Bases de données avancées

Organisation physique des données

Dan VODISLAV

CY Cergy Paris Université

Master Informatique M1

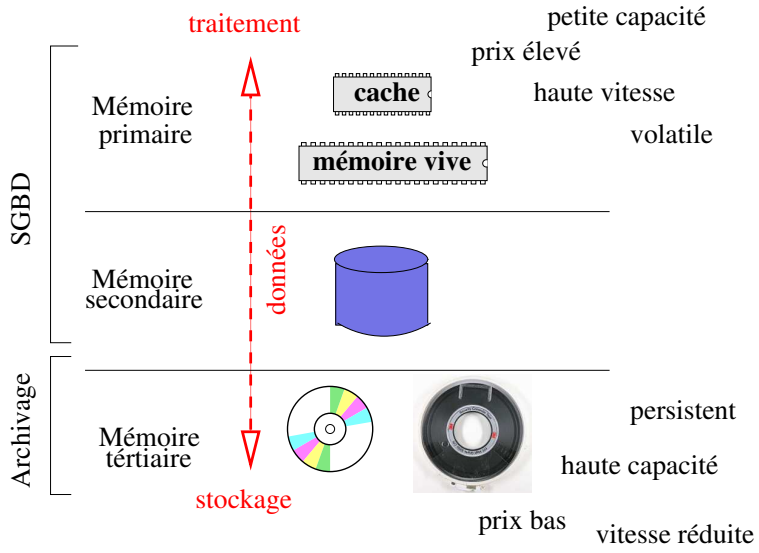
Cours BDA

Plan

- Organisation en mémoire secondaire
  - Hiérarchie de mémoires
  - Pages, articles, fichiers
- Types d'organisation physique
  - Séquentielle
  - Séquentielle triée
  - Séquentielle indexée
  - Arbres B et B+
  - Hachage
- Organisation physique dans Oracle

Organisation en mémoire secondaire

- Hiérarchie de mémoires
  - Différences de vitesse d'accès, prix, capacité



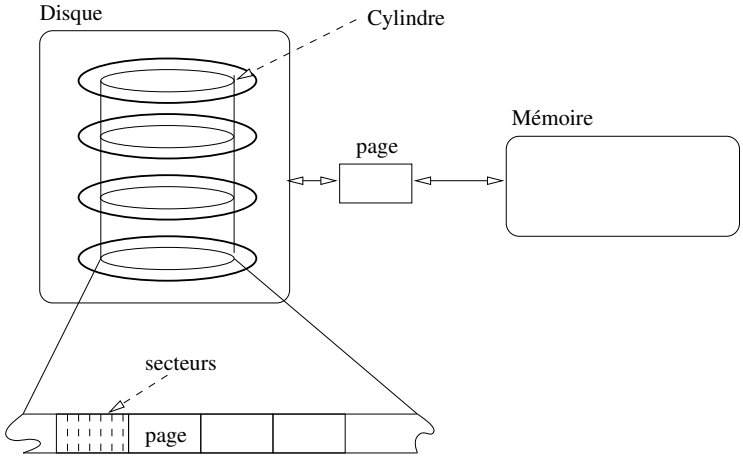
Comparaison mémoire principale / secondaire

- Capacité
  - Mémoire principale: quelques Go
  - Mémoire secondaire: centaines de Go, To
- Prix: comparables à capacité normale
  - Prix / Go : des centaines de fois plus petit pour la mémoire secondaire
- Temps d'accès
  - Mémoire principale: dizaines de nanosecondes
  - Mémoire secondaire: millisecondes

  → Les performances des SGBD sont mesurées essentiellement en nombre d'accès disque

Architecture d'un disque

- Disque divisé en *pages* de taille égale (même nombre de secteurs contigus)
- La page est *l'unité d'échange* entre les mémoires secondaire et principale
  - Coût des opérations: nombre de lectures/écritures de pages
- *Adresse page*: numéro cylindre + numéro face + numéro du premier secteur
- Têtes de lecture synchrones sur chaque face d'un cylindre



Fichiers et articles

- Les données sur disque sont stockées dans *des fichiers*
  - Un fichier occupe plusieurs secteurs/pages sur disque
  - L'accès aux fichiers est géré par le système de gestion de fichiers du système d'exploitation
  - Un fichier est identifié par son nom
- Un fichier stocke un ensemble *d'articles* (enregistrements, n-uplets, lignes)
  - Un article est une séquence de *champs* (attributs).
- Deux types d'articles
  - Articles en format fixe : la taille de chaque champ est fixée
  - Articles en format variable : la taille de certains champs est variable

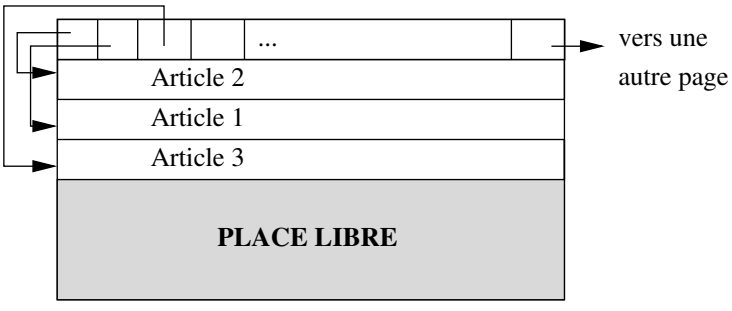
Articles et pages

- Les articles sont stockés dans des pages
  - En général taille article < taille page
  - *Adresse article* (ROWID) = (adresse page, indice de l'article dans la page)

  Important: ne pas modifier l'adresse d'un article si celui-ci est référencé !
- Organisation des pages
  - Nombreuses variantes, en fonction du type d'article (taille fixe/variable), de l'organisation du fichier (séquentielle, triée, …)
  - Page = entête de page + zone articles
  - La zone d'articles est divisée en *cellules*
    - De taille constante (articles de taille fixe)
    - De taille variable (articles de taille variable)
  - Adresse d'un article dans la page: numéro d'ordre de la cellule occupée
  - Suppression d'un article → sa cellule devient disponible pour accueillir ultérieurement un autre article

Organisation générale d'une page

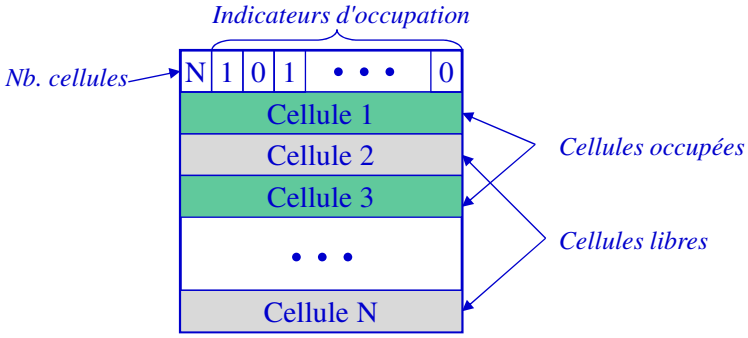
- Entête: répertoire de cellules + infos de chaînage des pages
  - *Répertoire de cellules*: nb. de cellules allouées + tableau de descripteurs de cellule
  - *Descripteur de cellule*: adresse dans la page + indicateur d'occupation
- Zone articles = zone cellules + zone libre
  - Zone cellules: cellules occupées + cellules libres
  - *Suppression* article: cellule marquée libre dans le répertoire
  - *Ajout* article: on cherche une cellule libre de taille suffisante, sinon on alloue une nouvelle cellule dans la zone libre



Articles à taille fixe

- Plus simple à gérer
  - Pas besoin de garder dans le répertoire la position de chaque cellule
    - On garde seulement un bit d'occupation par cellule
  - L'accès aux champs d'un article est simple
    - La position de chaque champ est toujours la même
  - Pas de problèmes de débordement d'un article ou d'une page

→ Économie de place dans la page et accès plus rapide aux articles/champs

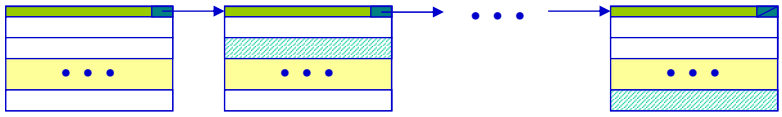


Types d'organisation physique

- Organisation séquentielle
  - Stockage des articles suivant l'ordre d'insertion / suppression
- Organisation séquentielle triée
  - Articles triés selon une clé
- Organisation séquentielle indexée
  - Articles triés selon une clé + index d'accès rapide par clé
- Indexation avec arbres B / B+
  - Amélioration du séquentiel indexé pour supporter des mises à jour
- Organisation par hachage
  - Regroupement des articles suivant une fonction de hachage sur la clé

Organisation séquentielle

- Enchaînement de pages contenant des cellules occupées et des cellules libres (après suppression)



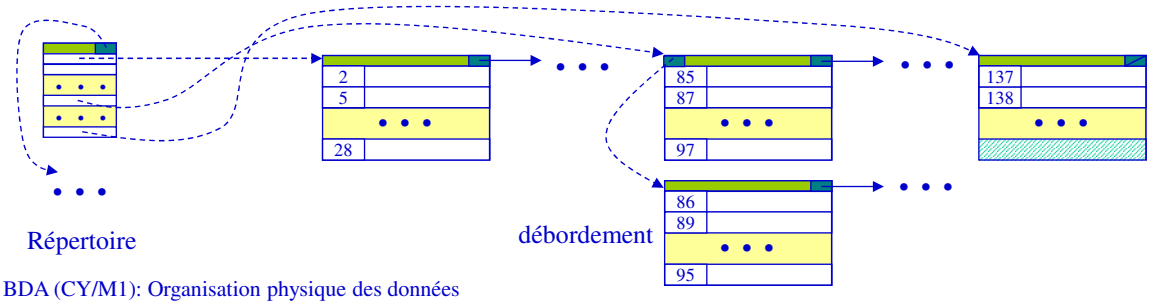
- En pratique : liste de pages pleines + liste de pages avec cellules libres
  - Question: pourquoi deux listes séparées?

Coût des opérations

- **Rappel**: ce n'est pas le nombre d'opérations sur les articles qui compte, mais *le nombre d'opérations sur les pages*
- Soit un fichier avec N pages
  - Coût exprimé en nombre de lectures/écritures de pages
- Recherche par valeur de champ ("clé")
  - Clé unique: en moyenne $(N+1)/2$ lectures de pages → $C_r \approx N/2$
  - Clé non-unique: N lectures → $C_r = N$
- Insertion: écriture dans la première page avec espace libre
  - $C_i = 2$ (1 lecture page + ajout article + 1 écriture page modifiée)
- Suppression: recherche de la page + suppression dans la page
  - Coût = coût recherche + nombre de pages écrites
  - Clé unique: $C_s = N/2 + 1$
  - Clé non-unique: $C_s = N + p$ (p: nb de pages avec des suppressions)
- Modification: même chose que la suppression

Organisation séquentielle triée

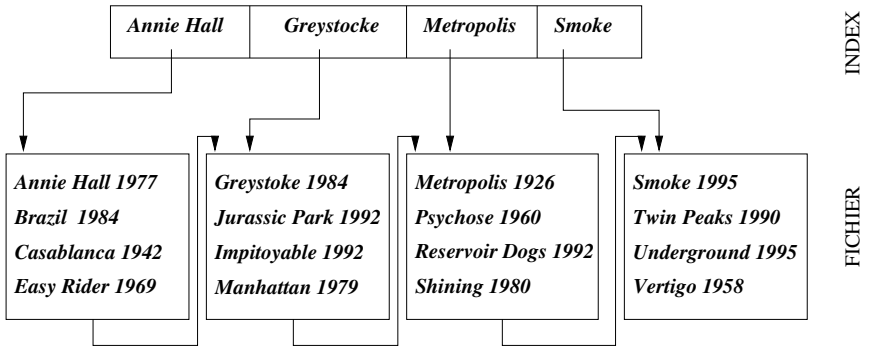
- Le fichier séquentiel est trié suivant la valeur de la clé
  - Répertoire des pages pour accès rapide à la *n*-ième page
- Recherche par clé: recherche dichotomique → $C_r \approx \log_2 N$
- Insertion: recherche position + insertion à cette position
  - Le décalage des articles après la position d'insertion changerait leur adresse!
  - Solution: *pages de débordement* → liste séquentielle de pages de débordement
  - Insertion dans la liste de pages de débordement
    - $C_i = C_r + 2 = \log_2 N + 2$ (insertion dans liste séquentielle)
- Suppression: recherche position + suppression dans la page
  - Si *d* pages de débordement: $C_s = C_r + d/2 + 1 = \log_2 N + d/2 + 1$ (si clé unique)



Organisation séquentielle indexée

- Amélioration de l'organisation séquentielle triée
  - *Fichier de données* avec les articles triés sur la clé
  - On rajoute *un index* sur les clés → répertoire amélioré
- *Index* : second fichier, contenant des clés et les adresses des pages du fichier de données qui correspondent à ces clés
  - Article index: couple (*c*, *a*), *c*=clé, *a*=adresse page
  - Signification: dans la page *a* le premier article a la clé *c*

  → l'index est lui aussi trié sur la clé

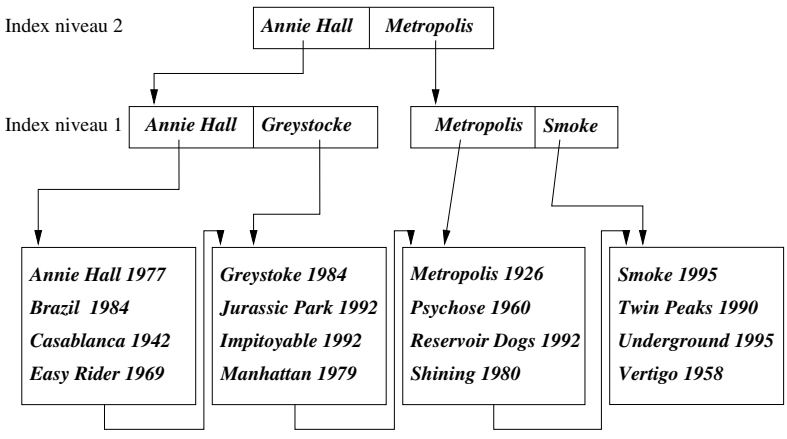


Recherche séquentielle indexée

- Recherche d'articles de clé *c*
  - On cherche dans l'index la plus grande clé *c'* tel que $c' < c$ (recherche dichotomique)
  - On lit la page qui correspond à la clé *c'* et on cherche les articles de clé *c* dans cette page (et éventuellement dans les pages suivantes)
- Avantage: recherche plus rapide car l'index occupe moins de pages que le fichier de données
  - Si l'index a *I* pages, la recherche prend $C_r = \log_2 I + 1$ (ou $\log_2 I + p$)
- Exemple: recherche dans un fichier de *N*=1000 pages, index de *I*=8 pages, clé unique
  - Organisation séquentielle: C_r=1000/2=500
  - Organisation triée: $C_r=\log_2(1000)=10$
  - Organisation indexée: $C_r=\log_2(8)+1=4$

Séquentiel indexé général

- L'index est lui-même un fichier trié → on peut l'indexer
  - On obtient un index à plusieurs niveaux → un arbre
    - Feuilles = fichier de données
    - Nœuds internes = index



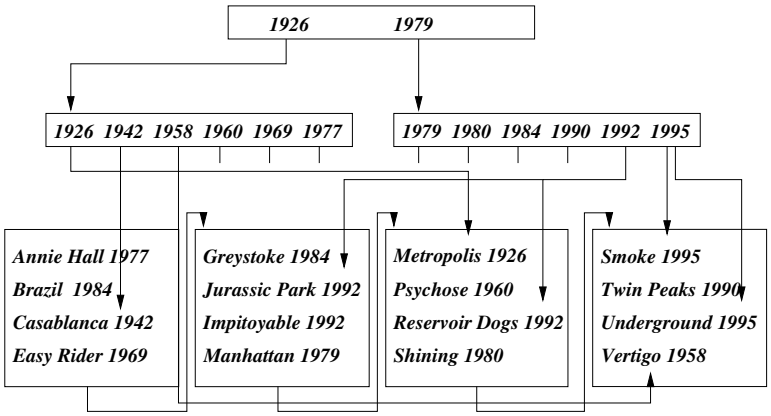
Séquentiel indexé : insertion et suppression

- L'insertion et la suppression se font dans le fichier de données (organisation triée) et peuvent induire des mises à jour de l'index
- Si l'on voulait maintenir le fichier compact → l'index change
- Pages de débordement
  - Avantage: pas de modification de l'index
  - Inconvénient: le temps de recherche se dégrade et varie en fonction de la clé recherchée
- Conclusion: le séquentiel indexé est mal adapté aux fichiers avec mises à jour fréquentes

Catégories d'index

- Selon le nombre de clés dans l'index
  - *Index dense*: toutes les clés du fichier de données se retrouvent dans l'index
  - *Index non-dense*: seulement une partie des clés du fichier de données se retrouvent dans l'index
    - Taille réduite, mais le fichier de données doit être trié
    - Un seul index non-dense par fichier (fichier trié sur la clé)
- Selon l'organisation du fichier de données
  - *Index clustérisé*: l'index et le fichier de données ont la même organisation
    - Proximité de clés dans l'index → proximité des articles dans le fichier
    - Un seul index clustérisé par fichier (fichier organisé suivant l'index)
  - *Index non-clustérisé*: les articles du fichier ont une organisation aléatoire par rapport à celle de l'index
- Question: de quel type est l'index séquentiel indexé précédent?

Exemple d'index dense

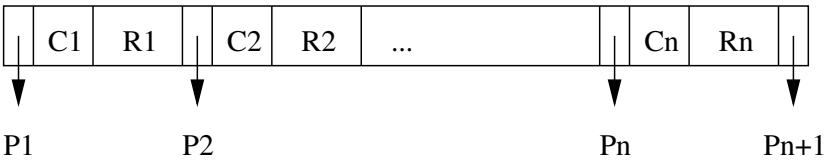


- Question: cet index est-il clustérisé ou non?

Arbre B

- Amélioration du séquentiel indexé pour supporter les mises à jour
  - L'arbre B (et ses variantes) est utilisé dans tous les SGBD relationnels
- Arbre B d'ordre k
  - Arbre *équilibré* (tous les chemins racine-feuille ont la même longueur)
  - *Chaque nœud occupe une page* et contient entre k et $2k$ articles
    - Seule exception: *la racine* qui contient entre 0 et $2k$ articles
  - Les articles d'un nœud sont triés sur la clé
  - Chaque nœud interne avec n articles a n+1 fils
    - Chaque nœud interne est un index pour ses fils/descendants

Structure du nœud d'un arbre B

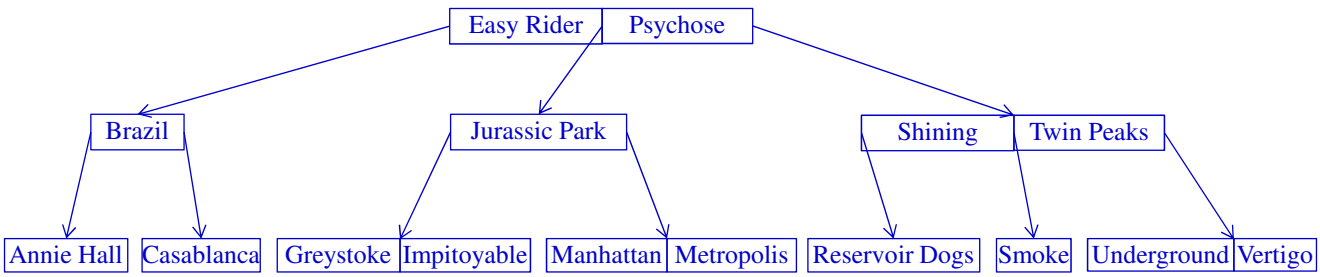


- n articles, en ordre croissant des clés ($k \leq n \leq 2k$)
  - $C_1 \leq C_2 \leq \ldots \leq C_n$: clés
  - $R_1, \ldots, R_n$: information associée aux clés
    - En général R_i = l'adresse physique (ROWID) de l'article de clé C_i
- n+1 pointeurs vers les nœuds fils dans l'index: $P_1, \ldots, P_{n+1}$
  - ***Contrainte***: toutes les clés x dans le sous-arbre pointé par P_i respectent la condition $C_{i-1} \leq x \leq C_i$ $(2 \leq i \leq n)$
    - Pour i=1 → $x \leq C_1$, pour i=n+1 → $C_n \leq x$

  → Les n clés déterminent n+1 intervalles, chaque pointeur P correspond à l'un de ces intervalles

Exemple d'arbre B

- Arbre B d'ordre k=1 sur le titre du film

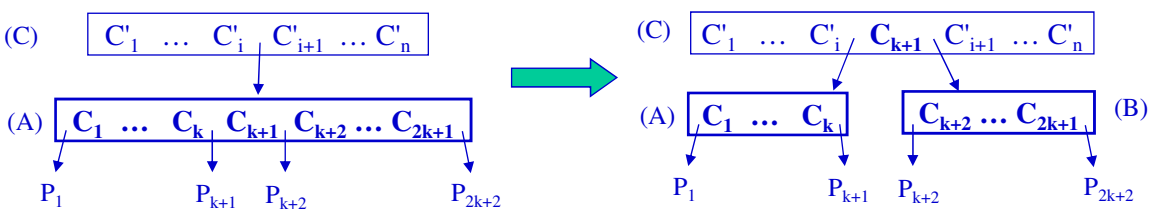


Recherche dans un arbre B

- Recherche des articles de clé c
  - Pour simplifier, considérons que les clés dans l'arbre sont *distinctes*
- On part de la racine de l'arbre
  - Soit C_1, C_2, …, C_n les clés du nœud courant
  - Si c est parmi ces clés (c=C_i) → ***résultat*** = *l'article récupéré à l'aide de* R_i
  - Sinon, si le nœud courant est une feuille → ***résultat*** = *aucun article*
  - Sinon on choisit l'intervalle $C_{i-1} < c < C_i$ (ou $c < C_1$ ou $c > C_n$) dans lequel se trouve c et on continue récursivement avec le nœud fils indiqué par P_i
- Si les clés ne sont pas distinctes:
  - Une clé peut dans ce cas apparaître plusieurs fois dans le même nœud et/ou dans des nœuds différents
  - Si c est parmi les clés du nœud courant (c=C_i) → en plus de l'article récupéré à l'aide de R_i, on cherche aussi dans les fils à gauche (P_i) et à droite (P_{i+1}) de la clé
  - *Remarque*: technique similaire lorsqu'on cherche *un intervalle de clés*

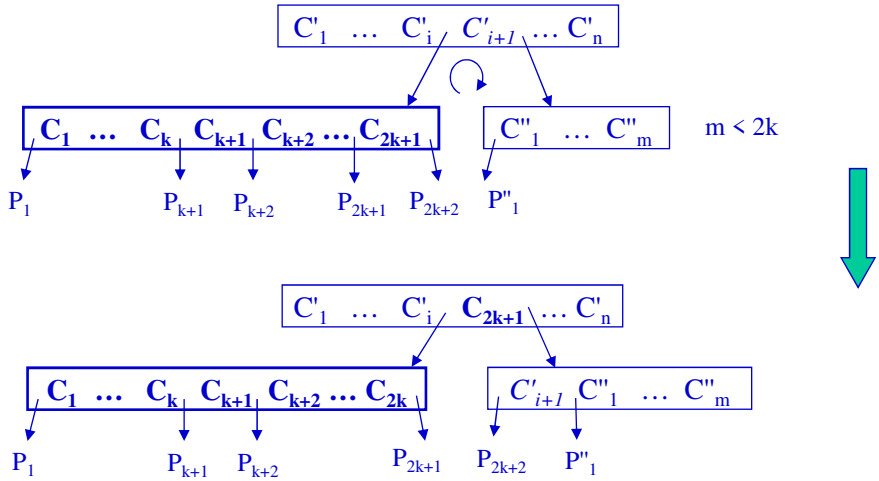
Insertion dans un arbre B

- On considère des clés distinctes et on insère une *nouvelle* clé c
- On cherche la feuille de l'arbre où doit se faire l'insertion (recherche de c)
  - On insère c à sa place dans la feuille (en gardant l'ordre croissant des clés)
- Si le nœud (A) déborde suite à l'insertion (c est la $2k+1^{\text{ième}}$ clé)
  - On crée un nouveau nœud B
  - On garde les k premières clés dans A et on met les k dernières dans B
  - La clé du milieu (la $k+1^{\text{ième}}$) est insérée dans le nœud parent C
    - À gauche de cette clé reste le nœud A, à droite se trouvera le nœud B
  - Si le nœud C déborde suite à cette insertion, on continue selon la même méthode
  - Si la racine déborde, la clé du milieu formera une nouvelle racine à une seule clé



Amélioration de l'insertion

- Pour éviter le débordement à l'insertion (éclatement de nœuds)
  → passage de clés vers des nœuds frères (technique de ***rotation***)
- Si l'un des frères de gauche ou de droite du nœud qui déborde n'est pas plein (moins de 2*k* clés) → on lui passe une clé

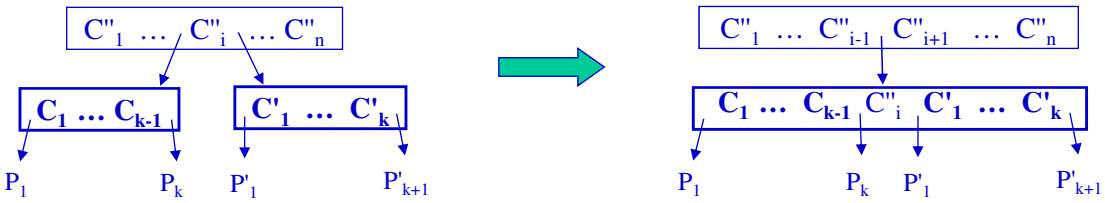


Suppression dans un arbre B

- On considère des clés distinctes et on veut supprimer une clé *c*
- On recherche la clé *c* dans l'arbre
  - Si on ne la trouve pas → rien à faire
  - Si elle est dans une feuille, on supprime la clé dans la feuille
    - Si la feuille reste avec moins de *k* clés (manque de clés) → on fait une *rotation inverse* pour récupérer une clé à partir de l'un des frères voisins
    - Si les frères voisins sont à la capacité minimale (*k* clés) → on ne peut pas emprunter de clé → on *fusionne* avec l'un des frères voisins en récupérant une clé du nœud parent
    - Si le nœud parent reste avec moins de *k* clés, on applique la même méthode
    - Si le nœud parent est la racine et qu'elle n'a qu'une seule clé, la racine disparaît et le nœud de fusion devient la nouvelle racine
  - Si elle n'est pas dans une feuille, on la remplace avec la clé *c'* immédiatement suivante (ou précédente), qui se trouve toujours dans une feuille
    - Ensuite on supprime *c'* de sa feuille suivant la méthode précédente
- Trouver la clé immédiatement suivante à une clé *c* d'un nœud interne
  - Dans le sous-arbre à droite de *c* on prend la clé la plus petite (qui se trouve dans la feuille la plus à gauche)
  - La clé immédiatement précédente: dans le sous-arbre à gauche de *c* on prend la clé la plus grande dans la feuille la plus à droite

    Ex. La clé après "Easy Rider" est "Greystoke", la clé avant "Psychose" est "Metropolis"

Fusion de nœuds à la suppression



Performances d'un arbre B

- Recherche: parcours d'un chemin dans l'arbre à partir de la racine
  - Nombre de lectures de pages ≤ le nombre de niveaux de l'arbre
    - Hauteur de l'arbre *h* = nombre d'arcs chemin racine-feuille = nombre de niveaux - 1
  - En moyenne entre *h* et *h*+1 lectures
  - Si clés non uniques (ou recherche par intervalle) → parcours de plusieurs chemins
- Insertion/suppression : parcours de recherche + insertion/suppression
  - Les éventuels éclatements/fusions suivent un chemin ascendant
    → le nombre d'écritures de pages reste au pire proportionnel à *h*
- Conclusion: les performances d'un arbre B dépendent de sa hauteur
- Question: quelle est la hauteur d'un arbre B d'ordre *k* contenant *n* clés ?

Évaluation de la hauteur d'un arbre B

- Pour un arbre B de hauteur *h*
  - Nombre maximum de clés: $2k$ clés dans tous les noeuds
    $n_{max} = (2k+1)^{h+1} - 1$
  - Nombre minimum de clés: 1 clé dans la racine + k clés dans les autres noeuds
    $n_{min} = 2(k+1)^{h} - 1$

  → la hauteur d'un arbre B d'ordre k stockant n clés

  $\log_{2k+1} (n+1) - 1 \leq h \leq \log_{k+1} ((n+1)/2)$

- Conclusion: la hauteur, donc la complexité des opérations sur l'arbre B est *logarithmique* par rapport au nombre de clés
- On a intérêt à ce que la valeur de k soit la plus grande possible

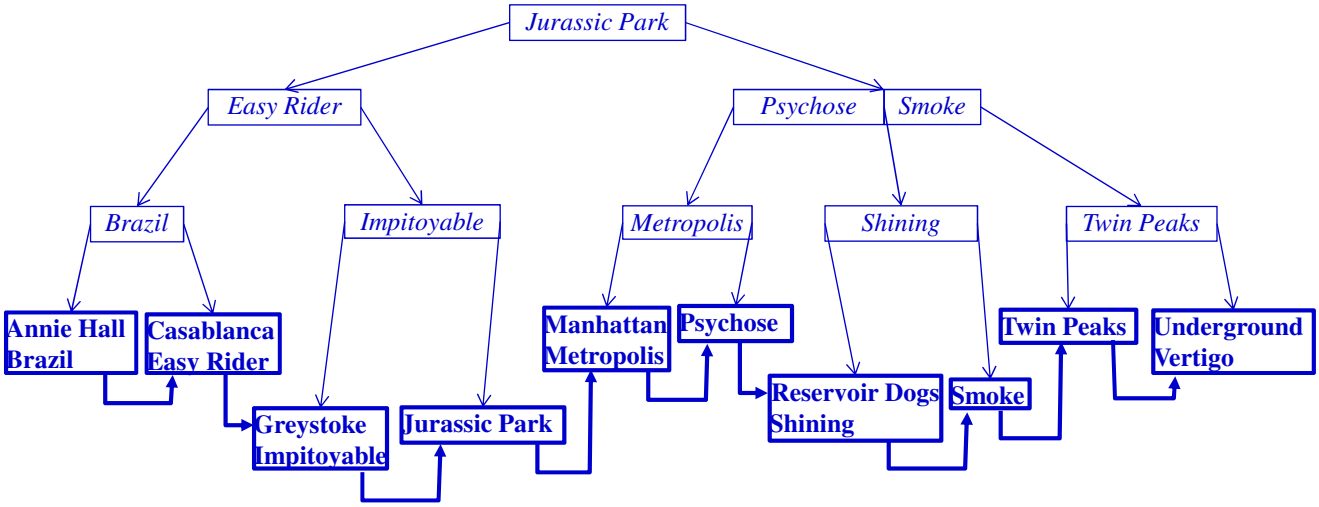
  Ex. pour k=100, si h=2 on peut stocker 8 millions de clés, si h=3 → 1,6 milliards
  - … mais la valeur de k est déterminée par le rapport entre la taille d'une page et la taille d'un article d'index

  → Stocker dans l'index juste *l'adresse* de l'article de données répond à cet objectif

Arbre B+

- Inconvénient de l'arbre B: la recherche par intervalle de clés / clés non uniques
  - Il faut parcourir tout un sous-arbre de l'arbre B
- Arbre B+: variante de l'arbre B qui simplifie la recherche par intervalles, tout en gardant les avantages de l'arbre B
- Principes
  - Toutes les clés de l'arbre se retrouvent dans les feuilles
    - Seules les feuilles contiennent les infos sur les articles de données
    - Les feuilles sont chaînées entre elles
  - Les clés des nœuds internes ne servent qu'à guider la recherche de feuilles
    - Redondance: les clés des nœuds internes se retrouvent aussi dans les feuilles
    - Les nœuds internes forment un arbre B au-dessus des feuilles!
- Variantes
  - Les feuilles contiennent les adresses des articles de données
  - Les feuilles contiennent les articles de données (index clustérisé)

Exemple d'arbre B+



- Remarques
  - Les nœuds internes stockent seulement des clés C_i (pas de R_i)
  - Les feuilles n'ont pas besoin de stocker des pointeurs P_i
  - → En pratique, le degré k des nœuds internes pourrait être différent de celui des feuilles
  - Le nombre de clés internes = nombre de feuilles – 1

Opérations sur l'arbre B+

- Recherche de clé
  - Similaire à l'arbre B, mais on descend toujours jusqu'aux feuilles
- Recherche par intervalle
  - On cherche la clé de début d'intervalle, ensuite on parcourt séquentiellement les feuilles jusqu'à la fin de l'intervalle
- Insertion
  - En cas de débordement de la feuille, la clé médiane qui remonte vers le nœud parent est aussi gardée dans l'une des deux feuilles résultantes
  - La rotation est un peu différente, à cause de la redondance des clés
- Suppression
  - La suppression se fait toujours dans une feuille
  - Modification de clé dans le nœud parent si l'on supprime une clé qui est également présente au niveau du parent
  - La fusion de feuilles est similaire, sauf que la clé qu'on récupère du nœud parent se trouve déjà dans les feuilles qui fusionnent

Performances d'un arbre B+

- Hauteur de l'arbre
  - Très proche de celle d'un arbre B
- Recherche de clé
  - On fait toujours h_{B+}+1 lectures (les adresses d'articles → dans les feuilles)
- Recherche par intervalle
  - Très efficace: h_{B+} + nb. feuilles avec clés dans l'intervalle
- Insertion, suppression
  - Proportionnelles à h_{B+}, donc très proches de celles pour l'arbre B

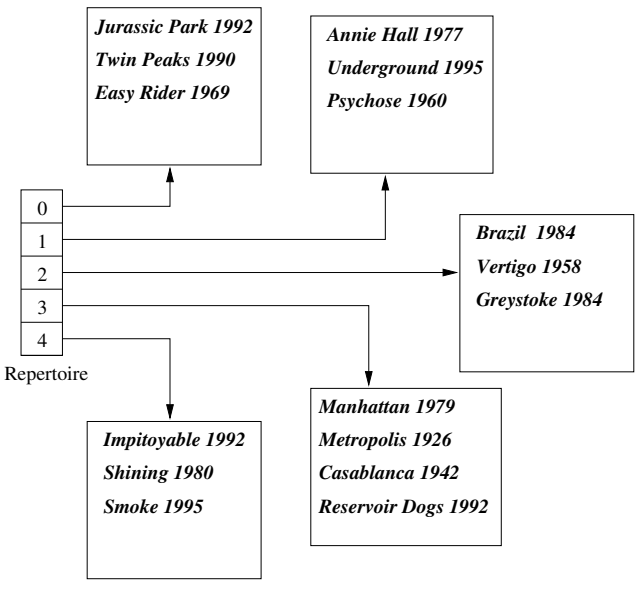
Organisation par hachage

- Les articles du fichier de données sont regroupés suivant une fonction de hachage sur la clé de l'article
- Soit N la taille (en pages) estimée pour le fichier de données
  - On choisit en principe une valeur plus grande que la taille courante du fichier, pour laisser de la place pour les insertions
  - On alloue N pages pour le fichier organisé par hachage
- Fonction de hachage sur $\mathbf{C}$ (l'ensemble de clés possibles)
  $H : \mathbf{C} \rightarrow \{0, 1, \ldots, N\text{-}1\}$
- Idée: tous les articles ayant une clé c tel que $H(c) = i$ sont rangés dans la page i
  - On utilise un répertoire à N entrées, qui renvoie vers chaque page
    - En principe ce répertoire entre en mémoire (sinon la recherche est plus chère)
  - La fonction H doit produire uniformément les valeurs $\{0, 1, \ldots, N\text{-}1\}$

Exemple de hachage

- Soit le fichier de films utilisé comme exemple
  - Clé = titre du film
  - On considère 4 articles par page, il y a en tout 16 articles
  - Les 16 articles pourraient n'occuper que 4 pages, mais on choisit *N*=5
- Fonction de hachage
  - On prend la première lettre du titre et on lui attribue une valeur de 1 à 26
    - No('a')=1, No('b')=2, …, No('z')=26
  - On prend le reste de la division de cette valeur par *N*

Résultat



Opérations par hachage

- Recherche d'articles ayant la clé *c*
  - On calcule $i = H(c)$
  - On consulte l'entrée *i* dans le répertoire *R*
  - On lit la page à l'adresse $R(i)$ et on cherche les articles de clé *c* là-dedans

→ Une seule lecture de page !

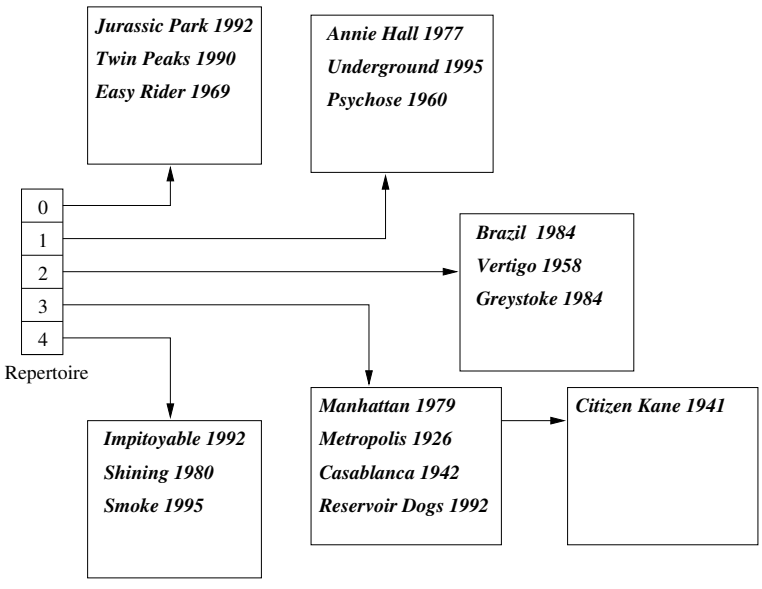
- Insertion d'article de clé *c*
  - On trouve la page qui correspond à $H(c)$ et on insère l'article
  - Si la page est pleine → on crée une *page de débordement* et on insère l'article dans cette page

→ La recherche peut avoir besoin de plus d'une lecture de page

- Suppression d'article de clé *c*
  - Recherche de l'article et suppression dans la page en question

Exemple avec insertion

- L'exemple précédent après insertion de ("Citizen Kane", 1941)



Hachage: avantages et inconvénients

- Avantages
  - Très rapide: une seule lecture de page (si pas débordement)
  - N'occupe pas d'espace disque dédié (si répertoire en mémoire)
- Inconvénients
  - Si le fichier évolue beaucoup il faut refaire un hachage
    - Recalcul de la bonne valeur pour *N*
    - Suppression sans recompactage → les listes de débordement diminuent rarement (on peut rester avec de nombreuses pages très peu remplies)
  - *On ne peut pas faire de la recherche par intervalles !*

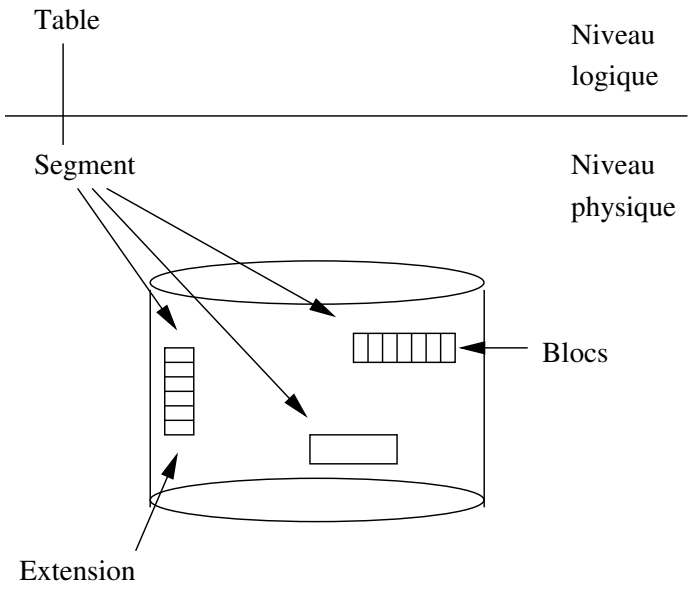
Comparatif des temps de recherche

Organisation	**Coût**	**Avantages**	**Inconvénients**
Séquentiel	$N/2$ - N	Simple	Coûteux
Séquentiel trié	$\log_2(N)$ + *const*	Efficace, intervalles	Coût des mises à jour
Séquentiel indexé	$\log_2(I)$ + *const*	Efficace, intervalles	Coût des mises à jour
Arbre B / B+	$\log_k(N)$ + *const*	Efficace, intervalles	Traversée de l'arbre
Hachage	*const*	Le plus efficace	Pas d'intervalles

Organisation physique dans Oracle

- Structures d'organisation physique
  - **Le bloc** (la page): l'unité physique d'entrée-sortie
    - Multiple de la taille des blocs du système de fichiers
  - **L'extension**: ensemble de blocs *contigus*, contenant un même type d'information
    - La lecture/écriture d'une extension de *n* blocs est nettement plus rapide que celle de *n* blocs quelconque
  - **Le segment**: ensemble d'extensions stockant un objet logique (table, index, etc.)

Blocs, extensions et segments

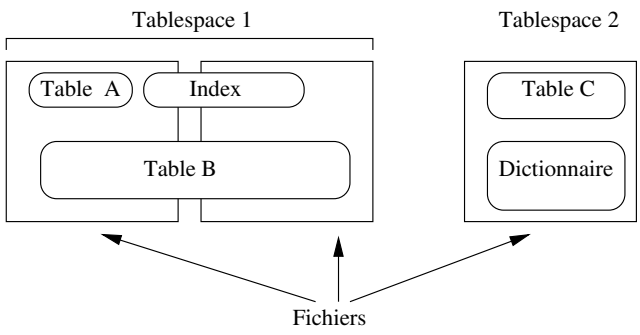


Types de segments

- *Segments de données*
  - Tables, clusters de tables
- *Segments d'index*
- *Segments temporaires*
  - Utiles pour les calculs temporaires (tri, group by, construction index, …)
- Remarque: un autre type de segment était utilisé dans les versions précédentes → segment "*rollback*"
  - Utilisé pour l'annulation des transactions
  - Versions plus récentes: gestion automatique des annulations

Tablespace, fichiers

- Espace Oracle sur disque: deux niveaux d'abstraction
  - *Logique*: ensemble de "***tablespace***"
  - *Physique*: ensemble de ***fichiers***
- Tablespace
  - Organisé en un ou plusieurs fichiers, dont il cache l'utilisation exacte
  - Simplifie la gestion des données: répartition, sauvegarde, protection, …
    - Règles spécifiques pour chaque tablespace, pas pour toute la base



Fichiers, segments, extensions

- Fichiers et segments
  - Un segment est toujours dans un tablespace
  - Par contre un segment peut être étalé sur plusieurs fichiers
- Fichiers et extensions
  - Une extension est toujours dans un seul fichier
  - L'allocation des extensions dans un segment est transparente entre les fichiers du tablespace

Stockage de tables

- Deux façons principales
  - *Indépendante*: la situation la plus courante
    - Le segment de données alloué automatiquement par Oracle
    - Paramètres segment
      - Taille initiale
      - Pourcentage d'espace libre dans chaque bloc
      - Taille des extensions
  - *Dans un cluster*: deux tables souvent jointes, organisées pour accélérer la jointure

Exemples de paramétrages

- Tablespace

```
CREATE TABLESPACE exemple
  DATAFILE 'dataex.ora' SIZE 100M
  DEFAULT STORAGE (INITIAL 10K NEXT 50K PCTINCREASE 20
                   MINEXTENTS 1 MAXEXTENTS UNLIMITED)
  ONLINE
```

  - Tablespace *'exemple'* dans le fichier *'dataex.ora'* (plusieurs fichiers sont possibles) de taille 100 Mo, rendu disponible (online)
  - Pour tout objet/segment créé dans le tablespace
    - La taille de la première extension est de 10 Ko, de la seconde de 50 Ko et les suivantes augmentent de 20% à chaque fois (60 Ko, 72 Ko, …)
    - Il y a au minimum une extension et pas de limite supérieure

Exemples de paramétrages (suite)

- Tables

```
CREATE TABLE salgrade (
    grade NUMBER PRIMARY KEY USING INDEX TABLESPACE user,
    salmin NUMBER,
    salmax NUMBER,
    description VARCHAR2(50))
  TABLESPACE exemple
  PCTFREE 10 PCTUSED 75
```

  - Table *'salgrade'* stockée dans le tablespace *'exemple'*
  - Index sur la clé primaire stocké dans un autre tablespace (*'user'*)
  - PCTFREE: pourcentage d'un bloc réservé pour les mises à jour
  - PCTUSED: pourcentage de remplissage d'un bloc à partir duquel on peut insérer des lignes (n-uplets)
    - Quand l'occupation du bloc descend sous 75% (PCTUSED), on peut insérer des n-uplets sans dépasser un remplissage de 90% (100-PCTFREE)

ROWID

- Un n-uplet est stocké dans un bloc (en général)
- ROWID = l'adresse physique d'un n-uplet
  - Ne change pas suite à des mises à jour
  - Le moyen le plus rapide d'accéder à un n-uplet
  - Quatre composantes (la dernière à partir d'Oracle 8)
    - L'adresse dans le bloc
    - Le numéro du bloc dans le fichier
    - Le numéro du fichier dans le tablespace
    - Le numéro de l'objet/segment dans le tablespace

Index B+ Oracle

- Le plus utilisé, peut être créé sur tout attribut ou liste d'attributs d'une table

```
CREATE INDEX salgrade_salmin ON salgrade(salmin)
CREATE INDEX salgrade_salminmax ON salgrade(salmin,salmax)
```

- Stockage: dans un segment propre, différent de celui de la table
  - Les nœuds internes contiennent les valeurs de l'attribut (des attributs) indexé(s)
  - Les feuilles contiennent chaque valeur indexée + le ROWID correspondant

Clusters

- Cluster (regroupement): structure permettant d'optimiser la jointure de deux tables

Ex. **Cinéma** (*ID-cinéma*, Nom-cinéma, Adresse)
Salle (*ID-cinéma, ID-salle*, Nom-salle, Capacité)

Supposons que *Cinéma* et *Salle* sont souvent jointes sur *ID-cinéma*

- Organisation du cluster
  - On groupe les n-uplets de *Cinéma* et de *Salle* ayant la même valeur pour l'attribut *ID-cinéma*
  - On stocke ces groupes de n-uplets dans les pages du cluster
  - On crée un index sur *ID-cinéma*

Exemple de cluster

Clé de regroupement (ID–cinéma)	**Nom–cinéma**	**Adresse**	
1209	Le Rex	2 Bd Italiens	
	ID–salle	**Nom–salle**	**Capacité**
	1098	Grande Salle	450
	298	Salle 2	200
	198	Salle 3	120
1210	**Nom–cinéma**	**Adresse**	
	Kino	243 Bd Raspail	
	ID–salle	**Nom–salle**	**Capacité**
	980	Salle 1	340
	...	...	...

Exemple de cluster (suite)

- Création cluster ("index cluster")

```
CREATE CLUSTER cluster-cinéma (ID-cinéma NUMBER(10))
  SIZE 2K
  INDEX
  STORAGE (INITIAL 100K NEXT 50K)
```

  - SIZE: taille données pour une valeur de *ID-cinéma*
  - INDEX: le type de cluster (ici "index")
  - STORAGE: les tailles des extensions du segment cluster

- Création index sur le cluster

```
CREATE INDEX idx-cluster ON CLUSTER cluster-cinéma
```


Exemple de cluster (suite)

- Création des tables dans le cluster

```
CREATE TABLE Cinéma (
  ID-cinéma NUMBER(10) PRIMARY KEY,
  Nom-cinéma VARCHAR2(32),
  Adresse VARCHAR2(64))
CLUSTER cluster-cinéma(ID-cinéma)

CREATE TABLE Salle (
  ID-cinéma NUMBER(10),
  ID-salle NUMBER(3),
  Nom-salle VARCHAR2(32),
  Capacité NUMBER)
CLUSTER cluster-cinéma(ID-cinéma)
```

Hachage

- Cluster de hachage ("hash cluster")
  - Cluster qui stockera la table en question suivant les caractéristiques définies pour le hachage

```
CREATE CLUSTER hash-cinéma (ID-cinéma NUMBER(10))
  HASHKEYS 1000
  HASH IS ID-cinéma
  SIZE 2K
```

  - SIZE: taille des données pour une valeur de *ID-cinéma* (taille n-uplet)
  - HASHKEYS: nombre de valeurs de la clé de hachage (taille répertoire de hachage)
  - HASH IS: valeur sur laquelle on fait le hachage (ex. 2**ID-cinéma*+1)
- Création table: référence vers le cluster de hachage

```
CREATE TABLE Cinéma (...)
CLUSTER hash-cinéma(ID-cinéma)
```