Plan du module

- **Partie 1- I. Introduction aux SGBDs**
  - Chapitre 1: Présentation des SGBDs
  - Chapitre 2: Définition et Evolution des données
  - Chapitre 3: Contrôle des données
  - **Chapitre 4: Gestion des objets utilisateurs**
    - **(Vues, séquences et Index)**
- **Partie 2- II. Langage procédural: PL/SQL**
- **Partie 3- III. Les Transactions**

ENICARTHAGE
المدرسة الوطنية للمهندسين بقرطاج
Ecole Nationale d'Ingénieurs de Carthage

I.4 Séquences, Vues & Index

2ème Ingénieurs info

Année Universitaire 2020-2021

R.MGHIRBI

C- Les index

Plan

- Motivations
- Définitions
- Structures d'index
- Index sous Oracle
  - Ordres de définition
  - Index sur fonction
  - Vues d'index dans le dictionnaire de données
- Bilan

1. Motivations(1/2)

- Exemple : Soit la table film suivante, comportant:
  - 1 000 000 (un million) de films
- Répondre à la requête:
  - select * from Film where année='1992'??? /
  - Un moyen pour récupérer la ou les lignes répondant à la clause année='1992' est ***de balayer toute la table***.
  - Le temps de réponse sera prohibitif!!!.

1. Motivations(2/2)

- Exemple (suite): On part des hypothèses suivantes:
  - Taille d'un enregistrement = 1200 octets
  - Taille d'un bloc = 4K ➔ 4096/1200➔ 3 enregistrements par bloc
  - Nb de blocs nécessaires: 1000000/3 => environ 300 000 blocs, ~1,2 Go

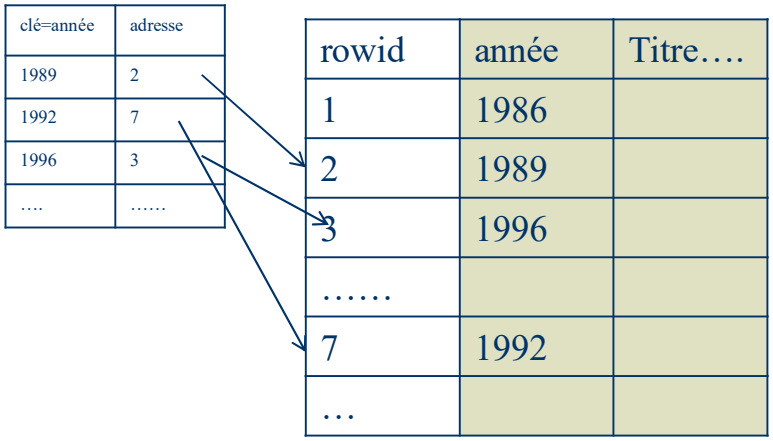
titre	année	...	titre	année	...
Vertigo	1958	...	Annie Hall	1977	...
Brazil	1984	...	Jurasic Park	1992	...
Twin Peaks	1990	...	Metropolis	1926	...
Underground	1995	...	Manhattan	1979	...
Easy Rider	1969	...	Reservoir Dogs	1992	...
Psychose	1960	...	Impitoyable	1992	...
Greystoke	1984	...	Casablanca	1942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

  - Afin d'optimiser ce type de requête, on pourra ***indexer*** l'attribut " année".
  - Une clé de recherche = une *liste* d'un ou plusieurs attributs sur lesquels portent des critères.

$

2. Principe

- Le principe d'un index est l'association de l'adresse de chaque enregistrement (ROWID) avec la valeur des colonnes indexées.



- *Le **ROWID** identifie un enregistrement d'une table dans la base de données, à partir de l'adresse physique du bloc et du numéro d'enregistrement dans le bloc.*

2. Principe

- Le principe de base d'un index est de construire une structure permettant d'optimiser les *recherches par clé* sur un fichier.
  - la clé peut avoir une certaine valeur :
    - SELECT *
      FROM Film
      WHERE titre = 'Vertigo'
  - Ou désigne une recherche par intervalle.
    - SELECT *
      FROM Film
      WHERE annee BETWEEN 1995 AND 2002
  - clés composées de plusieurs attributs
    - SELECT *
      FROM Artiste
      WHERE nom = 'Alfred' AND prenom='Hitchcock
  - L'index ne sert à rien pour toute recherche ne portant pas sur des valeurs de clé.

3. Définitions (1/2)

- Un *index* est un objet facultatif associé à une table pour accélérer les requêtes sur cette table.
- C'est une structure de données organisée de manière à accélérer certaines recherches.
- C'est un fichier !: Les enregistrements (ou *entrées*) de l'index sont de la forme [valeur, Addr]
- Le fichier d'index est trié sur valeur
- Les index peuvent être créés afin d'améliorer les performances de mise à jour et d'extraction des données.
- Un choix judicieux des index, ni trop ni trop peu, est donc un des facteurs essentiels de la performance d'un système.

3. Définitions (2/2)

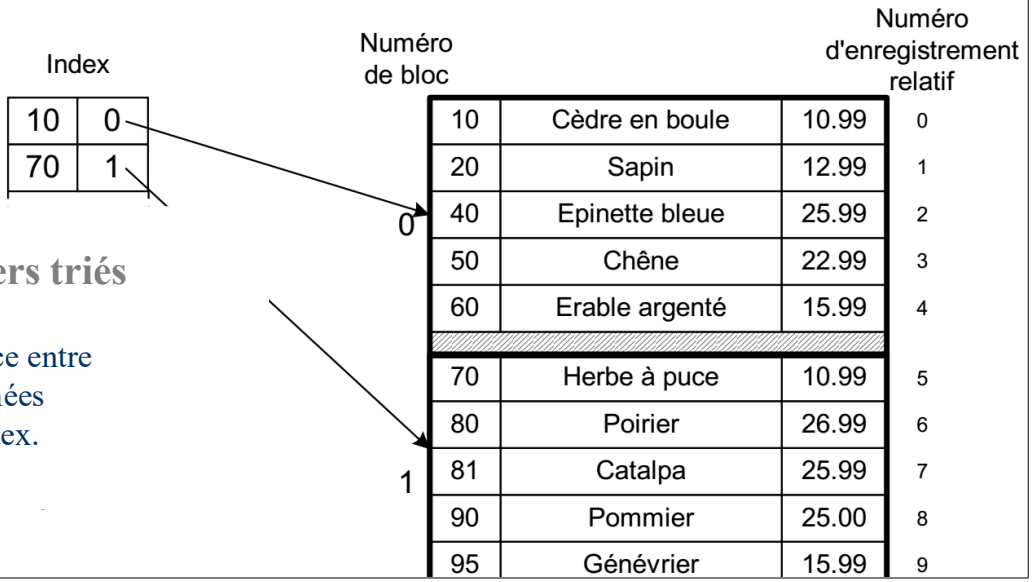
- Un index est créé
  - Soit automatiquement par le noyau (indexation implicite)
  - Soit à la demande du développeur (indexation explicite)
  - Indexation implicite:
    - Est mise en œuvre lorsqu'une clé primaire ou une contrainte d'unicité est définie sur une table
  - Indexation Explicite:
    - Pour accéder plus vite aux lignes d'une table à partir d'autres attributs de la table : index créés manuellement (explicites)
- Un index est mis à jour automatiquement lors de modifications de la table.
  - Peut concerner plusieurs attributs d'une même table (*index composé*).
    - Possibilité de créer plusieurs index sur une même table.
    - Dans le cas où il existe plusieurs index pour une table, l'optimiseur de requête sait choisir le meilleur index en fonction de la requête à exécuter.

4.Structures d'index

- Organisation de l'index
  - Non dense
  - Séquentiel (index dense)
  - Séquentiel indexé
  - Arbres B (par défaut dans ORACLE)
  - Bitmap
  - Hachage

4.1 Index creux/épars (non dense) (1/2)

- Un seul enregistrement d'index par bloc du fichier de données=> le fichier de données doit rester trié sur les valeurs de la clé d'index!!!
- Non dense (creux)
- Index de taille réduite: Accès séquentiel rapide (accès direct à un bloc puis séquentiel dans le bloc) est représenté dans le fichier d'index

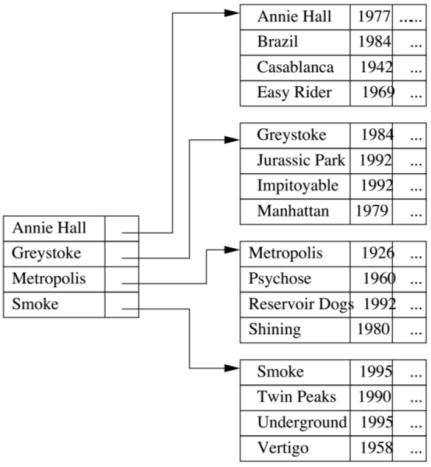


Maintenir des fichiers triés : garder une étroite correspondance entre l'ordre du fichier de données et l'ordre du fichier d'index.

4.1 Index non dense (2/2)

Exemple: On veut indexer un fichier de 16 films par un index non-dense sur le titre des films (clé d'indexation)

- On suppose que chaque bloc du fichier de données contient 4 enregistrements:
  - Ceci donne un minimum de 4 blocs.
  - Il suffit alors de quatre paires [titre, Adr] pour indexer le fichier



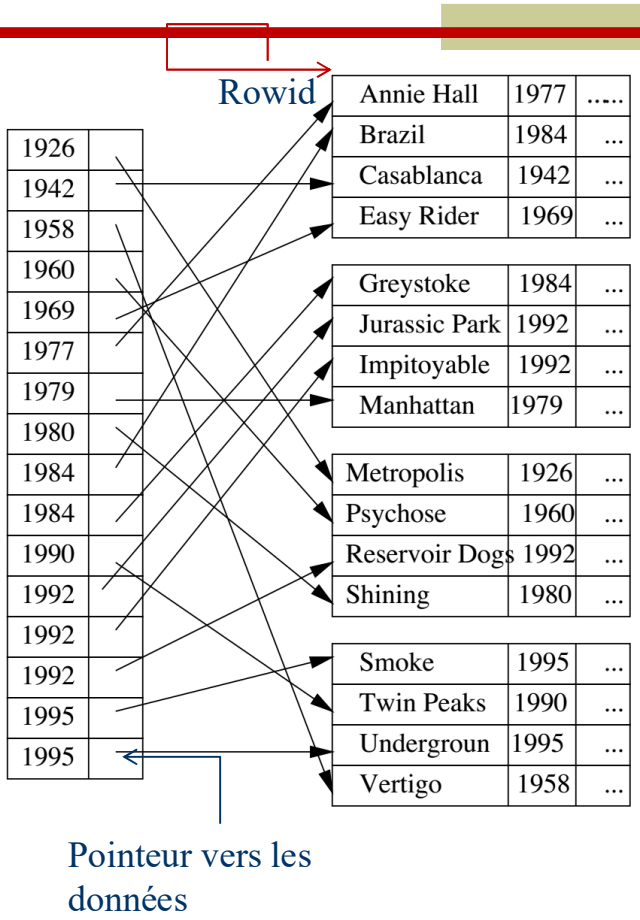
- Pour le fichier de 1000000 de films,
  - Il est constitué de 300.000 blocs.
  - Supposons qu'un titre de films occupe 20 octets en moyenne, et que l'adresse d'un bloc 8 octets.
  - La taille de l'index est donc
  - 300000*(20+8)=8,4 MO octets, à comparer aux 1,2Go du fichier de données. (0,6% de la taille du fichier de données)

4.2 Index séquentiel (dense) (1/3)

- L'index est un fichier trié sur la clé.
- Que se passe-t-il quand on veut indexer un fichier qui n'est pas trié sur la clé de recherche ?
- On ne peut tirer parti de l'ordre des enregistrements pour introduire seulement dans l'index la valeur de clé du premier élément de chaque bloc.
- Il faut donc baser l'index sur *toutes* les valeurs de clé existant dans le fichier, et les associer à l'adresse d'un enregistrement, et pas à l'adresse d'un bloc.
- Un tel index est *dense*.
- **Tous les enregistrements sont représentés. i.e** Toutes les clés de recherche sont présentes dans l'index

4.2 Index séquentiel (dense) (2/3)

- La figure montre le même fichier contenant seize films:
  - Trié sur le titre, et
  - Indexé maintenant sur l'année de parution des films.
- On constate d'une part que:
  - toutes les années du fichier de données sont reportées dans l'index, ce qui accroît considérablement la taille de ce dernier, et d'autre part que:
  - la même année apparaît autant qu'il y a de films parus cette année là



4.2 Index séquentiel (dense) (3/3)

- **Coût sur l'exemple**
  - Sur notre exemple de la table film (fichier de 1,2 Go)
    - Select * from films where année='1992': la clé d'index étant l'année
  - Calculons la taille du fichier d'index:
    - Une année = 4 octets, une adresse 8 octets (le pointeur vers l'année dans le fichier de données)
    - Taille de l'index : 1000000 *(4+ 8) = 12 Mo
    - Indexer année revient à occuper 12 Mo pour un fichier d'index
  - Cent fois plus petit que le fichier :

année	adr
…….	

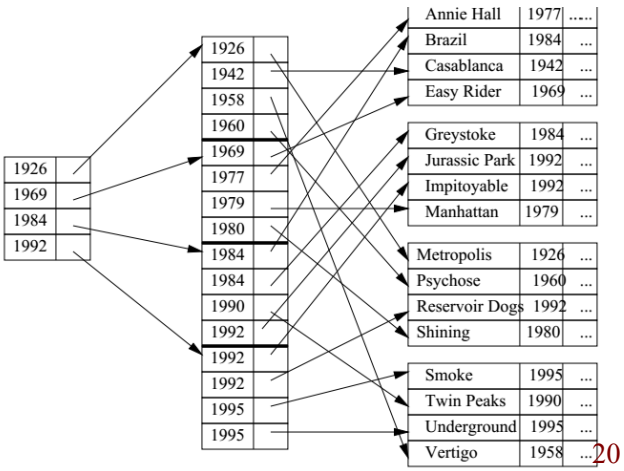
Dense / Non dense ?

- Un index dense peut coexister avec un index non-dense.
- Comme le suggèrent les deux exemples qui précèdent, on peut envisager de:
  - Trier un fichier sur la clé primaire et de créer un index non-dense, puis
  - d'ajouter des index denses pour les attributs qui servent fréquemment de critère de recherche.
- Il est possible en fait de créer autant d'index denses que l'on veut puisqu'ils sont indépendants du fichier de données.
- Cette remarque n'est plus vraie dans le cas d'un index non-dense puisqu'il s'appuie sur le tri du fichier et qu'un fichier ne peut être trié que d'une seule manière.

4.3 Index multi-niveaux

- Il peut arriver que la taille du fichier d'index devienne elle-même si grande que les recherches dans l'index en soit pénalisantes.
- On l'indexe à son tour.
  - **Essentiel** : l'index est trié, donc on peut l'indexer par un second niveau (**Non-dense**)
  - On parle du **séquentiel indexé**
  - **Exp:** chercher les films de 1990
  - Les index multi-niveaux sont très efficaces en recherche, et ce même pour des jeux de données de très grande taille.
  -

Le problème est, comme toujours, la difficulté de maintenir des fichiers triés sans dégradation des performances

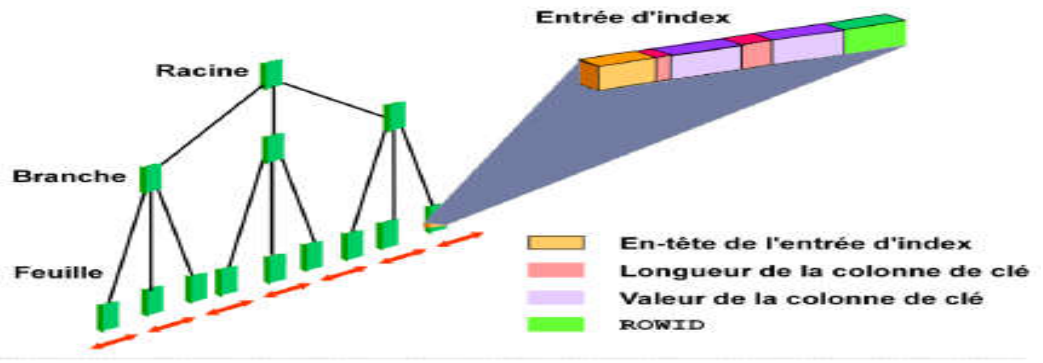


4.4 Index B-Arbre

- Aboutissement des structures d'index basées sur l'ordre des données
- C'est un arbre équilibré
- Chaque nœud est un index local
- Il se réorganise dynamiquement
- Utilisé universellement !

4.4 Index B-Arbre

- Organisation:
  - Au sommet de l'index se trouve la racine, qui contient les entrées pointant vers le niveau suivant de l'index.
  - Le niveau suivant comprend les blocs 'branches', qui pointent vers les blocs du niveau suivant de l'index.
  - Au plus bas niveau se trouvent les nœuds feuilles, qui contiennent les entrées d'index pointant vers les lignes de la table.
  - Ajoute des algorithmes de réorganisation dynamique qui résolvent la question de la maintenance d'une structure triée.



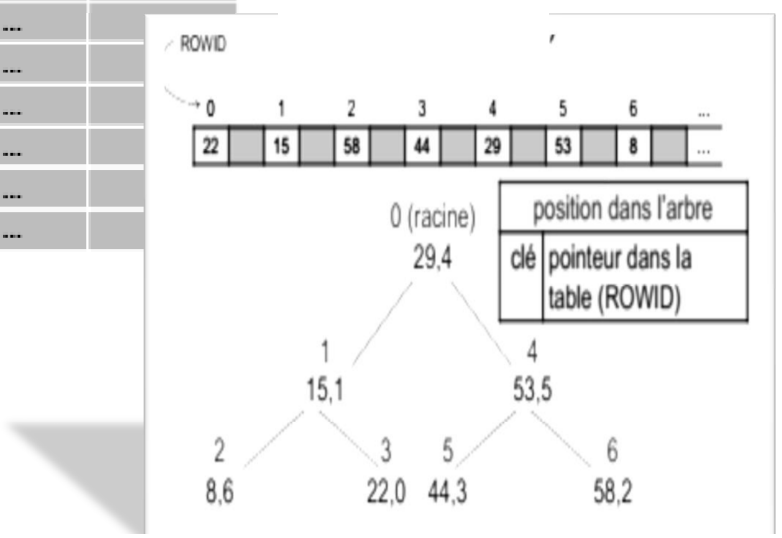
4.3 Index par Arbre-B

- La particularité de ce type d'index est qu'il conserve en permanence une arborescence symétrique (balancée).
- Toutes les feuilles sont à la même profondeur.
- Le temps de recherche est ainsi à peu près constant quel que soit l'enregistrement cherché.
- Le plus bas niveau de l'index (*leaf blocks*) contient les valeurs des colonnes indexées et le *rowid*.
- Toutes les feuilles de l'index sont chaînées entre elles.
- Ces index sont très fiables et performants,

4.3 Index B-Arbre

Exemple2:

RowID	numFilm	titre	realisateur
0	22	...	...
1	15	...	
2	58	...	
3	44	...	
4	29	...	
5	53	...	
6	8	...	

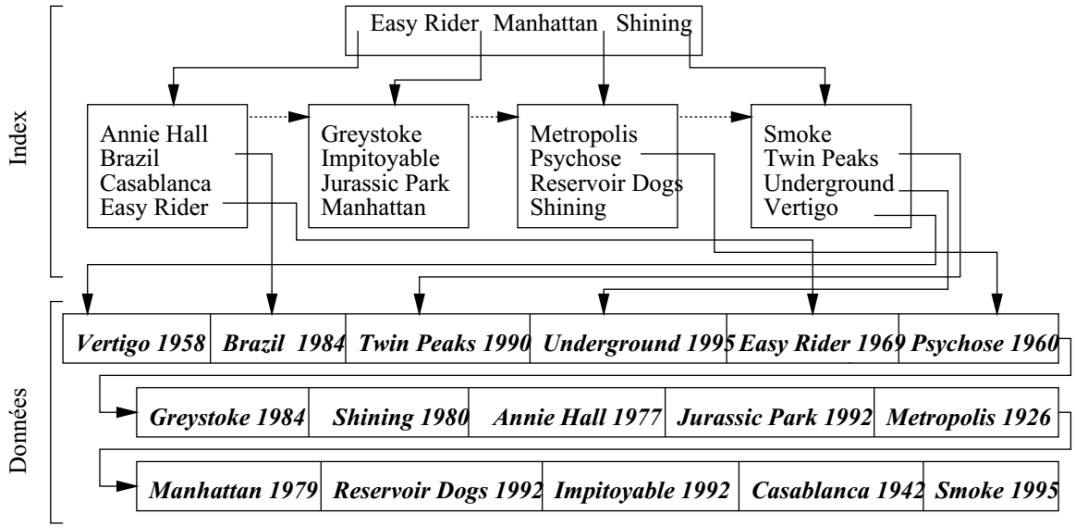


pointeur fils gauche	pointeur fils droit	clé	pointeur dans la table (ROWID)

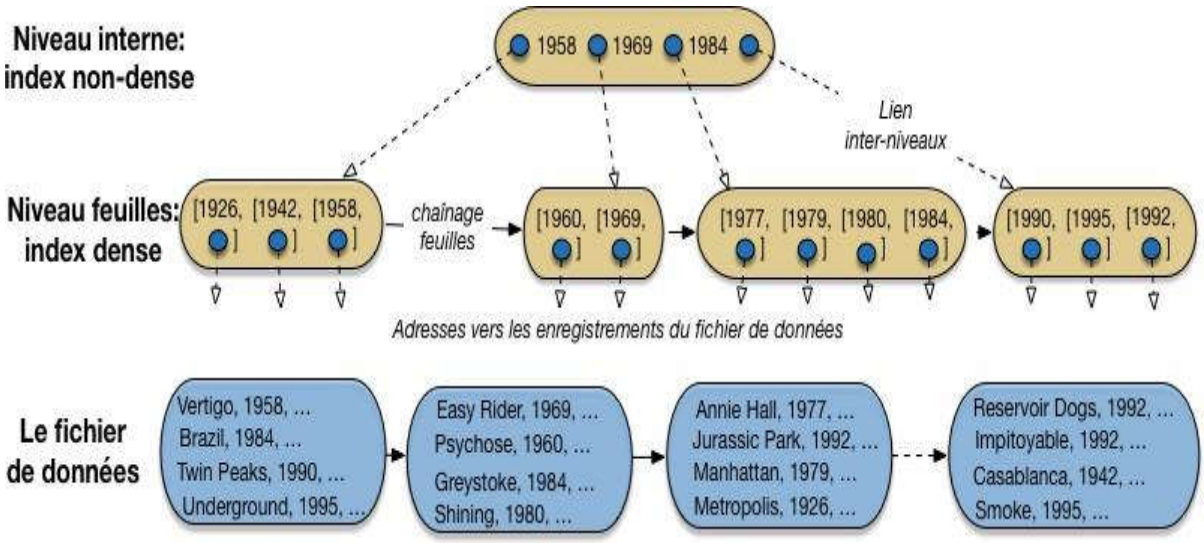
1,4,29,4	2,3,15,1	-1, -1, 8, 6	-1, -1, 22, 0	5, 6, 53, 5	-1, -1, 44, 3	-1, -1, 58, 2	...
0	1	2	3	4	5	6	...

4.3 Index par Arbre-B+

- Au lieu d'une clé par nœud, chaque nœud est un bloc contenant k clés triées et k+1 fils
-



Exemple d'un arbre B

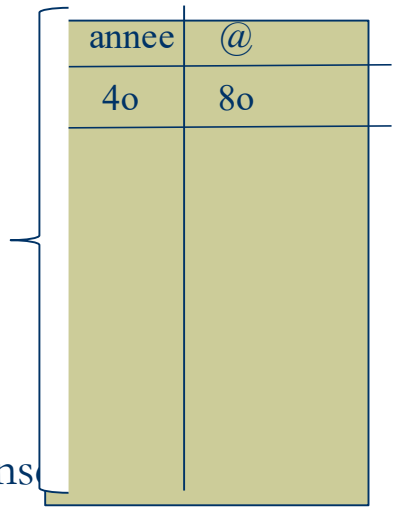


Avec notre fichier de 1M de films

- Nb_enregistrement par bloc:
  - 4096/12= 341 enregistrements
- Nb_blocs nécessaires pour le niveau dense (afin d'indexer toutes les valeurs de l'année)

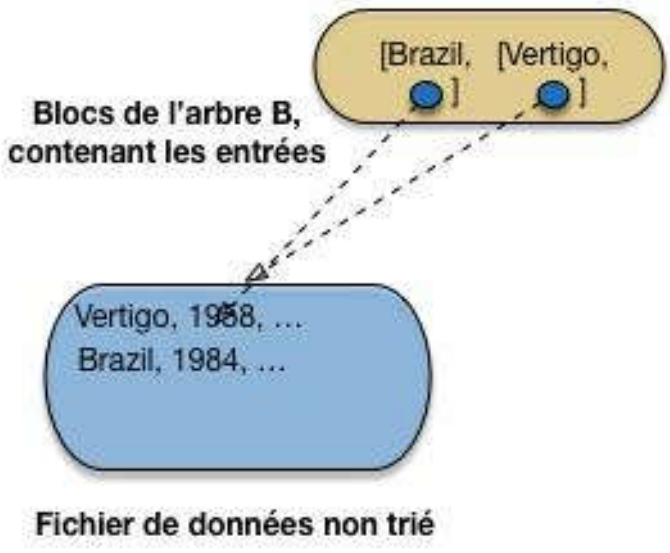
Est au pire: 1.000.000/341= 2 932 blocs

- Le deuxième niveau est non dense.
  - Il comprend autant d'entrées que de blocs ➔ 2932
- Nb_de_blocs necessaire pour le niveau non dens
  - 2932/341=~8 Blocs au deuxième niveau et 1 bloc racine à 8 clés.

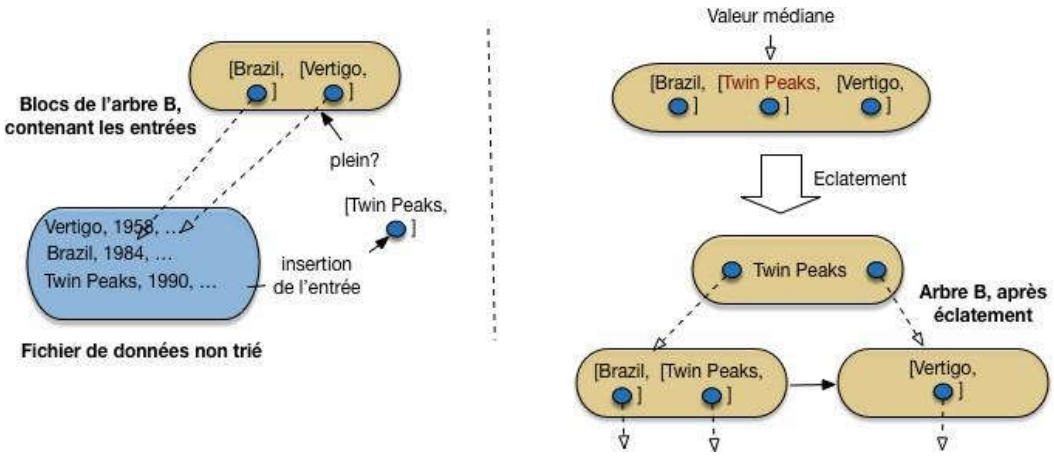


Insertion dans un arbre B

On construit l'index sur le titre des films. Situation initiale.



La procédure d'éclatement



Quand un nœud est plein, il faut effectuer un **éclatement**.

Conclusion Arbre B+

- Ces index sont très fiables et performants,
- Ils ne se dégradent pas lors de la montée en charge de la table *(La hauteur de l'arbre est logarithmique par rapport au nombre d'enregistrements).*
- Structure de données à la fois simple, très performante et propre à optimiser plusieurs types de requêtes : recherche par clé, recherche par intervalle, et recherche avec un préfixe de la clé.

4.4 Bitmap

- Exemple1: select * from film where genre='Drame';
- Pour 16 films

	1	2	3	4	5	6	7	8
Drame	0	0	0	1	1	1	0	0
Science-Fiction	0	1	0	0	0	0	0	0
Comédie	0	0	0	0	0	0	0	0

	9	10	11	12	13	14	15	16
Drame	0	0	0	0	0	0	1	0
Science-Fiction	0	1	1	0	0	0	0	0
Comédie	1	0	0	1	0	0	0	1

- On prend le tableau pour la valeur Drame
  - On garde toutes les cellules à 1
    On accède aux enregistrements par l'adresse
    => très efficace si *n*, le nombre de valeurs, est petit.

4.4 Bitmap

- Un index bitmap considère toutes les valeurs possibles pour un attribut.
- Pour chaque valeur, on stocke un tableau de bits (dit bitmap) avec autant de bits qu'il y a de lignes dans la table. (taille égale au nombre d'enregistrements de la table)
- **Très utile pour les colonnes qui ne possèdent que quelques valeurs distinctes et pour des tables peu volumineuses**
- Est recommandé quand on compare le stockage necessaire arbre-B est > stockage necessaire de tableaux de bits.
- Les index *bitmaps* sont très bien adaptés à la recherche d'informations basée sur des critères d'égalité (exemple : genre=' Drame'), mais ne conviennent pas du tout à des critères de comparaison (exemple : annee > 1992)
- La valeur bitmap est stockée en mode compressé, ce qui rend la place occupée par l'index inférieure à celle requise par un index de type B-arbre
- Efficace pour les opérations AND et OR.

3.4 Bitmap

- Exemple 5:
  - Une table contenant des élèves repérés par leur numéro (clé primaire), avec un attribut qui est leur classe : 1, 2 ou 3.

numeleve	...	numclasse	...
54		1	
18		2	
26		1	
30		3	
15		3	
12		2	
61		2	
44		1	

→

numeleve	numclasse=1	numclasse=2	numclasse=3
12	0	1	0
15	0	0	1
18	0	1	0
26	1	0	0
30	0	0	1
44	1	0	0
54	1	0	0
61	0	1	0
	Bitmap classe 1	**Bitmap classe 2**	**Bitmap classe 3**

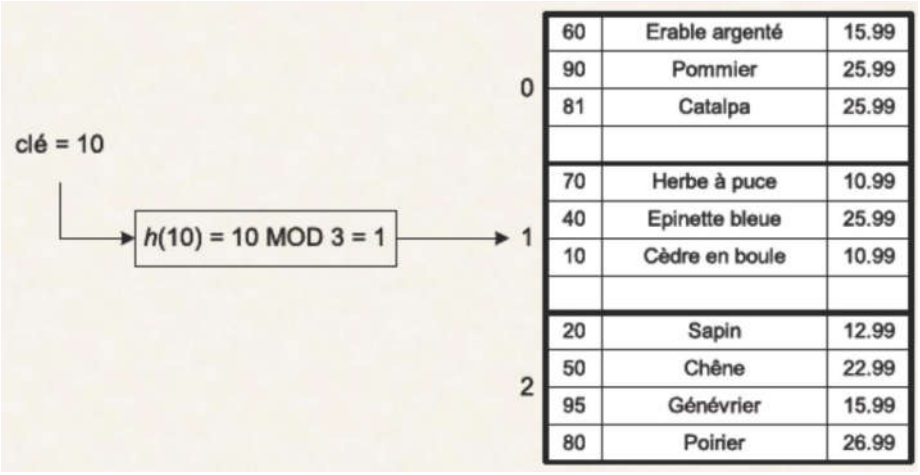
  - Pour numclasse = 2, chaîne de bits : 10100001
  - Peu de place occupée

3.5 Hachage

Principe

- Le stockage est organisé en N fragments constitués de séquences de blocs.
- La répartition des enregistrements se fait par un calcul.
  - Une fonction de hachage h prend une valeur de clé en entrée et retourne une adresse de fragment en sortie
  - Stockage : on calcule l'adresse du fragment d'après la clé, on y stocke l'enregistrement
  - Recherche (par clé) : on calcule l'adresse du fragment d'après la clé, on y cherche l'enregistrement Simple et efficace !

3.5 Hachage

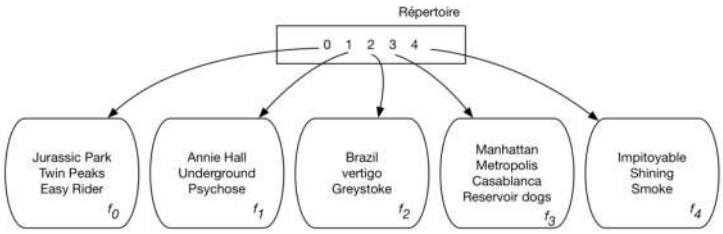


3.5 Hachage

- Une structure concurrente de l'arbre B est le *hachage* qui offre, dans certains cas, des performances supérieures, mais ne couvre pas autant d'opérations.
- Il existe différentes variantes des structures d'index basées sur le hachage:
  - La plus simple, le *hachage statique:*
    - Ne fonctionne correctement que pour des collections de tailles fixes: se réorganise difficilement
    - Ceci exclut des tables évolutives (le cas le plus courant).
  - Le *hachage dynamique*,
    - S'adapte à la taille de la collection indexée. Mais;
    - Il repose sur un répertoire (*directory*) dont la taille peut croître au point de devenir un problème.
  - Enfin la troisième structure est le *hachage linéaire*, une structure qui apporte toute l'efficacité du hachage tout en maintenant une taille de répertoire réduite.
- Problème avec le hachage => ne supporte pas les recherches par intervalle

Exemple

- On veut créer une structure de hachage pour nos 16 films
- Par simplification, on suppose que chaque fragment fait un bloc et contient 4 enregistrements au plus
  - on alloue 5 fragments (pour garder une marge de manœuvre)
  - un répertoire à 5 entrées (0 à 4) pointe vers les pages
  - On définit la fonction h(titre) = rang(titre[0]) mod 5
    - Donc on prend la première lettre du titre (par exemple 'I' pour 'Impitoyable'),
    - on prend sont rang d'alphabet (ici 9) et on garde le reste de la division par 5,
    - le nombre de fragments. h(rang[Impitoyable]) = 4



La seule structure additionnelle est le répertoire, qui **doit** tenir en RAM

3.5 Hachage

Recherche avec une structure basée sur le hachage

- Par clé , Oui
  SELECT * FROM Film WHERE titre = 'Impitoyable'
- Par préfixe ? Non
  SELECT * FROM Film WHERE titre LIKE 'Mat%'
- Par intervalle : non !
  SELECT *
  FROM Film
  WHERE titre BETWEEN 'Annie Hall' AND 'Easy Rider'

3.5 Hachage

Inconvénients du hachage statique

- **Très important** : *h* doit répartir uniformément les enregistrements dans les *n* fragments
  - Notre fonction est un contre-exemple : si une majorité de films commence par une même lettre ('L' par exemple) la répartition va être déséquilibrée.
- **Plus grave** : La structure simple décrite précédemment n'est pas **dynamique:** On ne peut pas changer un enregistrement de place
  - Donc il faut créer un chaînage de blocs quand un fragment déborde
  - Et donc les performances se dégradent...
  - Autre PBM: on ne peut pas changer le nombre de fragments car cela implique le changement de la fonction de répartition.
- Ces deux problèmes ont été résolu avec la variante de hachage linéaire

3.5 Hachage

Le hachage linéaire:

- L'apport du hachage linéaire est d'incrémenter à la fois le répertoire et les fragments *proportionnellement* aux besoins de stockage, en évitant le doublement systématique du répertoire.

4. Index dans Oracle

- Plusieurs types d'index dans Oracle presque tous mais
- Les types d'index les plus courants sous Oracle sont les suivants:
  - **Arbres équilibrés (Balanced arbre: B-arbre+) :**
    - Les index B-Tree sont le type d'index par défaut quand on ne précise rien.
    - Comme son nom l'indique l'index B-Tree est organisé en arbre,
    - Toutes les branches de l'arbre ont la même longueur,
  - **Bitmap** :
    - Ce type d'index est particulièrement efficace lorsque le nombre de valeurs est petit ainsi que pour les opérations AND et OR.

4.1 Création

- Indexation Explicite (suite):
  - Syntaxe:

```
CREATE [UNIQUE | BITMAP] INDEX
[<schema>.]<nom index>
ON <nom de table>
(<nom de colonne> [ASC|DESC]
[,<nom de colonne> [ASC|DESC]]
, ...)
```

  - UNIQUE permet de créer un index B-Tree qui ne supporte pas les doublons.
  - BITMAP crée un index « chaîne de bits ».
  - ASC et DESC précisent l'ordre (croissant ou décroissant).
- Exemple 1 (suite)

```
CREATE INDEX IndTitre
ON Film (titre);
```

4.2 Suppression et recréation

- Suppression d'un index

```
DROP INDEX nom-index
```

- Recréer un index

```
ALTER INDEX nom-index REBUILD
```


4.3 Index basés sur des fonctions

- Une fonction de calcul (expressions arithmétiques ou fonctions SQL, PL/SQL ou C) peut définir un index.
  - Celui-ci est dit « basé sur une fonction » (*function based index*).
- Dans le cas des fonctions SQL, il ne doit pas s'agir de fonctions de regroupement (SUM, COUNT, MAX, etc.).
- Ces index servent à accélérer les requêtes contenant un calcul pénalisant s'il est effectué sur de gros volumes de données.
- Dans l'exemple suivant, on accède beaucoup aux comptes bancaires sur la base du calcul bien connu de ceux qui sont souvent en rouge : (credit-debit)*(1+(txInt/100))-frais.

Un index basé sur une fonction peut être de type *B-tree* ou *bitmap*.

ROWID	Index fonction (credit-debit)*(1+(txInt/100))-agios
C	-7,29
D	228,67
A	450,14
B	574,1

CompteEpargne

ROWID	ncompte	titulaire	debit	credit	txInt	agios
A	C1	Guibert	560	1000	3.6	5.7
B	C2	Soutou	250	850	4.1	50.5
C	C3	Teste	40	45	4.2	12.5
D	C4	Albaric	670	900	3.9	10.3

Exemple

CREATE TABLE CompteEpargne
(ncompte VARCHAR(4), titulaire VARCHAR(30), debit NUMBER(10,2),
credit NUMBER(10,2), txInt NUMBER(2,1), agios NUMBER(5,2));

```
CREATE UNIQUE INDEX
 idx_titulaire_CompteEpargne
 ON CompteEpargne (titulaire DESC);
```
Index *B-tree*, ordre inverse.

```
CREATE INDEX
 idx_debitenFF_CompteEpargne
 ON CompteEpargne (debit*6.56);
```
Index *B-tree*, expression d'une colonne.

```
CREATE BITMAP INDEX
 idx_bitmap_txInt_CompteEpargne
 ON CompteEpargne (txInt);
```
Index *bitmap*.

```
CREATE INDEX
 idx_fct_Solde_CompteEpargne
 ON CompteEpargne
 ((credit-debit)*(1+(txInt/100))-agios,
 credit, debit, txInt, agios);
```
Index basé sur une fonction.

4.4 Vues dans le dictionnaire de données

- Pour vérifier l'existence d'index :
  - Consulter les vues (ORACLE) dans le **DICTIONNAIRE DE DONNEES**
    - **USER_INDEXES:** Index créés par l'utilisateur
    - **USER_IND_COLUMNS**
    - **ALL_INDEXES**
    - **ALL_IND_COLUMNS**
    - **DBA_INDEXES**
- Les vues DBA contiennent des informations sur les objets de tous les schémas.
- Les vues ALL incluent les enregistrements des vues USER et des informations sur les objets pour lesquels des privilèges ont été octroyés au groupe PUBLIC ou à l'utilisateur courant.
- Les vues USER contiennent des informations sur les objets appartenant au compte qui exécute la requête.

5. Bilan (1/2)

- On choisira de créer un index sur :
  - Les attributs utilisés comme critère de jointure,
  - Les attributs servant souvent de critères de sélection,
  - Sur une table de gros volume (d'autant plus intéressant si les requêtes sélectionnent peu de lignes).
  - Les index *bitmaps* sont conseillés quand il y a peu de valeurs distinctes de la (ou des) colonne(s) à indexer.
  - Dans le cas inverse, utilisez un index *B-tree*.
- L'adjonction d'index accélère la recherche des lignes. Mais …

5. Bilan (2/2)

- Impact négatif sur la mise à jour de la table.
  - Impact négatif sur les commandes d'insertion et de suppression
    - Mise à jour de tous les index portant sur la table ➔ à coût.
  - Attributs souvent modifiés (index à recréer...) vu le coût des opérations de mise à jour des index
  - Attributs ne faisant pas objet d'interrogations fréquentes
  - Table de petit volume,
  - Si requêtes sur NULL car les NULL, ne sont pas stockées dans l'index. (ex : WHERE ... IS NULL).

Exercices

1. Produits (Modele: *entier, Constructeur : caractère, Type : chaine,)*
2. PC (npc: *entier, Vitesse: entier, RAM: entier, HD: réel, CD: chaine, Prix: réel, #modele: entier)*
3. Portables (nport: *entier, Vitesse: entier, RAM: entier, HD: réel, Ecran: réel, Prix: réel, #modele: entier)*
4. Imprimantes (nimp : *entier, Couleur : booléen, Type : chaine, Prix : réel, #modele: entier)*

Contraintes
$Les\ clefs\ donnees\ ci-dessus$
$Pas\ de\ valeur\ nulle$
$PC[Modele] \subseteq Produits[modele]$ $Portables[Modele] \subseteq Produits[modele]$ $Imprimantes[Modele] \subseteq Produits[modele]$
$dom(Produits.Type) = \{laptop, pc, printer\}$ $dom(PC.CD) = \{6x, 8x\}$ $dom(Imprimantes.Type) = \{dry, ink-jet, laser\}$

Créer un index sur les types de produits

```
CREATE BITMAP INDEX Prodtype_index ON
Produits (type);
```


Exercice 2

- Soit les deux tables suivantes:
  - create table R (idR varchar(20) not null,
    primary key (idR));
  - create table S (idS int not null,
    idR varchar(20) not null,
    primary key (idS),
    foreign key idR references R);

Dites si la requête est optimisée par l'index

1. select * from R where idR = 'Bou'
   ➔ optimisée par l'index sur la clé primaire
2. select * from R where idR like 'B%'
   ➔ Recherche par préfixe sur la clé primaire: idem.
3. select * from R where length(idR) = 3
   ➔ La troisième applique une fonction à la clé: l'index ne peut pas être utilisé car la critère de recherche n'est plus une valeur de clé, sauf si on crée un index sur les valeurs de la fonction.

Exercice 2 (suite)

- Soit les deux tables suivantes:
  - create table R (idR varchar(20) not null,
    primary key (idR));
  - create table S (idS int not null,
    idR varchar(20) not null,
    primary key (idS),
    foreign key idR references R);

4. select * from R where idR like '_ou'

➔La quatrième est une recherche par suffixe: l'index n'est pas utilisable

5. insert into S values (1, 'Bou')

➔L'insertion tire partie de l'index sur la clé primaire: il faut vérifier que la valeur de clé n'existe pas déjà. En l'absence d'index il faudrait parcourir toute la table à chaque insertion.

6. Select * from S where idS between 10 and 20

➔La requête par intervalle peut tirer partie de l'index, si l'intervalle est limité.

7. Delete from R where idR like 'Z%'

➔Enfin la destruction dans R implique la vérification qu'il n'existe pas de nuplet référençant dans S, sinon la contrainte d'intégrité serait violée. L'index sur la clé étrangère est ici très utile.

Exercice 3

- Déterminer si les requêtes suivantes bénéficieront d'index? De quel type? Sur quelle(s) colonne(s)?
  - SELECT * FROM emp WHERE nom LIKE 'M%'
    - La requête suivante bénéficiera d'un index sur la colonne nom.
  - SELECT * FROM emp WHERE nom LIKE '?????????' ;
    - La requête suivante ne bénéficiera pas d'un index sur la colonne nom.
      - Valeurs NULL
- Soit le schéma relationnel suivant:
  - personne (<u>nump,</u> nom, prenom, localisation, departement, unite)
  - Implémentez une solution d'optimisation permettant d'améliorer la vitesse d'exécution suivante :
    - SELECT * FROM Personne ORDER BY Nom, Prenom
      - CREATE INDEX myindex ON Personne (Nom, Prenom) ;

Références

Livres:

1. C. MAREE et G. LEDANT, SQL-2 : Initiation, Programmation, 2ème édition, Armand Colin, 1994, Paris.
2. P. DELMAL, SQL2 – SQL3 : application à Oracle, 3ème édition, De Boeck Université, 2001, Bruxelles.
3. C.SOUTOU: SQL pour Oracle, 3ème et 7ème édition, Eyrolles, Paris.

Supports de cours

1. P.Rigaux: Sructures d'index
2. D.BAYERS et L.SWINNEN, Laboratoire 5:Les vues et contraintes