Master d'informatique- M1

UE BDR- 4I803
Bases de données réparties
Année 2016

site Web :
www-bd.lip6.fr/wiki/doku.php/site/enseignement/master/bdr

Hubert Naacke
Hubert.Naacke@lip6.fr
et Stéphane Gançarski

1

Support de cours

- Contenu de ce poly
  - Contenu partiel à compléter en séance
  - Volontairement incomplet
  - Assiduité fortement recommandée
    - Présence au cours nécessaire

2

Objectifs

Présenter les architectures des systèmes de gestion de bases de données (réparties) et les techniques permettant de les implémenter.

Techniques d'implémentation des SGBD relationnels.

Architecture des bases de données réparties et du Web.

Conception, interrogation et manipulation de données réparties.

Mise en pratique : chaque séance comporte 2h de TME

3

Plan

- Méthodes d'accès et indexation
- Structure d'index (hachage, arbre B+)
- Optimisation de requêtes
- Bases de données réparties: fragmentation
- Interrogation de bases de données réparties
- SGBD parallèles
- Transactions réparties
- Reprise sur pannes
- Gestion de données hétérogènes et réparties
- TME : Oracle, JDBC, SimJava…

4

Bibliographie

- H. Garcia-Molina, J.D.Ullman, J. Widom : *Database System Implementation*, Prentice Hall, 2000.
- M.T.Özsu, P. Valduriez : *Principles of Distributed Database Systems*, 3rd edition, Prentice Hall, 2011
- R. Ramakrishnan – J. Gehrke : *Database Management Systems*, Mc-Graw Hill
- S. Abiteboul, P. Buneman, D. Suciu : *Data on the Web : from relations to semistructured data and XML*, Morgan Kaufmann, 1999.

5

BDR
Master d'Informatique

Niveau M1

Cours 1- Méthodes d'accès

Stéphane Gançarski et Hubert Naacke

6

Plan

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

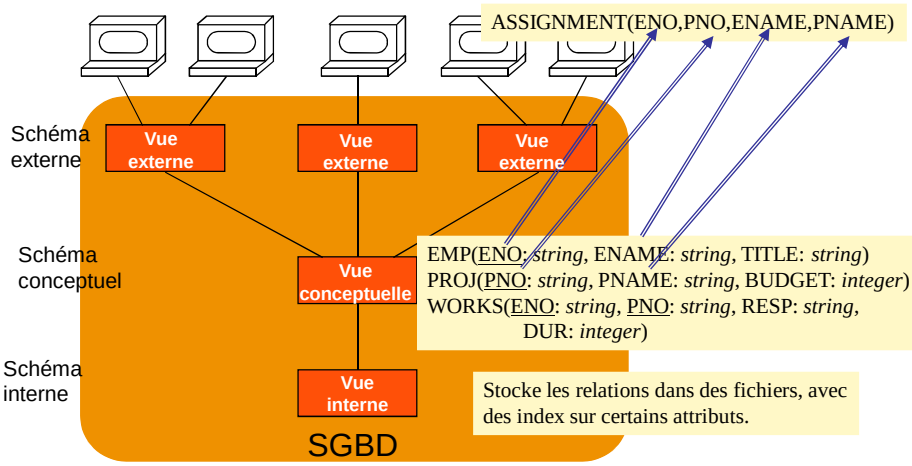
7

Objectifs des SGBD (rappel)

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

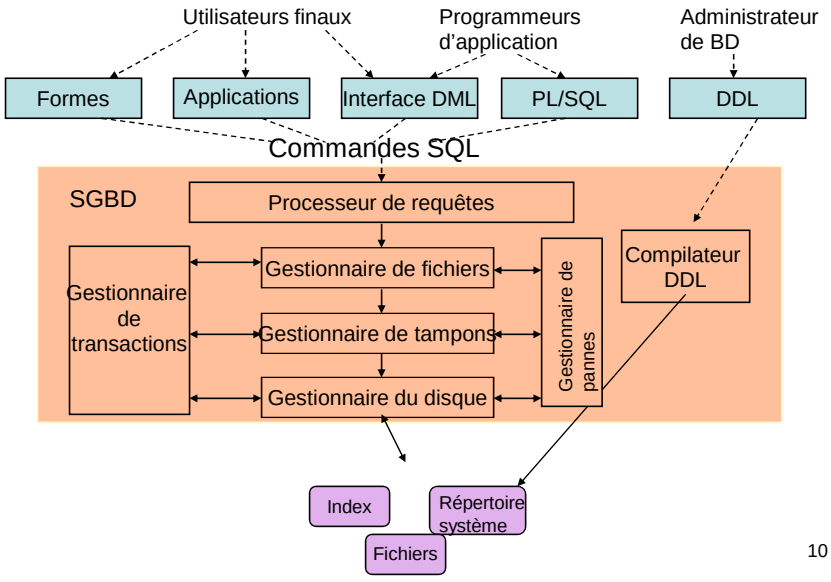
8

Architecture ANSI/SPARC



9

Architecture système



10

Stockage des données

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

11

Organisation des données en fichiers

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

12

Organisation séquentielle

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

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

13

Organisations Indexées

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

recherche dichotomique, index d'un livre…

14

Entrée d'un index

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

15

Index non plaçant

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

16

Index plaçant

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

17

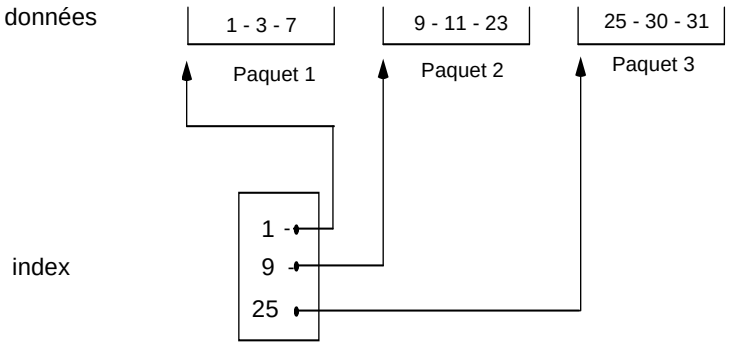
Index plaçant non dense

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

- Rmq: Un index contenant **toutes** les clés est dit **dense**

18

Exemple d'index plaçant non dense



19

Index unique

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

20

Accès aux données par un index

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

21

Index couvrant une requête

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

22

Index composé

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

23

Index composé: exemple

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

24

Choix entre un accès séquentiel ou un accès par index

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

25

Index hiérarchisé

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

26

Arbre B+

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

27

Arbre B+ : coût d'accès

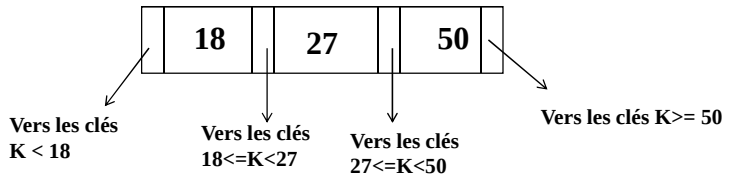
- Le coût d'accès est
  - proportionnel à la longueur d'un chemin
  - Identique quelle que soit la feuille atteinte

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

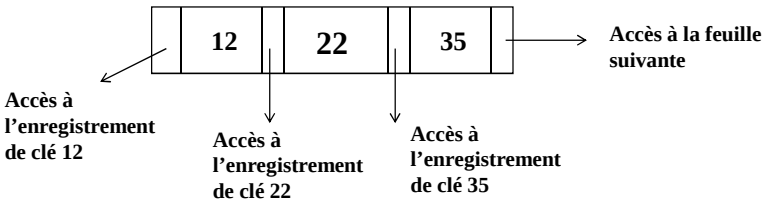
28

Arbre B+

- Les nœuds internes servent à atteindre une feuille



- Les feuilles donnent accès aux enregistrements



29

Arbre B+: 3 types de nœuds

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

30

Ordre d'un arbre, degré d'un noeud

- La capacité d'un nœud de l'arbre s'appelle l'**ordre**
- Un arbre-B+ est d'**ordre d** ssi
  - Pour un nœud intermédiaire et une feuille : $d \leq n \leq 2d$
  - Pour la racine: **1** $\leq n \leq 2d$
- Degré sortant d'un nœud
  - Un nœud intermédiaire (et la racine) ayant n valeurs de clés a **n+1** pointeurs vers ses fils
  - Une feuille n'a pas de fils

31

Nombre de clés dans les feuilles

- Dépend de l'ordre d et du nombre de niveaux p
- Nombre maxi de clés dans l'arbre
- Arbre à 1 niveau (arbre réduit à sa seule racine): 2d clés maxi
- Arbre à 2 niveaux :
  - racine: 2d clés maxi
  - 2d+1 feuilles, soit 2d×(2d+1) clés maxi dans les feuilles
- Arbre à p niveaux :
  - Nbre maxi de clés dans les feuilles: $2d(2d+1)^{(p-1)}$
- En pratique, un arbre B+ a rarement plus de 4 niveaux car d est grand (de l'ordre de la centaine)
- Nombre mini de clés dans les feuilles : ...

32

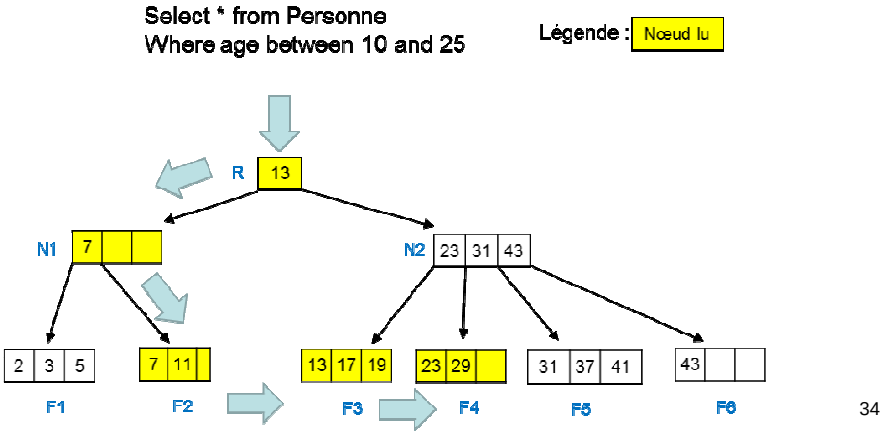
Arbre-B+ : chainage des feuilles

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

33

Parcours du chainage

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



34

Insertion

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

35

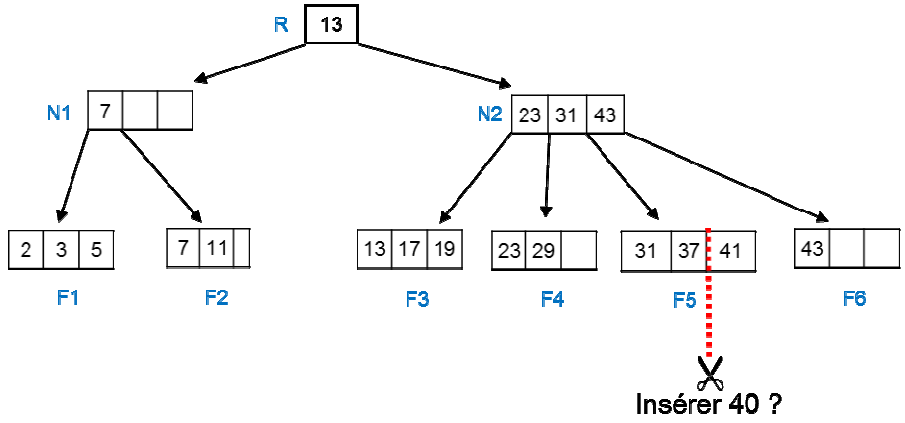
Insertion (cont.)

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

36

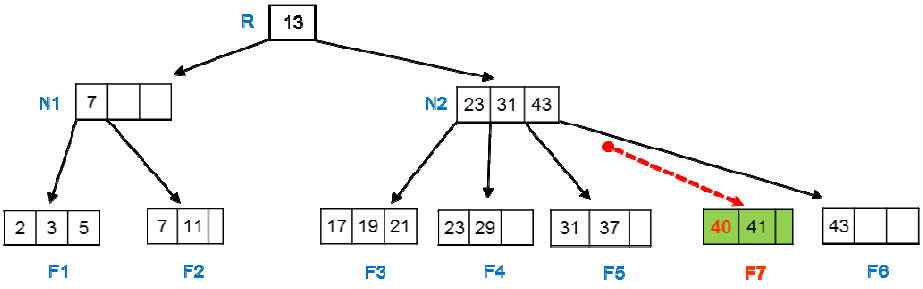
Insertion
Exemple 1 : état initial

Arbre initial (on suppose un capacité de 1 à 3 valeurs par nœud)



37

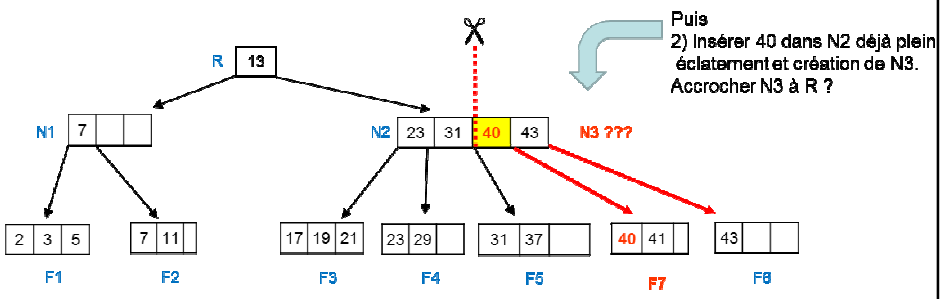
Exemple 1 (suite)



1) Insérer 40 dans F5 déjà pleine
éclatement et création de F7
Accrocher F7 à N2 ?

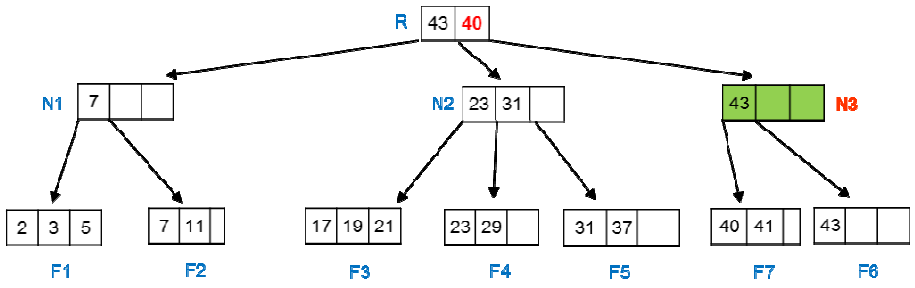
38

Exemple 1 (suite)



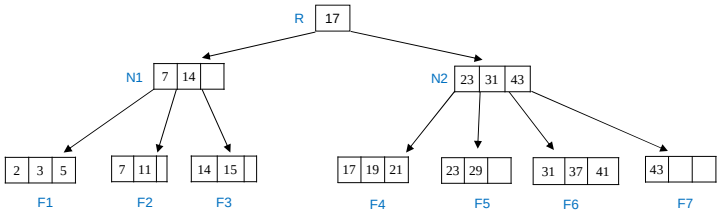
39

Exemple 1 (fin)



40

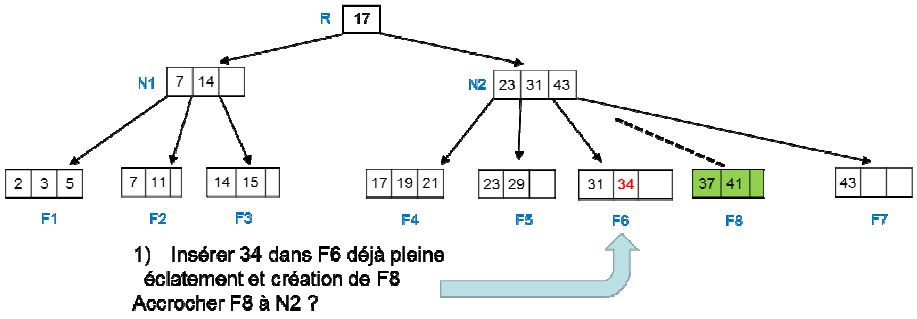
Insertion
Exemple 2 : état initial



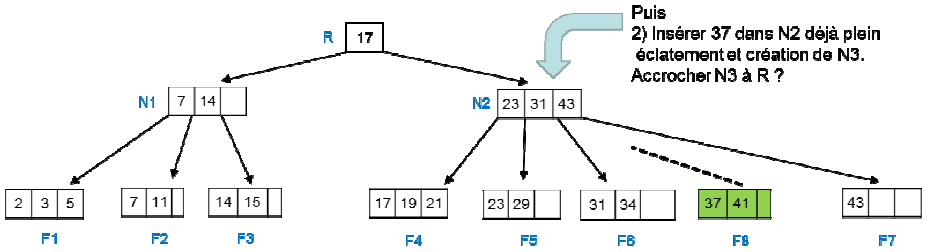
Insérer 34 ?

41

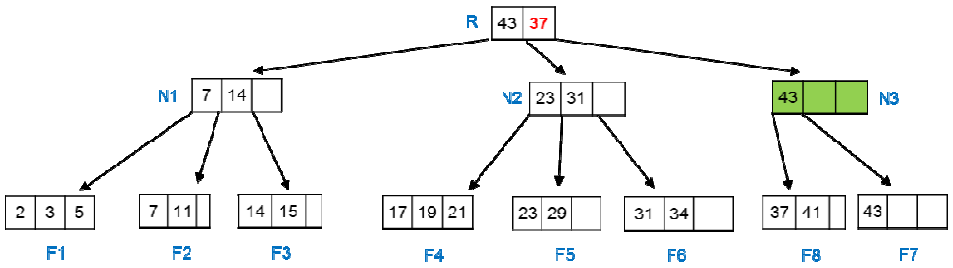
Exemple 2 (suite)



Exemple 2 (suite)



Exemple 2 (fin)



Suppression

- Supprimer la valeur de la feuille (et le pointeur vers l'enregistrement)
- Si la feuille est encore suffisamment pleine, il n'y a rien d'autre à faire.
- Sinon, redistribuer les valeurs avec une feuille **ayant le même parent**, afin que toutes les feuilles aient le nombre minimum de valeurs requis.
  - Ajuster, dans le nœud père, la valeur délimitant les 2 nœuds impliqués dans la redistribution
- Si la redistribution est **impossible**, alors fusionner 2 feuilles
  - Supprimer une feuille et la 'décrocher' en supprimant une valeur dans son père.
- Si le parent n'est pas suffisamment plein, appliquer récursivement l'algorithme de suppression
- Remarque1 : la propagation récursive peut entraîner la perte d'un niveau.

45

Redistribution entre 2 nœuds intermédiaires

- Redistribution entre deux nœuds intermédiaires ayant le même parent
- La valeur contenue dans le parent participe à la redistribution
  - la valeur du parent "descend" dans le nœud à remplir,
  - la valeur à redistribuer "monte" dans le parent

46

Résumé des opérations

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

47

Avantages et Inconvénients

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

48

Exercice Arbre B+

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

49

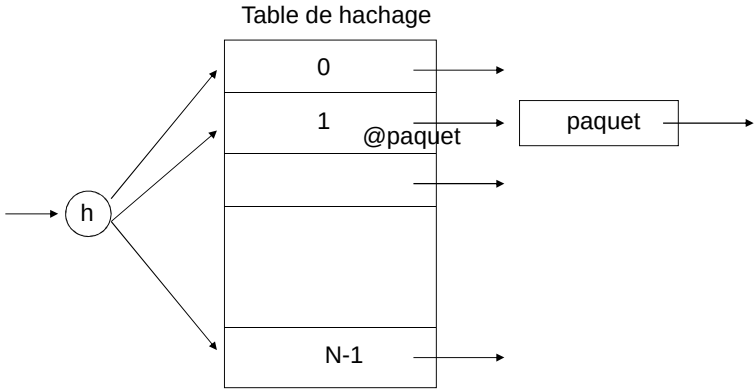
Organisations par Hachage

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

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

50

Hachage statique



51

Hachage statique

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

52

Techniques de débordement

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

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

53

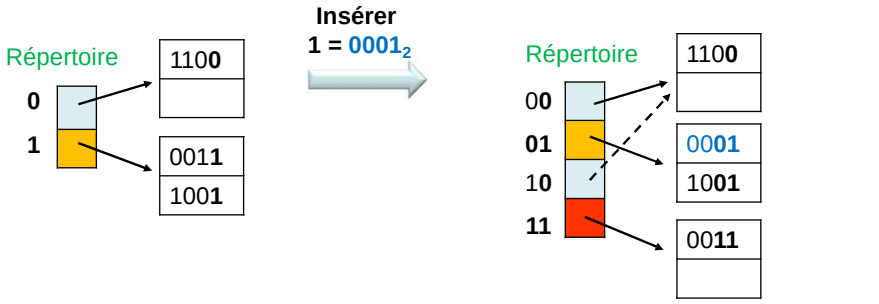
Hachage dynamique

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

54

Hachage extensible

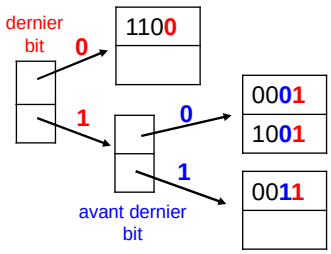
- Ajout d'un niveau d'indirection vers les paquets (tableau de pointeurs), qui peut grandir (considérer + de bits) : répertoire
- Jamais de débordement
  - Accès direct à tout paquet via le répertoire (i.e, une seule indirection)



55

Hachage extensible

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



➲ Suffixe utilisé pour l'indexation = profondeur

56

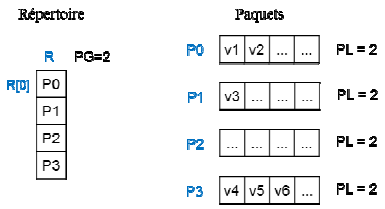
Hachage extensible
Profondeur Globale et Locale

- Profondeur globale (PG)
  - Sert à atteindre (retrouver) une entrée
    - En invoquant la fonction $h_{PG}(v) = v \bmod 2^{PG}$ pour lire la case $h_{PG}(v)$
  - Avantage d'utiliser la fonction modulo 2^{PG}
    - évite de re-répartir toutes les valeurs de la table quand on l'agrandit.
- Profondeur locale (PL)
  - Sert à ne pas éclater tous les paquets en même temps
    - Avantage: répartir parmi plusieurs insertions, le surcoût d'agrandissement de la table de hachage
  - Indique pour chaque paquet, s'il peut éclater "rapidement" sans agrandir le répertoire

57

Hachage extensible : notations

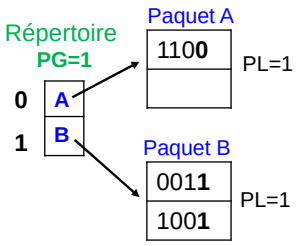
- Le répertoire est noté **R[**P_0, P_1, P_2,..., P_k **] PG=***pg* avec
  - P_i ... les noms d'un paquet,
  - *pg* la profondeur globale.
- Rmq : le répertoire contient k cases avec $k = 2^{pg}$
- Un paquet est noté P_i**(**v_j, ..., ...**) PL=***pl* avec
  - P_i le nom du paquet, par exemple A,B, ...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



58

Hachage extensible
Création du répertoire

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



59

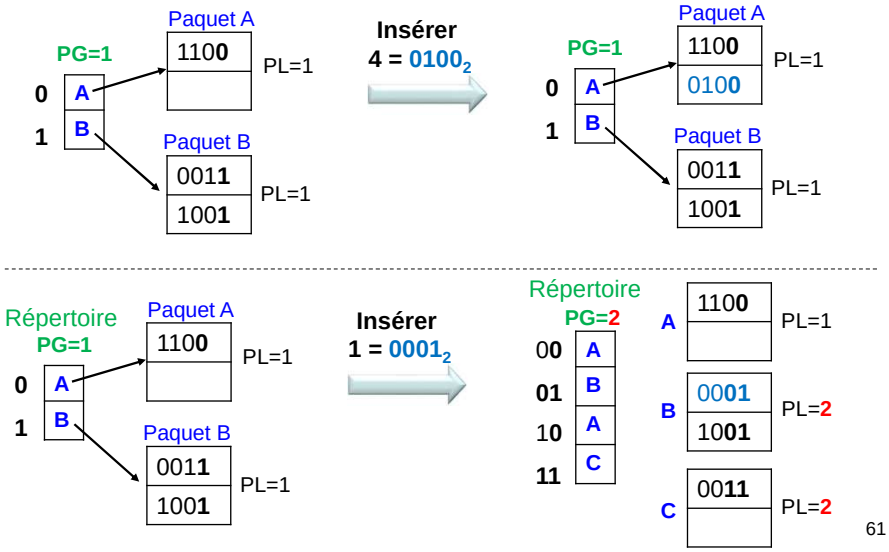
Hachage extensible : Insertion

- Insertion de v dans le paquet Pi

- **Cas 1)** P_i n'est **pas** plein, insertion immédiate dans P_i
- **Cas 2)** P_i est **plein** et $PL_i < PG$ alors éclater P_i
  - Créer un nouveau paquet P_j et l'accrocher dans le répertoire
    - Remplacer l'adresse de Pi par celle de Pj dans la 2ème case contenant Pi
    - Si 4 cases contiennent Pi, "recopier" le remplacement dans le reste du répertoire
  - Incrémenter les profondeurs locales de P_i et P_j (PL = PL+1)
  - Répartir les valeurs de Pi et v entre Pi et Pj
    - Si Pi est encore plein, réappliquer l'algo d'insertion : cas 2) ou 3)
- **Cas 3)** Pi est **plein** et **PLi = PG** alors doubler le répertoire
  - Recopier le contenu des k premières cases dans les k nouvelles cases suivantes
  - PG = PG+1 puis appliquer le **Cas 2)**

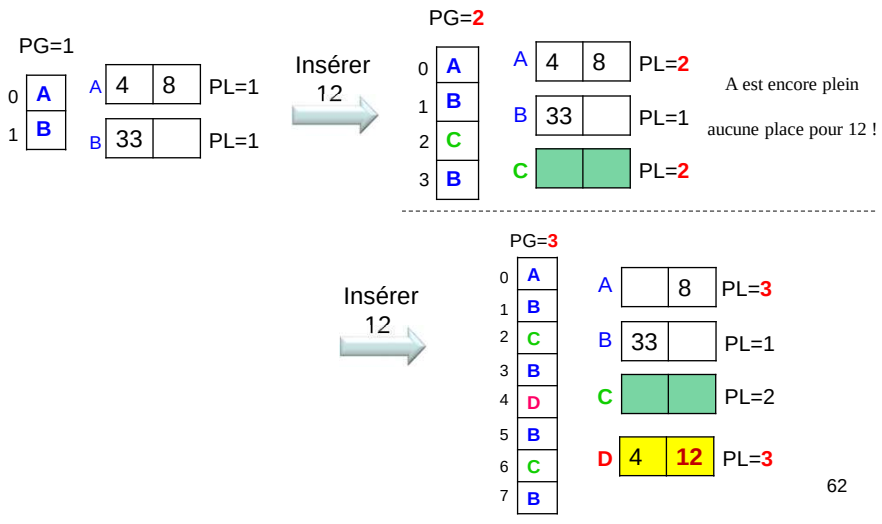
60

Hachage extensible : Insertions



61

Hachage extensible : Insertion avec plusieurs éclatements



62

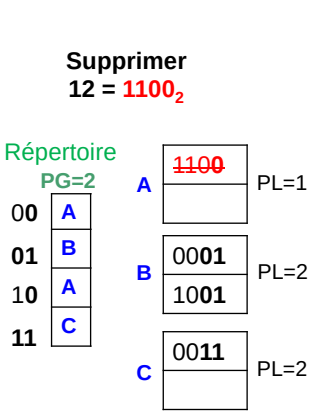
Hachage extensible
Suppression et fusion

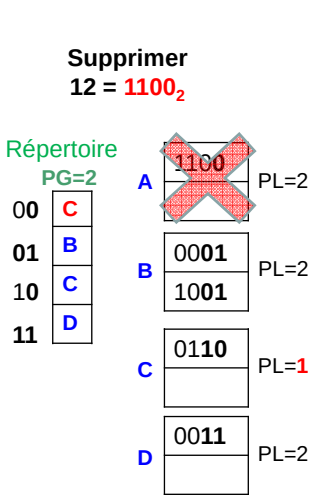
- Lors d'une suppression, si un paquet Pi devient vide :
- Si PL_i=PG alors
  - Fusionner Pi avec le paquet Pj référencé dans la case ayant le même suffixe que celle qui référence Pi
  - Suffixe commun (en base 2) de longueur PG -1
  - Exple si PLi=PG=3, les cases ayant le même suffixe (de longueur 2) sont :
    - R[0] et R[4]
    - R[1] et R[5]
    - ...
    - R[3] et R[7]
  - Supprimer Pi et le "décrocher" du répertoire : dans la case contenant Pi, remplacer Pi par Pj
  - Décrémenter la profondeur locale de Pj
- Sinon (on a PLi < PG) : pas de fusion, Pi reste vide.

- Si pour tous les paquets restants on a PL < PG alors ne garder que la première moitié du répertoire (division par 2) et décrémenter PG.

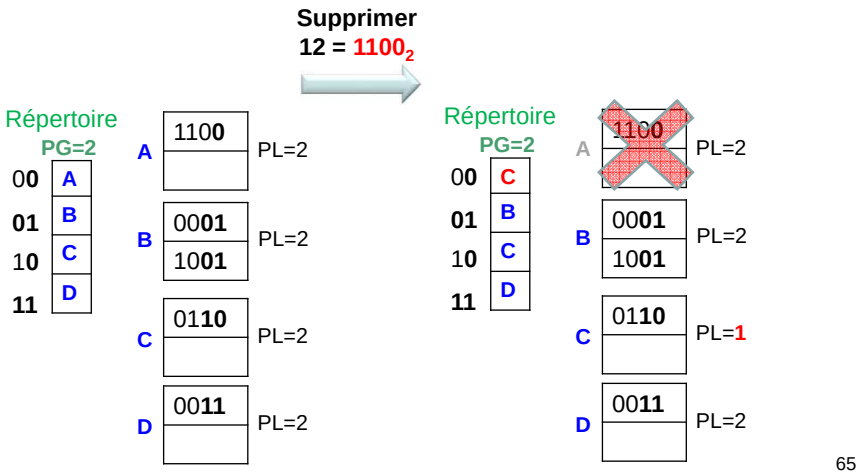
Hachage extensible
Exemples de suppression

-





Hachage extensible
Exemples de suppression



65

Hachage extensible (suite)

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

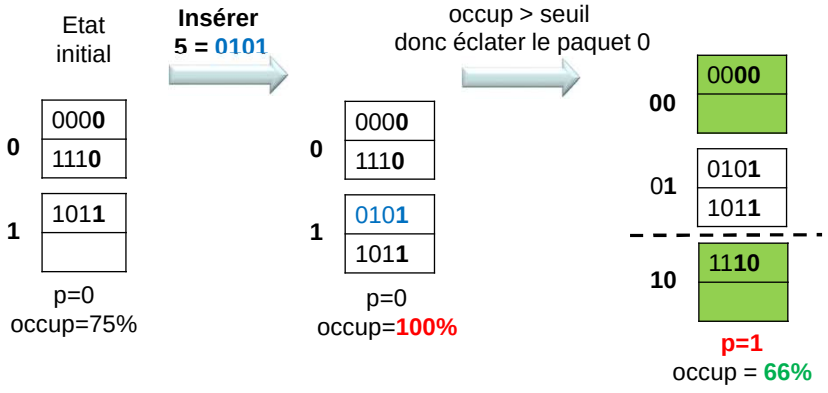
66

Hachage linéaire (1)

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

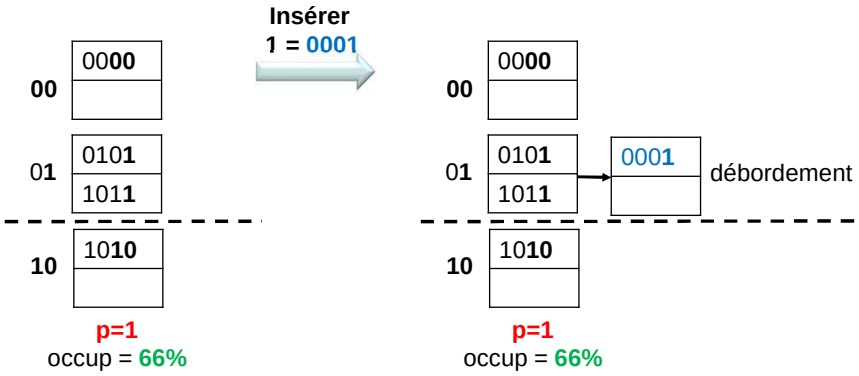
Hachage linéaire (2)

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



Hachage linéaire (3)

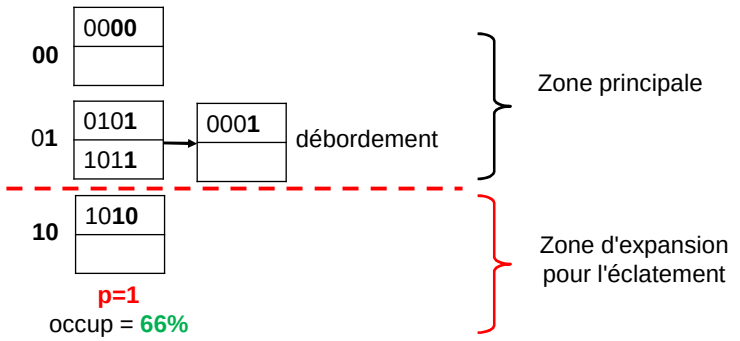
- Insérer 1 dans le paquet 1
- Débordement du paquet (pas d'éclatement car le taux d'occupation reste inférieur au seuil)



69

Hachage linéaire (4)

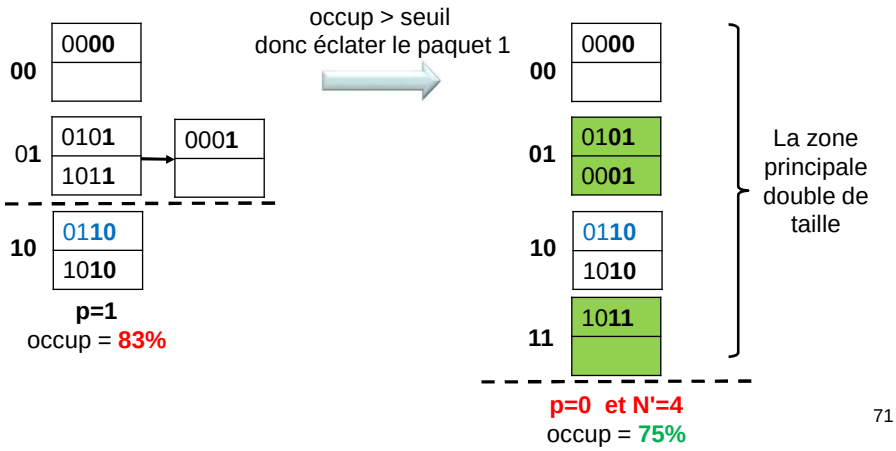
- Accès à l'enregistrement de clé c ?
- Soit i = $h_0(c)$ et i'= $h_1(v)$
- Si i >= p alors lire le paquet i sinon lire le paquet i'



70

Hachage linéaire (5)

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



71

Exercice Hachage extensible

- Chaque paquet **contient au plus 2 valeurs.**

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

72

Conclusion

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

73

Perspectives:

- autres index arborescents :
  - quadtree (quadrants), k-d-tree (d dimensions), R-tree (region), cache conscious tree
- Autres structures : bimap index
- Index pour les SGBD en mémoire
  - "Generalized prefix tree source,"
    - http://wwwdb.inf.tu-dresden.de/research-projects/projects/dexter/core-indexing-structure-and-techniques
    - M. Boehm, Vainqueur Sigmod contest 2009
  - The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases
    - Viktor Leis, Alfons Kemper, Thomas Neumann
    - ICDE 2013
- Index répartis

74