Optimisation et performance

dans les bases de données relationnelles

Bernard ESPINASSE
Professeur à Aix-Marseille Université (AMU)
Ecole Polytechnique Universitaire de Marseille

Mars 2013

1. Evaluation de l'activité sur la base de données
2. Optimisation physique
3. Optimiseurs de requêtes
4. Optimisation logique : dénormalisation

Plan

1. Evaluation de l'activité sur la base de données

2. Optimisation physique

- Performance et structures de données : choix d'une organisation/méthode d'accès
- Compression des données
- Index secondaires
- Partitionnement, clustérisation

3. Optimiseurs de requêtes

- Optimiseurs de requêtes
- Statistiques sur l'état des tables

4. Optimisation logique : dénormalisation

- Problématique de la dénormalisation
- Regroupement de tables, Création de redondances d'attribut non clé, Ajout de clé étrangère redondante, Création de transitivité, Création de tables de jointure

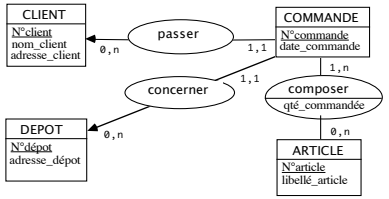
1. Valorisation de l'activité des traitements sur la BD

- **Transformation de sous-schémas conceptuels (SSC) en sous-schémas logiques (SSL)**
- **Valorisation des primitives logiques**

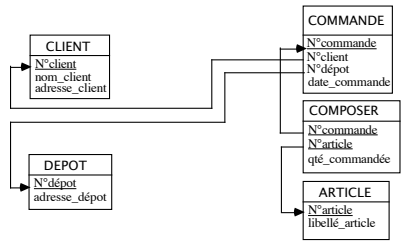
Valorisation de l'activité des traitements sur la BD (1)

1. Transformation de sous-schémas conceptuels (SSC) en sous-schémas logiques (SSL)

Sous-Schéma Conceptuel (SSC)



Sous-Schéma Logique (SSL) associé



Valorisation de l'activité des traitements sur la BD (2)

2. Fréquence d'une tâche associée à un modèle externe

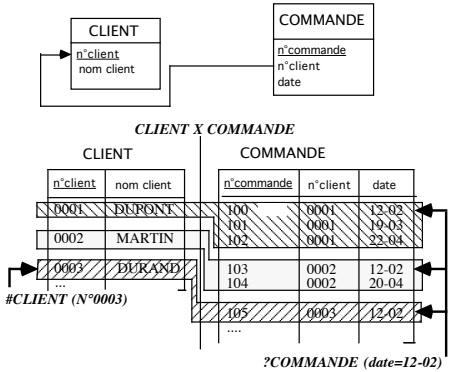
- fréquence du message associé au modèle externe exprimée par rapport à la période de référence (généralement le mois moyen)
- basée sur la fréquence de survenance des événements initiateurs des procédures où figure cette tâche

3. Transformation des actions en primitives

- **#T** : Accès à un tuple d'une table relationnelle par accés sur clé primaire, connaissant une valeur de cette clé
- **?T** : Accès à un tuple d'une table relationnelle par accès sur un autre attribut que la clé primaire, ou sélection/restriction de tuples d'une table selon une qualification
- **T+** : Ajout d'un tuple à une table relationnelle
- **T-** : Suppression d'un tuple à une table relationnelle. Cette primitive est notée T- nom de la table.
- **Tm** : Modification d'un tuple d'une table relationnelle, par la modification de valeurs d'un ou plusieurs attributs de ce tuple
- **TXT'**: Jointure entre deux tables relationnelles T et T', par la modification de valeurs d'un ou plusieurs attributs de ce tuple. Cette primitive est notée TXT' noms des tables concernées .

Valorisation de l'activité des traitements sur la BD (3)

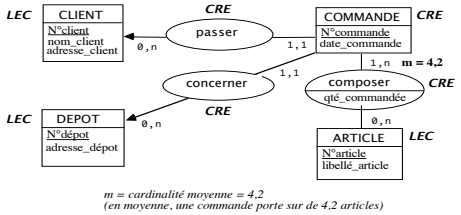
Illustration des primitives relationnelles :



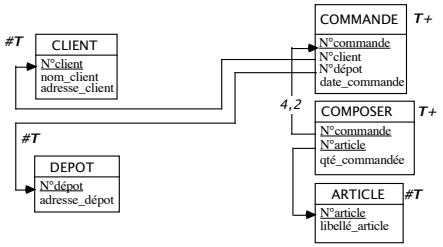
Valorisation de l'activité des traitements sur la BD (4)

- **actions associées à la prise en compte d'une nouvelle occurrence de COMMANDE :**

Sous-Schéma Conceptuel (SSC)



Sous-Schéma Logique (SSL) associé



Valorisation de l'activité des traitements sur la BD (4)

4. Récapitulatif de l'activité valorisée par primitive

- pour chaque modèle externe : traduction des actions, en primitives
- effectuer ce calcul d'abord pour l'activité unitaire de la tâche, puis de la multiplier par la fréquence de la tâche sur la période de référence.
- cumul par primitive de l'activité supportée par le modèle (total par ligne). Du fait de l'hétérogénéité des primitives en termes d'équivalent-accès, seul le cumul par primitive a une signification :.

primitives	activité unitaire	activité mensuelle
#T CLIENT	**1**	**10 000**
#T DEPOT	**1**	**10 000**
T+ COMMANDE	**1**	**10 000**
T# ARTICLE	**1x4,2**	**42 000**
T+ COMPOSER	**1x4,2**	**42 000**

→ repérer les primitives à forte activité, et retrouver les modèles externes responsables

- **en pratique:**

en l'absence d'outil logiciel prenant en charge cette valorisation fortement algorithmique, pratiquer ces calculs que sur un échantillon d'une vingtaine de modèles externes remarquables par leur fréquence, d'aspect critique (comme dans d'autres domaines la loi des 80-20 se constate également ici).

Valorisation de l'activité des traitements sur la BD (5)

5. Poids relatif des primitives en accès logiques

Définir des **équivalents-accès-logiques** = poids relatifs entre les différentes primitives.

Conventions pour la valorisation des primitives logiques

1- unité d'accés logique = #T, toutes les autres seront exprimées en fonction de #T.
2- le nombre de tuples occupées par une table T sera noté **NbT(T)**
3- le nombre de pages occupées par une table T sera noté **NbP(T)**
avec :
- NbP(T) = NbT(T)/nombre moyen de tuples dans la page
- le nombre moyen de tuples dans la page dépendra :
  - de la taille de la page, noté P
  - de la taille du tuple, noté T
  - du taux de remplissage de la page
- ces valeurs seront liées au SGBD et à l'organisation de la table adoptés.
- on pourra faire des calculs en première approximation avec un taux de remplissage de 80% et des pages de 4K.

Exemple :

NbT(T) = 10000; P = 4Koctets; taux de remplissage de 80%; T = 200 octets;
Nombre moyen de tuples par page = 4000x0,8/200 = 16
NbP(T) = 10000/16 = 625 pages.

Valorisation des primitives logiques (1)

- valeurs **moyennes** pour SGBDR actuellement commercialisés, une réflexion nécéssaire sur la pertinence de ces valeurs dans le contexte de la BD

#T =1 : sélection selon clé primaire = unité d'accés logique

?T= NbP(T) : sélection selon un attribut non clé primaire sans index secondaire : balayage séquentiel

?T= Sx(NbP(T)+NP(I)): si le critère de sélection comprend plusieurs termes dont un faisant intervenir un index secondaire I *(avec S = facteur de sélectivité du critère, cad la probabilité qu'un tuple satisfasse le critère; NbP(T) = nb pages de la table T; NbP(I) = nb pages de l'index I)*

T+ = T- = 4: le tuple est atteint par 1 accès à l'index + 1 accès à la page contenant le tuple cherché + 1 mise à jour d'une page + 1 mise à jour de l'index.

Tm = 3: le tuple est atteint par 1 accès à l'index + 1 accès à la page contenant le tuple cherché + 1 écriture sur page.

TXT' = S(NbP(T) + (N(T)xNbP(T'))) : jointure, *avec NbP(T) = nombre page de la table T, S = facteur de sélectivité du critère sur T, N(T) = nombre de tuples de T satisfaisant le critère, NbAP(T') = nombre page de la table T'*

Valorisation des primitives logiques (2)

Globalement on pourrait retenir :

primitive	équivalent accès
#T	1
Tm	3
T+=T-	4
?T	NbP(T)
?T si critère/index	Sx(NbP(T)+NP(I))
TXT'	S(NbP(T) + (N(T)xNbP(T'))

2. Optimisation physique d'une BD relationnelle

- **Performance et structures de stockage des données**
- **Conseils pour le choix d'organisation de stockage**
- **Compression**
- **Clé primaire et clé plaçantes**
- **Index secondaires**
- **Partition de tables (vertical et horizontal)**
- **Clustérisation de tables**

Performance et structures de stockage des données

Problème :

le SGBD permet toujours de trouver une donnée mais un **choix judicieux de structure de stockage** (organisation des tables) permettra de réduire ce temps de recherche.

→ pour certaines requêtes : certaines organisations seront plus performantes

mais :

→ ces organisations affecteront :

- l'espace disque
- la concurrence (encore plus délicate à gérer).

Certains SGBD proposent plusieurs organisations

dans INGRES, une table peut être organisée en :

- **fichier séquentiel (heap)**
  - organisation par défaut : INDEX PRINCIPAL (clé primaire de placement)
- **hash (clé calculée)**
  - accès sur une valeur exacte de la clé
  - la fonction de randonisation ne peut être modifiée.
- **ISAM**
  - les données sont triées selon une clé
  - permet un accès rapide sur valeur exacte ou partie de clé
  - index statique: nécessite réorganisation qd table augmente
- **BTREE**
  - données triées selon une clé
  - accès rapide sur valeur exacte ou partie de clé
  - index dynamique

On peut changer l'organisation d'une table à tout moment :

Modify emp **To** heap
Modify emp **To** hash **On** name
Modify emp **To** btree **On** name, age

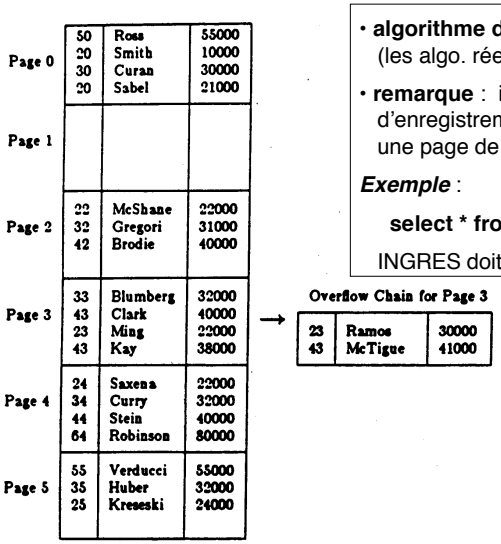
Structures de stockage : comparatif produits

produit	chemin d'accés	méthode d'accés
DB2 SQL/DS	index	VSAM,
INGRES	index	hash, ISAM, cheap, Btree,...
ORACLE	index+proximité	VSAM sous OS IBM Sinon hash, ISAM, cheap, Btree,...
UNIFY	index + liens explicites	hash, ISAM, Btree, ...
DATACOM-DB	index	hash, Btree,
…	…	…

Heap : présentation simplifiée

Page 1	Kreseski	27	24000 ...
	Ramos	33	30000
	Saxena	25	22000
	Clark	43	40000
Page 2	Aitken	53	50000 ...
	Blumberg	35	32000
	McShane	25	22000
	Mandic	46	43000
Page 3	Stannich	36	33000 ...
	Stein	43	40000
	Stover	38	35000
	Zimmerman	28	25000
Page 4	McTigue	44	41000 ...
	Kay	41	38000
	Cameron	38	35000
	Brodie	43	40000
Page 5	Curran	33	30000 ...
	Greeen	29	26000
	Huber	35	32000
	Gregori	34	31000
Page 6	Curry	35	32000 ...
	Ross	58	55000
	Verducci	58	55000
	Shigio	31	28000
Page 7	Sabel	24	21000 ...
	Robinson	64	80000
	Sullivan	38	35000
	Smith	13	10000
Page 8	Ming	25	23000 ...
	Giller	49	46000
	Gordon	30	27000

Hash : présentation simplifiée



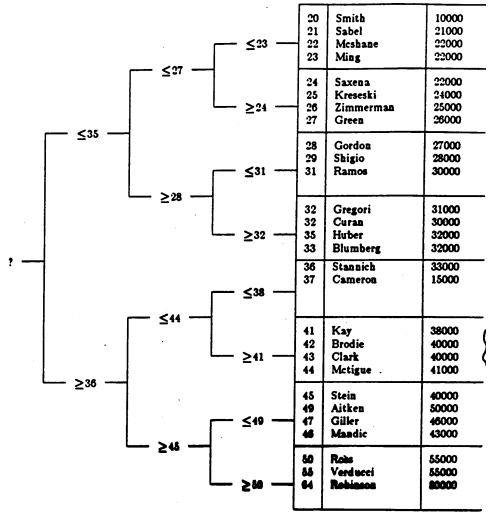
- **algorithme de hachage** : page = age mod 10 (les algo. réels sont bien plus complexes)
- **remarque** : ici la page 1 n'a pas d'enregistrement ; la page 3 est pleine et a une page de débordement

Exemple :

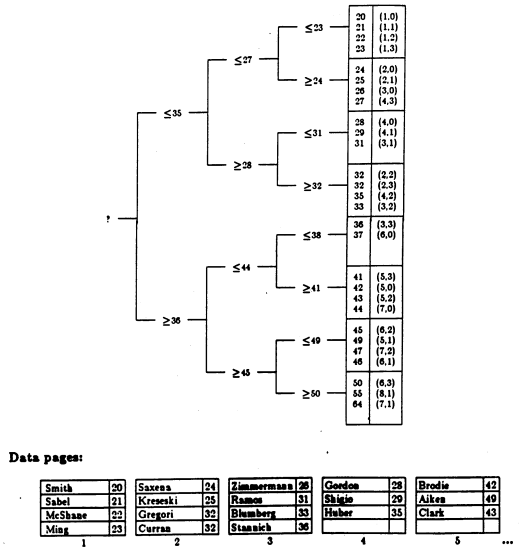
select * from emp where emp.age = 43 :

INGRES doit chercher dans la page 3 et sa

ISAM : présentation simplifiée



Btree : présentation simplifiée



Conseils pour le choix d'organisation de stockage

- **séquentiel (HEAP)**
  pour :
  - le chargement de données dans la base
  - petite table (occupation de peu de pages)
  - requêtes manipulant des tables entières
  - pour recupérer de l'espace dû à des DELETE

  Modify emp **To** heap
  contre
  - accès à 1 ou plusieurs tuples
  - grosses tables

- **HASH**
  pour :
  - recherche sur valeur exacte de clé (le plus rapide)

  contre :
  - recherche sur pattern matching (partie de clé)
  - traitement de table entière
  - joints naturels (systématique sans restriction)
  - débordements dès le départ
    (on ne peut ajuster la fonction de randomisation)

Conseils pour le choix d'organisation

- **ISAM:**
  - pour
    - requêtes nécessitant pattern matching ++
    - la table grossit lentement (peu de réorg.)
    - clé large
  - contre
    - si recherche sur clé complète (->HASH)
    - grosse table à croissance rapide
- **BTREE:**
  - *pour*
    - besoin de pattern matching +
    - la table grossit vite
    - table trop grosse pour être souvent réorganisée (Modify)
    - joints de tables entières
  - *contre*
    - table statique ou à croissance faible
    - large clé
    - si ajout de nouveaux tuples seulement en fin de table (plus grand risque de DEAD LOCK par utilisateurs)

Conseils pour le choix d'organisation

heap	hash	ISAM	Btree	meilleur pour
1	-	-	2	chargement de table
-	1	1	1	effacer tuples doubles
-	1	2	2	recherche sur clé complète
-	-	1	1	intervalles/pattern matching
1	3	2	2	recherches séquentielles
-	-	1	1	recherche sur clé partielle
-	-	-	1	accés à données triées
-	-	1	1	joints sur larges tables
-	-	-	1	index croît comme table
1	-	-	-	très petite table
-	-	-	1	très grande table

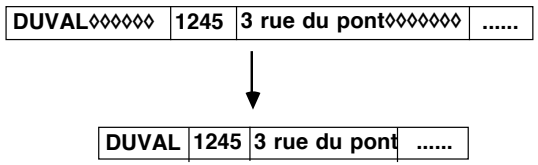
(1): préféré;
(2): acceptable;
(3): médiocre;
(-): déconseillé ou impossible

Compression

- **supprime les blancs**

pour
- gain en espace disque
- peut s'appliquer sur les structures seq, hash, isam, btree

contre
- la décompression est faîte à chaque accès => mise à jour plus coûteuse
- si données peu compréssible (pour adresse)
- si nombreux index secondaires



Clés primaires et clés plaçantes (1)

- **Clé primaire :**
  - le critère de recherche le plus fréquemment utilisé sur une table relationnelle concerne le plus souvent la **clé primaire**.
  - la clé primaire peut être porter sur un attribut (**clé primaire simple**) ou plusieurs attributs (**clé primaire composée**)
- **Clé plaçante :**
  - elle place les tuples d'une table sur disque selon la valeur de cette clé primaire facilitera grandement l'accès à la table selon ce critère

  → adopter pour la table une organisation indexée, dans laquelle la clé primaire de la table sera aussi **clé plaçante** et clé d'**index primaire**

Le choix judicieux pour chaque table d'une d'organisation parmi les organisations proposées par le SGBD adopté (principalement indexées), est un choix d'optimisation important.

Index secondaires (1)

- les applications sur la base de données effectuent aussi des accès aux données selon de multiples critères de recherche :

  -> l'installation d'**index secondaires** permet d'éviter un coûteux balayage séquentiel de la table

- **index secondaire** = un ré-ordonnancement logique des tuples de la table en fonction d'une clé d'accès ou **clé secondaire**, pouvant être discriminante ou non, simple ou composée, différent de la **clé plaçante**

LOGIQUE	*sémantique*	clé primaire clé externe	*simple ou composée*
PHYSIQUE	*accés et organisation*	clé plaçante clé secondaire	*simple ou composée*

Index secondaires (2)

- lors d'une recherche sur **index secondaire**, le balayage de l'index secondaire donne les identifiants de tous les tuples, appelé aussi "tuple identifiant", satisfaisant le critère de recherche
- les tuples sont ensuite **accédés directement** (à raison d'une entrée/sortie par page) par via l'identifiant et selon l'organisation de la table

Exemple : dans INGRES :

index_salaire

salaire	TIDP
8000	2
10000	0
12000	3
20000	1
...	...

emp

nom	salaire	....
jean	10000	
paul	20000	
renée	8000	
annie	12000	
...	...	

TIDP = pointeur sur le tuple (adresse de page et offset sur la page) de la table emp

Index secondaires (3)

Suite exemple :

index_salaire:

- est une table de la BdD
- a autant de tuples que emp
- est organisé par défaut en ISAM (Ingres)

Avantages :

- permet de spécifier des clés secondaires
- utilisé par l'optimiseur de requêtes

Inconvénients :

- gourmand en place disque
- mise à jour couteuse de la table *emp*
- détruit quand la table *emp* est réorganisé (Modify)

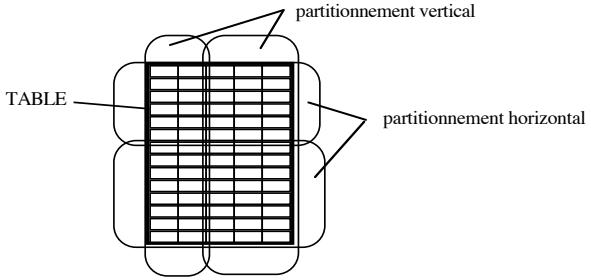
Partition de tables relationnelles (segmentation)

Objet : réduire les coûts d'accès aux données en **éliminant des informations inutilement transférées** de la mémoire secondaire à la mémoire centrale (entrées/sorties)

- les attributs et tuples des tables d'une BdDR **rarement tous utilisés** dans les requêtes : seulement 20 % des données effectivement stockées sont concernées par 80% des requêtes lancées sur la base (règle des 80/20 de Knuth).
- D'où l'idée **d'éclater certaines tables** de la BdDR afin de réduire le flux des informations en entrée/sortie :
  - soit verticalement en paquets d'attributs :
    partitionnement vertical
  - soit horizontalement en paquets de tuples :
    partitionnement horizontal

Partition de tables relationnelles (segmentation)

- le **paquet identifié comme le plus sollicité** pourra par exemple être stocké sur le support mémoire secondaire le **plus performant** :



- tout se passe comme si une table du MLD était **découpée en projections** ou **restrictions arbitraires** (non soumises aux règles de la normalisation) et stockée dans des sous-tables implantées physiquement dans des zones mémoires différentes.

Partitionnement vertical

- Consiste à **éclater une table en sous tables** regroupant **les attributs les plus souvent invoqués ensembles**
- Le partitionnement doit être **totalement transparent aux utilisateurs** qui ne connaissent que les schémas logiques des tables

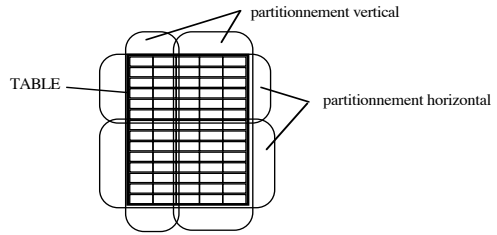
→ nécessite un prétraitement de leurs requêtes

- **Problèmes rencontrés lors de ce partitionnement sont [Richard 89] :**
  - choix d'une bonne partition (table à n attributs : $n^{n/2}$ partition possibles)
  - les coûts de certaines transactions utilisant des attributs répartis dans des paquets différents peuvent se révéler supérieurs à ce qu'ils seraient si les attributs n'étaient pas répartis.

Partitionnement horizontal

Consiste à subdiviser les tuples d'une table en sous tables de même schéma

- on retrouve la **table initiale** par simple **union algébrique** des **sous-tables**
- on regroupe ensemble les tuples avec **forte probabilité d'être accédés ensemble**
- chaque sous-relation peut avoir sa **propre organisation**, optimisant l'accès



Partitionnement horizontal

Intérêts du partitionnement horizontal [Richard 89] :

- apporte des solutions au problème de protection de données sensibles
- efficace pour limiter désorganisation des tables avec sous-ensemble de tuples :
  - soit de longueurs variables
  - soit ayant une valeur nulle soit dont la valeur est plus souvent modifiée
- la sous-table contenant ces tuples sera alors réorganisée aussi souvent que nécessaire, sans que les autres sous-tables le soit.

Doit être adopté avec prudence :

- s'il **favorise** les requêtes les plus fréquentes ou celles mettant en jeu les plus gros volumes d'information, il peut au contraire **pénaliser** fortement d'autres requêtes.

Clustérisation (cluster)

Objet : **regrouper physiquement, sur une même page** (une entrée/sortie), des données provenant d'une ou plusieurs tables et vérifiant des prédicats de sélection précisés

- l'utilisation des clusters est totalement **transparente** à l'utilisateur
- en général le **regroupement concerne 2 tables**, afin d'optimiser les jointures sur ces tables, c'est le principal intérêt de la clustérisation
- il est aussi possible de définir des clusters sur une **seule table**, ce qui permet d'implanter les deux variantes de partitionnement précédemment évoquées [Richard 89].
- peut être comparée à des optimisation plus structurelles **dénormalisantes**, mais elle présente le grand avantage **de ne pas modifier la structure logique** en se plaçant uniquement au niveau de **l'implantation physique** des données.

Clustérisation (cluster)

Exemple avec le SGBDR ORACLE : il faut créer tout d'abord les clusters, puis y placer une ou plusieurs tables (si plusieurs tables sont regroupées dans un cluster, elles doivent avoir au moins un attribut en commun)

- le regroupement des tables dans un cluster a pour conséquences que :
  - chaque valeur de cet attribut commun est stockée une fois dans la base
  - les tuples d'une table ayant la même valeur pour cet attribut sont stockées dans une même zone du disque (page)
  - SQL*plus crée un index, appelé cluster index, sur les attributs regroupés
- il est possible :
  - soit de créer une table et placer sa définition dans un cluster (création d'une table pour la première fois)
  - soit créer la table et la placer dans un cluster existant (pour ajouter une table existante dans le cluster)
- **Syntaxe de la création du cluster sous Oracle :**

```
CREATE CLUSTER nom_cluster
(attribut_cluster1 type_données, attribut_cluster2 type_données...)
[SIZE taille_logique]
[SPACE nom_zone]
[COMPRESS \ NOCOMPRESS];
```

- il est possible de supprimer des clusters, il faut alors d'abord en extraire les tables.

3. Optimisation des requêtes

- **Problématique de l'optimisation des requêtes**
- **Optimiseurs de requêtes**
- **Statistiques sur l'état des tables**

Problématique de l'optimisation des requêtes

Pourquoi ?

- **logique/physique**
  - → *l'utilisateur voit et manipule des tuples et des relations (description logique des infos.) sans savoir comment se fait le stockage sur mémoire secondaire (physique)*
  - → *dissociation logique/physique -> indépendance programme/données*
- **LMD assertionnels**
  - → langages de manipulation de données prédicatifs assertionnels:
    - définir le "quoi" pas le "comment"
    - ordre des spécifications indifférent
    - nombreuses formulation possible pour une même requête

=> le SGBD relationnel doit:

- → assurer le passage : « représentation relationnelle » -> « structure de stockage »
- → transformer : requête assertionnelle -> séquence d'opération/données
- → choisir la séquence optimale

=> Optimisation des requêtes

Optimiseurs de requêtes

- **La plupart des SGBD relationnels possèdent des optimiseurs de requêtes**
- **Les optimiseurs de requêtes ont pour rôle de :**
  - minimiser les I/O disques
  - minimiser le temps CPU
  - exploiter au maximum les index secondaires (requêtes mono-table)
  - établir une stratégie lors de requêtes multi-tables :
    - → *choix d'un ordre des jointures*
    - → *comment faire des jointures*
    - → *utiliser les index secondaires*
- **Pour cela les optimiseurs de requêtes utilisent des statistiques sur l'état des tables**

Statistiques sur l'état des tables

Dans INGRES commande OPTIMIZEDB :

pour chaque table R:
- NCARD(R): nb tuples de R (cardinalité de R)
- TCARD(R): nb de pages occupées par R

pour chaque attribut A de la table R:
- MIN(A): valeur minimale de A présente dans R
- MAX(A): valeur maximale de A présente dans R

pour un index I de la table R:
- ICARD(I): nb de clés distinctes dans I
- NINDX(I): nb de pages occupées par I
- CLUSTER(I): index plaçant ou non

- **Ces informations sont rangées dans une "méta-base"**
- **Informations mises à jour périodiquement par la commande UPDATE_STATISTICS**

Usage de Optimizedb (Ingres)

Soit la restriction sur la table emp :

emp.age > 100

- il y a de forte chance qu'aucun tuple de la table soit retourné
- le système considère cependant age comme un entier compris entre +/- 32767

-> l'optimiseur de requête peut adopter de meilleurs plans s'il sait à quoi ressemblent les données

Valeurs mini et maxi

- si tous les employés ont moins de 66 ans (valeur maxi = 66), l'optimiseur en déduit instantanément qu'aucun tuple ne repond à la requête
- sans autre information, l'optimiseur suppose alors une distribution uniforme entre valeur mini et valeur maxi

Commande:

Optimizedb -zx -remp **-a**age

génère les valeurs min et max pour l'attribut age de la table emp.

Usage de Optimizedb (Ingres)

(ingres)

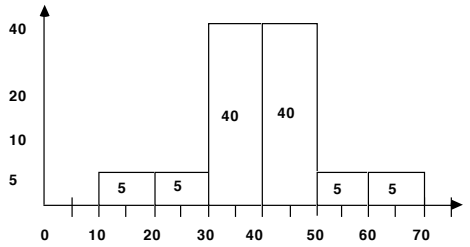
Histogrammes

- sans histogrammes, l'optimiseur de requêtes suppose une distribution uniforme entre les valeurs mini et maxi

Commande:

Optimizedb -zh -remp **-a**age

génére un histogramme de la distribution de l'âge de la table emp:



Usage de Optimizedb (Ingres)

Nombre de valeurs uniques

- utile pour anticiper le nombre de tuples répondant à une restriction ou une jointure
- étant donné la restriction emp.dept = "commercial", l'optimiseur suppose qu'un tuple sera retourné sans autre information
- san information sur les duplications, l'optimiseur suppose une jointure 1-1

Exemple

Base: personnel et Tables: emp (employés), dept (départements)
Attributs de emp: dept, div et attributs de dept: dnom, div, etage

avant de lancer cette requête:

```
Select e.name, d.etage From emp e, dept d
Where e.dept=d.dnom And e.div=d.div And d.etage >1
```

lancer la commande suivante:

Optimisedb personnel **-r**emp **-a**dept **–a**div **-r**dept **-a**dnom **-a**div **-a**floor

4. Optimisation logique d'une BD relationnelle : dénormalisation

- **Problématique de la dénormalisation**
- **Regroupement de tables**
- **Création de redondances d'attribut non clé**
- **Ajout de clé étrangère redondante**
- **Création de transitivité**
- **Création de tables de jointure**

Problématique de la dénormalisation (1)

La dénormalisation :

- ensemble de **règles d'optimisation structurelles** du modèle logique relationnel
- ces règles garantissent le **respect des spécifications initiales** :
- tout autre type de modification, sous couvert d'optimisation, peut **entraîner une altération de la sémantique du modèle conceptuel de données**
- pour chacune de ces règles il faut **estimer les gains espérés et les pertes générées**, ainsi que les **conséquences secondaires**

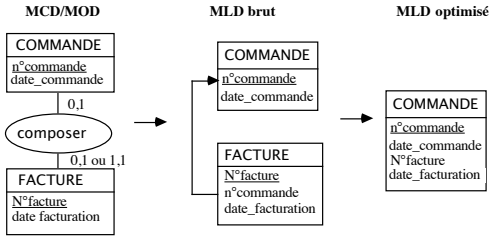
Problématique de la dénormalisation (2)

Les optimisations structurelles obtenues par dénormalisation :

- devront être **conduites avec prudence**, car elles ont des conséquences qui doivent être estimées
- l'efficacité de ces différentes optimisations **dépend du SGBD utilisé :** certaines même ne seront pas toujours possibles sur tel ou tel système.
- le concepteur optimise son modèle par **modifications successives** de la structure, en évaluant le bien fondé de son choix en **estimant le "gain" total** par rapport aux critères économiques définis.

Regroupement de tables

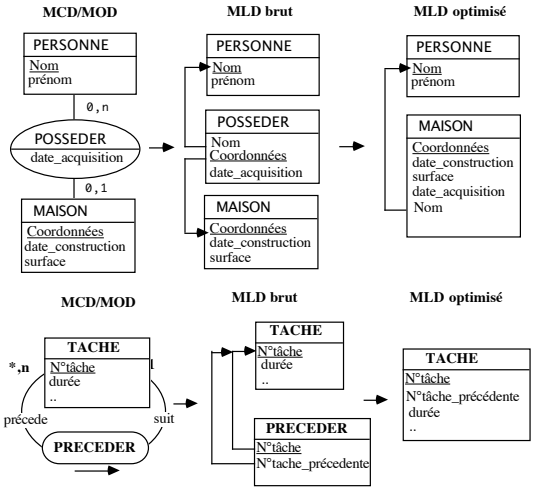
2 tables reliées par un lien un vers un, peuvent être regroupée en une même table, afin de réduire des activités en #T, ?T voir T+, T-, Tm et TXT' sur ces 2 tables :



Diagnostic : une forte activité en #T ou ?T sur des tables reliées par des liens un vers un.
Gain : suppression de la table B et donc de toutes les primitives associées.
Perte : augmentation du volume à mémoriser pour la table A = Σ taille des attributs migrés de la table B x nombre de tuples de la table A accueillant la migration.
Conséquences induites :
- si la taille des attributs de B migrant dans A est importante, il peut y avoir augmentation du coût de certaines primitives sur A.
- selon les cas, il faudra prévoir la gestion des valeurs nulles sur les attributs migrants.

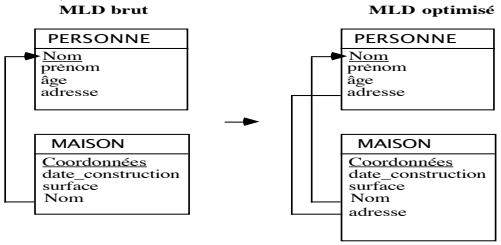
Regroupement de tables

Exemples :



Création de redondances d'attribut non clé

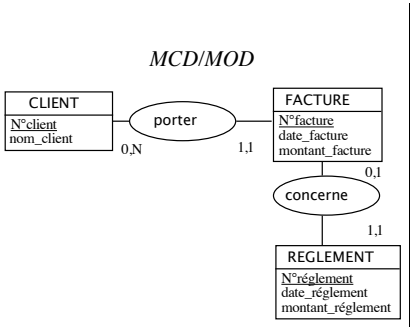
• Lorsque la consultation d'un ou plusieurs attributs d'une table A à partir d'une autre table B est importante et génère de nombreux #T, ?T voir TXT', il peut être intéressant de dupliquer, sous certaines conditions, ces attributs sollicités dans B:

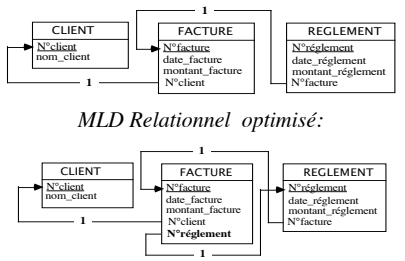


Optimisation justifiée pour une consultation importante de "adresse" à partir de la table MAISON.

Diagnostic : une forte activité en #T ou ?T voire TXT' sur des tables reliées par des liens un vers un ou un vers plusieurs.
Gain : réduction de #T ou ?T voire TXT'.
Perte : augmentation du volume à mémoriser pour la table accueillant la redondance d'attributs = S taille des attributs migrés x nombre de tuples de la table accueillant la duplication.
Conséquences induites :
- si la taille des attributs dupliqués est importante, il peut y avoir augmentation du coût de certaines primitives sur la table accueillant la redondance. selon les cas, il faudra prévoir la gestion des valeurs nulles sur les attributs dupliqués et d'une façon générale les conséquences de cette dénormalisation.

Ajout de clé étrangère redondante





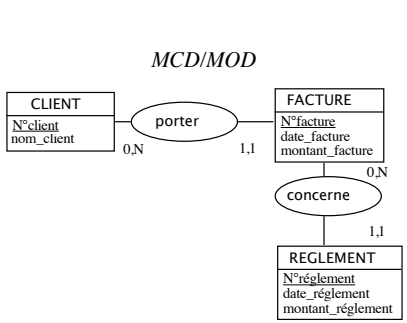
Schémas relationnels :

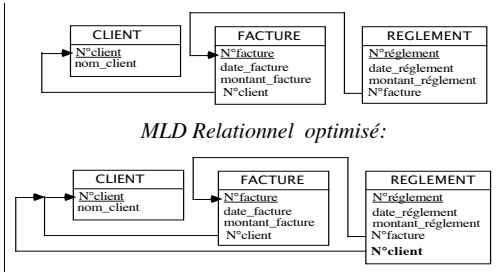
CLIENT(N°client, nom_client)
FACTURE(N°facture, **N°règlement**, date_facture, montant_facture, N°client)
REGLEMENT(N°règlement, date_réglement, montant_règlement, N°facture)

Diagnostic : une forte activité en #T ou ?T sur des tables reliées par des liens un vers un.
Gain : réduction de #T ou ?T.
Perte : augmentation du volume à mémoriser pour la table accueillant la nouvelle clé étrangère = S taille des attributs de cette clé x nombre de tuples de la table accueillant la migration.
Conséquences induites :
- si la taille de la nouvelle clé externe est importante, il peut y avoir augmentation du coût de certaines primitives sur la table l'accueillant.
- selon les cas, il faudra prévoir la gestion des valeurs nulles sur les attributs dupliqués et d'une façon générale les conséquences de cette dénormalisation.

Création de transitivité

Revient à créer une nouvelle dépendance transitive :





MLD Relationnel optimisé:

CLIENT(N°client, nom_client)
FACTURE(N°facture, date_facture, montant_facture, N°client)
REGLEMENT(N°règlement, **N°client**, date_réglement, montant_règlement, N°facture)

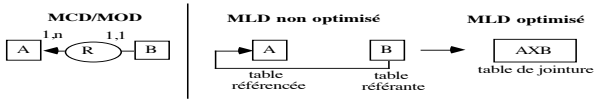
Diagnostic : une forte activité en #B ou ?B et tables relationnelles A, B, C reliées par des liens relationnels tels que précisés plus haut.
Gain : réduction de #B ou ?B.
Perte : augmentation du volume à mémoriser pour la table C accueillant la nouvelle clé étrangère = S taille des attributs de cette clé x nombre de tuples de C.
Conséquences induites :
- si la taille de la nouvelle clé étrangère est importante, il peut y avoir augmentation du coût de certaines primitives sur la table C.
- selon les cas, il faudra prévoir dans C la gestion des valeurs nulles sur les attributs de la clé externe et d'une façon générale les conséquences de cette dénormalisation.

Création de tables de jointure (1)

Il s'agit de réduire le temps d'exécution de requêtes en stockant physiquement les tables résultantes des jointures les plus fréquemment mise en oeuvre par ces requêtes :

Cas cardinalité mini * = 1 (ou >1)

- élimination de la table référencée A et de la table référençante B
- création de la table de jointure A X B
- les contraintes référentielles entre A et B sont transformées en contraintes de dépendance fonctionnelles sur les attributs de la table de jointure A X B

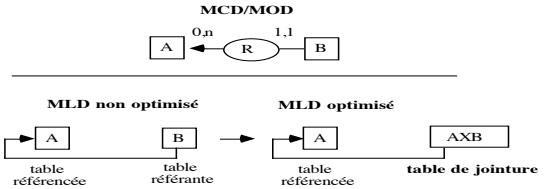


Diagnostic : une forte activité en A X B sur des tables avec dépendances fonctionnelles.
Gain : réduction du coût des jointures A X B.
Perte :
- augmentation du volume à mémoriser pour la table A X B (plus de redondance)
- table plus grosse, plus lourde à exploiter.
Conséquences induites :
- si taille des attributs dupliqués est importante, alors augmentation du coût de certaines primitives sur la table de jointure.
- selon les cas, il faudra prévoir d'une façon générale les conséquences de cette dénormalisation (MàJ sur attributs de A).

Création de tables de jointure (2)

Cas cardinalité mini * = 0 :

- la table référencée A doit être conservée
- élimination de la table référençante B
- création de la table de jointure A X B en éliminant éventuellement les attributs non utilisés par les requêtes utilisant cette jointure
- transférer les contraintes référentielles entre A et B dans la table de jointure A X B
- introduire les contraintes sur les dépendances fonctionnelles



Diagnostic : une forte activité en A X B sur des tables avec dépendances fonctionnelles.
Gain : réduction du coût des jointures A X B.
Perte :
- augmentation du volume à mémoriser pour la table A X B (plus de redondance)
Conséquences induites :
- si la taille des attributs dupliqués est importante, il peut y avoir augmentation du coût de certaines primitives sur la table de jointure.
- selon les cas, il faudra prévoir d'une façon générale les conséquences de cette dénormalisation (mises à jour sur les attributs de A qui se retrouvent dans A et A X B).