :

- Si (R,F) est décomposé en $R_1, \ldots, R_n$, alors dans chaque instance de R_i on ne peut valider que les DF $X \rightarrow Y$ **de** $\boldsymbol{F^+}$ *où R_i contient XY* (F est projeté sur les attributs de R_i : F_i)
- On veut être sûr que les contraintes définies par F sur R peuvent être vérifiées en vérifiant uniquement les DF F_i sur chaque instance R_i

$\Rightarrow$ sinon on serait obligé de *recalculer R (par des jointures)* pour vérifier les DF *perdues !*

Décomposition SPD (1/4)

- { R_1(CodePostal, Rue), R_2(CodePostal, Ville) }
  est une **décomposition *SPI*** de
  R(CodePostal, Ville, Rue) avec
  F={(Ville, Rue) $\rightarrow$CodePostal, CodePostal $\rightarrow$Ville}
  car $(R_1 \cap R_2) \rightarrow R_2 - R_1$ = CodePostal $\rightarrow$ Ville.

- **Mais:** la DF (Ville, Rue) $\rightarrow$ CodePostal ne peut plus être évaluée *efficacement* (sans faire des jointures) : **la décomposition ne préserve pas les dépendances** (SPD).
  => reste à le prouver formellement

Décomposition SPD (2/4)

- **Définition :** Une DF X $\rightarrow$ Y est projetée dans R si R contient XY.

•Remarque : Une DF non-projeté dans les relations d'une décomposition n'est pas forcément « perdue »: elle peut faire partie de la *fermeture transitive* des DF projetées.

Ex: R(ABCD) décomposée en R1(AB), R2(BC), R3(CD)
et F(A $\rightarrow$B, B $\rightarrow$C, C $\rightarrow$ D, D $\rightarrow$ A)

•Il faut considérer F^+ !!!

- **Définition :** Soit F^+_R les DF de F^+ qui se projettent dans R. Une décomposition de R en $\{R_1, R_2, \ldots, R_n\}$ est préserve les dépendances (SPD) dans F si

$$F^+=[F^+_{R1} \cup \ldots \cup F^+_{Rn}]^+$$

Décomposition SPD (3/4)

Théorème : Il suffit de montrer que $F \subseteq [F^+_{R1} \cup \ldots \cup F^+_{Rn}]^+$

Preuve :

- $F^+ \supseteq [F^+_{R1} \cup \ldots \cup F^+_{Rn}]^+$: trivial, car on ne peut pas créer de nouvelles DF lorsqu'on fait une décomposition.
- $F^+ \subseteq [F^+_{R1} \cup \ldots \cup F^+_{Rn}]^+$: il suffit de montrer que toute DF de F peut se déduire des DF projetées : $F \subseteq [F^+_{R1} \cup \ldots \cup F^+_{Rn}]$

Décomposition SPD (4/4)

Entrée: décomposition $D=\{R_1, R_2, \ldots, R_n\}$ et F

Sortie : succès (D est SPD) ou échec

Algorithme : on montre que toutes les DF sont préservées

```
pour toutes les DF X →A ∈ F :
    Z := X
    tant que Z change
        pour toutes les Ri : Z := Z ∪ ([Z∩Ri]+F ∩Ri)
    si A ∉ Z retourne échec
    sinon retourne succès
```

Revient à considérer F+

Exemple : (trivial)

•R(ABCD) et F={A→B, B→C, C→D, D→A}

R_1(AB), R_2(BC), R_3(CD) est SPD car Z={A,C,B, D} pour D→A

Conception de bases de données

- Conception logique
- Dépendances fonctionnelles
- Décomposition de schémas et normalisation
- Formes normales
- Algorithme de décomposition

1e Forme Normale

- **Définition :** Une relation est en première forme normale (1NF ou 1FN en anglais) quand tous les attributs ont des valeurs atomiques.
- En particulier, une relation 1FN ***ne peut avoir*** une valeur d'attribut qui soit un *ensemble de valeurs* ou un *nuplet*.
- Hypothèse de base des SGBD relationnels.
- Dans les SGBD objet, relationnel-objet et XML, cette contrainte est relâchée.

2e Forme Normale

- L'attribut A *dépend complètement* de X si X→A ∈ F est une DF non-redondante à gauche :
  - X'⊂ X implique X' → A ∉ F^+
- A *dépend partiellement* de X sinon

Définition 2FN :

Une relation R est en deuxième forme normale (2FN) par rapport à un ensemble de DF F ssi (si et seulement si) tout attribut de R qui ne fait pas partie d'une **clé** dépend *complètement* de *chaque clé*.

Exemple 2FN

EmpProj(ENO, ENAME, TITLE, PNO, PNAME, RESP)

F = {ENO → (ENAME, TITLE), PNO → PNAME, (ENO, PNO) → RESP}
Une clé : (ENO, PNO)

- **EmpProj n'est pas en 2FN** parce que ENAME, TITLE et PNAME dépendent *partiellement de la clé*
- Ainsi : ENAME et TITLE sont répétés pour chaque produit et PNAME est répété pour chaque employé

Exemple 2FN et redondance

FournisseurProduit(NomF, NomP, Prix, Marque)
F = {NomP → Marque, NomF → (NomP, Prix)}
LA clé : NomF

- **FournisseurProduit est en 2FN :**
  - Aucun attribut non-clé ne peut dépendre partiellement d'une clé avec un attribut

Mais : il y a encore des redondances : la marque d'un produit est répétée pour chaque fournisseur.

3ème forme normale

Définition 3FN :

Un schéma de relation R est en troisième forme normale (3FN) par rapport à un ensemble de DF F, *ssi* pour toute DF *non-triviale* X → A applicable à R, A est *premier* (fait partie d'une clé) ou X est une surclé *de R*.

FournisseurProduit(NomF, Marque, NomP, Prix)
F = {*NomP → Marque*, NomF→ (NomP, Prix)}
Clé : NomF

FournisseurProduit est en 2FN mais pas en 3FN :
Dans *NomP → Marque*,
Marque n'est pas premier et NomP n'est pas une surclé (Marque dépend *transitivement* de la clé)

On peut montrer que toute relation en 3FN est aussi en 2FN...

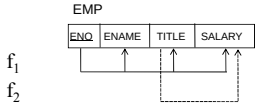
Exemple 3FN + redondance

R(CodePostal, Ville, Rue)
F={(Ville,Rue) → CodePostal, CodePostal →Ville}
Deux clés : (Ville Rue) et (Rue CodePostal)

- **FournisseurProduit est en 3FN :**
  Pas d'attributs non-premier...

Mais il y a encore des redondances : la ville est répétée pour chaque rue.

3FN – Exemple



EMP n'est pas en 3FN à cause de f_2

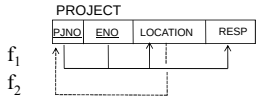
- TITLE → SALARY : TITLE n'est pas une surclé et SALARY n'est pas premier
- le problème est que l'attribut clé ENO détermine transitivement l'attribut SALARY : redondance

Solution : décomposition

- EMP (ENO, ENAME, TITLE)
- PAY (TITLE, SALARY)

Forme normale de Boyce-Codd

Observation : En 3FN, on peut encore avoir des DF transitives où les attributs dépendants sont premiers.



Définition FNBC :
Un schéma de relation R est en forme normale Boyce-Codd (FNBC) par rapport à un ensemble de DF F, si pour toute DF non-triviale X → A (de F^+) applicable à R, X est une surclé de R.

Forme Normale de Boyce-Codd

Propriétés de FNBC :

- Tout attribut non-premier dépend complètement de chaque clé.
- Tout attribut premier dépend complètement de chaque clé à laquelle il n'appartient pas.
- Aucun attribut ne dépend d'attributs non-premiers.

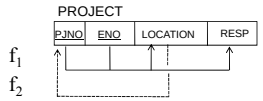
Remarques :

- Toute relation à 2 attributs est FNBC

FNBC = plus aucune redondance dûe aux DF

FNBC – Exemple

- Supposons la relation PROJECT où chaque employé sur un projet a un lieu et une responsabilité unique pour ce projet et il y a un seul projet par lieu



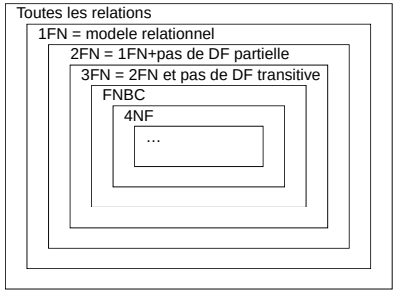
PROJECT est en 3FN mais pas en FNBC

Décomposition :

PR (PJNO, ENO, RESP), PLOC (LOCATION, PJNO)

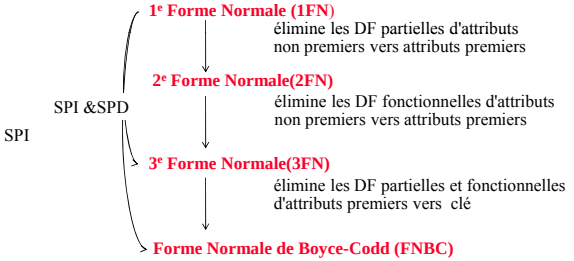
Cette décomposition évite toutes les anomalies mais n'est pas SPD ! (on a perdu la DF (PJNO ENO) → LOCATION)

Formes normales



Formes normales et DF

1FN élimine les attributs non atomiques



(la 4è forme normale implique d'autres contraintes que les DF…)

Conception de bases de données

- Conception logique
- Dépendances fonctionnelles
- Décomposition de schémas et normalisation
- Formes normales
- Algorithme de décomposition

Algorithme de passage en 3FN

Entrée : $R(A_1,A_2,\ldots,A_n)$ et un ensemble minimal de DF F

Sortie : une décomposition SPI et SPD $\{R_1,R_2,\ldots,R_n\}$ où tous les R_i sont en 3FN

Algorithme :

1.Regrouper les DF qui ont même partie gauche

2.Créer un schéma de relation R_i avec tous les attributs de chaque groupe de DF

3.Si aucune clé de R n'apparaît dans un R_i existant, rajouter un schéma de relation formé par les attributs d'une clé de R

4.Éliminer les schémas de relation inclus dans d'autres

Décomposition en 3FN

Exemple : FournisseurProduit(NomF, Marque, NomP, Prix)
avec F = **{ NomP→ Marque**, NomF→ (NomP, Prix) }

1.Regroupement : rien à faire

2.Création de

- Fournisseur(NomF, NomP, Prix) avec F1= NomF→ (NomP, Prix)
- Produit(NomP, Marque) avec F2={NomP → Marque}

3.Création d'une table pour la clé NomF : pas nécessaire

4.Élimination de schémas inclus dans d'autres schémas : rien à faire

Algorithme de passage en FNBC

Entrée : $R(A_1,A_2,\ldots,A_n)$ et F un *ensemble minimal* de DF F

Sortie : une décomposition SPI $\{R_1,R_2,\ldots,R_n\}$ où tous les R_i sont en FNBC (décomposition pas forcément SPD)

Algorithme :

1. S:={R}

2. **tant** qu'il existe un $R_i(Y)$ dans S et une DF non-triviale

$X \rightarrow A$ dans $[R_i]^+_F$ telle que X n'est pas surclé de R_i:

$$S := (S - R_i) \cup R_j(AX) \cup R_k(Y \backslash A)$$

(on décompose Ri en {Rj, Rk})

(noter que c'est bien SPI)

Exemple de décomposition FNBC

Soient

- EMP(ENO, ENAME, TITLE, PNO, PNAME, RESP)
- *F* = {ENO → ENAME , ENO → TITLE => on peut regrouper),
  PNO → PNAME,
  (ENO PNO) → RESP}

- EMP n'est pas en FNBC, car ENO et PNO ne sont pas surclés, donc ENO → (ENAME TITLE) et PNO → PNAME posent problème

Exemple de décomposition FNBC

Commençons avec

D0 = {**EMP(ENO, ENAME, TITLE, PNO, PNAME, RESP)**}

Itération 1

- prendre une des DF qui posent problème : ENO → ENAME TITLE

D1 = {R_1, R_2} où

- **R_1(ENO, PNO, PNAME, RESP)** avec F1 = { PNO → PNAME, (ENO PNO) → RESP}
- **R_2(ENO, ENAME, TITLE)** avec F2 = {ENO → (ENAME TITLE) }

R_2 est en FNBC, mais pas R_1

Exemple de décomposition FNBC

Itération 2

- *D1* contient R_1(ENO, PNO, PNAME, RESP) qui n'est pas en FNBC
- prendre une des DF qui posent problème : PNO → PNAME

D2 = {R_2, R_3, R_4} où

- R_3(ENO, PNO, RESP) avec F1 = { (ENO PNO) → RESP }
- R_4(PNO, PNAME) avec F1 = { PNO → PNAME }

R_2, R_3, R_4 sont alors en FNBC (et on a pu préserver les DF, ce qui n'est pas toujours le cas)

UFR 919 Ingénierie – module LU3IN009
Cours 10 : SQL : contraintes/triggers/vues

- Contraintes d'intégrité (rappel)
- Triggers (rappel)
- Vues

Contraintes d'intégrité

Une *contrainte d'intégrité* est une *condition logique* qui doit être satisfaite par les données stockées dans la BD.

But : maintenir la cohérence/l'intégrité de la BD :

- Vérifier/valider *automatiquement* (en dehors de l'application) les données lors des mises-à-jour (insertion, modification, effacement)
- La cohérence est liée à la notion de transaction

cohérent — Non visible de l'extérieur de la transaction → cohérent
Début trans. — Fin trans.

- Déclencher *automatiquement* des mises-à-jour entre tables pour maintenir la cohérence globale.

Contraintes d'attributs

PRIMARY KEY
- désigne un *ensemble d'attributs* comme la *clé primaire* de la table

FOREIGN KEY
- désigne un *ensemble d'attributs* comme la *clé étrangère* dans une *contrainte d'intégrité référentielle*

NOT NULL
- spécifie qu'un attribut ne peut avoir de valeurs nulles

UNIQUE
- spécifie un *ensemble d'attributs* dont les valeurs doivent être distinctes pour chaque couple de n-uplets.

D'un point de vue logique, pas de différence entre PK et Unique, mais la relation peut être organisée (Arbre-B+) selon la clé primaire en utilisant la directive *organization index* en fin de *create table*.

Exemple : Clés

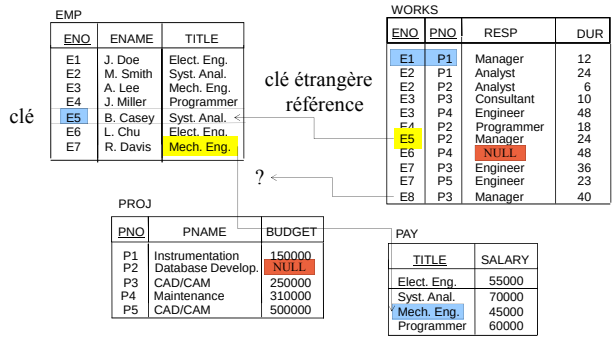
Créer la table Project(<u>Pno</u>, Pname, Budget, City) :

```
CREATE TABLE Project
       (Pno   CHAR(3),
        Pname VARCHAR(20)UNIQUE NOT NULL,
        BudgetDECIMAL(10,2)DEFAULT 0.00,
        City CHAR(9),
        PRIMARY KEY (Pno));
```

ou Pno **CHAR(3) PRIMARY KEY**

Remarque: plusieurs clés possibles (Pno et Pname), 1 seul est primaire
PRIMARY KEY implique UNIQUE et **NOT NULL.**

Exemple clé étrangère



Maintenance automatique de l'intégrité référentielle

La suppression (ON DELETE) ou la mise-à-jour (ON UPDATE) d'un n-uplet référencé (de clé primaire) nécessite une action sur le n-uplet avec la clé étrangère :

- RESTRICT : l'opération est *rejetée* (par défaut)
- CASCADE : supprime ou modifie tous les n-uplets avec la clé étrangère si le n-uplet référencé est supprimé ou *sa clé* est modifiée
- SET [NULL | DEFAULT] : mettre à NULL ou à la valeur par défaut quand le n-uplet référencé est effacé/sa clé est modifiée. L'attribut doit accepter la valeur NULL

Pas de *on update cascade* en Oracle => trigger

Contraintes de n-uplets

Contraintes portant ***sur une seule table***.

(d'autres tables peuvent apparaître dans des sous-requêtes)

La condition est vérifiée *chaque fois* qu'un **n-uplet** est inséré ou modifié dans la table; la mise-à jour (transaction) est rejetée si la condition est fausse.

```
CREATE TABLE Works
  (Eno      CHAR(3),
   Pno      CHAR(3),
   Resp     CHAR(15),
   Dur      INT,
PRIMARY KEY (Eno,Pno),
FOREIGN KEY (Eno) REFERENCES Emp(Eno),
FOREIGN KEY (Pno) REFERENCES Project(Pno),
CHECK (NOT(PNO<'P5') OR Dur>18));
```


Contraintes de tables : Assertions

Contraintes ***globales sur plusieurs relations***.

```
CREATE ASSERTION name CHECK (condition)
```

Exemple: le salaire total des employés du projet P5 ne peut dépasser 500

```
CREATE ASSERTION salary-control CHECK
  (500 >= (SELECT     SUM(Salary)
           FROM       Emp E, Pay P, Works W
           WHERE      W.Pno = 'P5'
           AND        W.Eno = E.Eno
           AND        E.Title = P.Title));
```

Remarque : pas disponible dans les SGBD actuels => triggers
Remarque : vérification des contraintes = problème d'efficacité

SQL : contraintes

- Contraintes d'intégrité
- Triggers
- vues

Triggers : utilisation

« Règles actives » (ECA) *généralisant* les contraintes d'intégrité **:**

- génération automatique de valeurs manquantes (ex. valeur dérivée, par défaut)
- éviter des modifications invalides (C: test, A: abort)
- implantation de règles applicatives (« business rules »)
- génération de traces d'exécution, statistiques, ...
- maintenance de répliques
- propagation de mises-à-jour sur des vues vers les tables
- intégrité référentielle entre des données distribuées
- interception d'événements utilisateur / système (LOGIN, STARTUP, …)

Trigger ou règle active ou règle ECA

Définition ECA :

Événement (E) :

- une mise-à-jour de la BD qui active le trigger, ou d'autres (opérations DDL, logon, serverror, ..)

ex.: réservation de place

Condition (C):

- un test ou une requête devant être vérifié lorsque le trigger est activé (une requêtes est *vraie* si sa réponse n'est pas vide)

ex.: nombre de places disponibles ?

Action (A):

- une procédure exécutée lorsque le trigger est activé et la condition est *vraie* : E,C → A

ex.: annulation de réservation

Exécution des triggers (1)

Moment de déclenchement du trigger par rapport à l'événement *E* (maj. activante) :

- avant (before) E
- après (after) *E*
- à la place de (instead of) de *E* (spécifique aux vues => màj des données de la base)

Nombre d'exécutions de l'action A par déclenchement :

- une exécution de l'action *A* par n-uplet modifié (ROW TRIGGER)
- une exécution de l'action *A* par événement (STATEMENT TRIGGER)

Exécution des triggers (2)

Delta structure : « les données considérées par le trigger »

- *:old* avant l'événement, *:new* après l'événement (peuvent être renommés)
- *for each row* : un n-uplet, *for each statement*: un ensemble de n-uplets
- *:new* peut être modifié par l'action, mais effet seulement si *before*
- Pour agir avec un trigger *after*, il faut modifier directement la base
- *:old* (resp. *:new*) n'a pas de sens pour *insert* (resp. *delete*)

Syntaxe (Oracle)

```
{ CREATE | REPLACE } TRIGGER <nom>
// Événement
{BEFORE | AFTER | INSTEAD OF}
{INSERT | DELETE | UPDATE [OF <attribut, …>]}
    ON <table>
[REFERENCING [NEW AS <nouv>] [OLD AS <anc>]]
[FOR EACH ROW] // ROW TRIGGER
// Condition
[WHEN  (<condition SQL>) THEN]
// Action
<Procédure SQL>
```


Contrôle d'intégrité

Emp (Eno, Ename, Title, City)

Vérification de la contrainte de clé à l'insertion d'un nouvel employé :

```
CREATE TRIGGER InsertEmp
BEFORE INSERT ON Emp
REFERENCING NEW AS N
FOR EACH ROW
WHEN EXISTS
     (SELECT * FROM Emp WHERE Eno=N.Eno)
THEN
    ABORT;
```


Contrôle d'intégrité

Emp (Eno, Ename, Title, City) **Pay**(Title, Salary)

Suppression d'un titre et des employés correspondants (« ON DELETE CASCADE ») :

```
CREATE TRIGGER DeleteTitle
BEFORE DELETE ON Pay
REFERENCING OLD AS O
FOR EACH ROW
BEGIN
    DELETE FROM Emp WHERE Title=O.Title
END;
```

Mise-à-jour automatique

Emp (Eno, Ename, Title, City)

Création automatique d'une valeur de clé (autoincrément) :

```
CREATE TRIGGER SetEmpKey
BEFORE INSERT ON Emp
REFERENCING NEW AS N
FOR EACH ROW
BEGIN
  N.Eno := SELECT COUNT(*) FROM Emp
END;

/* le premier Eno sera 0 */
```


Mise-à-jour automatique

Pay(Title, Salary, Raise)

Maintenance des augmentations (raise) de salaire :

```
CREATE TRIGGER UpdateRaise
AFTER UPDATE OF Salary ON Pay
REFERENCING OLD AS O, NEW AS N
FOR EACH ROW
BEGIN
     UPDATE Pay
     SET Raise = N.Salary - O.Salary
     WHERE Title = N.Title;
END
```


Analyse des triggers

Plusieurs triggers de type différent peuvent être affectés au même événement :

- Ordre par défaut : BEFORE STATEMENT → BEFORE ROW → AFTER ROW → AFTER STATEMENT

Un trigger activé peut en activer un autre :

- longues chaînes d'activation => problème de performances
- boucles d'activation => problème de terminaison

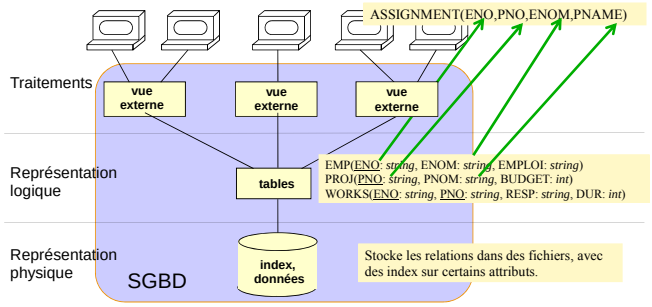
Recommandations :

- pour l'intégrité, utiliser si possible le mécanisme des contraintes plus facile à optimiser par le système.
- associer les triggers à des règles de gestion.

SQL : contraintes

- Contraintes d'intégrité
- Triggers
- Vues

Architecture ANSI/SPARC



Pourquoi définir des vues

Une BD peut contenir des *centaines de tables avec des milliers d'attributs* :

1. Les requêtes sont complexes :
   - difficiles à formuler
   - Ne portent que sur un sous-ensemble des attributs
   - source d'erreurs
2. Une modification du schéma nécessite la modification de beaucoup de programmes.

Solution : *Adapter* le schéma et les données à des applications spécifiques → vues

Définition d'une vue

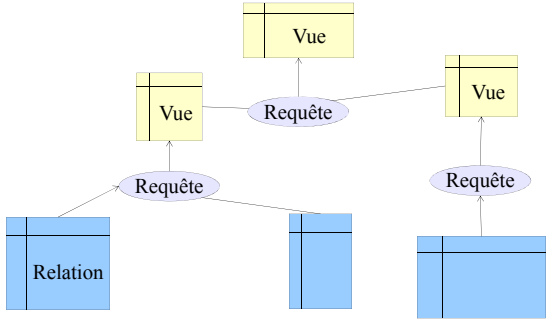
Définition : Une vue **V(a1, a2, … an)** est une relation avec n attributs qui contient le résultat d'une requête Q(x1, x2, …xn) évaluée sur une base de données BD :

V(a1, a2, …an) :- Q(x1, x2, …,xn, BD)

Remarques :

- V possède un schéma relationnel avec des attributs a1,…an.
- V *reflète* l'état actuel d'une base de données BD
- V peut être interrogée et il est possible de définir des vues à partir d'autres vues.
- On distingue les relations « matérialisées » (tables) et les relations « virtuelles » (vues)

Relations et Vues



Définition d'une vue dans SQL

CREATE VIEW *nom_vue* [(*att1*, *att2*...)]
AS *requête_SQL* [**WITH CHECK OPTION**]

- *nom_vue* désigne le nom de la relation
- *att1*, ... (optionnel) permet de nommer les attributs de la vue (attributs de la requête par défaut)
- *requête_SQL* désigne une requête SQL standard qui définit le « contenu » (instance) de la vue
- **WITH CHECK OPTION** (voir mises-à-jour de vues)

Exemple

Emp (Eno, Ename, Title, City)
Pay(Title, Salary)
Project(Pno, Pname, Budget, City)
Works(Eno, Pno, Resp, Dur)

Définition de la vue **EmpProjetsParis** des employés travaillant dans des projets à Paris :

```
CREATE VIEW EmpProjetsParis(NumE, NomE, NumP, NomP, Dur)
AS SELECT Emp.Eno, Ename, Works.Pno, Pname, Dur
   FROM Emp, Works, Project
   WHERE Emp.Eno=Works.Eno
   AND Works.Pno = Project.Pno
   AND Project.City = 'Paris'
```


Interrogation de vues

Emp (Eno, Ename, Title, City)
Pay(Title, Salary)
Project(Pno, Pname, Budget, City)
Works(Eno, Pno, Resp, Dur)

Les noms des employés de projets Parisiens :

Requête sans vue:

```
SELECT Ename
FROM Emp, Works, Project
WHERE Emp.Eno=Works.Eno
AND Works.Pno = Project.Pno
AND Project.City = 'Paris'
```

Requête avec vue:

```
SELECT NomE
FROM EmpProjetsParis
```

On obtient le même résultat

Évaluation de requêtes sur des vues

Vue :

```
CREATE VIEW EmpProjetsParis
AS SELECT Emp.Eno, Ename, Project.Pno,
Pname, Dur
FROM Emp, Works, Project
WHERE Emp.Eno=Works.Eno
AND Works.Pno = Project.Pno
AND Project.City = 'Paris'
```

Requête :

```
SELECT Emp.Eno FROM EmpProjetsParis
WHERE Dur > 3
```

Mise-à-jour de vues

Problème de mise-à-jour : une vue est une relation virtuelle et toutes les modifications de cette relation doivent être "transmises" aux relations (tables) utilisées dans sa définition.

La plupart du temps il n'est pas possible de mettre à jour une vue (insérer un n-uplet, …).

Exemple:

CREATE VIEW V AS SELECT A,C
FROM R,S WHERE R.B = S.B

- Insertion d'un n-uplet [A:1,C:3] dans la vue V
- Quelle est la modification à faire dans R et S (valeur de B) ?

Vues modifiables (dépend beaucoup du sgbd)

Une vue *n'est pas modifiable* en général:

- quand elle ne contient pas tous les attributs définis comme NON NULL dans la table interrogée (cas de l'insert)
- quand elle contient une jointure
- quand elle contient une fonction agrégat

Règle : Une vue *est modifiable* quand elle est définie comme une sélection/projection sur une relation R (qui peut aussi être une vue modifiable) sans utilisation de SELECT DISTINCT ni de group by.

Lorsqu'une vue n'est pas modifiable, on peut créer un trigger *instead of*, qui va exécuter les répercussions sur les tables de la base de la mise à jour sur la vue au lieu de le faire sur la vue (attention, dans un trigger *instead of*, on ne peut pas préciser l'attribut d'un update, donc *update on* et non pas *update of Att on*)

Mises-à-jour

Emp (Eno, Ename, Title, City) **Project**(Pno, Pname, Budget, City)
Pay(Title, Salary) **Works**(Eno, Pno, Resp, Dur)

```
CREATE VIEW ProjetParis
AS SELECT Pno,Pname,Budget
      FROM Project
      WHERE City='Paris';

UPDATE ProjetParis
SET Budget = Budget*1.2;
```


WITH CHECK OPTION

WITH CHECK OPTION protège contre les « disparitions de n-uplets » causées par des mise-à-jour :

```
CREATE VIEW ProjetParis
WITH CHECK OPTION
AS SELECT Pno,Pname,Budget,City
      FROM Project
WHERE City='Paris';
```

```
UPDATE ProjetParis
SET City = 'Lyon'
WHERE Pno=142;
```

← Mise-à-jour rejetée

Vues et tables

Similitudes :

- Interrogation SQL
- UPDATE, INSERT et DELETE sur vues modifiables
- Autorisations d'accès
- Evaluation et optimisation

Différences:

- On ne peut pas créer des index sur les vues
- On ne peut pas définir des contraintes (clés)
- Une vue est recalculée à chaque fois qu'on l'interroge
- *Vue matérialisée* : stocker temporairement la *vue* pour améliorer les performances. => pb de performance si les tables sont mises à jour
  - rafraichissement des vues incrémental : détecter quelles vues matérialisées doivent être rafraichies après une mise à jour

UFR 919 Ingénierie – module LU3IN009
Cours 11 : Ouverture sur les cours de M1

+ de modèles : MLBDA
+ de système et réparti : BDR

Objectifs MLBDA

Présenter des modèles et langages pour le développement de nouveaux types d'applications (Web 2.0, réseaux sociaux, réseaux de capteurs, open data, recherche d'information, …).

Apprendre les technologies récentes de gestion de données (XML, JSON, RDF, SPARQL…)

Apports du modèle relationnel

- Simplicité des concepts et du schéma
- Bon support théorique
- Langage d'interrogation déclaratif
- Haut degré d'indépendance des données
- Optimisation des accès à la BD
  - bonnes performances
- Gestion de contraintes d'intégrité

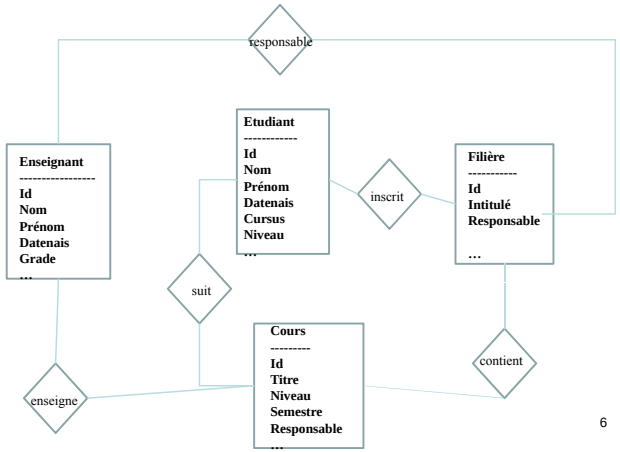
Limites du modèle relationnel

- Trop grande simplicité du modèle de données
  - 1ère forme normale de Codd
    - attributs mono-valués : n-uplets plats
  - Pauvreté du système de typage
    - Types prédéfinis (entier, réel, chaîne, …) : pas de possibilité d'extension
  - Inadapté aux objets complexes (ex: documents structurés)
    - Un objet du monde réel est modélisé à l'aide de plusieurs relations : mauvaise lisibilité, perte d'information sémantique, nombreuses jointures
- Très bon support pour les applications de gestion, mais mal adapté pour d'autres types d'applications
  - CAO, CFAO
  - BD Géographiques
  - BD techniques, documentation
  - …

Limites du modèle relationnel

- Langage d'interrogation et de manipulation non complet
  - Pas de récursion
  - Pas de structures de contrôle : conditionnelles, boucles
  - L'utilisation de deux langages (SQL + un langage de programmation) provoque un dysfonctionnement du système
    - Sql déclaratif, LP procédural
    - Systèmes de typage différent
    - Espaces de noms différents
    - Utilisation de curseurs pour manipuler les ensembles
    - Mauvaises performances
- Ensemble fermé d'opérateurs : algèbre relationnelle
  - Critères d'optimisation liés à ces opérateurs
- Index restreints aux types de base
- Pas de versions, pas de transactions longues

Données complexes



Constat

- Multiplication des relations, nombreuses jointures
- Eclatement des entités, informations dispersées dans plusieurs relations
- Informations redondantes, non factorisées (étudiant, enseignant)
- Difficile de représenter simplement les objets complexes (constitués d'autres objets et/ou d'ensembles)

⇨

- Modèle de données plus riche
- Conception plus proche du monde réel

Et aussi

- Gestion de gros objets (données multimedia) avec structures de stockage adaptées
- Nouveaux modèles de transactions (transactions longues, distribuées, imbriquées..)
- Prise en compte des versions
- Indépendance des objets et des traitements
- Extensibilité
- Meilleure intégration des langages d'interrogation et de manipulation

Approches

- Extensions du modèle relationnel
- Langages de programmation persistants
- Systèmes orienté-objet
  - Modèle objet : ODMG et OQL
  - Modèle relationnel-objet : SQL3

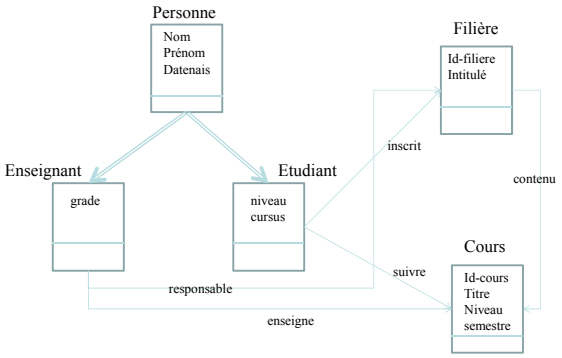
9

Modélisation en objets

- Les entités du monde réel sont modélisées par des objets.
- Un objet peut être atomique, ou composé d'autres objets.
- Chaque objet a un identificateur unique.
- Un objet a une interface et des opérations qui lui sont applicables.
- Les objets de même nature sont regroupés dans des classes, reliées par des liens de composition et/ou d'héritage.
- Un schéma de base de données objet est un graphe de classes.

10

Exemple



11

Données du Web

- Des données très hétérogènes
  - Documents HTML, SGML, Latex, PDF, txt, …
  - Multimédia : son, graphique, images, vidéos, photos, dessins, etc.
- Des applications très diverses
  - Commerce électronique
  - Portail d'information
  - Intranet
  - Publication en ligne
- Toutes les catégories d'utilisateurs

12

Constat

- Des informations différentes pour décrire des données similaires
- Des présentations différentes, mais on retrouve une certaine régularité
- Des modifications fréquentes, une évolution rapide.

13

Caractéristiques des données du Web

- Structure irrégulière :
  nécessité de modéliser et d'interroger ces structures
- Structure implicite :
  ex: un document électronique a un texte et une grammaire. On peut parser pour détecter des informations et des relations entre ces informations. Mais cela nécessite des outils spéciaux.
- Structure partielle :
  il manque des informations, certaines informations sont stockées hors de la base, et ne sont pas structurées.
- Structure indicative (et non pas contrainte, comme dans les BD) :
  pas de typage, pas de schéma (trop contraignants)

14

Caractéristiques des données du Web

- Un schéma a posteriori, et non a priori :
  en semi-structuré, le schéma est souvent postérieur aux données (c'est l'inverse en BD). Mais parfois, on peut suggérer ou guider (ex: pages personnelles).
- Un schéma très vaste (on ne sait pas tout),
  => il faut pouvoir l'interroger
- Un schéma ignoré (on parcourt tout à la recherche d'une chaîne)
  => pas possible en SQL, il faut trouver d'autres langages
- Un schéma qui évolue rapidement
  => à prendre en compte dans le langage de requêtes
- La distinction entre schéma et données est peu claire.
- Le typage est flou.

15

Données semi-structurées

- Les données semi-structurées sont
  - sans schéma
  - autodescriptives, càd.
    - pas de séparation entre données et types
    - une donnée contient son propre type
    - elles peuvent être interprétées indépendamment de toute autre information.

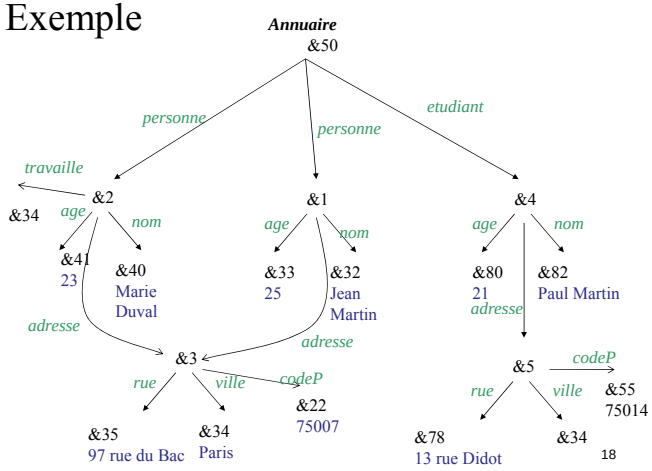
16

Modèles semi-structurés

- Graphes dont les nœuds sont des objets et dont les arcs sont étiquetés.
- Les données sont stockées dans les feuilles (objets atomiques).
- Les étiquettes donnent des informations sur le schéma.

17

Exemple



18

XML

- XML : représentation des données
- Xschema : description du schéma
- Xpath : navigation dans l'arbre XML

/A//B/@att1 : on cherche les A sous la racine, puis les B descendants de ces A, puis l'attribut att1 de ces B

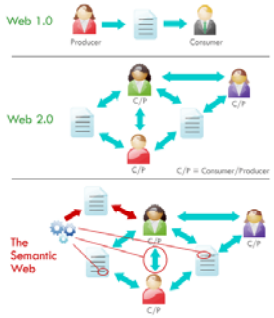
//C : tous les élements C rencontrés

- Xquery : interrogation

```
for $x in document("bib.xml")//book
where $x/author/last = " Ullman "
return $x/title
```

19

Le Web sémantique



Rendre le contenu des ressources du Web plus accessibles et plus utilisables.

Exploiter sémantiquement les données.

20

Applications

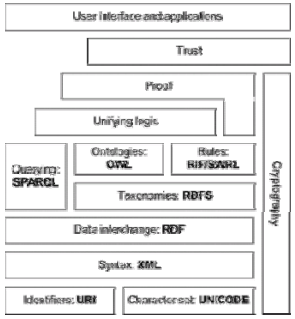
21

Besoins

- Représenter la sémantique
- Établir des liens entre les données
- Exploiter ces liens
  - Déduire de nouvelles informations
- Indexer et interroger

22

Les standards du Web sémantique



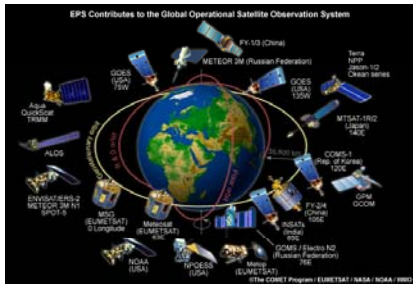
23

Modèle de données sémantique

- RDF (Resource Description Framework) : description des données, annotations sémantiques
- Déclaration RDF : triplet (sujet, prédicat, objet) reliant une ressource à une propriété et une valeur.
  Ex (Picasso,peint,Guernica). Graphe de concepts
- RDFS : description des ontologies
- SPARQL : interrogation des données RDF

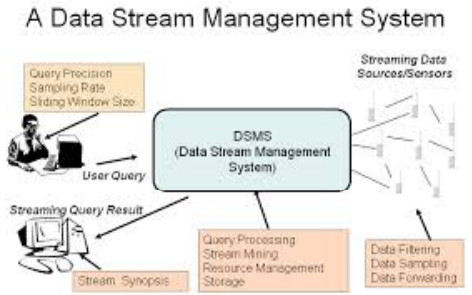
24

Flux de données



25

Gestionnaire de flux de données



26

Caractéristiques

- Très grosses quantités de données, très évolutives
- Données volatiles
- L'ordre d'arrivée des données est important
- Taille mémoire limitée

Comment interroger ?

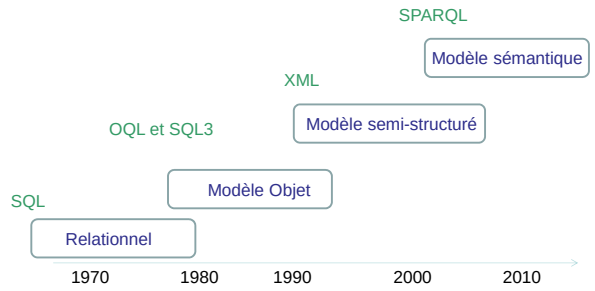
27

Requêtes continues

- Interrogation par échantillonage
- Interrogation par fenêtrage
  - Les n derniers éléments du flux
  - Les éléments des n dernières secondes
- Mise à jour incrémentale des résultats en temps réel

28

Conclusion



29

Conclusion

- Evolution des modèles en fonction du type des données
- Rester proche du 'monde réel'
- Evolution des langages
- Adaptation des techniques des bases de données

30

Objectifs BDR

- Aller plus loin sur l'optimisation des requêtes
- Etudier les problèmes lorsque les données sont réparties
  - BD réparties
  - BD fédérées
  - BD parallèles
  - Autres (P2P, …)
  - Transactions réparties

31

Bases de données réparties

- Système d'une grosse institution
- Répartition intégrée à la définition de la BD
  - Favoriser la localité
  - Gérer la réplication
- Hétérogénéité possible
  - Puissance des serveurs
  - Capacité du SGBD
  - Débit réseau

32

Bases de données réparties

- Conception de la BD
  - Les données doivent être au plus près des requêtes qui les demandent
  - Réplication synchrone vs asynchrone
  - Configuration des répliques :
    - Gain lecture
    - Coût de maintenance en écriture
- Traitement des requêtes
  - Encore plus complexe qu'en client serveur
  - Coût de transfert réseau >> accès disk

33

Bases de données réparties

Exemple :

- R(A,B,C) sur S1, S(B,D) sur S2, S1 demande R⋈S
- Stratégies :
  - Transférer R sur S2, jointure sur S2 et transfert du résultat
  - Transférer S sur S1, jointure sur S1

  Dans les deux cas, on transfère toute une relation…
  - Semi-jointure :
    - Transférer T1 = $\Pi_B(R)$ vers S2, calcul de T2 = T1 ⋈ S sur S2
    - Transférer T2 sur S1 et calcul de T2 ⋈ R = S ⋈ R

34

Bases de données fédérées / médiateurs

- Concevoir une BD répartie à partir de BD existantes
- Pb d'intégration de données
  - Identifier les éléments communs
  - Résoudre les conflits de noms, de types
  - Garantir l'unicité des clés
  - Etablir des fonctions de conversions
  - Connaitre les capacités des sites
    - Machine
    - Algorithmes
    - …

35

Bases de données parallèles

- Utiliser des machines parallèles (ex. cluster) pour accélérer les longs traitements
  - Beaucoup de données : parallélisme intra opérateurs
  - Requêtes complexe : parallélisme inter opérateurs
- Influence de l'architecture :
  - Shared memory / shared disk / shared nothing
- Différence avec BD réparties
  - Pas de localité
  - Les données sont allouées aux nœuds pour une requête :
    - Round robin
    - Vecteur de partitionnement / hachage
- Hadoop / Map Reduce

36

Transactions réparties

- Transactions qui s'exécute sur des données de plusieurs sites = (sous-transactions)
- ACID plus complexe à assurer
  - A : toutes les sous transactions doivent valider ou toutes doivent abandonner
  - C : pas uniquement CI (ex. cohérence mutuelle des répliques)
  - I : graphe de précédence globale, verrouillage réparti, synchroniser les horloges
- Interblocages répartis difficiles à détecter

37