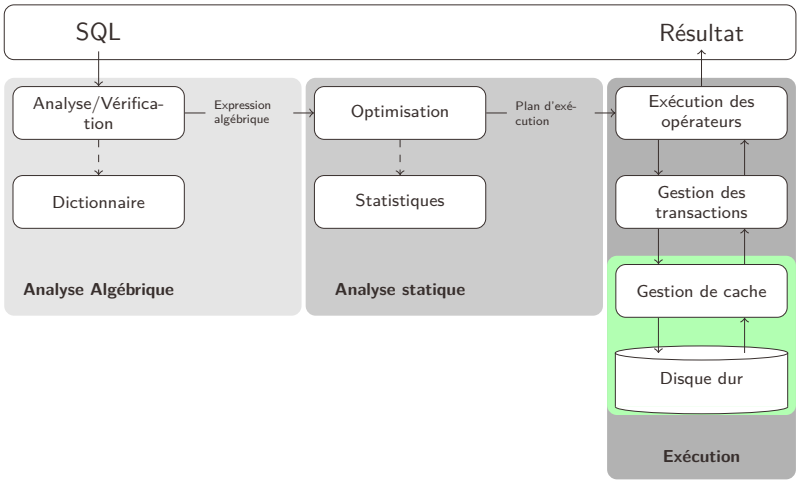
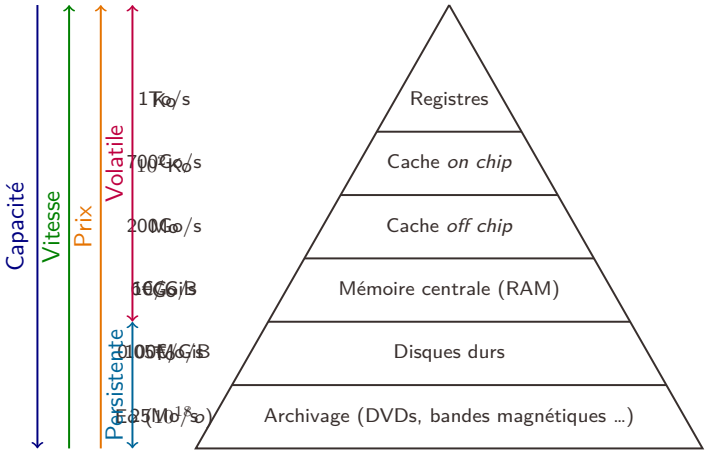
Gestion de la mémoire et principes d'indexation dans les bases de données

Anne-Cécile Caron, Anne Étien, Mikaël Monet, Sylvain Salvati

Traitement d'une requête SQL



Les types de mémoire



Problématique des données massives

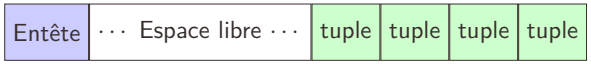
- le traitement des données nécessite de les transférer au sein de la hiérarchie mémoire des couches lentes aux couches rapides,
- le temps de transfert des données depuis la mémoire secondaire (disque dur) est le plus important, c'est lui qui limite la chaîne de traitement,
- on néglige le temps de calcul par rapport à ce temps de transfert.

Un modèle de coût

L'évaluation de la qualité d'un algorithme employé dans une base de données se fait en fonction du nombre d'accès à la mémoire secondaire.

Le bloc l'unité de base de mémoire

Chaque accès, écriture dans la base de données se fait à partir du bloc (page disque). Il est structuré pour pouvoir rapidement récupérer les données à l'intérieur :



Représentation des tuples

Les données sont typées, et chaque type est représenté par un nombre d'octets fixé ou variable :

Types numériques

Type	Taille	Interval de valeurs
TINYINT	1 octet	0 à 255 ou -128 à 127
SMALLINT	2 octets	-32768 à 32767 ou 0 à 65535
MEDIUMINT	3 octets	-8388608 à 8388607 ou 0 à 16777215
INT, INTEGER	4 octets	-2147483648 à 2147483647 ou 0 à 4294967295
BIGINT	8 octets	$\approx 10^{19}$
FLOAT(p)	4 octets si $0 \leq p \leq 24$ 8 octets si $25 \leq p \leq 53$	p nombre de décimales
FLOAT	8 octets	53 décimales
DECIMAL(n,p)	variable	
BIT(n)	$\lceil \frac{n+7}{8} \rceil$	vecteur de bits

Types, Dates et Heures

Type	Taille
DATE	3 octets
TIME	3 octets
DATETIME	8 octets
TIMESTAMP	4 octets
YEAR	1 octet

Types textuels

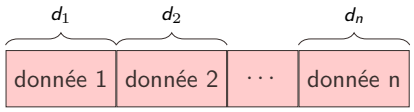
Type	Taille
CHAR(n)	n octets, $0 \leq n \leq 255$
BINARY(n)	n octets, $0 \leq n \leq 255$
VARCHAR(n)	n+1 octets si $n \leq 255$, n+2 octets si $256 \leq n \leq 65535$
BLOB, TEXT	L + 2 octets avec $L \leq 65535$
ENUM('v1', ..., 'vn')	1 octet si $n \leq 255$, 2 octets si $256 \leq n \leq 65535$
SET('v1', ..., 'vn')	1, 2, 3, 4 ou 8 octets (64 valeurs au plus)

Tuples avec données de tailles fixes

Si un tuple est constitué de données de tailles fixes de types suivants :

$$(\mathtt{t}_1, \ldots, \mathtt{t}_n)$$

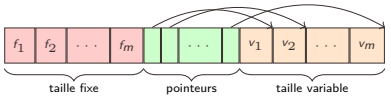
de tailles respectives d_1, ..., d_n, il suffit d'accoler les champs les uns derrière les autres :



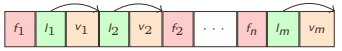
Tuple avec des champs de tailles variables

Si un tuple contient des données de tailles variables, il y a plusieurs représentations possibles :

- placer toutes les données de tailles fixes d'abord, puis les données de tailles variables en les faisant précéder de pointeurs pour trouver la donnée suivante,



- placer sa longueur devant chaque champs de taille variable :



Retrouver un tuple : adresse physique

- Les systèmes de bases de données commerciaux gèrent eux-mêmes l'espace du disque dur.
- La plupart des SGBD libres utilisent le système de fichiers pour stocker les données.
- Dans les deux cas, un SGBD associe à chaque tuple une adresse physique.

Nous faisons ici l'hypothèse que les tuples ont des tailles qui sont en octets, on pourrait compter en nombre de bits pour utiliser mieux l'espace. Cela présente néanmoins des problèmes par rapport à l'efficacité des calculs, les processeurs fonctionnant mieux avec certaines longueurs de mots.

Ajouter un tuple

Pour ajouter un tuple dans une table, il faut trouver un bloc où il y a suffisamment de place :

- les SGBD maintiennent sur disque l'information concernant l'espace disponible dans chaque bloc pour trouver un endroit convenable,
- cependant, si on cherche à maintenir un ordre dans la table suivant une clé, il peut s'avérer qu'il n'y pas assez de place dans la table.

Modifier un tuple

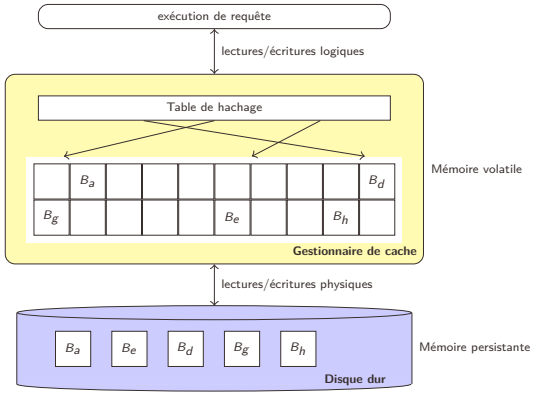
On peut également chercher à modifier la valeur associée à un tuple. Si celui-ci contient des données de taille variable, cela peut poser plusieurs problèmes :

- il peut ne pas y avoir assez de place dans son bloc pour le placer dans celui-ci,
- si sa taille n'est pas la même qu'avant, il faut éventuellement changer l'organisation des tuples dans le bloc.

Dans tous les cas, il faut également mettre à jour tous les endroits où les adresses physiques sont employées (typiquement dans tous les index – ceux-ci recensent et organisent les adresses physiques de tuples). Le même genre de problème se pose pour les effacements de tuples.

Le gestionnaire de cache

- Les SGBD, utilisent une mémoire tampon afin de conserver une image des blocs sur le disque et accélérer les accès aux tuples,
- cette mémoire tampon peut être paramétrée, sa taille peut être un facteur essentiel de l'efficacité d'un SGBD.



Déroulement d'une lecture logique

Demande d'une donnée → deux possibilités :

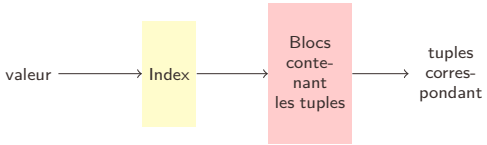
- le bloc qui la contient est chargé dans le cache, auquel cas, la donnée est retournée,
- sinon, il faut lire le bloc sur le disque, le charger dans le cache et on se retrouve dans le cas précédent. Il y a cependant deux situations possibles :
  - le cache contient encore de l'espace libre, et il n'y a pas de problème,
  - le cache est saturé et il faut libérer un espace à l'intérieur,
    - un SGBD doit avoir en mémoire la plus grande partie des données possible,
    - il doit maintenir en mémoire les blocs les plus utilisés,
    - une bonne gestion du cache doit maximiser le *hit ratio* :

$$\text{hit ratio} = \frac{\text{nb lectures logiques} - \text{nb lectures physiques}}{\text{nb lectures logiques}}$$

    - Le *hit ratio* dépend de la distribution de l'utilisation des tables : certaines tables sont plus lues que d'autres.

Accélérer l'accès aux données

- Le parcours complet d'une table peut prendre un temps très important,
- pour accélérer l'accès aux données, il convient de maintenir des structures de données qui limitent le nombre de lectures sur le disque,
- en particulier il s'agit de trouver rapidement les tuples qui ont un champs ayant une certaine valeur,
- ces structures de données portent le nom d'index,
- de plus, si les couples (valeur,pointeur) sont petits relativement aux tuples indexés, il y a des chances que l'index tienne en mémoire, limitant encore les accès disque.

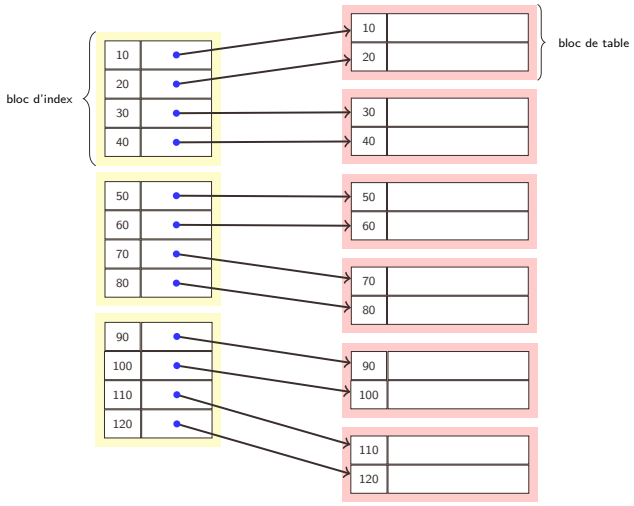


Typologie des index

- Index primaire : un index primaire requiert que la relation soit ordonnée, ils sont typiquement utilisés sur les clés primaires (mais pas nécessairement) ;
- Index secondaire : l'organisation des données sur disque est indépendant de l'index.
- Index dense : il y a une entrée par valeur possible,
- Index non-dense : il y a une entrée par bloc.

	Primaire	**Secondaire**
Dense	Fichier ordonné, chaque tuple est indexé	Organisation des données quelconque (ordonnées ou non), chaque tuple est indexé
Non-Dense	Fichier ordonné, chaque bloc est indexé	Ce type d'index ne fait pas de sens. Il n'existe pas.

Exemple d'index primaire dense



Index pour une relation triée

Un index primaire suppose que la relation est stockée ordonnée, en général selon sa clé primaire. Plus précisément :

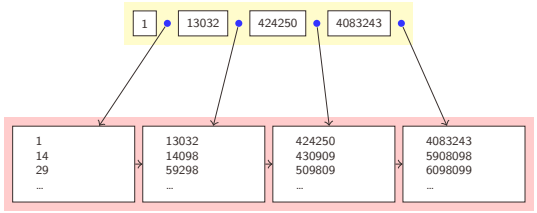
- il y a au plus un tuple par clé,
- le fichier a la structure d'un dictionnaire.

Dans ce cas, on peut faire appel à une recherche dichotomique :

- cela suppose que la table soit constituée de blocs consécutifs,
- il est rarement possible de maintenir cette propriété en plus de l'ordre pour des grandes tables,
- en conséquence les blocs sont chaînés les uns aux autres et la dichotomie devient impossible.

Dans cette situation, on peut indexer chaque bloc par la première clé qui s'y trouve, on peut alors accéder directement au bloc. C'est la particularité d'un index non dense.

Exemple d'index primaire non-dense

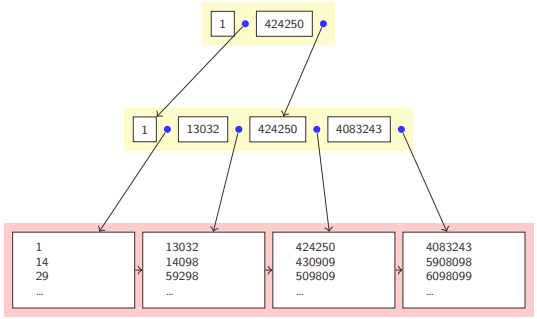


Hypothèse : Le fichier de données est trié selon la clé. L'index ne référence que la première valeur de chaque bloc.

Recherche dans un index non-dense

- Si l'index est ordonné dans le fichier (ce qui est vrai dans la plupart des cas), on peut utiliser une recherche dichotomique
- Sinon,
  - on peut effectuer un `Full Scan`,
  - ou encore créer un nouvel index non dense, pour indexer le premier index.

Exemple index non-dense sur plusieurs niveaux



Taille des index non-dense

- Un index non-dense contient au plus une entrée par bloc qui décrit la relation.
- Les entrées sont petites : un champs de relation (quelques octets), et une adresse (8 octets).

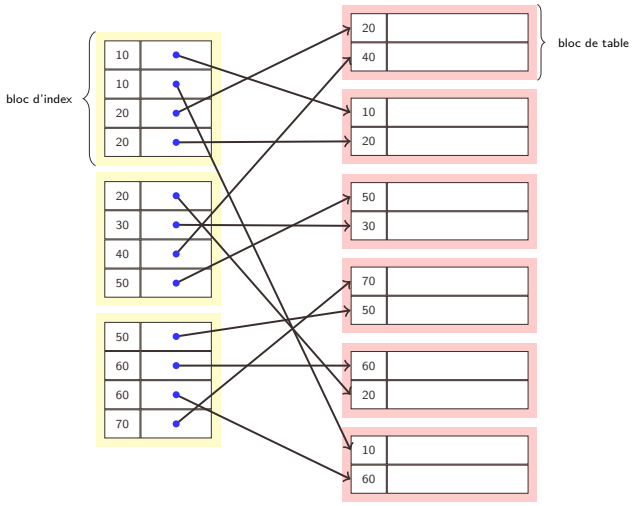
Si on prend une relation avec les caractéristiques suivantes :

- elle contient 10^6 tuples dans 300000 blocs,
- la relation occupe $1.3 Go$,
- le champs indexé compte en moyenne 20 octets,
- l'index aura pour taille : $300000 * (20 + 8) = 8.4 Mo$.

Index secondaires

- Les index secondaires servent essentiellement à retrouver des tuples en connaissant la valeur d'un champs qui n'est pas une clef primaire.
- À la différence des index primaires pour lesquels la relation valeur $\rightarrow$ tuple est fonctionnelle, le plus souvent il y a plus d'un tuple associé à une valeur.

Index secondaire : exemple



B-arbres

- Les B-arbres sont des structures d'index qui généralisent plusieurs idées :
  - celle des index à plusieurs niveaux (cf index non-denses),
  - celle des arbres AVLs afin de garantir une recherche qui requiert un temps logarithmique dans le nombre de valeurs indexées.
- Si on suppose que l'on a 10^6 valeurs, si on les stocke dans :
  - un arbre binaire, il faut suivre 20 pointeurs ce qui correspond à 20 accès disque,
  - un arbre d'arité 1000, il faut suivre seulement 2 pointeurs, ce qui divise par 10 le nombre d'accès disque.

B-arbre et B+-arbre

Les B-arbres présentent les propriétés suivantes :

- Ils sont équilibrés : ils maintiennent automatiquement le nombre de niveau d'indexation approprié à la taille de l'index,
- leurs nœuds sont stockés sur des blocs du disque,
- les blocs qui les constituent sont toujours au moins à moitié plein (au plus ils utilisent un espace deux fois supérieur au nombre de valeurs indexées),
- pour leur variante, les B+-arbres, les feuilles sont chaînées dans l'ordre. Cela présente l'avantage de pouvoir répondre facilement à des requêtes portant sur des intervalles (ex : films tournés entre 1950 et 1960),

Par la suite, nous présentons les B+-arbres, appelés B-arbres par la plupart des SGBD.

B-arbre : structure

- les nœuds des B-arbres sont des blocs sur disque,
- chaque bloc qui contient k valeurs et $k+1$ pointeurs,
- le nombre maximal de valeurs par nœud est déterminé par le nombre maximal n de valeurs et de $n+1$ pointeurs qui peut tenir dans un bloc.

Parmi les nœuds d'un B-arbre, on distingue :

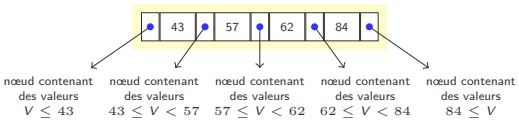
- la racine,
- les nœuds internes,
- les feuilles

B-arbre : la racine

- La racine possède toujours au moins deux pointeurs (s'il n'y a qu'un pointeur, le B-arbre ne pointe que vers un tuple ; on laisse ce cas de côté),
- chaque pointeur de la racine pointe vers un nœud du B-arbre.

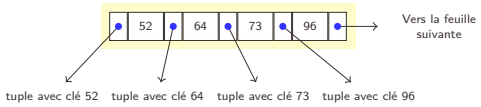
B-arbre : les nœuds intérieurs

- Dans un nœud intérieur tous les pointeurs présents pointent vers un bloc du B-arbre,
- il y a au moins $\lceil (n+1)/2 \rceil$ pointeurs utilisés (et donc au moins $\lceil (n+1)/2 \rceil - 1$ valeurs),
- si le nœud contient les valeurs V_1,..., V_k, alors le pointeur P_j pointe vers des blocs qui contiennent des valeurs comprises entre V_{j-1} et V_j

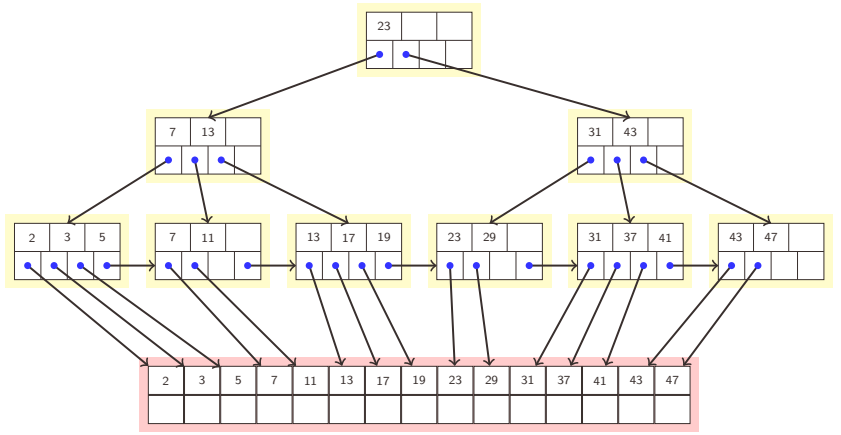


B-arbre : les feuilles

- Au niveau des feuilles les valeurs sont celles du fichier de données, elles sont ordonnées entre les feuilles de gauche à droite,
- le dernier pointeur de chaque feuille pointe sur la feuille suivante,
- parmi les k pointeurs dans une feuille, au moins $\lfloor (k+1)/2 \rfloor$ pointent vers des tuples de données.



Exemple : $n = 3$



Insertion dans un B-arbre

- on essaie de placer la paire (valeur,pointeur) dans la feuille appropriée,
- si il n'y a pas de place, on découpe la feuille en deux :
  - on a n+1 valeurs donc on peut toujours découper en deux paquets de valeurs de taille supérieure à $\lfloor (n+1)/2 \rfloor$,
  - la valeur qui découpe les deux parties est insérée dans le nœud parent et on applique cette stratégie récursivement,
  - si on insère dans la racine et qu'il n'y a plus de place, on crée une nouvelle racine.

Suppression dans un B-arbre

- lorsque l'on supprime une paire (valeur, pointeur) d'une feuille, et que cette feuille reste suffisamment pleine, on en reste là,
- sinon, il y a $\lfloor (n+1)/2 \rfloor - 1$, paires, on essaie les opérations suivantes :
  - si on peut fusionner avec une feuille adjacente on le fait, on supprime une clé d'un des parents et on applique la procédure récursivement,
  - sinon, on peut transvaser des paires depuis l'un des voisins.

B-arbres - en résumé

- structure dynamique, c'est la structure d'indexation par défaut des SGBD ;
- index primaire (denses ou non) ou secondaire ;
- index unique ou non, i.e. avec plusieurs paires (clés, valeurs) ayant la même clé ;
- coût d'une recherche par clé : $1 +$ le nombre de niveaux. Le nombre de niveaux dépasse rarement 3.
- recherche par clé, par préfixe de la clé et par intervalle ("range query").
- utilisé bien au delà de l'indexation des relations d'un SGBD (indexation des fichiers dans un OS, pages jaunes pour un SGBD distribué, etc)

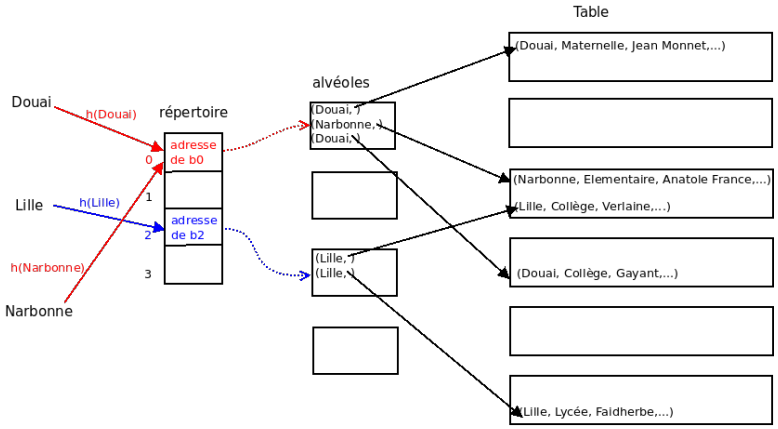
Il existe d'autres types d'index dans les SGBD relationnels comme les *Hash Index* et les *BitMap Index*.

Hachage

On peut définir des indexes de type table de hachage. Au delà des indexes persistants, le hachage est très utilisé en mémoire RAM lors de l'exécution d'une requête.
Les principes du hachage sont les suivants :

- Une fonction de hachage h distribue les valeurs dans M alvéoles (buckets) en fonction de la clé.
- Chaque alvéole contient un nombre maximum de valeurs.
- On suppose qu'on lit une alvéole sur le disque en temps constant.
- Difficulté : la fonction h doit répartir uniformément les enregistrements dans les alvéoles.
- Ici la structure est statique. Le nombre d'alvéoles ne change pas. Éventuellement on peut ajouter des blocs en les chaînant, mais l'accès n'est plus en temps constant.

Hachage : exemple



Hachage dynamique

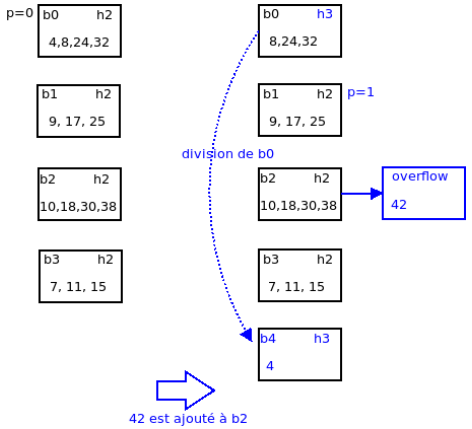
La structure de données s'adapte quand une alvéole est remplie, en modifiant le nombre M d'alvéoles, donc la fonction de hachage.
A titre d'exemple, nous allons étudier le **hachage linéaire**.

- Quand une alvéole b est pleine :
  - on chaîne une nouvelle alvéole à b, sans changer M
  - on divise une alvéole b_p, souvent distincte de b. p est un indice incrémenté à chaque division. M est augmenté de 1.

  Très astucieux ; on a des blocs de débordement mais ils sont rectifiés à terme par la division.
- Initialement, $p = 0$, l'alvéole b_0 est donc la première qui doit se diviser.
- Comme M augmente, il faut changer la fonction de hachage, mais on veut que cette fonction reste valable pour les alvéoles non divisées $\rightarrow$ on utilise 2 fonctions de hachage.
  1. $h_n : k \rightarrow (k \bmod 2^n)$ pour les alvéoles b_i t.q. $i \in [p, M - p - 1]$
  2. $h_{n+1} : k \rightarrow (k \bmod 2^{n+1})$ pour les autres alvéoles

Hachage linéaire : exemple

- Chaque alvéole stocke au plus 4 valeurs.
- Au départ, le nombre d'alvéoles $M = 4 = 2^2$, donc on utilise les fonctions de hachage (h_2, h_3).



Pour simplifier, on ne montre que les clés.

Hachage linéaire : exemple

- Algorithme utilisé pour savoir dans quelle alvéole est la clé k :
  $i := h_n(k)$;
  $if (i < p) i := h_{n+1}(k)$;
  return addr(b_i);
- La prochaine alvéole pleine déclenchera la division de b_1, puis de b_2 puis de b_3.
- Quand les 4 premières alvéoles auront été divisées, on aura $M = 8 = 2^3$. On considère alors les fonctions de hachage (h_3, h_4), on réinitialise p à 0.

Hachage - en résumé

- index primaire (denses ou non) ou secondaire ;
- index unique ou non ;
- coût d'une recherche par clé : constant, sauf si les alvéoles sont étendues avec des listes chaînées ;
- le hachage linéaire s'adapte dynamiquement ;
- le répertoire prend très peu de place, il tient en mémoire ;
- recherche par clé uniquement, pas par intervalle ni préfixe ; c'est le grand inconvénient de ces index.

Index bitmap

Un index bitmap est organisé sous la forme d'un tableau de bits, très pratique pour des opérations booléennes.

- tableau qui contient autant de colonnes que de valeurs possibles de la clé, et autant de lignes que la relation à indexer.
- chaque case (x, y) contient un bit qui indique si la ligne x a la valeur de clé y ; la ligne ne comporte que des 0 si la valeur de la clé vaut `null`.

Rowid	M	F
213	1	0
234	0	1
395	1	0
423	0	0
...	...	...

Index Bitmap

Rowid	Id_Employe	sexe	age	Id_service	...
213	1	'M'	46	null	
234	2	'F'	52	13	
395	3	'M'	28	2	
423	4	null	34	2	
...	...	...			

Table RH

Index bitmap : exemple de combinaison booléenne

```
select * from Personne
where sexe='M' and situation = 'Marié'
```

rowid	'M'
123	0
153	1
264	1
391	0
...	...

AND

rowid	'marié'
123	1
153	1
264	0
391	0
...	...

=

rowid	Req
123	0
153	1
264	0
391	0
...	...

Index bitmap - en résumé

- adapté aux colonnes qui ont une faible cardinalité (type énuméré, booléen pour le cas extrême ...) ;
- prend peu de place (les vecteurs de bits sont généralement compressés), mais peu efficace si la relation est mise à jour fréquemment ; pour cette raison, il est utilisé surtout dans le cadre des entrepôts de données où l'on fait essentiellement des `select`, avec des critères peu sélectifs ;
- calcul rapide de combinaisons de critères ; opérations booléennes sur les tableaux de bits.