Cours
"Bases de données"

Stockage des données, indexation, optimisation des requêtes

3° année (MISI)

Antoine Cornuéjols

www.lri.fr/~antoine

antoine.cornuejols@agroparistech.fr

Stockage, indexation, optimisation

1. ***Stockage des données***
   - 1.1. Supports physiques
   - 1.2. Techniques de stockage
2. ***Structures de données***
   - 2.1. Indexation de fichiers
   - 2.2. L'arbre-B
   - 2.3. Hachage
   - 2.4. Les index *bitmap*
   - 2.5. Structures de données multidimensionnelles
3. ***Évaluation des requêtes***
   - 3.1. Algorithmes de base
   - 3.2. Jointures
   - 3.3. Compilation d'une requête et optimisation

Stockage, indexation, optimisation

1. ***Stockage des données***
   - 1.1. Supports physiques
   - 1.2. Techniques de stockage
2. ***Structures de données***
   - 2.1. Indexation de fichiers
   - 2.2. L'arbre-B
   - 2.3. Hachage
   - 2.4. Les index *bitmap*
   - 2.5. Structures de données multidimensionnelles
3. ***Évaluation des requêtes***
   - 3.1. Algorithmes de base
   - 3.2. Jointures
   - 3.3. Compilation d'une requête et optimisation

Organisation des données

Conception physique de la base

Les bases de données sont généralement trop volumineuses pour tenir entièrement en mémoire centrale.

La ***conception physique de la base de données*** et le choix des solutions techniques est à la charge des concepteurs, des administrateurs et des développeurs du SGBD.

Chaque fois qu'une portion de données est nécessaire, elle doit être localisée sur le disque (à l'intérieur d'un *segment*), copiée en mémoire centrale pour être traitée puis réécrite sur le disque si des modifications ont été apportées.

Les données stockées sur le disque sont organisées en **fichiers d'enregistrements**.

- Il faut pouvoir les **localiser rapidement**
- et **minimiser le nombre d'accès** car la mémoire secondaire est lente.

Organisation des données

Conception physique de la base

Segments et pages

Les fichiers sont décomposés en *pages*

- Portions de taille fixe d'informations contiguës dans le fichier
- Unité d'échange entre le disque et la mémoire principale

Le disque est divisé en *segments* [physique] (de la taille d'une page [logique])

- Une page [logique] peut être stockée dans n'importe quel bloc [physique]

Les demandes des applications (e.g. SGBD) **pour lire un item** sont satisfaites par :

- la lecture de la page contenant l'item dans le buffer du SGBD
- le transfert de l'item du buffer vers l'application

Les demandes du SGBD **pour modifier un item** sont satisfaites par :

- la lecture de la page contenant l'item dans le buffer du SGBD (si pas encore là)
- la modification de l'item dans la mémoire du SGBD
- la copie de la page du buffer sur le disque

Organisation des données

Conception physique de la base

Réduction de la "latence"

Stocker les pages contenant des informations corrélées sur des segments proches physiquement sur le disque

- *Justification* : si le SGBD accède à *x*, il est probable qu'il va accéder ensuite à des données reliées à *x*

Compromis sur la taille des pages

- ***Grande taille*** -> les données liées à *x* ont des chances d'être stockées sur la même page, d'où réduction des accès aux pages
- ***Petite taille*** -> réduit le temps de transfert et réduit la taille du buffer en mémoire centrale
- **Taille typique** : 4096 octets

Organisation des données

Conception physique de la base

Réduction du nombre de transferts de pages

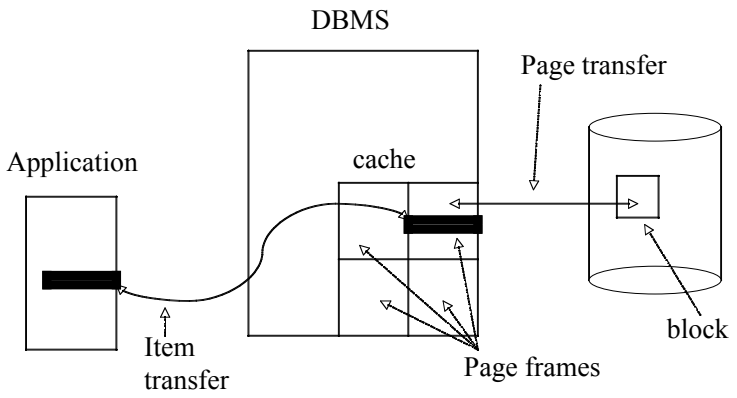
Conserver en cache les pages récemment traitées

- *Justification* : les demandes de pages peuvent alors être satisfaites à partir de la mémoire cache plutôt qu'à partir du disque (beaucoup plus rapide)
- On purge la mémoire cache lorsqu'elle est pleine

Organisation des données

Conception physique de la base

Accès aux données par une mémoire cache



Organisation des données

Conception physique de la base

Les systèmes RAID

Un ***système RAID*** (Redundant Array of Independent Disks) est un ensemble de disques configurés de telle manière qu'ils apparaissent comme un seul disque avec :

- Une **vitesse de transfert accrue**
  - Des demandes à des disques différents peuvent être traitées indépendamment
  - Si une requête concerne des données stockées séparément sur plusieurs disques, les données peuvent être transférées en parallèle
- Une **plus grande fiabilité**
  - Les données sont stockées de manière redondante
  - Si un disque tombe en panne, le système peut continuer à opérer

Organisation des données

Conception physique de la base

À retenir

Ce sont les

- temps d'accès relatifs aux mémoires primaires et secondaires
- et les capacités mémoire

qui motivent les techniques d'indexation développées

Disque dur :

- Temps de recherche ≈ 45 à 60 ms
- Temps de lecture ≈ 1 à 2 ms

Mémoire centrale :

- Temps de recherche ≈ 10^{-6} s
- Temps de lecture ≈ 10^{-6} s

Organisation des données

Opérations sur les fichiers

Les ***opérations sur les fichiers*** concernent :

- les **extractions**
- les **mises-à-jour**

Ces opérations impliquent des sélections en fonction de ***conditions de sélection*** ou de ***filtrage***.

Il existe des :

- **conditions simples** (e.g. sélection *sur un attribut*)
- **conditions complexes** (e.g. sélection *sur plusieurs attributs*)

Une condition complexe doit être décomposée en conditions simples.

Organisation des données

Opérations sur les fichiers

Opérations couramment disponibles dans les SGBD :

- ***Open***. Ouvre le fichier pour lecture ou écriture.
- ***Reset***. Positionne le pointeur de fichier d'un fichier ouvert en début de fichier.
- ***Find*** (ou ***Locate***). Recherche le premier enregistrement qui satisfait la condition de recherche et le met dans le *buffer*.
- ***Read*** ou ***Get***. Copie l'enregistrement contenu dans le *buffer* dans la variable de programme du programme utilisateur.
- ***FindNext***. Recherche le prochain enregistrement qui satisfait la condition de recherche et le met dans le *buffer*.
- ***Delete***. Efface l'enregistrement courant et met à jour le fichier disque.
- ***Modify***. Modifie la valeur d'un ou plusieurs champs de l'enregistrement courant et met à jour le fichier disque.
- ***Insert***. Insère un nouvel enregistrement dans le fichier.
- ***Close***. Ferme l'accès au fichier en vidant les *buffers* et en libérant les ressources.

Organisation des données

Opérations sur les fichiers

Optimisation

Une organisation de fichier réussie devrait pouvoir *exécuter efficacement* les opérations que nous pensons *appliquer fréquemment* au fichier.

Exemples :

- Si condition de recherche fréquente sur le numéro de Sécurité Sociale, il faut favoriser la localisation par rapport à la valeur de `NoSS`.
  ==> Soit trier par rapport à `NoSS`, soit définir un index sur `NoSS`.
- Si, en plus, recherche d'employés par service, il faudrait stocker tous les enregistrements des employés dont la valeur de `SERVICE` est identique de façon contiguë. Mais cet arrangement est en conflit avec les enregistrements classés sur la valeur de `NoSS`.

  ==> Il faut trouver un compromis : **optimisation**

Stockage, indexation, optimisation

Organisation des données

Conception logique de la base

Il existe plusieurs organisations primaires de fichiers qui déterminent la façon dont les enregistrements sont *placés physiquement* sur le disque, et *donc la façon dont il est possible d'accéder à ces enregistrements*.

- Dans un ***fichier plat*** (ou ***fichier non ordonné***) les enregistrements sont placés sur le disque sans observer d'ordre particulier et les nouveaux enregistrements sont simplement ajoutés en fin de fichier.
- Dans un ***fichier trié*** (ou ***fichier séquentiel***), ils sont ordonnés en fonction de la valeur d'un champ particulier nommé clé de tri.
- Un ***fichier haché*** utilise une fonction de hachage appliquée à un champ particulier (la *clé de hachage*).
- Les ***arbres-B*** utilisent des structures arborescentes.
- Les ***organisations secondaires***, ou ***structures d'accès auxiliaire***, permettent d'accéder rapidement aux enregistrements en se basant sur d'autres champs que ceux qui ont servi pour l'organisation primaire. La plupart de ceux-ci sont des *index*.

Organisation des données

Fichiers non ordonnés ("*heap files*")

Type d'organisation le plus simple.

Les enregistrements sont placés dans le fichier dans l'ordre de leur insertion.

Insertions/modifications d'un tuple :

- En moyenne $F/2$ pages sont lues (donc transférées) si la page existe déjà
- $F+1$ pages transférées si la page n'existe pas

Effacement d'un tuple :

- En moyenne $F/2$ pages sont lues (donc transférées) si la page existe déjà
- F pages transférées si la page n'existe pas

Organisation des données

Fichiers ordonnés ("*sorted files*")

Les enregistrements sont ordonnés physiquement sur disque en fonction des valeurs d'un de leurs champs (*champ d'ordonnancement*).

Si le champ d'ordonnancement est la clé primaire, le champ est appelé *clé d'ordonnancement.*

Avantages :

- Les requêtes basées sur le champ d'ordonnancement ont un coût moyen de log_2F par **recherche binaire**
- Si les requêtes concernent des tuples successifs, l'**utilisation de cache** est très efficace

Aucun avantage si l'accès se fait par d'autres attributs que le champ d'ordonnancement.

Organisation des données

Fichiers ordonnés ("*sorted files*")

Problème (maintien de l'ordre):

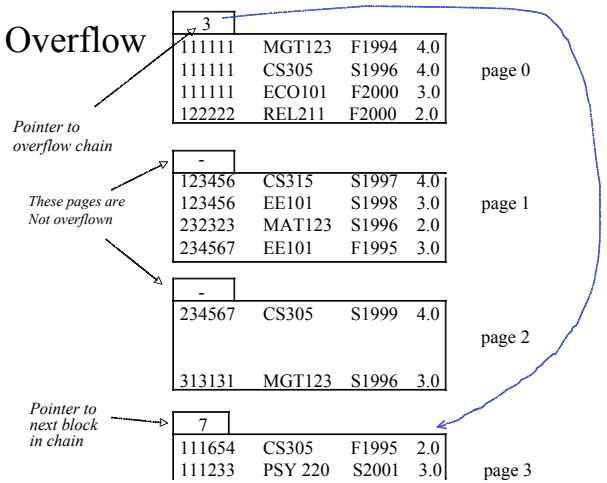
- ***Coût moyen*** : $F/2$ lectures (pour trouver l'endroit) + $F/2$ écritures (afin de pousser les enregistrements suivants

- **Solution partielle 1** : Laisser des espaces libres

- **Solution partielle 2** : Utilisation de pages d'*overflow* (et tri de temps en temps)
  - ***Désavantages***
    - Les pages successives ne sont plus nécessairement stockées à la suite
    - Les pages overflow ne sont pas dans l'ordre

Organisation des données

Fichiers ordonnés ("*sorted files*")



Organisation des données

Utilisation d'index

Mécanisme permettant de localiser efficacement les tuples (ou pages) sans avoir à scanner l'ensemble de la table (ou du fichier).

Basé sur une ou plusieurs *clés de recherche*.

Les entrées d'index peuvent contenir :

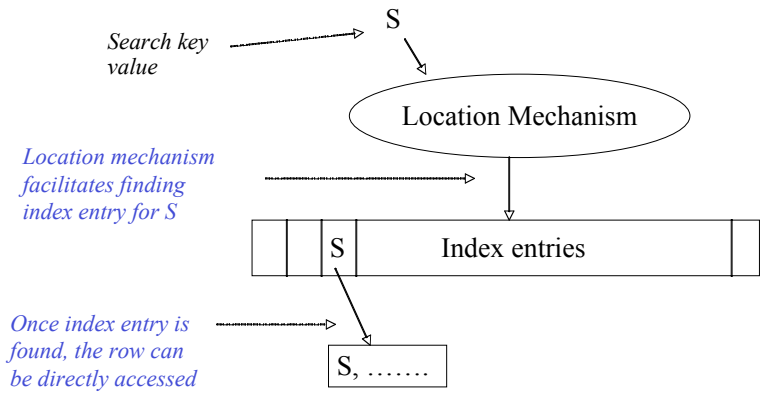
- les données elles-mêmes (l'index et le fichier sont dits *intégrés*)
- la valeur de la clé de recherche et le pointeur vers un enregistrement ayant cette valeur (l'index est dit *non intégré*).

Les entrées d'index sont stockée suivant les valeurs de la clé de recherche :

- des entrées avec la même clé de recherche sont stockées ensemble (hachage, B-arbres)
- les entrées peuvent être triées suivant la valeur de la clé (B-arbres)

Organisation des données

Utilisation d'index



Organisation des données

Structures de stockage

Index intégré

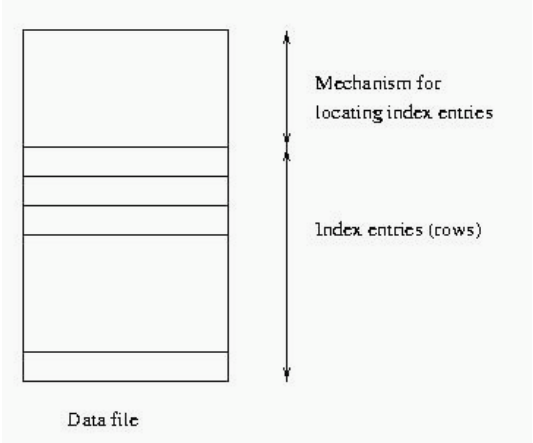
- Fichiers non triés ("*heap files*")
- Fichiers triés ("*sorted files*")
- Fichiers intégrés contenant l'index et les enregistrements
  - ISAM (*Index Sequential Access Method*)
  - B+ arbres
  - Hachage

Organisation des données

Structures de stockage

Index intégré

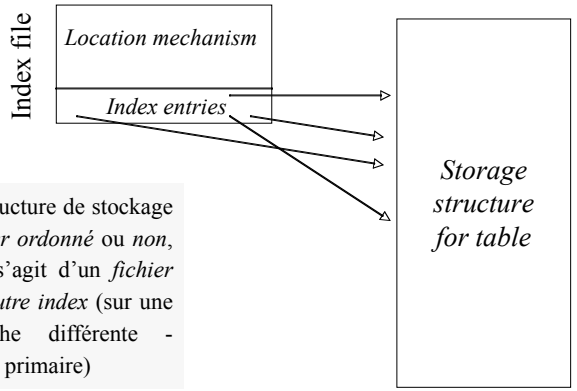
Contient la table et l'index (principal)



Organisation des données

Structures de stockage

Index avec une structure de stockage séparée



Dans ce cas, la structure de stockage peut être un *fichier ordonné* ou *non*, mais **souvent** il s'agit d'un *fichier intégré avec un autre index* (sur une clé de recherche différente - typiquement la clé primaire)

Organisation des données

Structures de stockage

Les index de regroupement (clusters index)

Organisation des données

Structures de stockage

Les index denses et les index clairsemés

Index dense :

- Contient une entrée d'index pour chacune des valeurs de la clé de recherche, et donc pour chacun des enregistrements, du fichier de données.

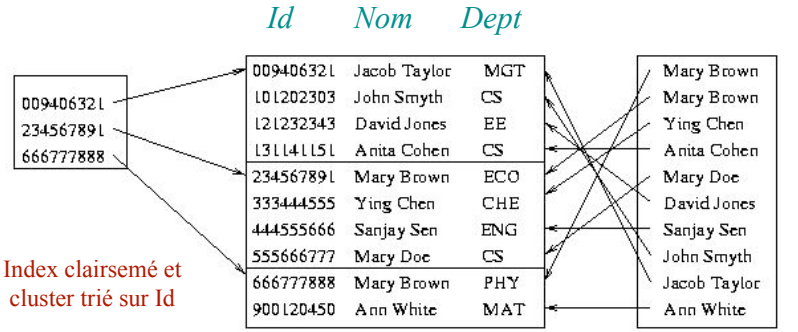
Index clairsemé :

- Ne contient d'entrées d'index que pour certaines des valeurs sur lesquelles il y a des recherches à effectuer.

Organisation des données

Structures de stockage

Les index denses et les index clairsemés



Index clairsemé et cluster trié sur Id

Fichier de données trié sur les Id

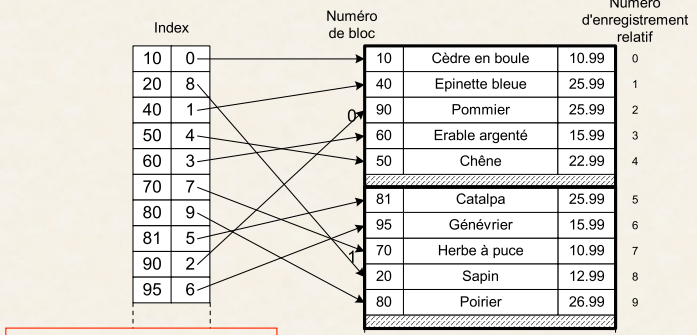
Index dense et non cluster trié sur Nom

8 Organisations unidimentionnelles : indexage et hachage

- *Sélection basée sur une clé d'accès*
  - recherche associative
  - Ex: Chercher le plant dont le *noCatalogue* = 10
- *Sériel*
  - lire tout le fichier en pire cas
  - *O*(N)
- *Indexage*
  - *O*(log(N))
  - *sélection par intervalle*
- *Hachage*
  - ~*O*(1)

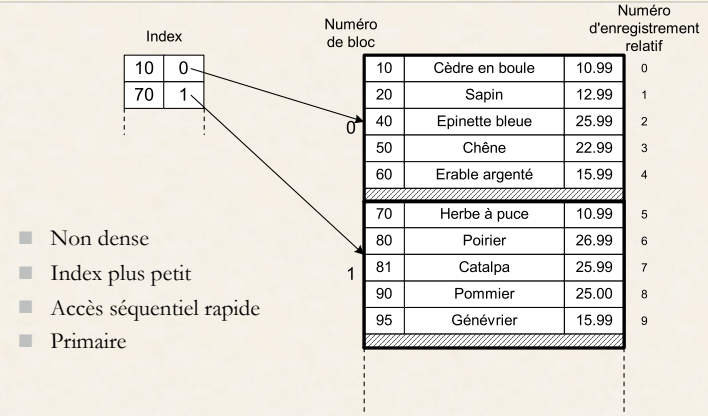
8.1 Indexage

- *Index* et *clé d'index* (*index key*)
  - valeur de la clé =>adresse de(s) l'enregistrement



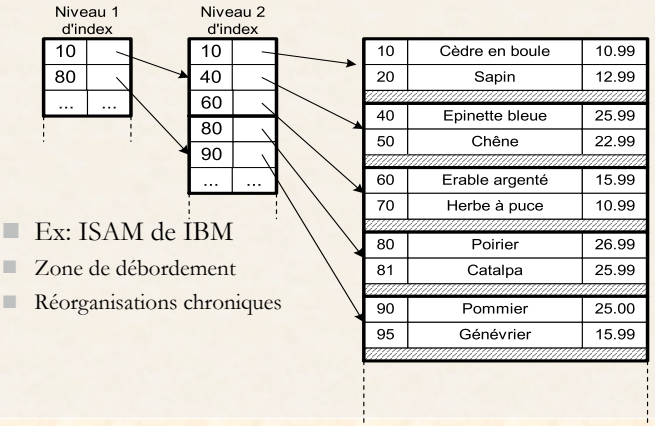
Index dense secondaire

Fichier séquentiel indexé



- Non dense
- Index plus petit
- Accès séquentiel rapide
- Primaire

Index séquentiel hiérarchique

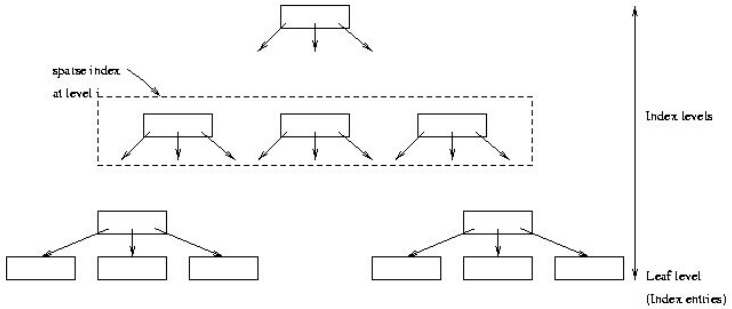


- Ex: ISAM de IBM
- Zone de débordement
- Réorganisations chroniques

Organisation des données

Structures de stockage

Les index multi-niveau



Coût de la recherche = nombre de niveaux

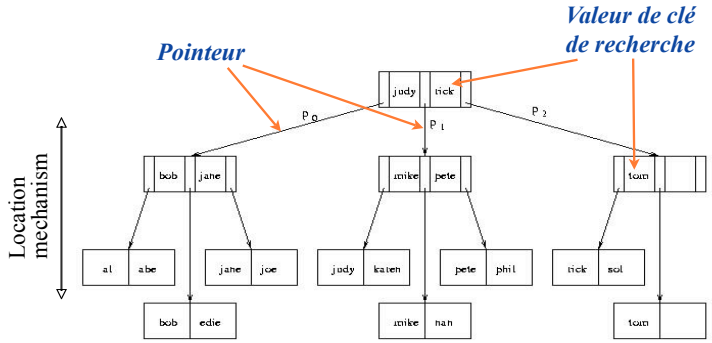
Si facteur de branchement = *b*, coût = $log_b F + 1$

E.g. : *F= 10000* et *b = 100, coût = 3*

Organisation des données

Structures de stockage

Index Sequential Access Method (ISAM)



Organisation des données

Structures de stockage

Index Sequential Access Method (ISAM)

Index statique :

- Les niveaux de séparation ne changent pas
- Le nombre et la position des feuilles restent fixes

Performant pour :

- les recherches d'égalité
- les recherches d'intervalles

Organisation des données

Structures de stockage

Les arbres B+

Soutiennent :

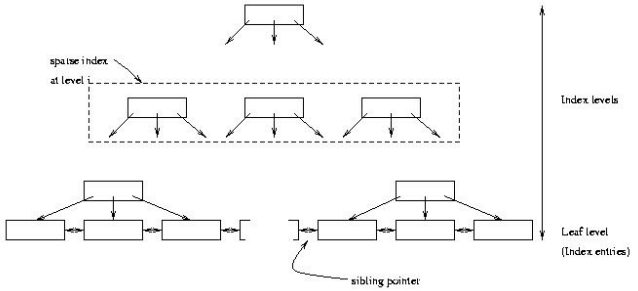
- les recherches d'égalité et d'intervalle
- les clés à attributs multiples
- les recherches à clé partielle

- **Dynamiques**

Organisation des données

Structures de stockage

Les arbres B+



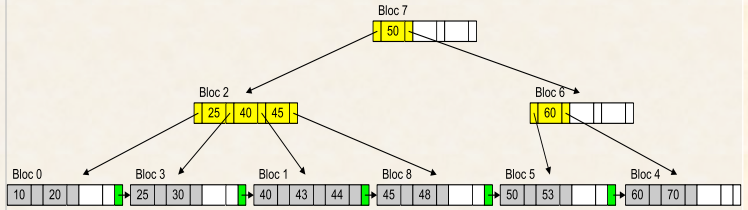
- Le niveau feuille est une liste (triée) chaînée des entrées d'index
- Les pointeurs entre frères permettent la recherche par intervalle, même si allocation et dé-allocation de feuilles (les feuilles n'ayant pas à être physiquement contiguës)

8.1.1 Indexage par Arbre-B et variantes

- *Arbre-B* (*B-arbre, B-tree*)
  - forme d 'index hiérarchique
  - équilibré
  - $O(log(N))$ en pire cas
- Réoganisation dynamique
  - division/fusion des blocs
  - taux d 'occupation minimum de 50%

8.1.2 Arbre-B+

- Hypothèse initiale : clé simple et unique
- Nœud = bloc



Structure d 'une feuille

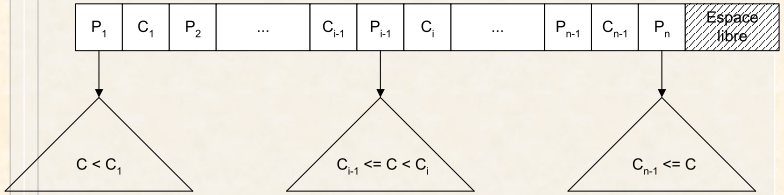
- 1. Remplie à moitié au minimum $\lceil FBM_f/2 \rceil \leq n = nombre\ de\ clés \leq FBM_f$
- 2. Clés triées : i < j ⇒ $C_i < C_j$
- 3. Clés d'une feuille < clés de la suivante
- 4. Au même niveau (équilibré)

C_1	R_1	...	C_n	R_n	Espace libre	S

- C_i : Clé
- R_i : reste de l'enregistrement ou référence
- S : Pointeur sur le bloc suivant dans la liste des feuilles

Structure d'un bloc interne

- 1. Remplie à moitié au minimum:
  - $\lceil Ordre_I /2 \rceil \leq n = nombre\ de\ pointeurs \leq Ordre_I$
- 2. Clés triées : i < j ⇒ $C_i < C_j$
- 3. C_{i-1} <= Clés sous $P_{i-1} < C_i$



8.1.2.1 Recherche dans un arbre-B+

RechercherClé (*noBloc*, *clé*, *indice*)

Entrée
- *noBloc* : typeNuméroBloc {numéro du bloc à chercher; racine au premier appel}
- *clé* : typeClé {la clé à chercher}

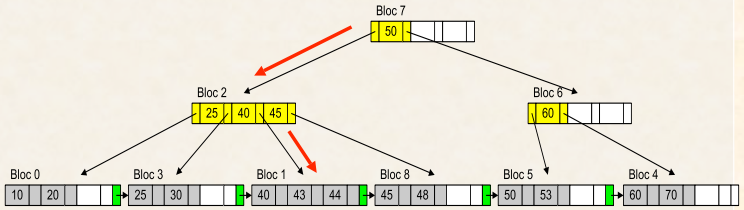
Sortie
- *noBloc* : typeNuméroBloc {numéro du bloc où est la clé }
- *indice* : 1..FBM_F { si clé trouvée, indice de la clé dans le tableauClés du bloc}
- valeur de retour : BOOLÉEN;{vrai si clé a été trouvée}

Pré-condition : arbre n'est pas vide

DÉBUT
- *lireBloc*(*fichierArbre-BPlus*, *noBloc*, *bloc*);
- TANT QUE bloc n'est pas une feuille
  - *indice* := l'indice de la plus petite clé de *tableauClés* plus grande que clé (ou *n*);
  - *noBloc*:= *tableauEnfants*[*indice*];
  - *lireBloc*(*fichierArbre-BPlus*, *noBloc*, *bloc*);
- FIN TANT QUE;
- SI la *clé* est présente dans *tableauClés*
  - *indice* := l'*indice* de la *clé* dans le *tableauClés*;
  - retourner VRAI;
- SINON
  - retourner FAUX
- FIN SI

FIN

Rechercher 43



8.1.2.2 Complexité de la recherche et hauteur de l'arbre

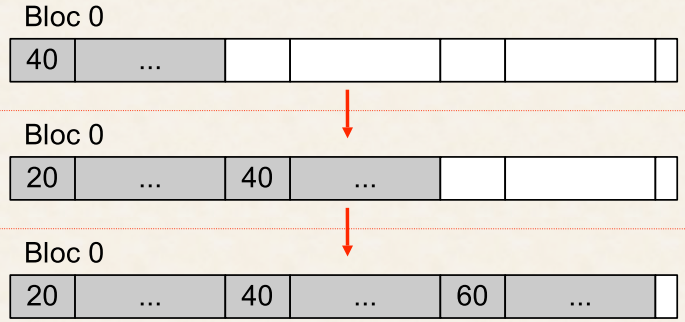
- $FBM_f = 20$ et $Ordre_I = 200$
- Hauteur = nombre de niveaux
- Hauteur 2 N ≥ 2 * 10 = 20 clés (pire cas)
- Hauteur 3 N ≥ 2 * 100 * 10 = 2 000 clés
- Hauteur 4 N ≥ 2 * 100 * 100 * 10 = 200 000 clés
- Hauteur 5 N ≥ 2 * 100 * 100 * 100 * 10 = 20 000 000 clés
- Hauteur H $N \geq 2 * \lceil Ordre_I / 2 \rceil^{H-2} * \lceil FBM_f / 2 \rceil$ pour $H \geq 2$
- $H \leq 2 + log_{\lceil Ordre_I / 2 \rceil} (N / (2 * \lceil FBM_f / 2 \rceil))$
  - $\mathbf{O}(log\ N)$

Hauteur moyenne

- $H \sim 1 + \lceil log_{OrdreMoyenI} (N / FB_f) \rceil$
  - $OrdreMoyen_I = \lfloor 2/3\ Ordre_I \rfloor$
  - $FB_f = \lfloor 2/3\ FBM_f \rfloor$
- Index secondaire
  - $FB_f \sim OrdreMoyen_I$
  - $H = \lceil log_{OrdreMoyenI} (N) \rceil$

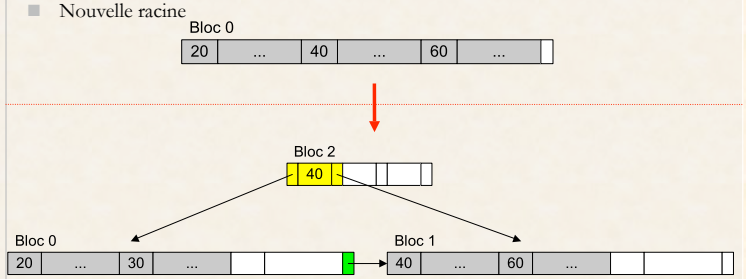
8.1.2.3 Insertion dans un arbre-B+

- $FBM = 3$, $Ordre_I = 4$

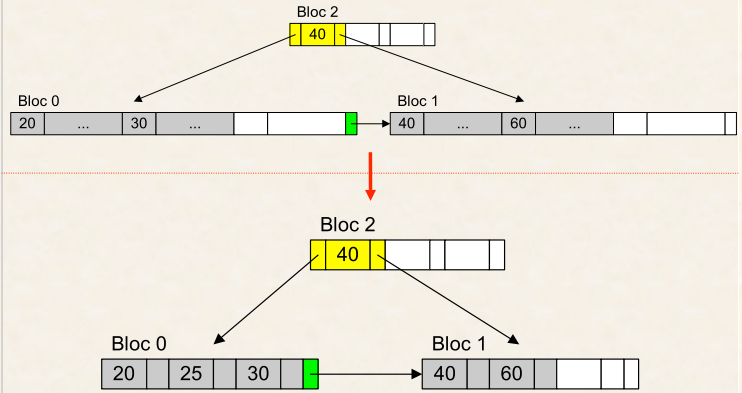


Débordement et division

- Insertion de 30
- Débordement et la division du bloc 0
- 40 est promue
- Nouvelle racine

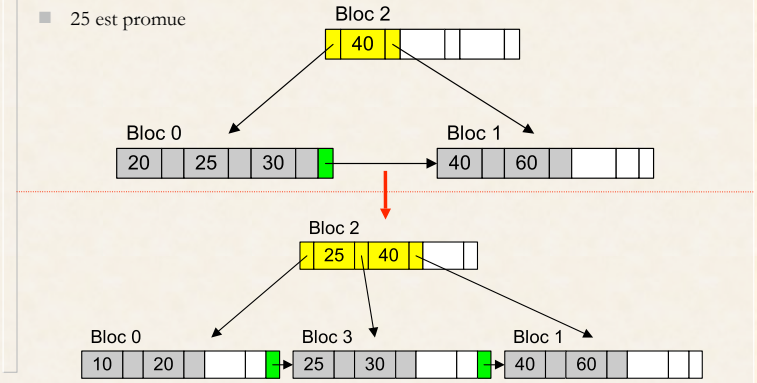


Insertion de 25

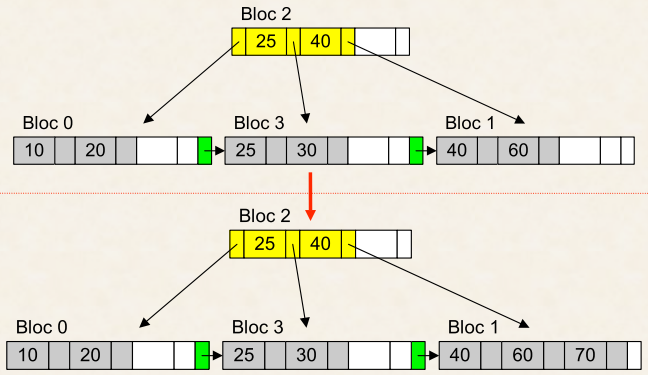


Insertion de 10

- Débordement et la division du bloc 0
- 25 est promue

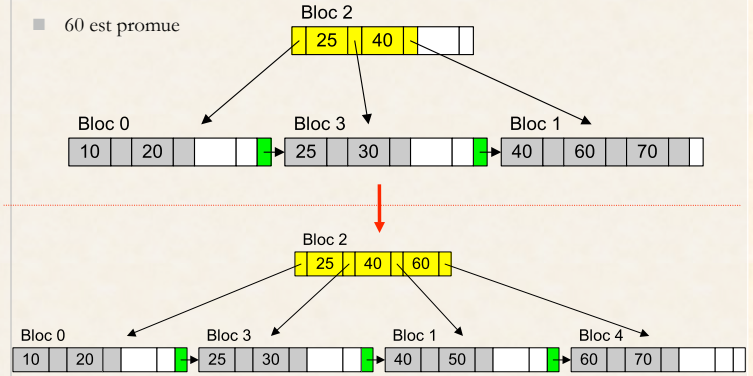


Insertion de 70

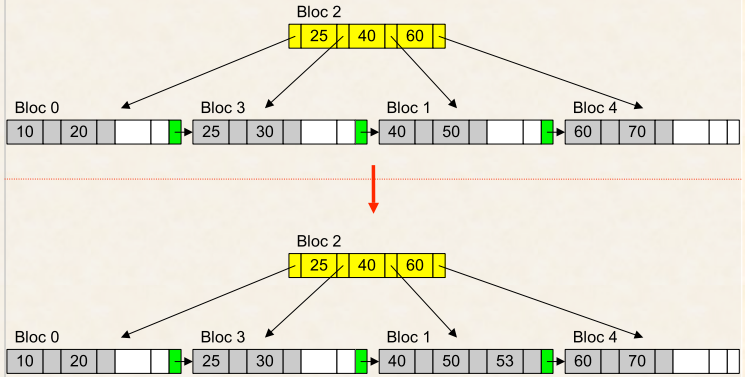


Insertion de 50

- Débordement et la division du bloc 1
- 60 est promue

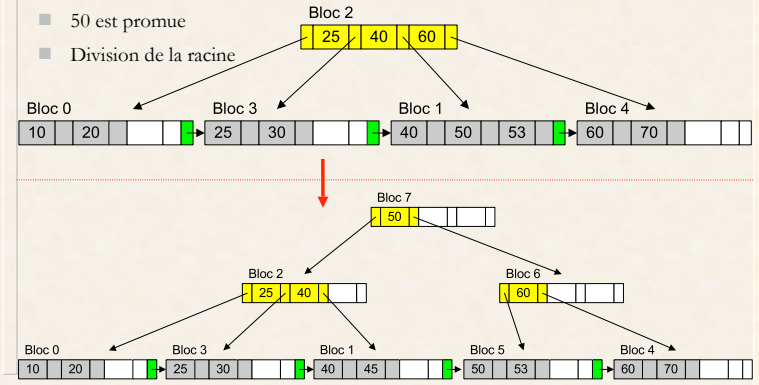


Insertion de 53



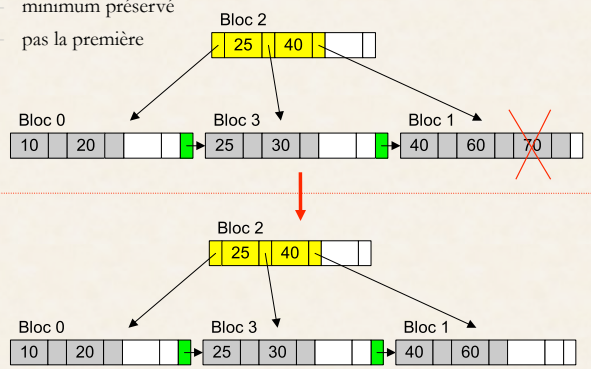
Insertion de 45

- Division du bloc 1
- 50 est promue
- Division de la racine



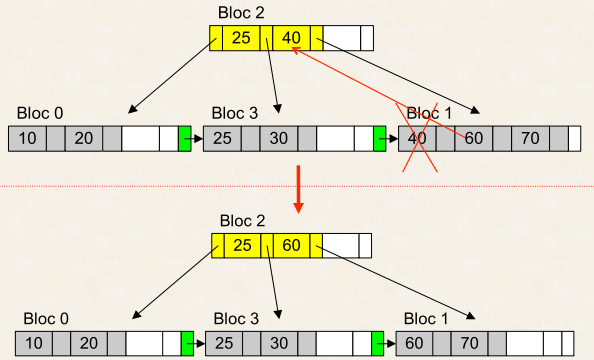
8.1.2.4 Suppression dans un arbre-B+

- Cas simple
  - minimum préservé
  - pas la première



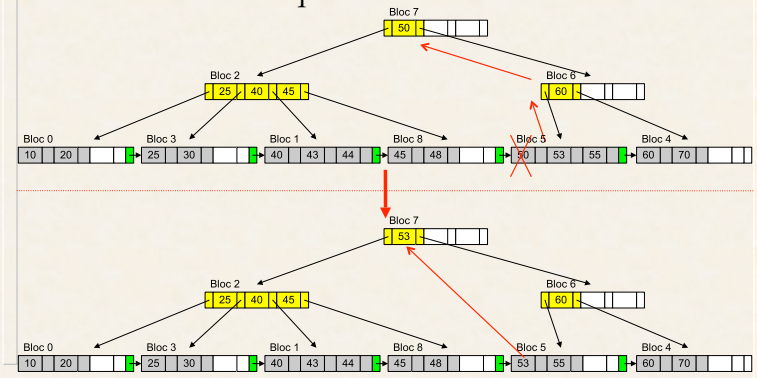
Première clé du bloc et pas la première feuille

- Remplacer dans le parent (si pas « aîné »)



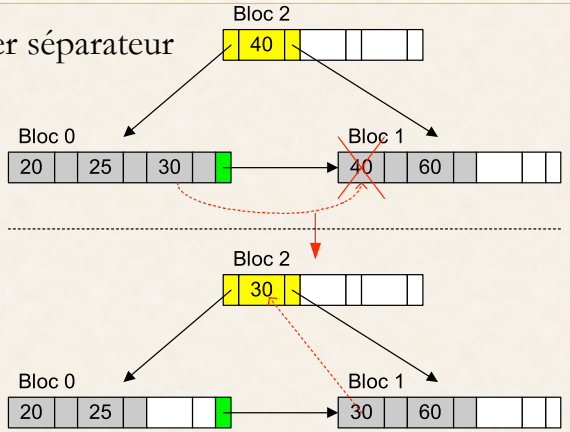
Première clé du bloc et pas la première feuille

- Remonter tant que l'enfant est l '« aîné »

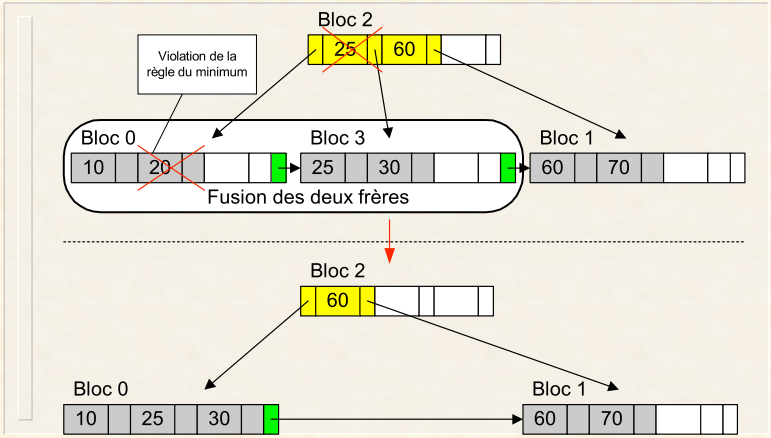


Violation du minimum : redistribution si possible

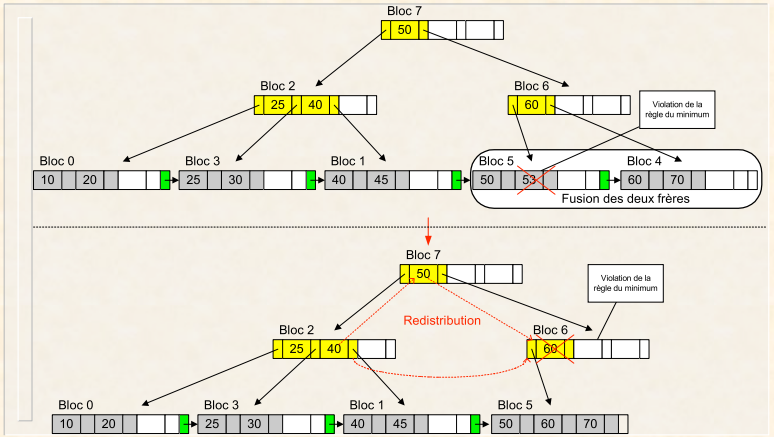
- Ajuster séparateur



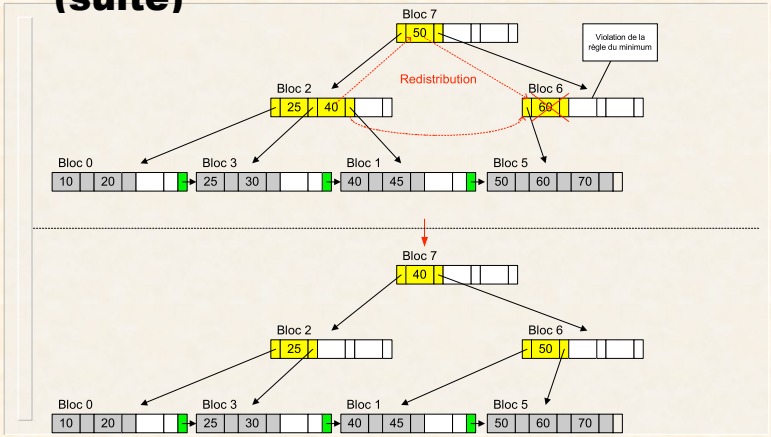
Violation du minimum : fusion



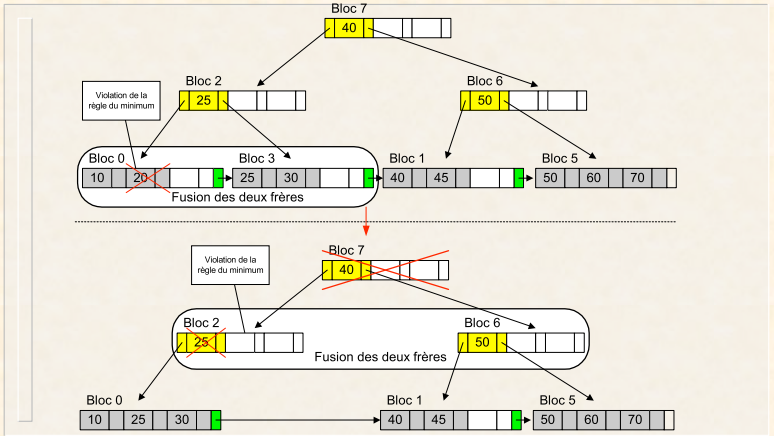
Cas de fusion de feuilles et de redistribution au niveau du parent



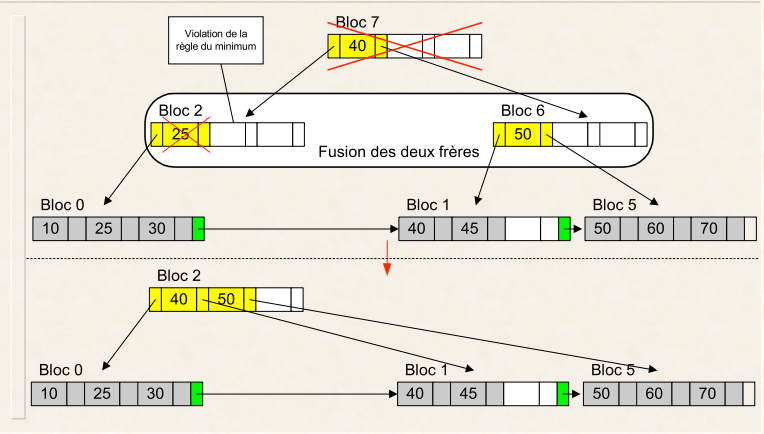
Cas de fusion de feuilles et de redistribution au niveau du parent (suite)



Cas de fusion en cascade

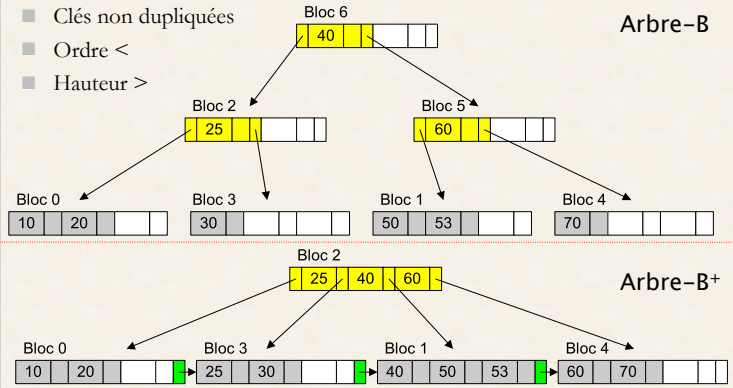


Cas de fusion en cascade (suite) : réduction de la hauteur



8.1.3 Arbre-B

- Clés non dupliquées
- Ordre <
- Hauteur >



8.1.4 Autres variantes du concept d'arbre-B

- Redistribuer plutôt que diviser
  - occupation moyenne 67% => 86%
- Diviser deux en trois
  - Arbre-B*
- Ordre variable
  - clés de taille variable
- Arbre B préfixe
  - comprimer les clés diminue la hauteur
- Algorithme de chargement en lot
  - feuilles consécutives
  - taux de remplissage prédéterminé

Cas d'une clé non unique

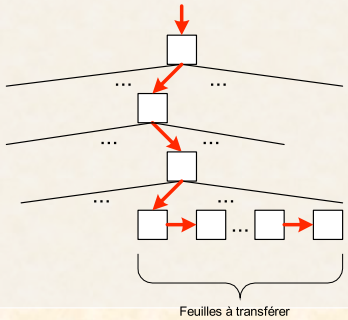
- Arbre-B+ primaire sur une clé non unique
  - IDE difficile
- Arbre B+ secondaire avec clés répétées
  - clé d 'accès + pointeur (unique)
- Arbre B+ secondaire avec collection de références
  - *listes inversées* dans les feuilles
- Arbre B+ secondaire avec référence à une collection d'enregistrements
  - *Index groupant* (" *clustering index* ")
    - organisation primaire par grappe et index secondaire sur même clé
- Arbre B+ secondaire avec référence à collection de références
  - listes inversées à part
- Arbre B+ avec vecteurs booléens
  - index « bitmap »

8.1.5 Réalisation de l'accès par IDE à l'aide d'une organisation par index

- Index primaire
  - IDE = *id_fichier, valeur de la clé unique*
  - nécessite le passage par l 'index
- IDE logique
  - index secondaire
  - clé d 'index = IDE

8.1.6 Sélection par intervalle ou préfixe

- Arbre B^+
  - recherche de la valeur minimale
  - parcours des feuilles jusqu 'à la valeur maximale



8.1.7 Index sur une clé composée

- Clé composée ~ clé simple formée de la concaténation des champs
- Sélection par préfixe de la clé composée

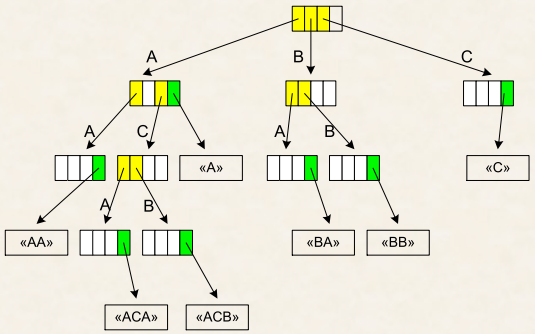
8.1.8 Index bitmap

indice	noPermis	sexe	couleurYeux	bitmap sexe = M	bitmap sexe = F	bitmap couleurYeux = bleu	bitmap couleurYeux = brun	bitmap couleurYeux = rouge
1	G111	M	brun	1	0	0	1	0
2	G555	M	bleu	1	0	1	0	0
3	G222	F	brun	0	1	0	1	0
4	G888	M	brun	1	0	0	1	0
5	G777	F	brun	0	1	0	1	0
6	G666	M	rouge	1	0	0	0	1
7	G333	F	bleu	0	1	1	0	0
8	G444	F	brun	0	1	0	1	0

- *couleurYeux* = *brun* et *sexe* = *M*
  - 10111001 ET 11010100 = 10010000

8.2 Arbre digital (trie)

- Chaque niveau : position d'un symbole de la clé vue comme une séquence de symboles $s_1 s_2 \ldots s_n$



8.3 Hachage

- *Hachage* **ou** *adressage dispersé* **(***hashing***)**
- Fonction *h(clé de hachage)* => l'adresse d'un *paquet*
- Fichier = tableau de *paquets* **(***bucket***)**
  - ~ARRAY paquet [0..*TH*-1]
  - *TH* : *taille de l'espace d'adressage primaire*
- Habituellement paquet = bloc
- Pas d 'index à traverser : *O*(1) en meilleur cas
- Sélection par égalité (pas intervalle)

8.3.1 Hachage statique

clé = 10

h(10) = 10 MOD 3 = 1

0	60	Erable argenté	15.99
	90	Pommier	25.99
	81	Catalpa	25.99
1	70	Herbe à puce	10.99
	40	Epinette bleue	25.99
	10	Cèdre en boule	10.99
2	20	Sapin	12.99
	50	Chêne	22.99
	95	Génévrier	15.99
	80	Poirier	26.99

8.3.1.1 Problème de débordement dû aux collisions

- Méthode de résolution des collisions
  - Adressage ouvert
    - AC+1, AC+2,....., *n*-1, 0, 1, ...AC-1
  - Chaînage

0	60	Erable argenté	15.99
	90	Pommier	25.99
	81	Catalpa	25.99
1	70	Herbe à puce	10.99
	40	Epinette bleue	25.99
	10	Cèdre en boule	10.99
	43	Magnolia	28.99
2	20	Sapin	12.99
	50	Chêne	22.99
	95	Génévrier	15.99
	80	Poirier	26.99

52	Pin	18.99

Fonction de hachage

- Répartition uniforme des clés dans [0..*TH*-1]
  - $h(clé) = clé \text{ MOD } TH$
    - *TH* est premier
  - $h(clé) = clé\ p \text{ MOD } TH$
    - *TH* et *p* sont relativement premiers
  - $h(clé) = (\sum s_i) \text{ MOD } TH$
    - s_i est une sous-séquence des bits de la clé
- Clé non numérique
  - représentation binaire vue comme un entier

Hachage vs indexage

- $O(1)$ en meilleur cas vs $O(\log(N))$
- Pas d 'espace supplémentaire d 'index
- Gaspillage d 'espace si *TH* trop >
- Performance dégradée si TH trop <
- Gestion plus délicate
  - déterminer *h* et *TH*
  - maintenance : réorganisations
- Clé non numérique ?
  - représentation binaire vue comme un entier

Calcul d 'espace

- Heuristique : *Taux d 'occupation moyen* ~ 80%
  - $TauxOccupation = N/(TH \times FB) \cong 0.8$
- *Taux de débordement moyen* sous distribution uniforme [Merrett, 1984 #217] :
  - $FB = 1$ ~ 30%
  - $FB = 10$ ~ 5%
  - $FB = 100$ ~1%

8.3.1.2 Fonction de hachage préservant la relation d'ordre

- (*tidy functions*)
- $clé_1 < clé_2 \Rightarrow h(clé_1) < h(clé_2)$
- Connaissances préalables au sujet de la distribution des clés

8.3.2 Hachage dynamique

- Adaptation de *TH* et *h* aux variations du volume des données
- ~ arbre-B
  - division et fusion de paquets (blocs)
- Deux variantes de base
  - *linéaire*
  - *extensible*

8.3.2.1 Hachage linéaire

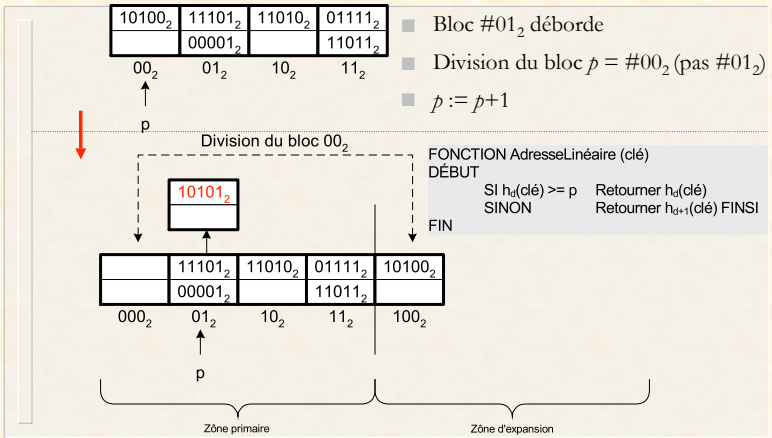
- Adaptation de *TH*
  - suite d 'expansions
- Début de *d*ième expansion, $d \in \{0, 1, \ldots\}$
  - *TH* passera de 2^d à 2^{d+1}
  - adresse du paquet : $h_d(clé) = b_{d-1}, b_{d-2}, \ldots, b_1, b_0$

d = 2

10100_2	11101_2	11010_2	01111_2
	00001_2		11011_2
00_2	01_2	10_2	11_2

p

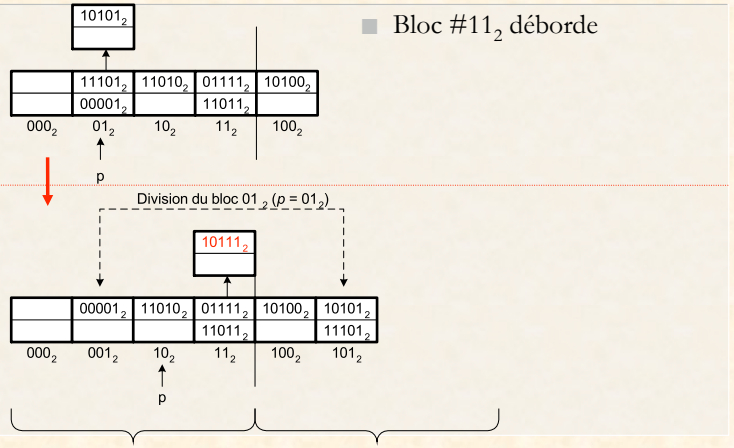
Insertion de h(clé) = 10101_2



- Bloc #01_2 déborde
- Division du bloc p = #00_2 (pas #01_2)
- $p := p+1$

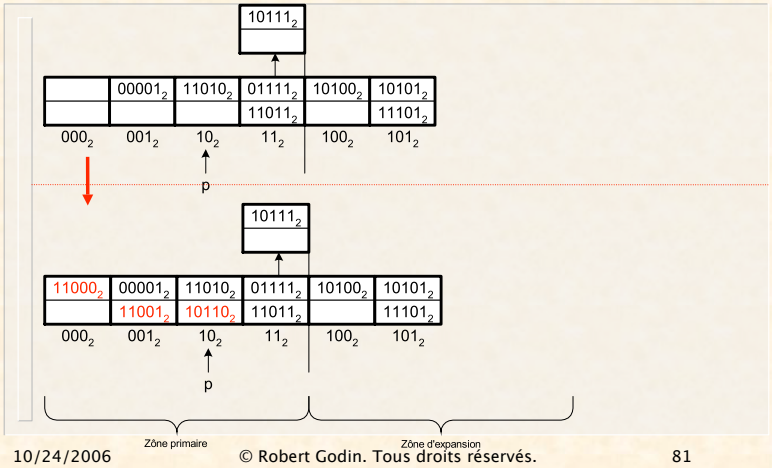
```
FONCTION AdresseLinéaire (clé)
DÉBUT
        SI hd(clé) >= p     Retourner hd(clé)
        SINON               Retourner hd+1(clé) FINSI
FIN
```


Insertion de h(clé) = 10111_2

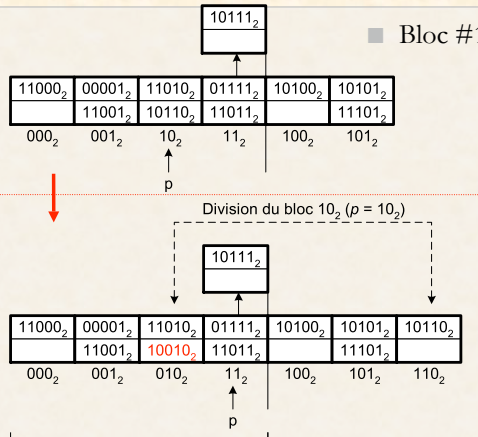


- Bloc #11_2 déborde

Insertion de 11000_2, 11001_2 et 10110_2

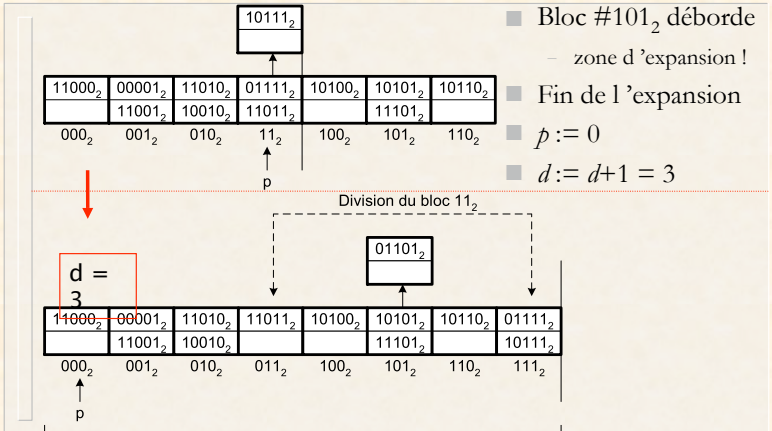


Insertion de 10010_2



- Bloc #10_2 déborde et est divisé

Insertion de 01101_2



- Bloc #101_2 déborde
  - zone d 'expansion !
- Fin de l 'expansion
- $p := 0$
- $d := d+1 = 3$

8.3.2.1.1 Variantes du hachage linéaire

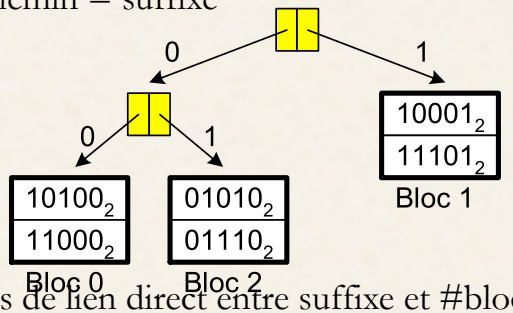
- Variante du contrôle de la division
  - algorithme de base
    - débordement => division : taux d 'occupation ~ 60%
  - division/fusion contrôlée par taux d 'occupation
- Variante de gestion des débordements
  - hachage linéaire au niveau suivant
- Variante de division
  - biais dans les chaînages (à droite de p)
  - expansions partielles
    - diviser n blocs en n+1
  - fonction de hachage exponentielle
- Gestion de l 'espace d 'adressage primaire
  - préserver la contiguïté de l 'espace malgré expansions ?

8.3.2.2 Hachage extensible

- Ajoute un niveau d 'indirection
- *Répertoire d'adresses de paquets*
  - espace supplémentaire
  - accès disque supplémentaire pour répertoire
    - antémémoire
- Bloc qui déborde est divisé
  - pas de dégradation due au chaînage
  - pire cas : 2 transferts

Analogie avec arbre digital

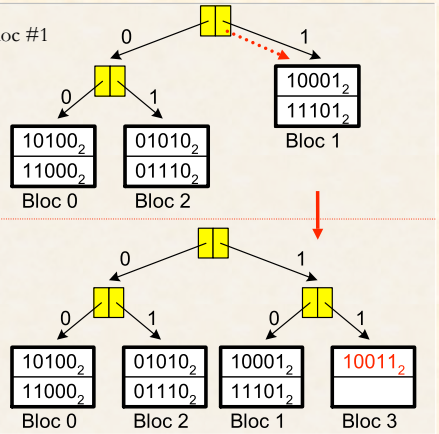
- Répertoire vu comme arbre digital
- Chemin = suffixe



- Pas de lien direct entre suffixe et #bloc

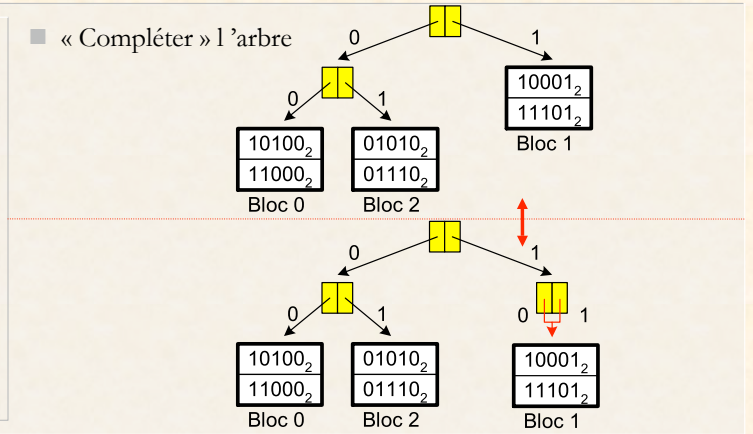
Insertion de h(clé) = 10011_2

- Débordement et division du bloc #1
- Utilisation d 'un bit de plus



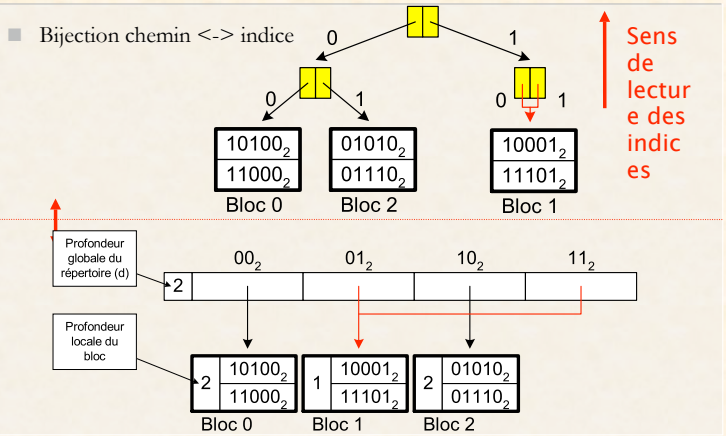
«Remplacer» l'arbre digital par un répertoire

- « Compléter » l 'arbre

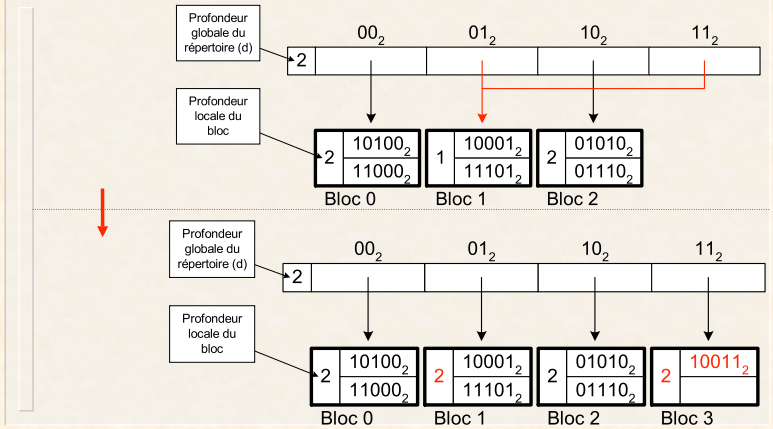


Arbre digital => un répertoire

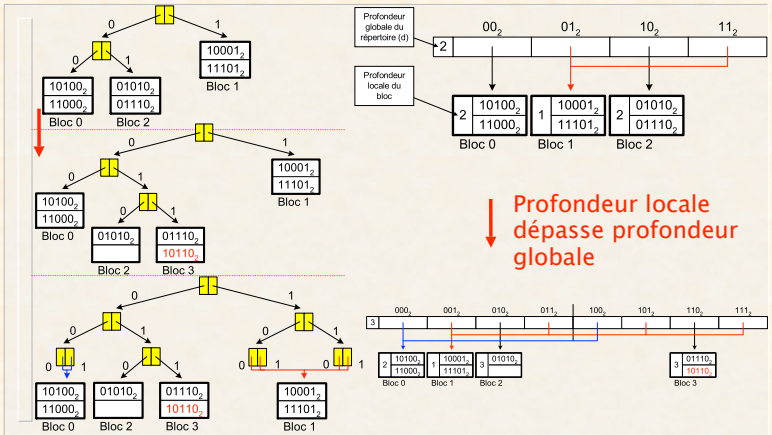
- Bijection chemin <-> indice



Insertion de h(clé) = 10011_2 avec répertoire



Cas de dédoublement de répertoire : insertion de h(clé) = 10110_2



Hachage extensible (suite)

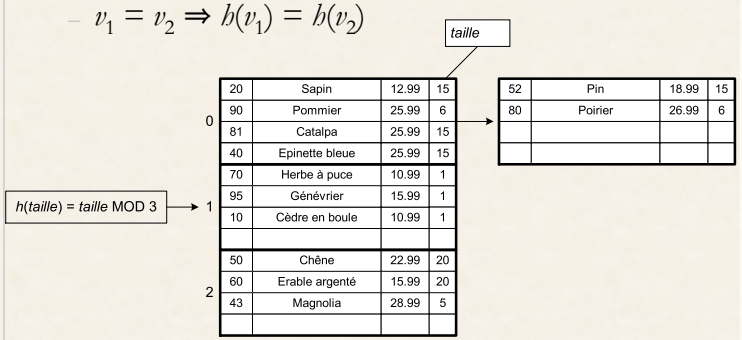
- Occupation d 'espace
  - comportement oscillatoire assez prononcé
    - entre .53 et .94
    - moyenne : $\ln 2 = .69$
- Variation
  - contrôle de la division par taux d 'occupation
  - gestion des débordements

8.3.3 Réalisation de l'accès par IDE avec le hachage

- Hachage statique
  - bloc d 'ancrage fixe
  - IDE = *idFichier, #blocAncrage, #séquence*
    - HASH CLUSTER d 'Oracle
  - applicable dans le cas non unique
- Cas d 'une clé de hachage unique
  - IDE = *id_fichier, valeur de la clé unique*
  - nécessaire avec hachage dynamique

8.3.4 Hachage sur une clé non unique et effet de grappe

- Regroupement des mêmes valeurs de clé
  - $v_1 = v_2 \Rightarrow h(v_1) = h(v_2)$



8.3.5 Hachage secondaire

- Hachage sur
  - (clé de hachage, IDE)

8.4Tableau comparatif des organisations

Critère	Sériel	Arbre-B⁺ primaire	Arbre-B⁺ secondaire	Hachage statique	Hachage dynamique
Sélection par égalité sur clé unique	*O(N/FB)* Cas moyen : *O*(N/FB/2)	*O(log (N))*	*O(log (N))*	Meilleur cas :*O*(1) Pire cas : *O(N/FB)*	Meilleur cas :*O*(1) Pire cas : *O(N/FB)*
Sélection par égalité sur clé non unique	*O(N/FB)*	*O*(log (*Card*(*clé*))+ *Sel*/*FB*)	*Approche par duplication* *O*(log (*N*)+ *Sel*)	Meilleur cas : O(*Sel*/*FB*) Pire cas : *O(N/FB)*	Meilleur cas : O(*Sel*/*FB*) Pire cas : *O(N/FB)*
Sélection par intervalle	*O(N/FB)*	*O*(log (*Card*(*clé*))+ *Sel*/*FB*)	*O*(log (*N*)+ *Sel*)	*O(N/FB)*	*O(N/FB)*
Itération sérielle	*O(N/FB)*	*O(N)*		*O(N/FB)*	*O(N/FB)*
Itération séquentielle	O(tri)	*O(N/FB)*	*O(N)*	O(tri)	O(tri)
Insertion/suppression	O(1)	*O*(log *N*)	*O*(log *N*)	Meilleur cas :O(1) Pire cas : *O(N/FB)*	Meilleur cas :O(1) Pire cas : *O(N/FB)*
Taux d'occupation mémoire secondaire	Maximal approche 100%	En moyennne : 66% (2/3)	Ajouter à l'organisation primaire	~80% À ajuster Pire cas non borné	~69% (ln 2) Peut augmenter au prix d'une diminution de performance.
Autres considérations		Support difficile de l'accès par IDE et donc des organisations secondaires	Peut y avoir plusieurs index secondaires sur la même table	Distribution des clés par fonction de hachage ? Problème avec tables volatiles Stress du DBA	Disponibilité restreinte

