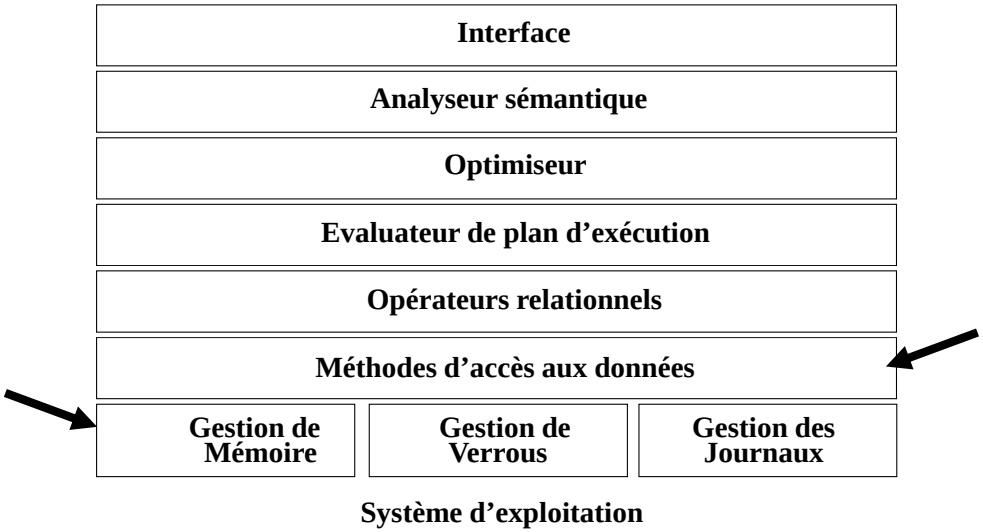
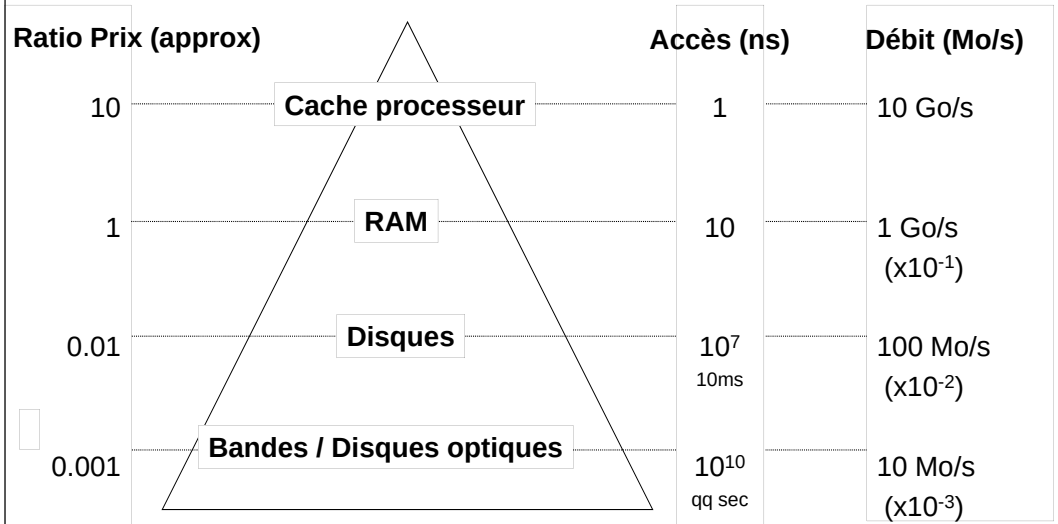
Architecture en couche d'un SGBD



Modèles de stockage et indexation

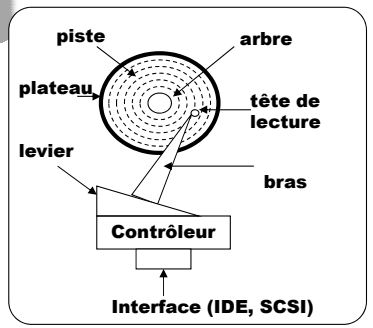
- 1. Hiérarchie mémoire
- 2. Modèles de stockage
- 3. Propriétés des index
- 4. Organisations arborescentes
- 5. Organisations hachées

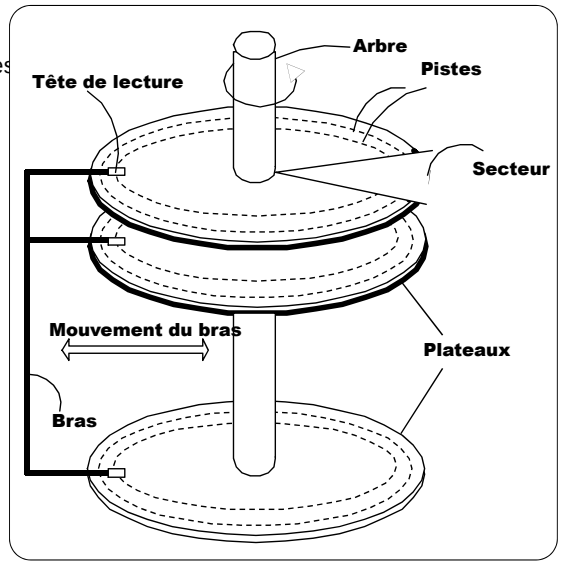
Hiérarchie mémoire



Disques Magnétiques

- Progression moyenne des disques
  - Capacité + 60% par an
  - Latence stable ← contraintes mécaniques …
  - Débit + 40% par an





Optimisations logicielles

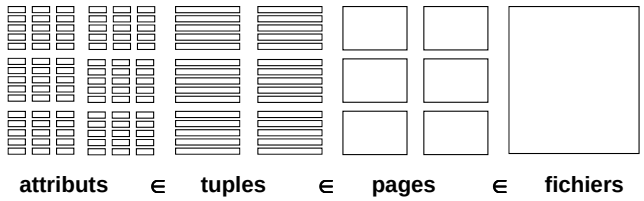
- Prefetching
  - Le SGBD peut anticiper les patterns d'accès aux données
  - Ainsi, il peut pré-charger (prefetch) des données en RAM
  - NB: le prefetching doit être géré au niveau du SGBD (vs. OS)
- Clustering
  - Regroupement physique des blocs de données
    - Quand ils contiennent des données souvent utilisées ensemble
  - Placement de ces blocs sur
    - La même piste physique
    - Le même cylindre physique
  - Compromis entre bon placement et trop de perte de place
    - Allocation d'un ensemble de blocs par «granule»
    - Calibrer la taille du granule pour chaque objet DB

Evolutions matérielles

- Futures technologies de stockage BD
  - FLASH NAND
    - 1 puce Flash ➔ 256GB aujourd'hui, 2TB au format carte SD demain
    - 1 SSD = k puces
    - Latence: read ≈ 100 µs, write ≈ 300 µs, random write ≈ variable
    - Débit ≈ 20 Mo/sec
    - Contraintes fortes : EraseBlock-before-RewritePage, nb cycles Erase limité
    - ≈ 10 fois plus cher qu'un disque magnétique
  - PCM, MEMS/NEMS (Micro/Nano Electro-Mechanical Systems)
    - Pas encore opérationnel
- SGBD actuels optimisés pour les disques (HDD)

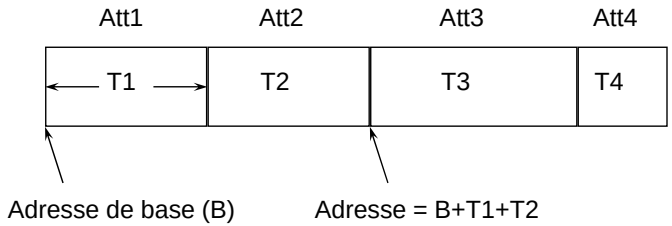
Modèles de stockage des données

- Les données sont stockées sous forme de
  - *attributs*, de taille fixe ou variable...
  - …stockés eux-mêmes sous forme de *tuples*...
  - …stockés dans des *pages* (blocs disques)...
  - …stockés dans des *fichiers*



- Rappel : le SGBD gère un cache de pages en RAM à la façon d'une mémoire virtuelle

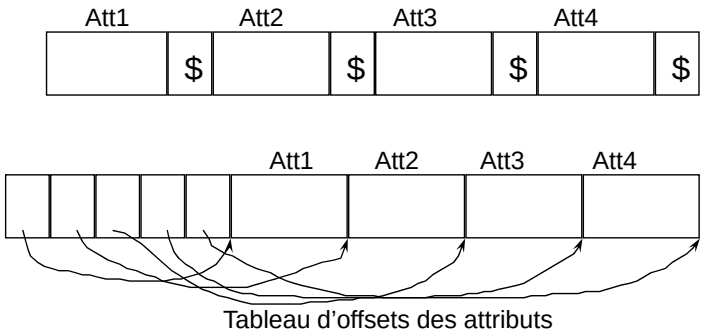
Stockage des tuples (attributs de taille fixe)



- Les informations concernant les types/tailles d'attributs
  - Sont partagées par tous les tuples d'un fichier
  - Sont stockées dans le catalogue système
- Accéder au i^{ème} attribut $\not\Rightarrow$ la lecture totale du tuple

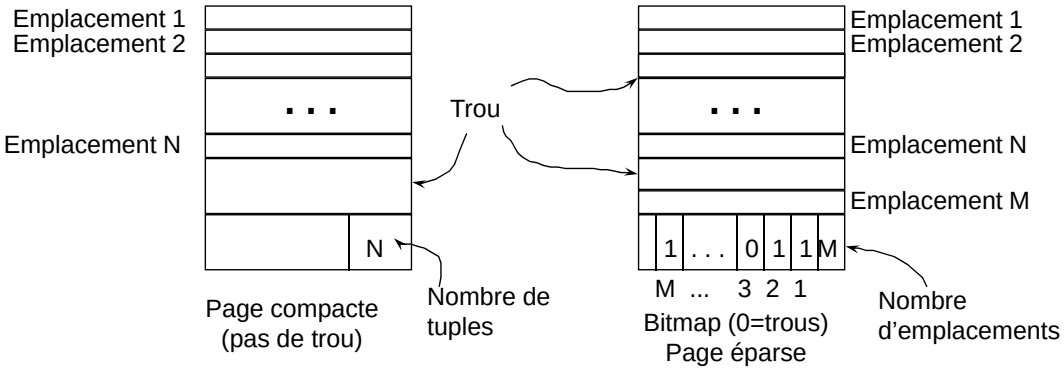
Stockage des tuples (att. de taille variable)

- Deux alternatives de stockage (le nombre d'attributs est fixe):



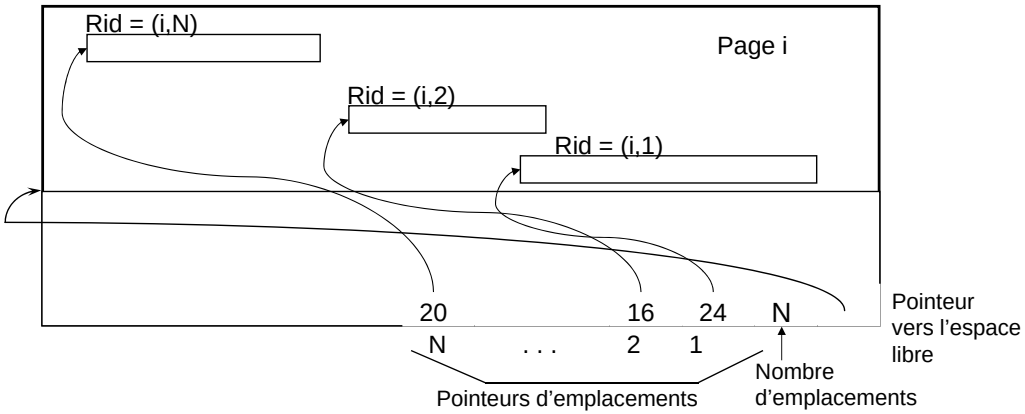
- La deuxième alternative offre
  - Un accès direct au i^{ème} attribut
  - Pas de caractère délimiteur réservé

Pages : tuples de taille fixe



- Identifiant d'un tuple (**R**ecord **id**) : *Rid* = < *id page, emplacement #*>
- Remarque
  - la *Clé* d'un tuple est un «pointeur logique», le *Rid* est un «pointeur physique»

Pages: tuples de taille variable



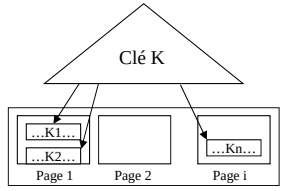
- Déplacement des tuples dans la page sans changer le *Rid…*

Indexation

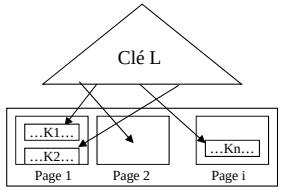
- Objectifs
  - Offrir un accès rapide à tous les tuples d'un fichier satisfaisant un même critère
    - A partir d'une clé de recherche (discriminante ou non, mono ou multi-attributs)
    - Sur des fichiers ordonnés sur cette clé, ou non ordonnés, ou ordonnés sur une autre clé
- Moyen
  - Créer une structure de données accélératrice associant des adresses de tuples aux valeurs de clés
- Index
  - Table (ou hiérarchie de tables) implémentant cet accélérateur

Catégories d'index (1)

- Index primaire ou plaçant (*primary* or *clustered*)
  - Tuples du fichier organisés par rapport à l'index
    - Les tuples sont stockés «dans» l'index
    - Dit autrement, l'adresse disque du tuple dépend de la valeur de la clé de recherche
  - 1 seul index primaire par table…
    - Souvent construit par le SGBD sur la clé primaire
- Index secondaire ou non plaçant (*secondary* or *unclustered*)
  - Tuples organisés indépendamment de l'index
    - Seuls les *Rid* des tuples sont «dans» l'index
  - Plusieurs index secondaires possibles par table



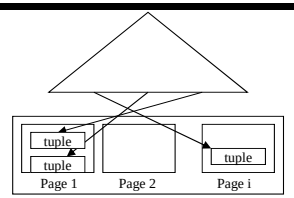
Tuples → triés sur K



Tuples non triés sur L

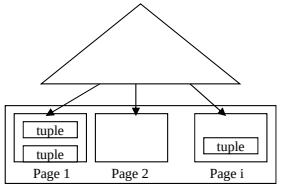
Catégories d'index (2)

- Index dense (*dense*)
  - Contient toutes les clés
  - Adresse tous les tuples



- Index non dense (*sparse*)
  - Ne contient pas toutes les clés
  - N'adresse pas tous les tuples (e.g., adresse une seule fois chaque page)
  - Nécessite que le fichier soit trié sur les clés

    Contient alors la plus grande clé de chaque bloc + l'adresse du bloc



- NB : Densité d'un index : $\frac{\text{nb clés dans l'index}}{\text{nb clés dans la table}}$

Catégories d'index (3)

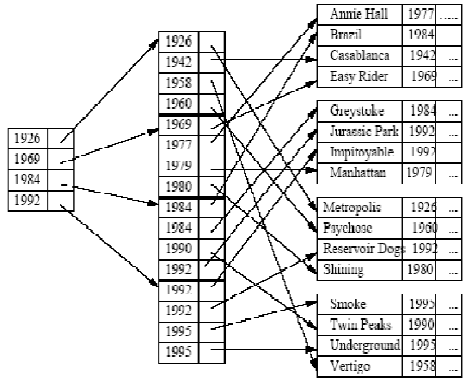
- Mais au fait …
  - Peut-on créer un index non dense non plaçant ?
  - Peut-on créer un index dense et plaçant ?
  - Peut-on créer un index primaire (c.à.d, plaçant) sur une clé secondaire (c.à.d, non discriminante) ?
  - Peut-on créer un index secondaire (c.à.d, non plaçant) sur une clé primaire (c.à.d, discriminante) ?

Catégories d'index (4)

- Index hiérarchisés ou multi-niveaux
  - Permet de gérer de gros index (i.e., très gros fichiers…)
  - Principe : chaque index est indexé
- Fonctionnement
  - L'index est trié (mais trop gros ➔ peu performant…)

    Pourquoi l'index à indexer doit-il être trié ?
  - On indexe l'index
    - Par un second index non dense (→ 2ème niveau)
  - On peut continuer…

    (jusqu'à obtenir un index non dense qui tiennent dans une seule page…)



Organisations arborescentes

- La structure d'arbre-B permet de construire des index équilibrés (B comme Bayer ou comme Balancés).

- Arbre-B (B-tree)
  - Un arbre-B d'ordre *m* est un arbre tel que:
    - *(i) chaque noeud contient k clés triées, avec m <= k <= 2m, sauf la racine pour laquelle k vérifie 1<= k <= 2m.*
    - *(ii) tout noeud non feuille possède (k+1) fils. Le ième fils contient des clés comprises entre la (i-1)ème et la ième clé du père.*
    - *(iii) l'arbre est équilibré.*

Structure interne d'un nœud d'arbre-B

P0	x1 a1 P1	x2 a2 P2	……	xi ai Pi	……	xk ak Pk

- Pi: pointeurs internes permettant de représenter l'arbre; les feuilles ne contiennent pas de pointeurs Pi;
- ai: pointeurs externes sur les données;
- xi: valeur de clé.

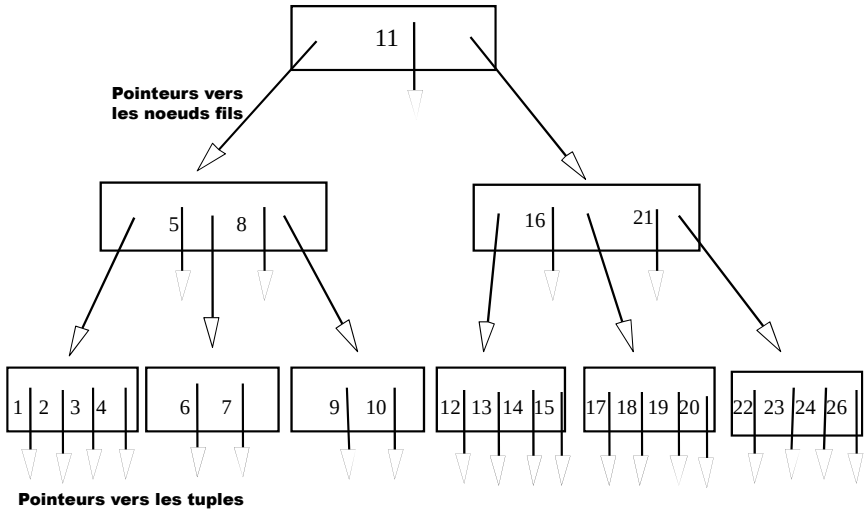
  (1) (x1, x2…xk) est une suite croissante de clés;

  (2) Toute clé y de K(P0) est strictement inférieure à x1;

  (3) Toute clé y de K(P1) est comprise entre xi et xi+1;

  (4) Toute clé y de K(PK) est strictement supérieure à xk.

Exemple d'arbre-B d'ordre 2



Hauteur d'un Arbre-B

- La hauteur d'un arbre-B est déterminé par son ordre et le nombre de clés contenues.
  - pour stocker N clés :

    $\log_{2m}(N) \leq h \leq \log_{m}(N)$

  - Soit h=3 pour N=8000 et m=10
  - Ou h=3 pour N=8 millions et m=100
  - Ou encore h=3 pour N=1 milliard et m=500

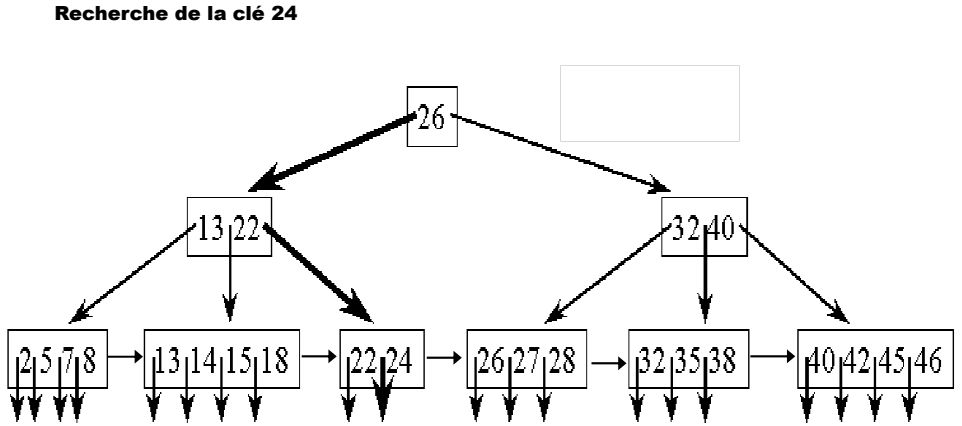
- h est très important car il détermine le nombre d'E/S nécessaire pour traverser l'arbre lors d'une sélection

- Dans la pratique, qu'est ce qui détermine m ?
- Et pourquoi est-ce important que l'arbre soit équilibré ?

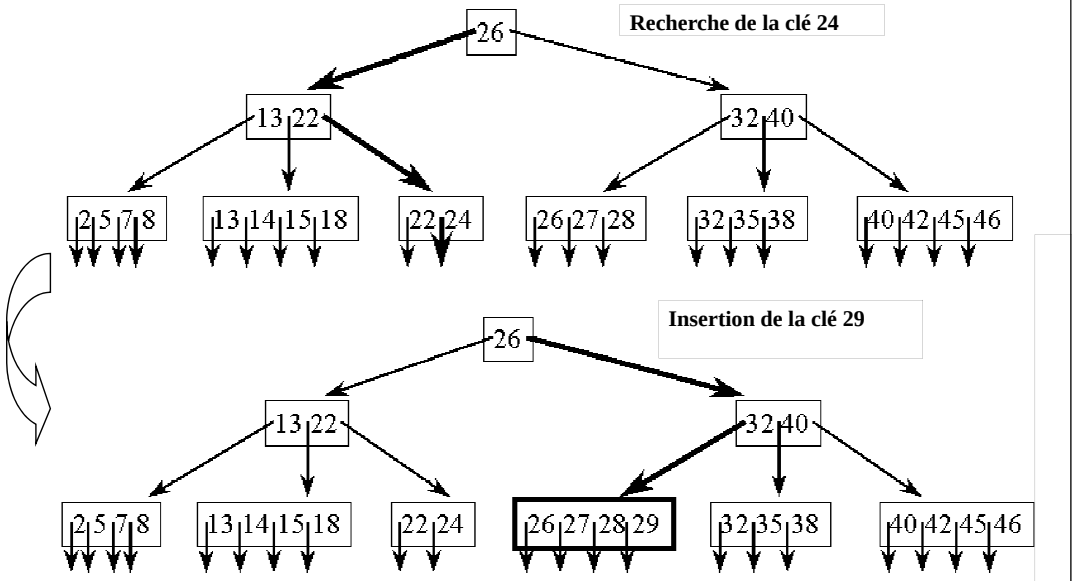
Arbre-B+

- Notion 15: Arbre B+ (B+ tree)
  - Arbre-B dans lequel on répète les clés des nœuds ancêtres dans chaque nœud et on chaîne les nœuds feuilles pour permettre un accès rapide en séquentiel trié.
- Les arbres-B+ sont utilisés pour gérer des index hiérarchisés :
  - 1) en mettant toutes les clés des articles dans un arbre B+ et en pointant sur ces articles par des adresses relatives ==> INDEX NON PLACANT
  - 2) en rangeant les articles au plus bas niveau de l'arbre B+ ==> INDEX PLACANT

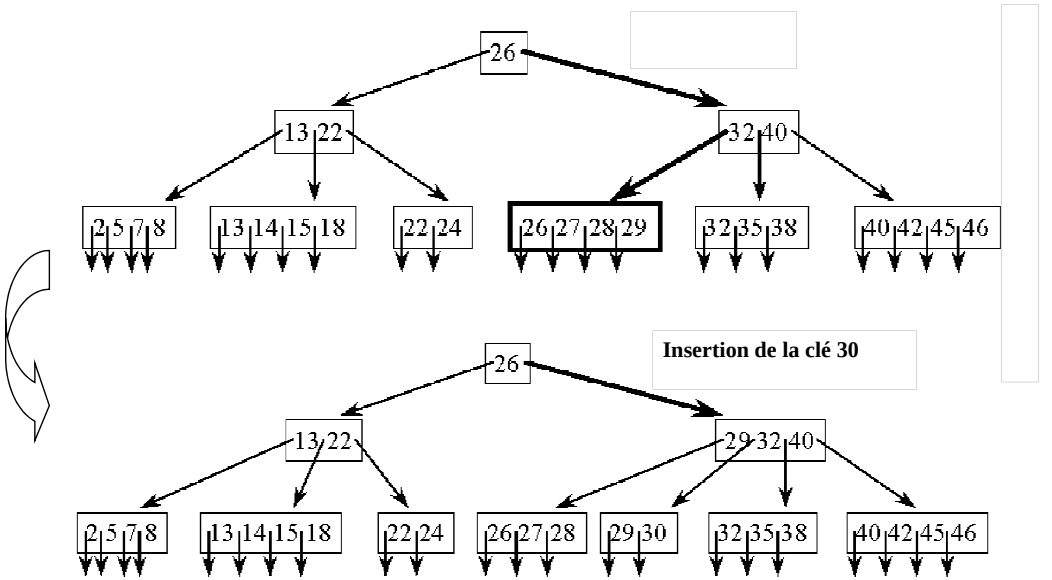
Exemple d'arbre-B+



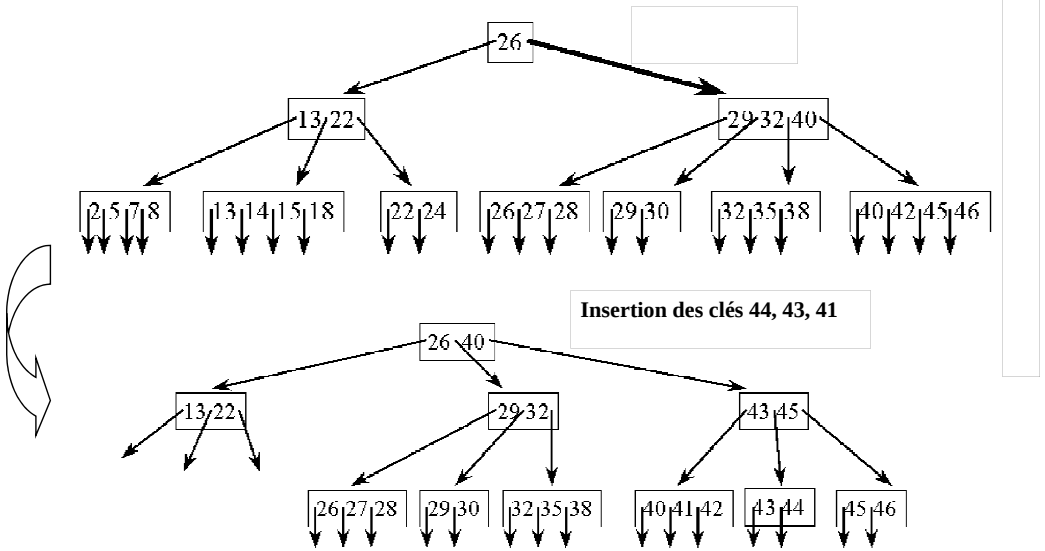
Exemple d'opérations sur B+Tree d'ordre 2



Exemple (cont')



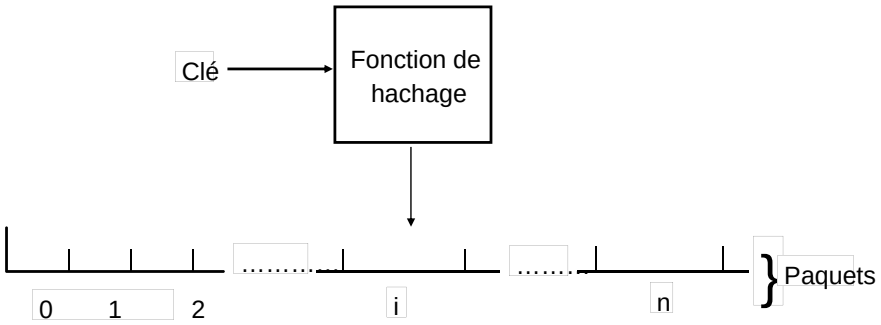
Exemple (cont')



Organisations par Hachage

- Objectif
  - Offrir un accès rapide à tous les tuples d'un fichier satisfaisant un même critère (idem orga. arborescentes)
  - Eviter le coût lié à la traversée d'une arborescence
- Moyen
  - Calculer l'adresse du tuple à l'aide d'une fonction de hachage appliquée à la clé de recherche
  - La sélection se fait directement en recalculant cette même fonction

Hachage statique



Fonction de Hachage

- DIFFÉRENTS TYPES DE FONCTIONS :
  - PLIAGE DE LA CLE
  - CONVERSION
  - MODULO P
  - ...
- BUT :
  - Obtenir une distribution uniforme pour éviter de saturer un paquet
  - Mauvaise fonction de hachage ==> Saturation locale, perte de place
- SOLUTION : AUTORISER LES DEBORDEMENTS

Techniques de débordement

- 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.
- Chaînage
  - constitue un paquet logique par chaînage d'un paquet de débordement à un paquet plein.
- Rehachage
  - applique une deuxième fonction de hachage lorsqu'un paquet est plein pour placer en débordement.

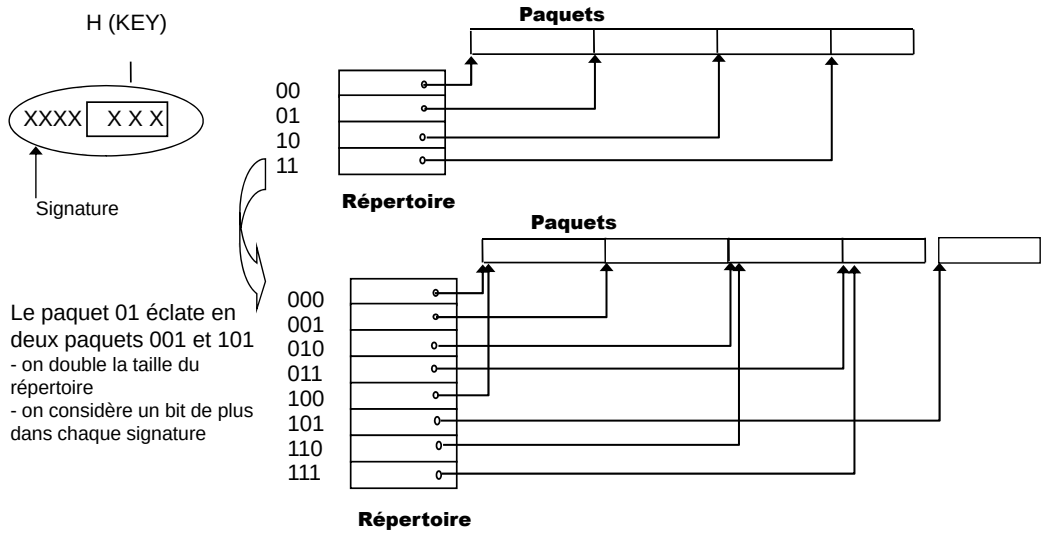
Problème du hachage statique

- Nécessité de réorganisation
  - Un fichier ayant débordé ne garantit plus de bons temps d'accès (1 + p), avec p potentiellement grand
  - Il ne garantit pas non plus l'uniformité (prédictibilité) des temps d'accès
  - Le nombre de paquets primaires est fixe, ce qui peut entrainer un mauvais taux de remplissage
- Solution idéale: réorganisation progressive
  - Il faudrait pouvoir changer dynamiquement la fonction de hachage
  - … mais comment savoir ensuite laquelle appliquer ?

Techniques de hachage dynamique

- Techniques permettant de faire grandir progressivement un fichier haché saturé en distribuant les articles dans de nouvelles régions allouées au fichier.
- LES QUESTIONS CLÉS :
  - (Q1) Quel est le critère retenu pour décider qu'un fichier haché est saturé ?
  - (Q2) Quelle partie du fichier faut-il agrandir quand un fichier est saturé?
  - (Q3) Comment retrouver les parties d'un fichier qui ont été agrandies et combien de fois l'ont-elles été ?
  - (Q4) Faut-il conserver une méthode de débordement et si oui laquelle?

Fichier haché extensible



Comparaisons B+Tree vs. Hachage

- Organisations arborescentes
  - Supporte les *point* (égalité) et *range* (inégalité) *queries*
  - Le nb d'E/S dépend de la hauteur de l'arbre (usuellement h reste très petit)
  - La taille de l'index est potentiellement importante
  - Le coût de mise à jour est potentiellement important

- Organisations hachées
  - Ne supporte que les *point queries*
  - Performance optimale (resp. mauvaise) quand les données sont bien (resp. mal) distribuées