SGBD-R : système

Indexation

Guillaume Raschia — Nantes Université

originaux de Philippe Rigaux, CNAM

Dernière mise-à-jour : 3 mars 2023

Plan de la session

- Structures d'indexation (S3.1)
- Arbre B (S3.2)
- Le hachage (S4)
  - Hachage statique
  - Hachage extensible
  - Hachage linéaire
- Index bitmap

Structures d’indexation (S3.1)

Structures d'index

Contenu de cette section :

- Qu'est-ce qu'un index ?
- Principes généraux
- Index denses et non-denses

En l'absence d'index, la seule solution est le parcours séquentiel du fichier.

Avec un index : accès direct à l'enregistrement.

Amélioration très significative des temps de réponse.

Exemple : une table de films

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

Volumétrie de l'exemple

La table des films, avec :

- 1 000 000 (un million) de films
- Un enregistrement = 1200 octets
- Un bloc = 4Ko, donc il y a 3 enregistrements par bloc
- Environ 300 000 blocs, soit 1,2 Go de fichier de données

Index = concept bien connu

Prenons un livre à contenu technique (pas une fiction !), par exemple un livre de recettes. Il contient—au moins—un index.

- L'index présente les termes importants, classés par ordre alphabétique
- À chaque terme sont associés les numéros de page où on trouve le terme
- En parcourant l'index (par dichotomie !) on trouve la ou les page(s) qui nous intéressent

Les index informatiques

Même principe ! C'est un fichier qui permet de trouver un enregistrement dans une table.

Vocabulaire :

- Clé d'indexation = une liste d'un ou plusieurs attribut(s)
- Une adresse (déjà vu) est une adresse de bloc ou une adresse d'enregistrement
- Entrée d'index = un enregistrement de la forme `[valeur, addr]`, où `valeur` est une valeur de clé d'indexation

L'index est trié sur la clé d'indexation (`valeur`).

Exemples

Clés de recherche :

- Le titre du film (c'est aussi la clé primaire)
- l'année du film
- Une paire constituée de l'année, suivie du titre. L'ordre importe !

Opérations de recherche :

- *Vertigo*
- les films parus entre 1960 et 1975
- les films commençant par 'V'
- les films de 1992 commençant par 'R'

L'index ne sert à rien pour toute recherche ne portant pas sur la clé.

Le cas des dictionnaires

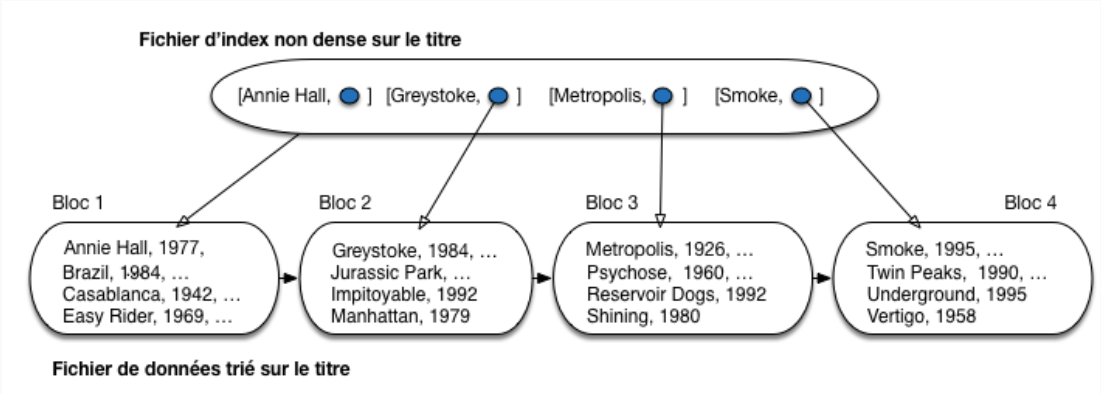
Les dictionnaires ont la particularité de trier les termes.

On peut alors créer un index qui ne référence que le premier mot de chaque page.

```
- ...
- ballon, page 56
- bille,  page 57
- bulle,  page 58
- cable,  page 59
- ...
```

On peut quand même utiliser cet index pour trouver n'importe quel mot.
(« armée », « crabe », « botte », « belle »).

Index non dense



Hypothèse : le fichier de données est trié sur la clé, comme un dictionnaire.
L'index ne référence que la première valeur de chaque bloc.

Opérations

Recherches :

- **Par clé** (ou par point) : je recherche *Shining*
  - recherche dichotomique sur le fichier d'index séquentiel
- **Par intervalle** : tous les films entre *Greystoke* et *Psychose*
- **Par préfixe** : tous les films commençant par 'M'

Mais pas par suffixe : tous les films terminant par 'e' ?

Exemple « concret »

Sur notre fichier de **1,2 Go**

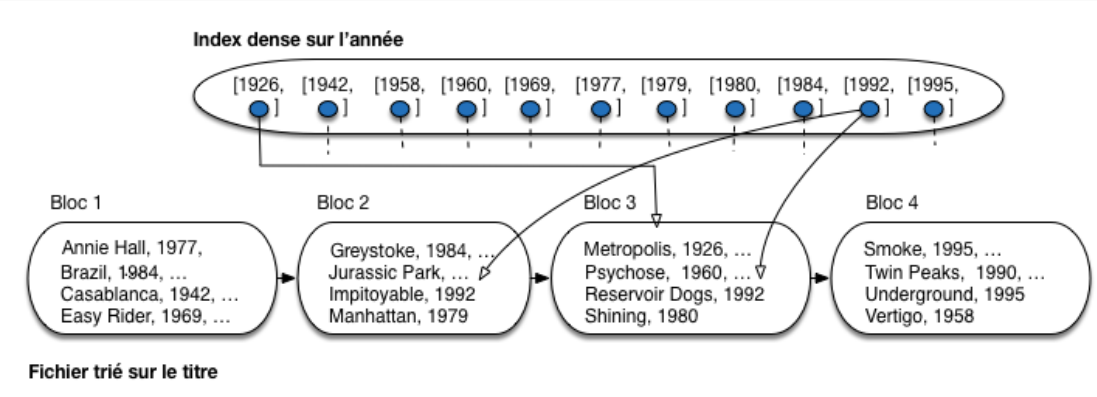
- En supposant qu'un titre occupe 20 octets, une adresse 8 octets
- Taille de l'index : $300\,000 \times (20 + 8)$ = **8,4 Mo**

Beaucoup plus petit que le fichier de données !

Problème

Maintenir l'ordre sur le fichier et sur l'index.

Index dense



Fichier de données non trié. Toutes les valeurs de clé sont représentées.

Exemple concret

Sur notre fichier de **1,2 Go**

- Une année = 4 octets, une adresse 8 octets
- Taille de l'index : $1\,000\,000 \times (4 + 8)$ = **12 Mo**

Encore $100\times$ plus petit que le fichier de données.

Recherches :

- **Par clé** : comme sur un index non dense
- **Par intervalle** (exemple [1950, 1979]) :
  1. recherche, dans l'index, de la borne inférieure
  2. parcours séquentiel *dans l'index*
  3. à chaque valeur : accès au fichier de données

La recherche par intervalle induit des accès aléatoires.

Index primaire et index secondaire

Sur un fichier de données, on peut créer :

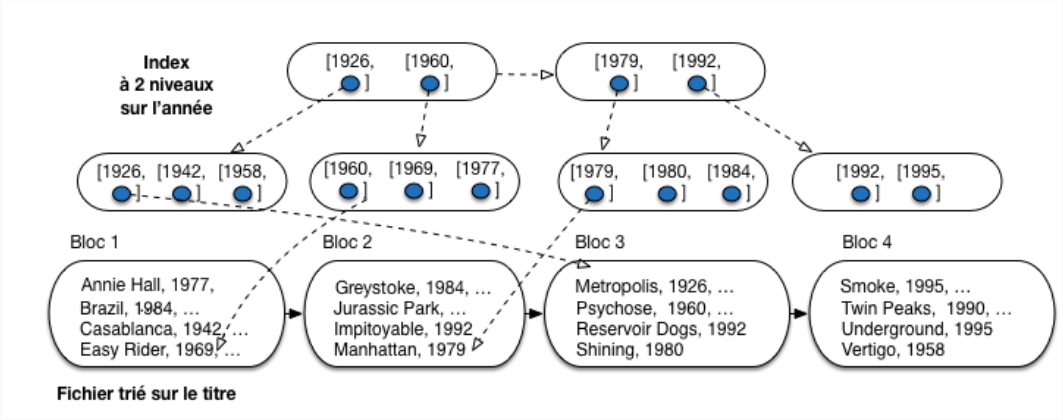
- un seul index non dense, il est plaçant (ou *cluster* en anglais)
- un nombre quelconque d'index denses, non plaçants

Index sur clé primaire

Une table est généralement indexée sur sa clé primaire : on parle d'index primaire. De plus, si les données sont triées sur la clé, l'index est non dense, donc plaçant !

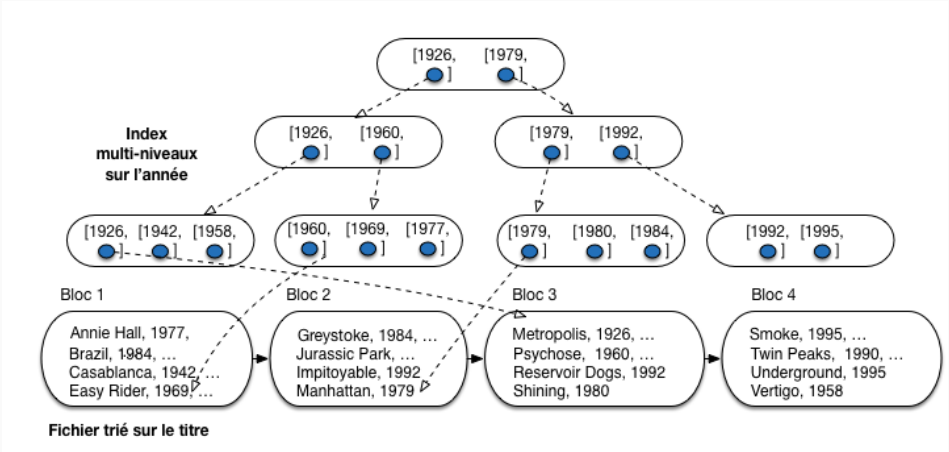
Tous les autres index sont dit secondaires, et non plaçants.

Index multi-niveaux : on indexe l'index



Essentiel : l'index est trié, donc on peut l'indexer par un second niveau non dense
Sinon ça ne servirait à rien : ✎ pourquoi ?

Index multi-niveaux : jusqu'où ?



- Arrêt quand la racine est constituée d'un seul bloc
- Structure hiérarchique ; recherche de haut en bas

Résumé

Retenir :

- Un index accélère des opérations de recherche portant sur la clé d'indexation ou sur un préfixe de la clé.
- La structure repose sur le tri des valeurs de clé indexées.
- On peut créer un index non dense sur un fichier trié, dense sur un fichier non-trié.
- Les index multi-niveaux s'appuient sur un premier niveau dense, puis sur des niveaux supérieurs non denses.

Les index ont un coût, il faut les maintenir au fur et à mesure des insertions dans le fichier de données indexé.

Arbre B (S3.2)

Arbre B

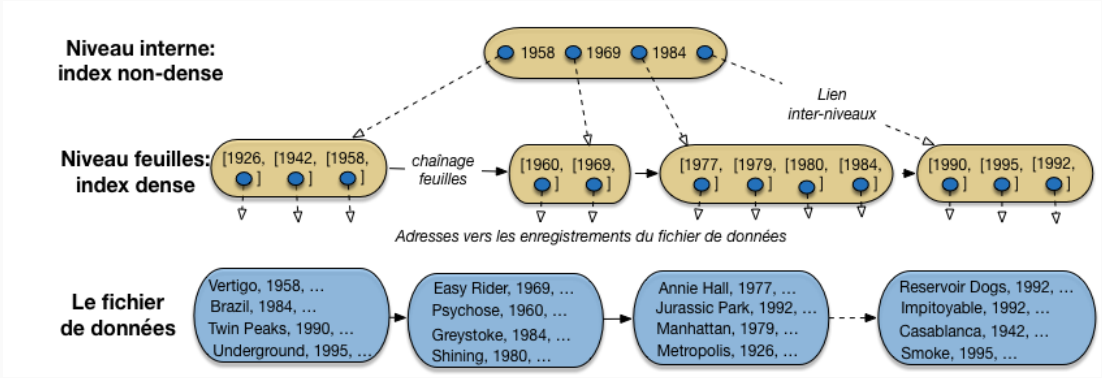
Aboutissement des structures d'index basées sur **l'ordre** des données

- c'est un arbre équilibré
- à chaque nœud correspond un bloc qui agit comme un index local
- il se réorganise dynamiquement
- B comme « Balanced » ? « Bayer » ? « Boeing » ?

Utilisé universellement !

Comparable à l'index séquentiel multi-niveaux, sans la maintenance (tri) des fichiers d'index.

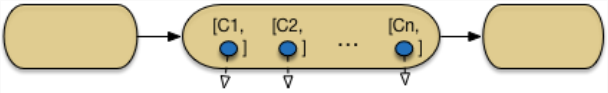
Exemple d'arbre B



Nœud feuille d'un arbre B

Un nœud feuille est un index dense local, contenant des entrées d'index.

Hypothèse : le fichier de données n'est pas trié sur la clé d'index (non plaçant)

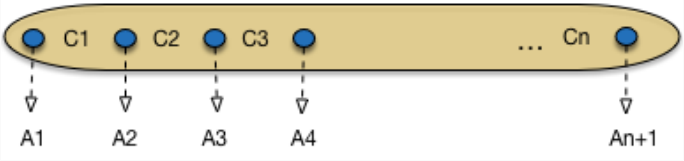


Chaque entrée référence un (ou plusieurs) enregistrement(s) du fichier de données : celui (ceux) ayant la même valeur de clé que l'entrée.

Nœud interne d'un arbre B

Un nœud interne est un index non dense local, les enregistrements servant de clé, intercalés avec des pointeurs.

L'arbre B+ : variante commune dans laquelle les nœuds internes ne contiennent que les clés et les pointeurs.



Le nœud interne contient n clés et $n+1$ pointeurs.

Les feuilles du sous-arbre référencé par A_2 contiennent tous les enregistrements dont la clé c est comprise entre $C_1 < c \leq C_2$.

Avec notre fichier de 1M de films

Admettons qu'une entrée d'index occupe 12 octets, soit 8 octets pour l'adresse, et 4 pour la clé (l'année du film).

Chaque bloc contient 4 096 octets. On place donc $\lfloor \frac{4\,096}{12} \rfloor = 341$ entrées (au mieux) dans un bloc.

- Il faut $\lceil \frac{1\,000\,000}{341} \rceil = 2\,933$ blocs pour le niveau des feuilles.
- Le deuxième niveau est non dense. Il comprend autant de pointeurs que de blocs à indexer, soit 2 933. Il faut donc $\lceil \frac{2\,933}{342} \rceil = 9$ blocs.
- Finalement, un troisième niveau, constitué d'un bloc avec 8 entrées et 9 pointeurs suffit pour compléter l'index.

Capacité d'un arbre B

- avec un niveau d'index (la racine seulement) on peut référencer 341 films ;
- avec deux niveaux, on indexe 342 blocs de 341 films chacun, soit $341 \times 342 = 116\,622$ films ;
- avec trois niveaux on indexe $341 \times 342^2 = 39\,884\,724$ films ;
- enfin avec quatre niveaux on indexe plus de 13 milliard de films !

La hauteur de l'arbre est logarithmique par rapport au nombre d'enregistrements.

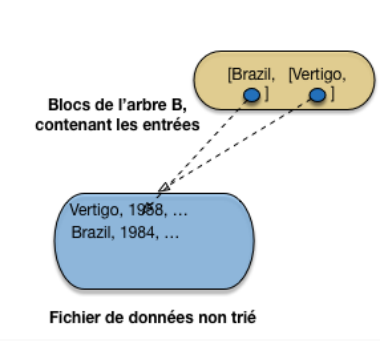
Inversement, avec un arbre de hauteur h, on indexe une collection de taille exponentielle en h.

Insertion dans un arbre B

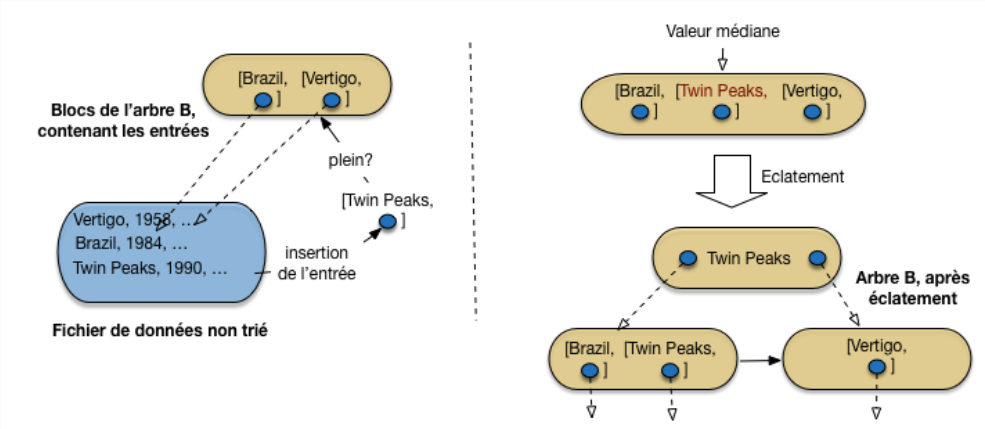
On construit l'index sur le titre des films. La taille variable et la longueur des clés d'index sont des facteurs aggravants pour le poids et donc pour l'efficacité de l'arbre B.

On suppose un arbre B dont les nœuds ont une capacité de 2 entrées.

Situation initiale :

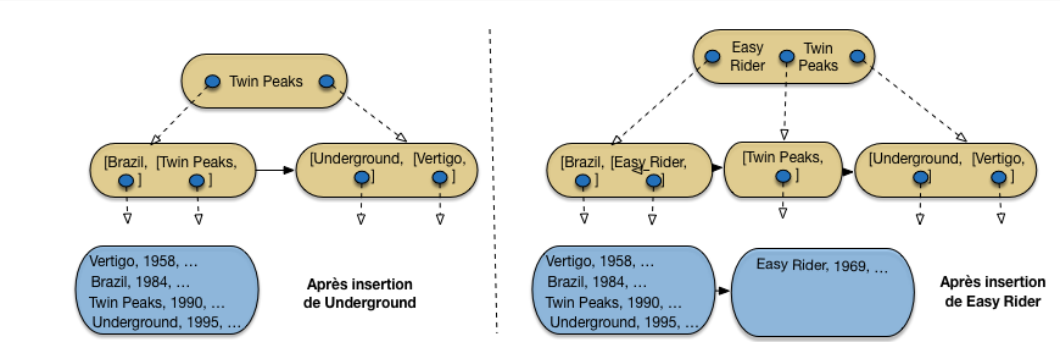


La procédure d'éclatement



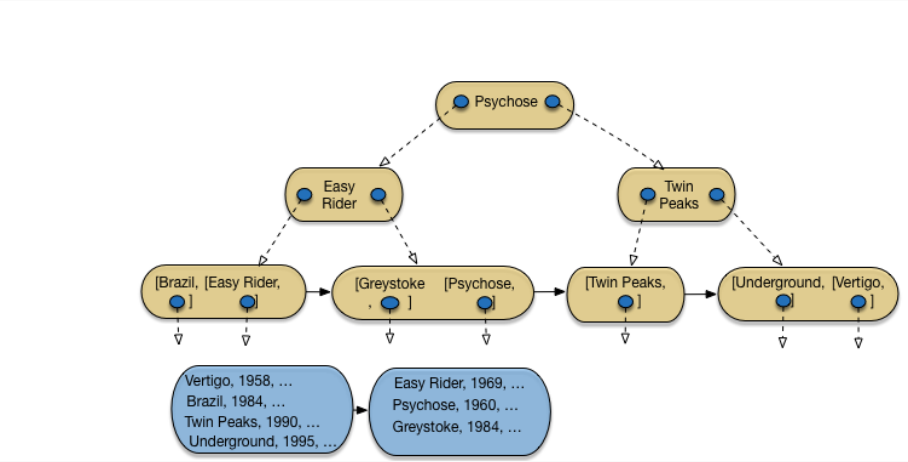
Quand un nœud est plein, il faut effectuer un éclatement.

Les insertions continuent



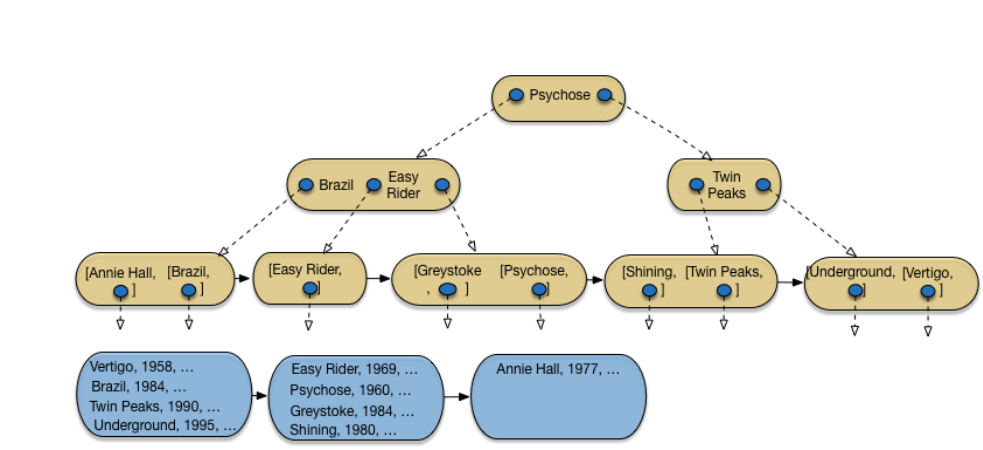
Underground, puis *Easy Rider*.

Ajout d'un niveau



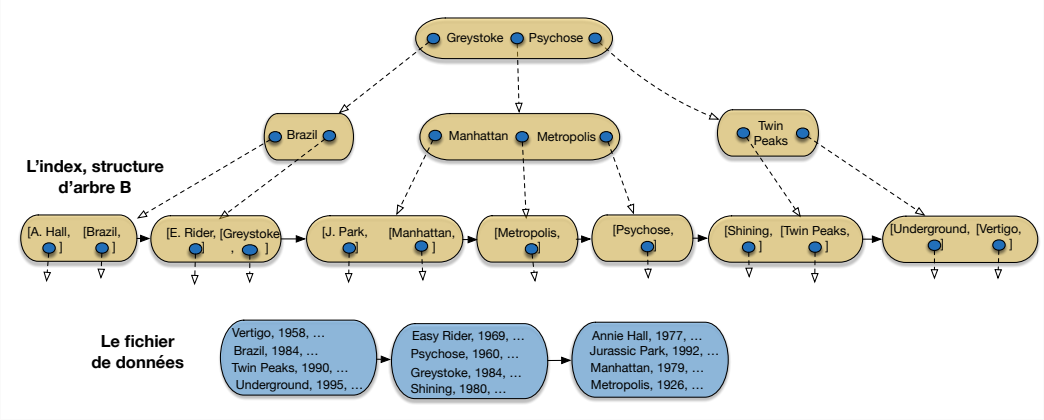
Après *Psychose* puis *Greystoke*, l'éclatement de la racine entraîne l'ajout d'un niveau. Les feuilles sont toutes à la même distance de la racine !

Encore des insertions



Shining et *Annie Hall*.

Toujours des insertions



Après l'insertion de *Jurassic Park*, *Manhattan* et *Metropolis*.

Ordre k d'un arbre B

L'ordre k d'un arbre B désigne le nombre minimal de clés dans les nœuds. Il traduit *de facto* le degré minimal des nœuds internes ($= k+1$).

Par extension, k fixe le taux de remplissage minimal (50%) de chaque bloc du fichier d'index :

$$k \leq n \leq 2k, \text{ où } n \text{ désigne le nombre d'entrées (clés) d'un nœud.}$$

✎ Seule la racine de l'arbre échappe à cette règle d'or. Pourquoi ?

En pratique, on observe un taux moyen de 70%.

L'exemple de l'index sur les années des films

On a déjà observé que la capacité d'un nœud vaut $\lceil \frac{4\,096}{12} \rceil = 341$. Donc l'ordre de l'arbre B vaut $k = \lfloor \frac{341}{2} \rfloor = 170$.

Insertion dans un arbre B d'ordre k

On recherche la feuille de l'arbre où l'enregistrement doit prendre place et on l'y insère. Si le bloc p déborde (il contient $2k + 1$ éléments) :

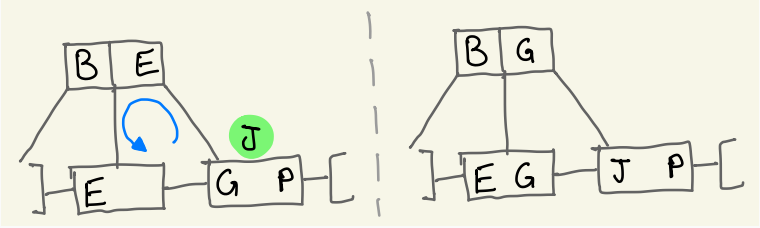
1. On alloue un nouveau bloc p'.
2. On place les k premiers enregistrements (ordonnés selon la clé) dans p et les k derniers dans p'.
3. On insère le $k + 1^{\text{e}}$ enregistrement (médian) dans le père de p. Son pointeur gauche référence p, et son pointeur droit référence p'.
   - Dans les feuilles d'un arbre B, l'enregistrement médian est aussi inséré dans p.
4. Si le père déborde à son tour, on réitère à partir de 1.

Amélioration de l'insertion

Pour limiter les procédures d'éclatement en cas de débordement ($n > 2k$)

Idée

Redistribuer les clés avec un frère voisin, s'il a lui-même moins de $2k$ clés.

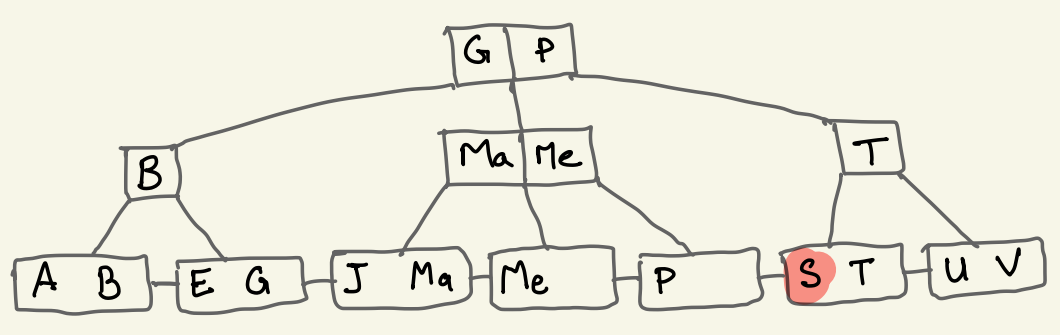


Opération de rotation, sans nouvelle allocation de bloc

La suppression : cas simple n°1

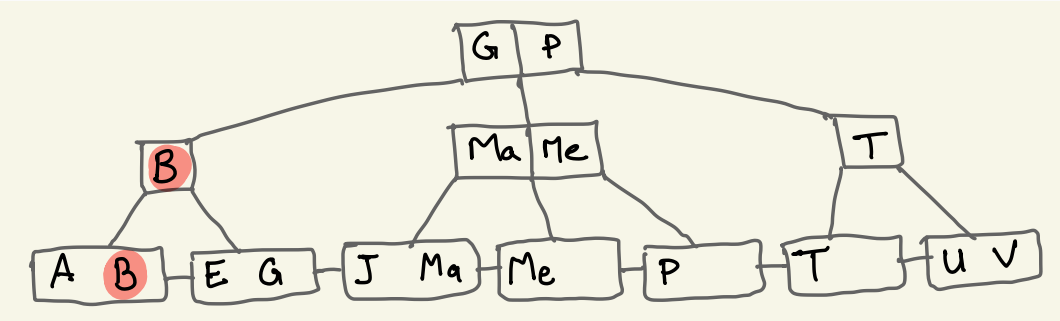
Arbre B des Films, d'ordre 1.

Suppression de l'entrée *Shining* : réorganisation du bloc feuille.



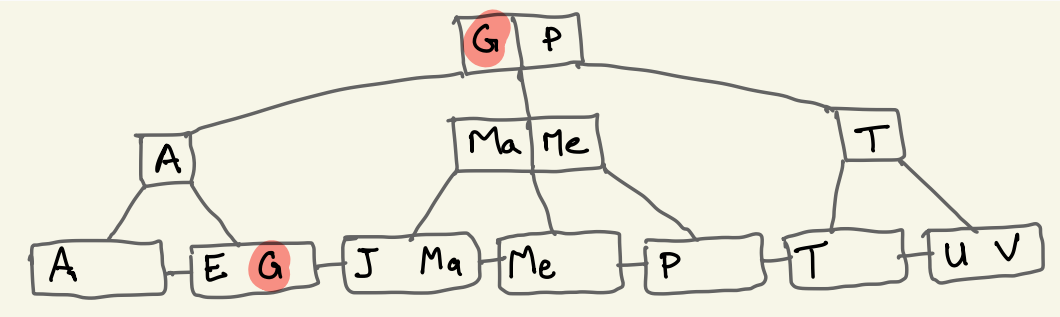
La suppression : cas simple n°2

Suppression de l'entrée *Brazil*, plus grande valeur du bloc : maj de la clé dans le parent.



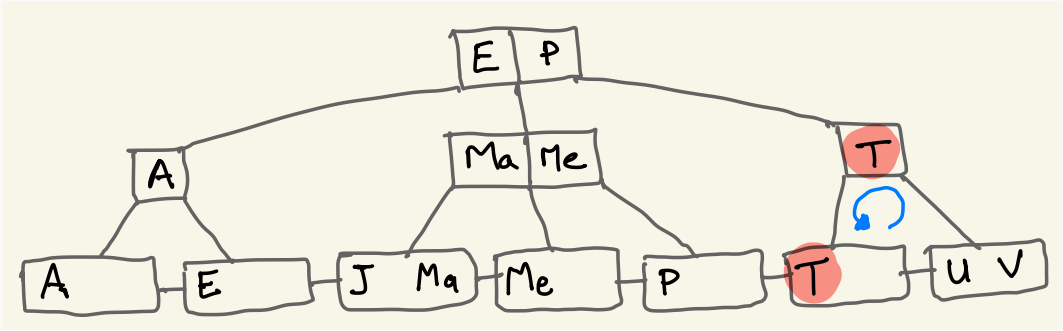
La suppression : cas simple n°3

Suppression de l'entrée *Greystoke*, plus grande valeur du « bloc aîné » : maj de la clé dans un ancêtre.



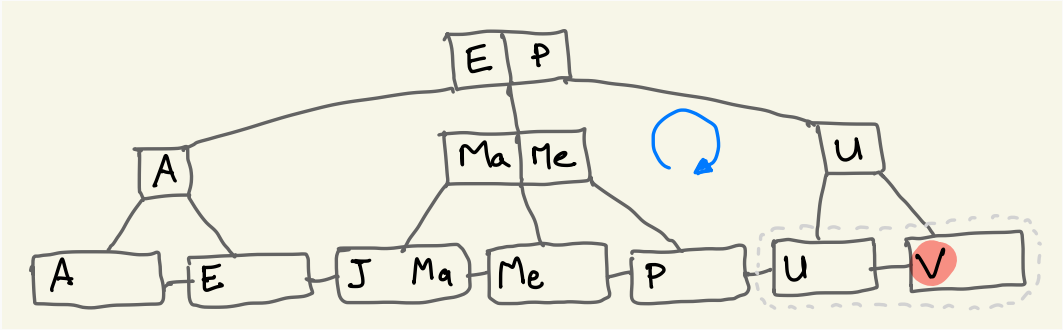
La suppression : cas n°4

Suppression de l'entrée *Twin Peaks*, laissant un bloc à $n < k$ entrées : récupération d'une entrée dans un bloc frère voisin, par rotation inverse.



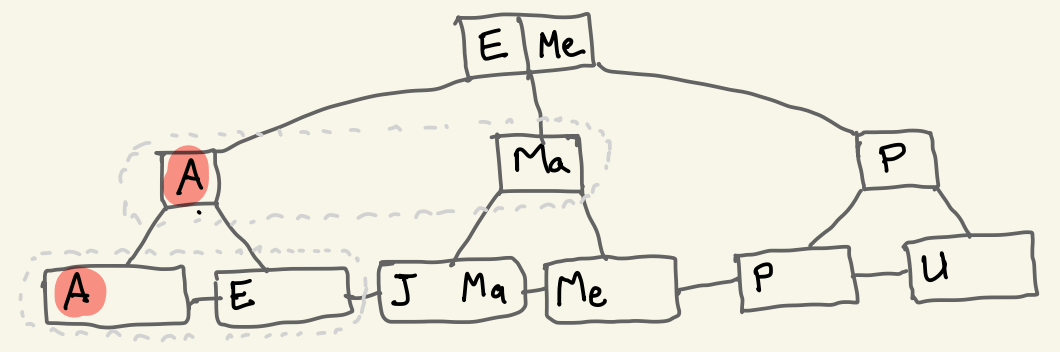
La suppression : cas n°5

Suppression de l'entrée *Vertigo*, laissant un bloc à $n < k$ entrées : fusion avec un bloc frère voisin, et rotation inverse pour rééquilibrer le nœud parent.



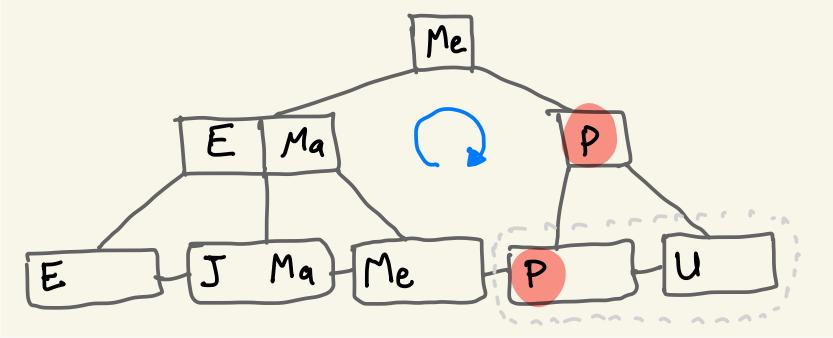
La suppression : cas n°6

Suppression de l'entrée *Annie Hall*, laissant un bloc à $n < k$ entrées : fusion avec un bloc frère voisin, et fusion du bloc parent par descente de la clé médiane.



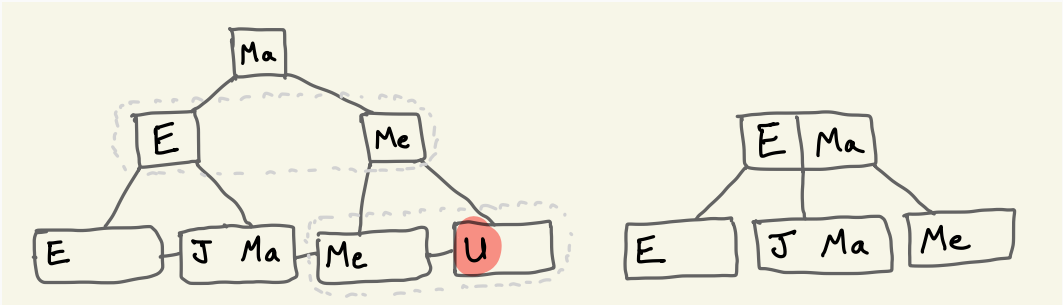
La suppression : cas n°7, semblable au cas n°5

Suppression de l'entrée *Psychose*.



La suppression : cas n°8

Suppression de l'entrée *Underground*. Fusion avec un frère voisin et fusion du parent.



Réduction de la hauteur, par suppression de la racine.

Suppression dans un arbre B d'ordre k

Principe

1. La suppression d'une clé c se fait toujours au niveau des feuilles ;
2. Si la suppression de la clé c d'une feuille (récursivement d'un nœud interne) entraine un nombre de clés $n < k$:
   - 2.1 On opère une rotation inverse pour redistribuer les clés avec un frère voisin ;
   - 2.2 Mais si les frères voisins ont eux-mêmes tout juste k clés, on fusionne avec l'un d'eux, en descendant la clé médiane située dans le nœud parent ;
   - 2.3 Si le parent a moins de k clés, on réitère sauf s'il s'agit de la racine ;
   - 2.4 Dans le cas où la racine ne contient plus aucune clé, le nœud de fusion devient la nouvelle racine (diminution de la hauteur de l'arbre).

Dans les arbres B originaux, et non la variante B+, il faut envisager la suppression d'une clé c dans un nœud interne.

Sur quelques propriétés structurelles des arbres B

- Structure hiérarchique dont l'unité logique est le bloc (= un nœud)
- À l'insertion et la suppression, réorganisation locale seulement
  - Avec propagation éventuelle au nœud père, jusqu'à la racine
- Taux de remplissage garanti à 50%
- Chaînage des blocs feuilles pour un parcours séquentiel
- Équilibrage parfait, induit par les procédures de mise à jour : progression de la hauteur par la racine uniquement

Retour sur la capacité d'un arbre B

Arbre B d'ordre k, et de hauteur h, indexant N enregistrements.

Encadrement du nombre d'entrées N : $2k \times (k+1)^{h-1} \leq N \leq 2k \times (2k+1)^h$

Le nombre total de clés dans les nœuds internes est exactement $N - 1$!

Encadrement et approximation de la hauteur h :

$$\log_{2k+1}\left(\frac{N}{2k}\right) \leq h \leq \log_{k+1}\left(\frac{N}{2k}\right) + 1, \quad \text{soit } h \approx \lfloor \log_{2k}(N) \rfloor$$

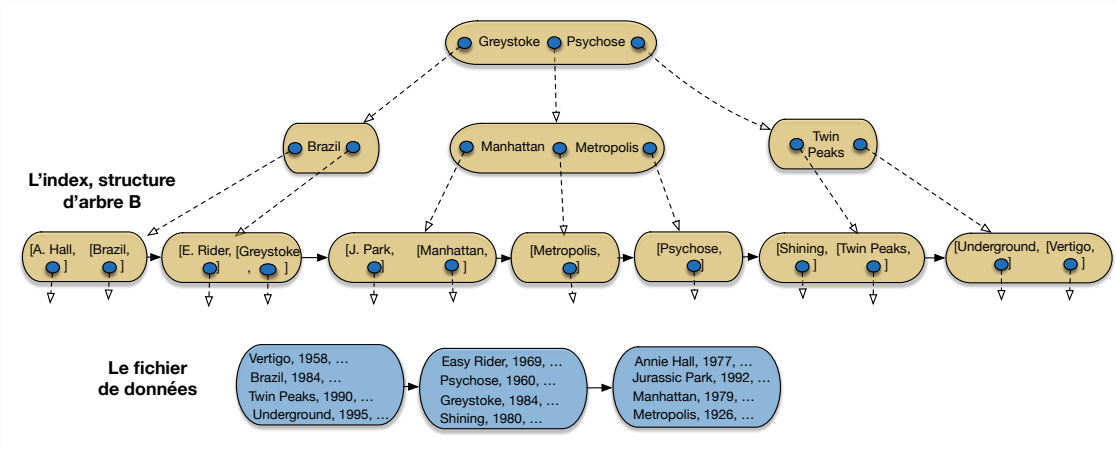
Exemple « réaliste » pour $k = 100$:

- si $h = 2$, $N \leq 8 \times 10^6$
- si $h = 3$, $N \leq 1,6 \times 10^9$

Les opérations d'accès coûtent au maximum h E/S (en moyenne $\geq h - 1$).

La recherche dans un arbre B

Rappel de l'arbre B sur les titres de films

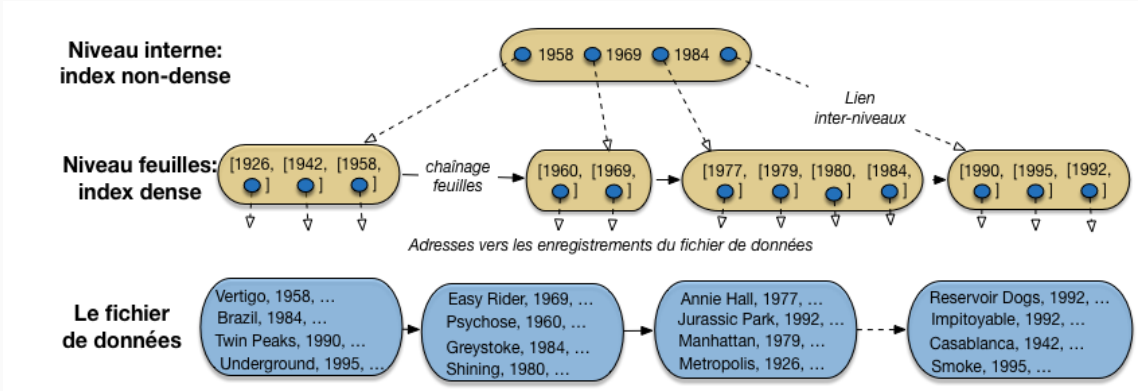


Recherche par clé

```
SELECT *
FROM  Film
WHERE titre = 'Manhattan'
```

- on lit la racine de l'arbre : *Manhattan* étant situé dans l'ordre lexicographique entre *Greystoke* et *Psychose*, on doit suivre le chaînage situé entre ces deux titres ;
- on lit le bloc intermédiaire, on descend à gauche ;
- dans la feuille, on trouve l'entrée correspondant à Manhattan ;
- il reste à lire l'enregistrement.

L'arbre sur les années



Recherche par intervalle

```
select *
from  Film
where annee between 1960 AND 1975
```

- On fait une recherche par clé pour l'année 60
- on parcourt les feuilles de l'arbre en suivant le chaînage, jusqu'à l'année 1975
- à chaque fois on lit l'enregistrement

Attention, les accès directs au fichier de données peuvent coûter très cher.

Recherche par préfixe

Exemple :

```
SELECT *
FROM  Film
WHERE titre LIKE 'M%'
```

Revient à une recherche par intervalle.

```
SELECT *
FROM  Film
WHERE titre BETWEEN  'MAAAAAA...' AND 'MZZZZZZ...'
```

Contre-exemple :

```
SELECT *
FROM  Film
WHERE titre LIKE '%e'
```

Ici index inutilisable

Création d'un arbre B

Automatique sur la clé primaire et tout attribut portant la contrainte `unique`.

On parle d'index unique.

```
create table Film (titre varchar(30) not null,
                   ...,
                   revenu int unique,    -- /!\ valeurs à null
                   primary key (titre)
                   );
```

Sur n'importe quel attribut ou liste d'attributs :

```
create index filmAnnee on Film (annee, réalisateur)
```

Le SGBD synchronise le contenu de la table et celui de l'index.

Résumé : efficacité de l'arbre B

Il est (presque) parfait !

- On a très rarement besoin de plus de trois niveaux
- Le coût d'une recherche par clé est le nombre de niveaux, plus 1
- Supporte les recherches par clé, par intervalle, par préfixe
- Dynamique

On peut juste lui reprocher d'occuper de la place.

Le hachage (S4)

Le hachage

Très utilisé comme structure en mémoire RAM.

Pour les bases de données, un concurrent de l'arbre B plaçant (*cluster*)

- Meilleur (un peu, et en théorie) pour les recherches par clé
- N'occupe aucune place

Mais

- Se réorganise difficilement
- Ne supporte pas les recherches par intervalle

Principe du hachage

Le stockage est organisé en N fragments (*buckets*) constitués de séquences de blocs.

La répartition des enregistrements se fait par un calcul. Une *fonction de hachage* h prend une valeur de clé en entrée et retourne une adresse de fragment en sortie

- Stockage : on calcule l'adresse du fragment d'après la clé, on y stocke l'enregistrement
- Recherche (par clé) : on calcule l'adresse du fragment d'après la clé, on y cherche l'enregistrement

Simple et efficace !

Exemple

On veut créer une structure de hachage pour nos 16 films

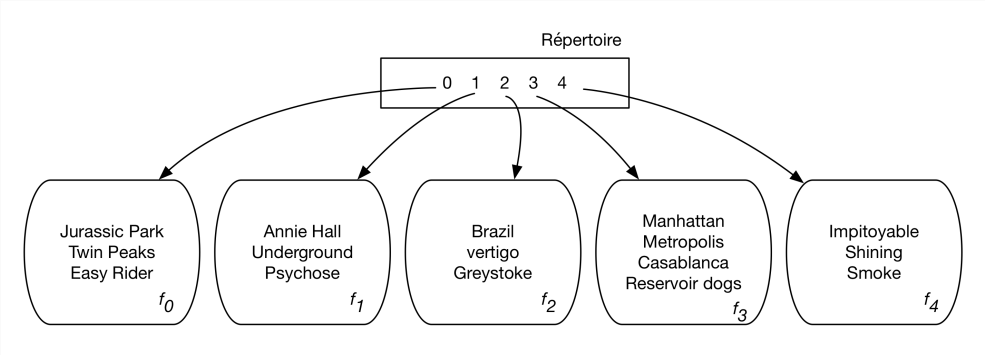
Simplifions : chaque fragment fait un bloc et contient 4 enregistrements au plus

- on alloue 5 fragments (pour garder une marge de manœuvre)
- un répertoire à 5 entrées (0 à 4) pointe vers les fragments
- On définit la fonction $h(\text{titre}) = rang(\text{titre}[0]) \mod 5$

Donc on prend la première lettre du titre (par exemple 'I' pour *Impitoyable*), on calcule son rang dans l'alphabet (ici 9) et on garde le reste de la division par 5, le nombre total de fragments.

$$h(\text{« } \textit{Impitoyable} \text{ »}) = 4$$

Le résultat



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

Recherches

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

Les difficultés

Très important : h doit répartir uniformément les enregistrements dans les n fragments

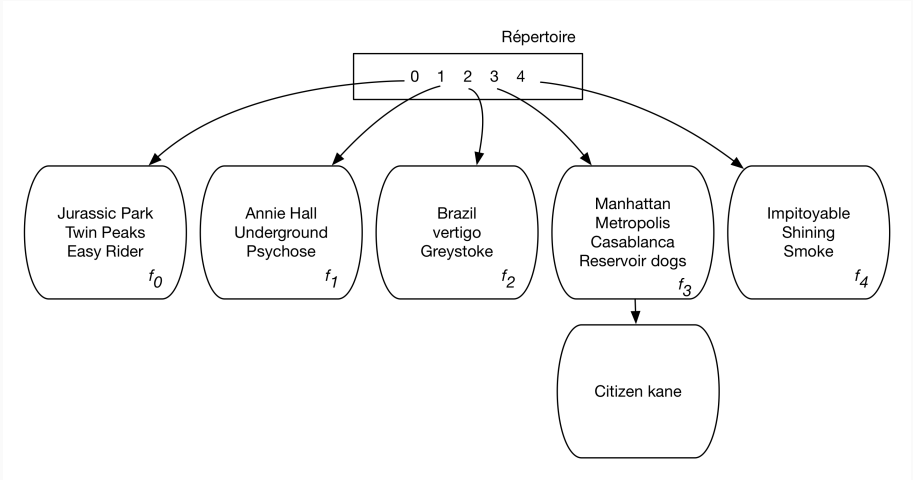
Notre fonction est un contre-exemple : si une majorité de films commence par une même lettre (‘L’ par exemple) la répartition va être déséquilibrée.

Plus grave : La structure simple décrite précédemment n’est pas dynamique.

- On ne peut pas changer un enregistrement de place
- Donc il faut créer un chaînage de blocs quand un fragment déborde
- Et donc les performances se dégradent...

Notez que l’on ne peut pas changer le nombre de fragments car cela implique le changement de la fonction de répartition.

Exemple : insertion de *Citizen Kane*



Il faut maintenant deux lectures pour accéder à *Citizen Kane*.

Résumé : avantages et inconvénients du hachage

En terme d'efficacité, il paraît optimal

- La structure (le répertoire) prend très peu de place
- Le coût d'une recherche par clé est constant
- De plus il est particulièrement simple à implanter

Mais

- Dans sa version de base, il se dégrade quand la situation évolue
- Pas de recherche pas intervalle ou préfixe
- Structure plaçante : il ne peut y avoir qu'une seule structure de hachage par table

Hachage extensible

Le hachage extensible permet de réorganiser la table de hachage en fonction des insertions et suppressions.

Principe

La fonction de hachage h est fixe, mais on utilise les n premiers bits du résultat $h(c)$ pour tenir compte de la taille de la collection.

Par rapport à la version de base du hachage, on ajoute deux contraintes :

1. le nombre d'entrées dans le répertoire est une puissance de 2
2. la fonction h donne toujours un entier sur 4 octets (32 bits)

La taille du répertoire tend à croître rapidement et peut poser problème.

Exemple : le hachage des 16 films

On ne montre que le premier octet de chaque code de hachage.

titre	h(titre)
Vertigo	01110010
Brazil	10100101
Twin Peaks	11001011
Underground	01001001
Easy Rider	00100110
Psychose	01110011
Greystoke	10111001
Shining	11010011

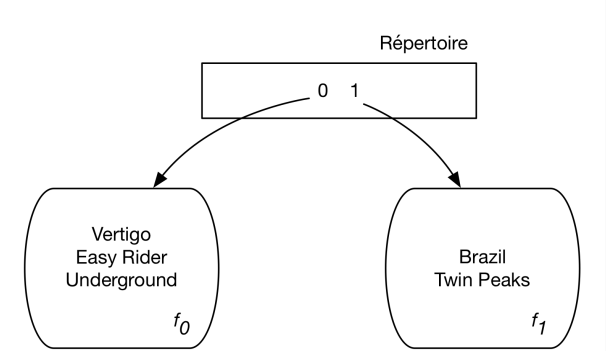
La fonction h est immuable.

Construction de la table

Au départ on utilise seulement le premier bit de la fonction

- Deux valeurs possibles : 0 et 1
- Donc deux entrées, et deux fragments
- L'affectation d'un enregistrement dépend du premier bit de sa fonction de hachage

Avec les 5 premiers films :



Insertions

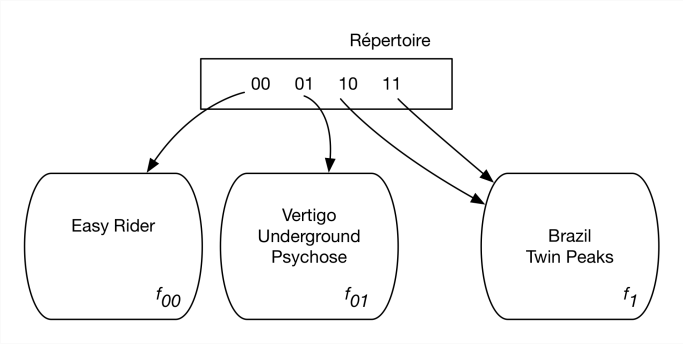
Supposons 3 films par fragment ; l'insertion de *Psychose* (valeur 01110011) entraîne le débordement du premier fragment.

- On double la taille du répertoire : quatre entrées 00, 01, 10 et 11.
- On alloue un nouveau fragment pour l'entrée 01
- Les entrées 10 et 11 pointent sur le même fragment.

Le répertoire grandit, mais dans l'espace de stockage on ajoute seulement un nouveau fragment.

Illustration

Les enregistrements anciennement stockés dans le fragment 0 sont répartis dans les fragments 00 et 01 en fonction de leur second bit.



Réorganisation locale, donc coût acceptable.

Insertions suivantes

Deux cas.

Cas 1 : on insère dans un fragment plein, mais plusieurs entrées pointent dessus

$\Rightarrow$ on alloue un nouveau fragment, et on répartit les pointeurs

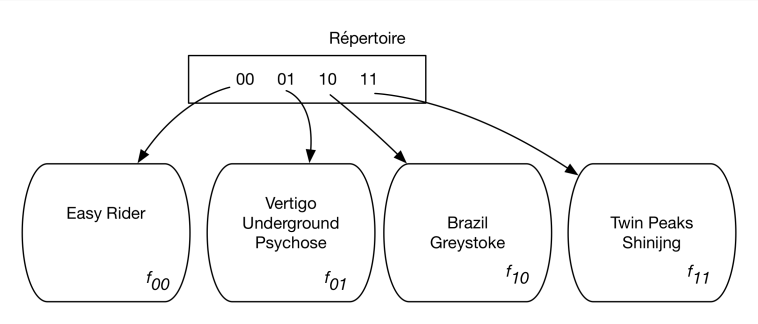
Cas 2 : on insère dans un fragment plein, associé à une seule entrée

$\Rightarrow$ on double à nouveau le nombre d'entrées

Simple, mais la croissance du répertoire est un potentiel problème.

Greystoke 10111001 et Shining 11010011

Le fragment 1 déborde. Il était référencé par deux entrées du répertoire.



On ajoute un fragment, on redistribue, le répertoire ne change pas. C'est le cas qui devient le plus courant quand la collection grandit.

À retenir : hachage extensible

Le hachage extensible résout en partie le principal défaut du hachage, l'absence de dynamicité.

Le répertoire tend à croître de manière exponentielle, ce qui peut soulever un problème à terme.

Il reste une structure plaçante qui doit être complétée par l'arbre B pour des index secondaires.

Le hachage linéaire

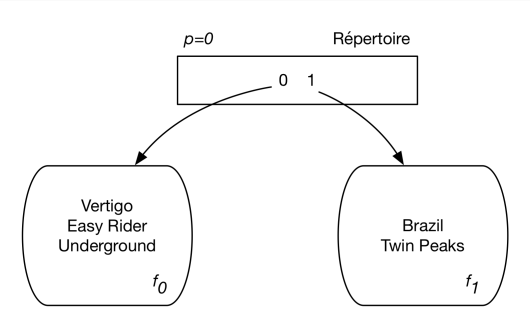
Hachage linéaire : un hachage qui s'adapte dynamiquement aux insertions et suppressions, avec une évolution minimale du répertoire et des fragments.

titre	h(titre)
Vertigo	14
Brazil	43
Twin Peaks	25
Underground	24
Easy Rider	8
Psychose	33
Greystoke	17
Shining	16
Citizen Kane	48

Une fonction $h(c)$ immuable s'applique à la clé et renvoie un entier.

La structure

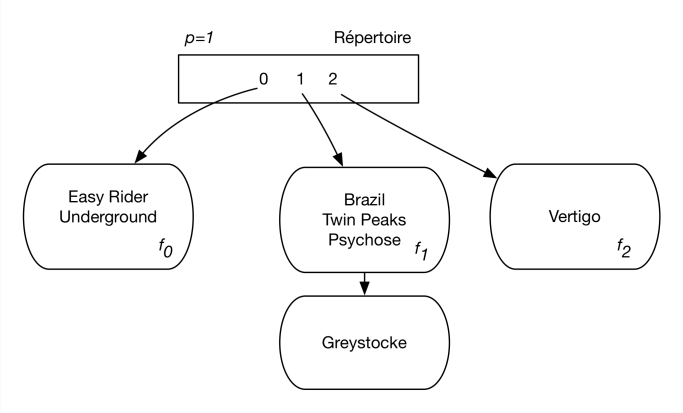
Comme les autres hachages, plus un paramètre : l'indice de partitionnement.



On considère la famille de fonctions $h_i(c) = h(c) \mod 2^i$.
Ici on appplique h_1 (modulo 2).

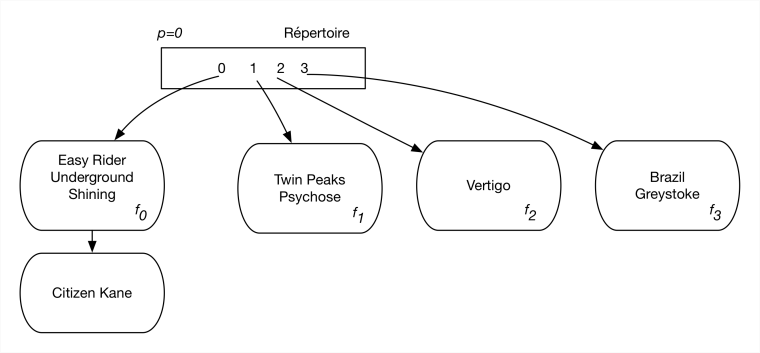
Insertions

On insère Psychose, puis Greystoke qui vont tous deux dans f_1. C'est f_1 qui déborde, mais c'est f_0 qui éclate en (f_0, f_2).



On insère Shining puis Citizen Kane

Tous les deux vont dans f_0 qui déborde. Mais maintenant c'est f_1 qui éclate.



On a découplé les éclatements et les débordements. On sait que les seconds seront rectifiés par les premiers, à terme.

Résumé du hachage linéaire

À tout moment, on a deux fonctions (h_n, h_{n+1}) et un indice de partitionnement p.

- La fonction h_{n+1} s'applique à tous les fragments avant p car ils ont déjà éclaté.
- La fonction h_n s'applique à tous les autres fragments qui n'ont pas encore éclaté.

Quand le dernier fragment éclate $(p = n)$ la fonction h_n ne sert plus et la paire de fonctions de hachage devient (h_{n+1}, h_{n+2})

Brillant !

Index bitmap

Le problème

Comment indexer une table sur un attribut qui ne prend qu'un petit nombre de valeurs ?

- Avec un arbre B : pas très bon car chaque valeur est peu sélective
- Avec un hachage : pas très bon non plus car il y a beaucoup de « collisions ».

Or situation fréquente, notamment dans les entrepôts de données

Exemple : codification des films

rang	titre	genre	...
1	Vertigo	Suspense	...
2	Brazil	Science-Fiction	...
3	Twin Peaks	Fantastique	...
4	Underground	Drame	...
5	Easy Rider	Drame	...
6	Psychose	Drame	...
7	Greystoke	Aventure	...
8	Shining	Fantastique	...
...	...	...	...

Principes de l'index bitmap

Soit un attribut A, prenant n valeurs possibles $[v_1, \dots v_n]$

- On créé n tableaux de bits, un pour chaque valeur v_i
- Ce tableau contient un bit pour chaque enregistrement e
- Le bit d'un enregistrement e est à 1 si $e.A = v_i$, à 0 sinon

Exemple

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Drame	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	0
Science-Fiction	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0
Comédie	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1
Suspense	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Fantastique	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
Aventure	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0

Soit une requête comme :

```
SELECT * FROM Film WHERE genre='Drame'
```

- On prend le tableau pour la valeur `Drame`
- On garde toutes les cellules à 1
- On accède aux enregistrements par l'adresse

Très efficace si n, le nombre de valeurs, est petit.

Autre exemple

```
SELECT COUNT(*) FROM  Film
WHERE genre IN ('Drame', 'Comédie')
```

- On compte le nombre de 1 dans le tableau `Drame`
- On compte le nombre de 1 dans le tableau `Comédie`
- On fait la somme et c'est fini !

Typique des **requêtes décisionnelles**