Polycopié de Cours

Structures de Fichiers

- Structures de Données Externes -

(SFSD / 2CP)

HIDOUCI Walid-Khaled & KERMI Adel
ESI - 2024

Plan

Introduction

L’objectif du cours est l’étude des méthodes d’accès aux fichiers (structures et algorithmes associés : recherche, insertion, suppression, …). Cela concerne principalement :

- Les performances des solutions algorithmiques manipulant des structures de fichiers (manipulation efficace de données volumineuses et massives).
- L’analyse de la complexité pour ce type particulier d’algorithmes.

A l'issue de ce cours, les étudiants seront capables de :

- Développer des solutions algorithmiques efficaces qui manipulent les structures de fichiers.
- Analyser la complexité temporelle et spatiale des différentes méthodes de structuration de fichiers.
- Répondre aux nouvelles problématiques liées aux données massives.
- Comprendre les éléments de base utilisés dans les systèmes de stockage modernes et dans l'optimisation/traitement des requêtes.

Les notions traitées dans ce cours permettent d’aborder sans difficultés, les futures enseignements dans les domaines des bases et ingénierie de données, ainsi que dans la programmation avancée et les systèmes d’exploitation modernes.

Nous nous intéressons à l’étude des algorithmes de base sur les grands ensembles de données en mémoire externe. C’est un aspect fondamental dans la formation de base d’un ingénieur d’état en informatique de haut niveau.

Les structures de fichiers permettent de représenter des ensembles de données (souvent très volumineux) à l’aide de structures de données internes et externes. Leurs manipulations induit des transferts de données entre mémoire centrale et mémoire externe (secondaire).
L’optimisation du nombre de transferts est le principal objectif permettant l’amélioration des performances pour ce type d’applications.

Le *langage **C*** sera utilisé pour la mise en pratique des notions abordées dans ce cours. La maîtrise de ce langage en particulier, constitue la meilleure voie pour une compréhension saine et précise du fonctionnement et des détails internes des systèmes informatiques en général.

Structures de Fichiers

Chapitre 1 Généralités

Dans ce chapitre nous allons présenter les notions générales relatives aux ***structures de fichiers***. Les principaux points abordés sont

- Les mémoires
- Les fichiers
- Les opérations de base sur les fichiers

1) Les différents types de mémoires

Il existe plusieurs types de mémoires dans un ordinateur. Chaque type de mémoire est caractérisé par la vitesse d'accès, la taille, la nature de l'accès permis (lecture, écriture, effacement, ...), le coût et la volatilité (pertes d'information en cas de crash système ou redémarrage).

Il y a des mémoires très rapides, très coûteuse et en très petites quantités : les *registres* du processeur. Leur utilisation est automatisée à travers le module de génération de code du compilateur utilisé.
Il y a des mémoires très lentes, peu coûteuses et en très grandes quantités : par exemple les volumes de bandes magnétiques. De nos jours on ne les utilise principalement que pour l'archivage des données et programmes à travers des commandes particulières du système d'exploitation.
Entre ces deux extrêmes on trouve (dans l'ordre croissant des vitesses d'accès et décroissant du coût à l'unité) les types de mémoires suivants :
- Les mémoires cache entre le processeur et la mémoire centrale (en lecture/écriture et volatile gérées principalement par le hardware). Leurs tailles sont supérieures aux registres et inférieures à la mémoire centrale.
- La Mémoire Centrale (***MC***) ou mémoire principale (en lecture/écriture et volatile) contenant les programmes et les données en cours de traitement. La taille est supérieure à celle des mémoires caches et inférieure à celle de la mémoire secondaire.
- La Mémoire Secondaire (***MS***), souvent constituée de dispositifs de mémoires externes à accès direct : les disques magnétiques , les flash disques, les disques SSD, … (en lecture/écriture et non volatile). La ***MS*** est utilisée principalement pour le stockage persistent des programmes et/ou données et aussi pour la virtualisation de la ***MC*** (comme par exemple le *paging* dans les systèmes d'exploitation). Sa taille est souvent supérieure à celle de la ***MC*** mais l'accès est beaucoup plus lent aussi. Souvent une petite partie de la ***MC*** est réservée par le système d'exploitation pour être gérée comme mémoire cache de la ***MS*** (par exemple le *buffer pool*).
- Les autres mémoires externes ou auxiliaires utilisées principalement pour l'archivage de données et de programmes sont constituées de dispositifs de mémorisation lents et peu coûteux. Souvent offrant la possibilité de gestion multi-volumes pour rendre leur capacité de stockage théoriquement illimitée. Nous retrouvons dans cette catégorie les disques optiques (en lecture seule ou en lecture/écriture limitée et non volatiles) et les bandes magnétiques à accès séquentielle (en lecture/écriture et non volatile).

En programmation de structures de fichiers, nous nous intéressons principalement à la mémoire centrale (***MC***) et à la mémoire secondaire (***MS***). Toutes les variables d'un programme en cours d'exécution sont allouées en ***MC*** et le contenu des fichiers manipulés se trouve en ***MS***. D'après le paragraphe précédent, nous retiendrons ce qui suit :
- La ***MC*** est volatile (son contenu est perdu en cas de crash système ou redémarrage)
- La ***MS*** n'est pas volatile (elle est supposée résister aux crash et redémarrages)
- L'accès à la ***MC*** est beaucoup plus rapide que celui à la ***MS***. Par exemple avec les technologies actuelles, l'accès à une unité d'information en ***MC*** est presque 10000 à 100000 fois plus rapide que celui d'un ***disque magnétique*** et aussi 1000 fois plus rapide que celui d'un disque ***SSD***.

1.1) Structure des disques magnétiques

Un des types de ***MS*** les plus utilisés de nos jours reste le ***disque magnétique*** (ou encore disque dur : ***DD*** ou ***HDD*** pour *Hard Disk Drive*).

Un ***HDD*** est un ensemble de *plateaux* en forme de disques superposés. Chacun d'eux est formé de deux faces magnétiques (*surfaces*) et tournent en continu avec une certaine vitesse de rotation. Chaque piste est découpée en *secteurs* de même taille. Le *secteur* représente l'unité de transfert entre la ***MS*** et la ***MC***. C'est la plus petite quantité d'information pouvant être lue ou écrite sur le disque.

Disque = ensemble de *plateaux* tournant avec une vitesse de rotation déterminée
Plateau = ensemble de *pistes* concentriques numérotées 0, 1, 2, ... sur chaque *face*
Cylindre = ensemble de *pistes* ayant un même diamètre sur les différents *plateaux*
Piste = ensemble de *secteurs* de même taille

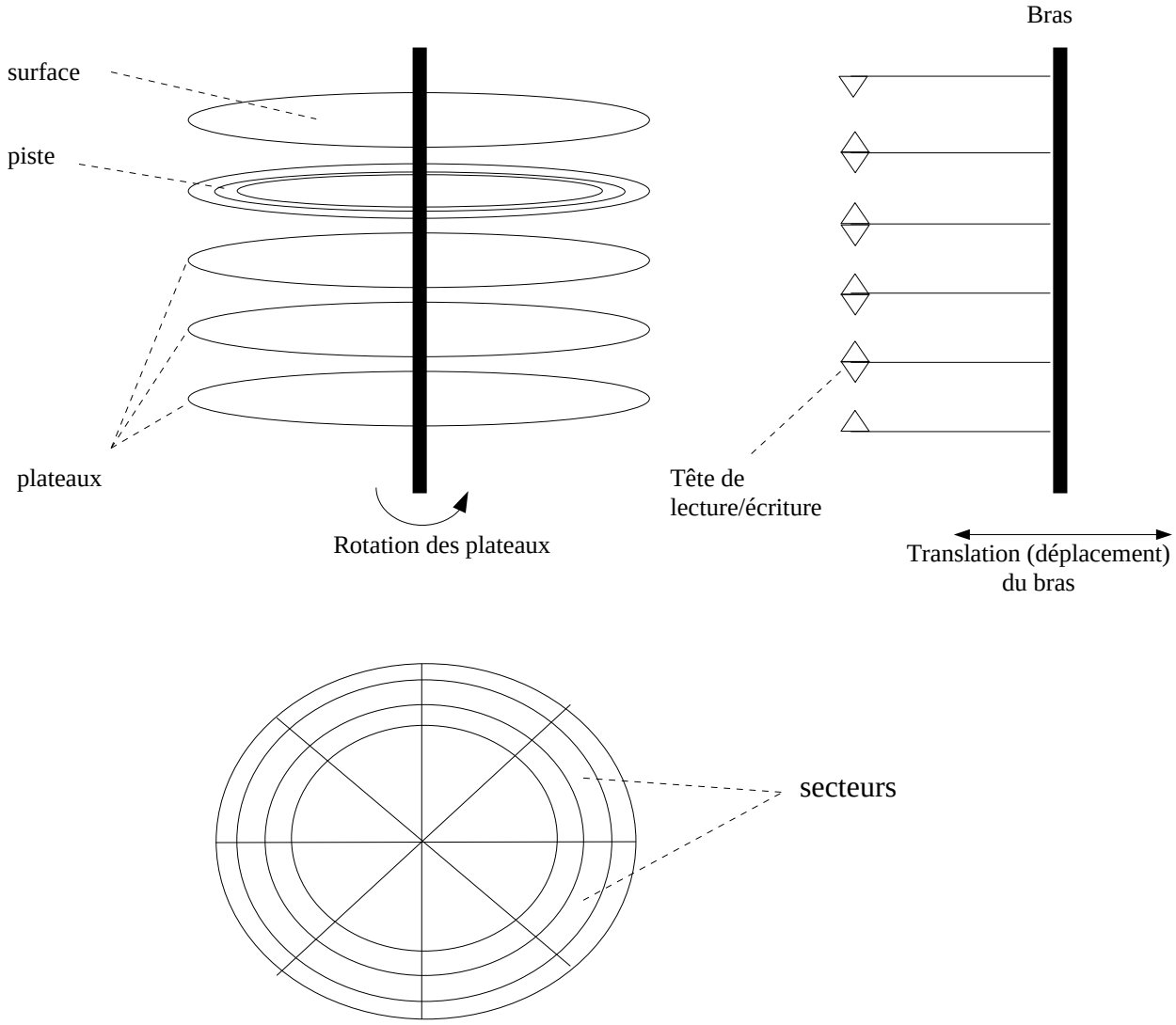
Le *bras* du disque contient les têtes de lecture/écriture (une pour chaque *face* d'un plateau). Il se déplace latéralement de *cylindre* en *cylindre* pour positionner les têtes de lecture/écriture sur les bonnes *pistes*.

Pour lire ou écrire un *secteur* donné, le contrôleur déplace d'abord le bras vers le bon *cylindre*, ensuite attend que le bon *secteur* passe sous la tête de lecture/écriture concernée avant de transférer les données.

Avec la technologie actuelle, le temps nécessaire à cette opération (temps d'accès = temps du déplacement du *bras* + le temps d'attente de passage du *secteur* cherché sous la tête de lecture/écriture, incluant le transfert de données ***MC*** $\leftrightarrow$ ***MS***) est au voisinage de quelques millisecondes en moyenne. Si on le compare au temps d'accès à la mémoire centrale (quelques dizaines ou centaines de nanosecondes), la différence est énorme (le temps d'accès au disque est plus lent que celui de la ***MC*** de plusieurs ordres de grandeurs). Le temps de déplacement du bras (temps de translation) est plus grand que le temps de rotation. Des lectures ne nécessitant pas de déplacement de *bras* (donc sur le même *cylindre*) coûtent aux environs de quelques μs en moyenne.

Parmi les méthodes utilisées (dans les systèmes d'exploitation) pour diminuer l'effet de cette grande différence dans le temps d'accès entre la ***MC*** et la ***MS***, on trouve le **buffering**. Il s'agit de réserver une zone en mémoire centrale (***MC***) pouvant contenir plusieurs blocs en même temps (le buffer-cache). Lors de la lecture de blocs non présents en ***MC***, des stratégies de remplacement (comme *LRU, FIFO, …*) sont utilisées afin de minimiser le nombre d'accès disque. Le **prefetching** (consistant à lire plusieurs blocs consécutifs lors d'une demande d'accès à un seul bloc donné) est très souvent utilisé, en combinaison avec le buffering, pour augmenter les chances de trouver les

blocs demandés, déjà présents en ***MC***, lors de leurs éventuels futurs accès.



D'autres techniques, comme la **multi-programmation** (durant l'opération d'E/S d'un programme, le processeur est alloué à un autre programme) avec le ***Time-Sharing*** (pseudo-parallélisme entre plusieurs processus en partageant le temps CPU) ou encore la **parallélisation des E/S** (les données sont réparties sur plusieurs disques – comme dans la technologie **RAID**) sont aussi utilisées pour diminuer le goulot d'étranglement sur les E/S.

De plus, comme le temps de déplacement du *bras* est plus grand que celui nécessaire pour la rotation du disque, l'accès à des *secteurs* physiquement proches (situés sur la même *piste* ou le même *cylindre* par exemple) est beaucoup plus rapide que l'accès à des secteurs nécessitant un déplacement du *bras*. Le **clustering** est une technique de stockage rassemblant les données susceptibles d'être traitées ensemble dans des *secteurs* physiquement proches.

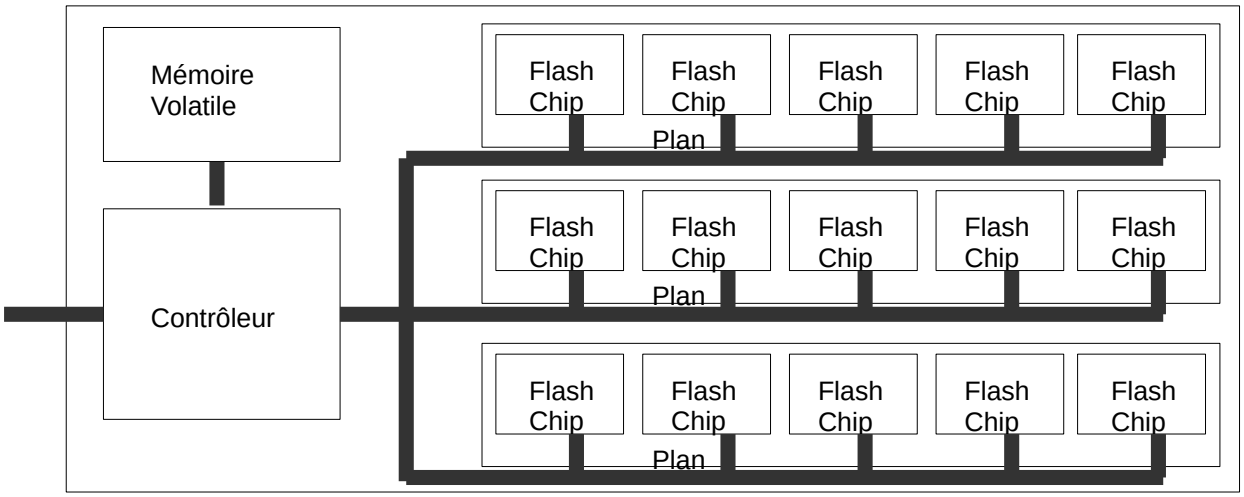
Le disque est contrôlé par un circuit (le contrôleur de disque) qui offre au système d'exploitation des **primitives d'E/S** (du type '**Lire_un_Bloc(...)** / **Ecrire_un_Bloc(...)**') cachant les détails internes concernant les déplacements du *bras* et la rotation des *plateaux*. Pour ce type de périphériques, les E/S sont dites "bufférisées" car les transferts se font bloc par bloc (ex: un bloc = un ou plusieurs *secteurs*).

Les primitives d'E/S permettent de manipuler les *secteurs* du disque (l'unité de transfert) comme étant des tableaux d'octets de taille fixe (ex 512, 1024, 4096 octets ...).

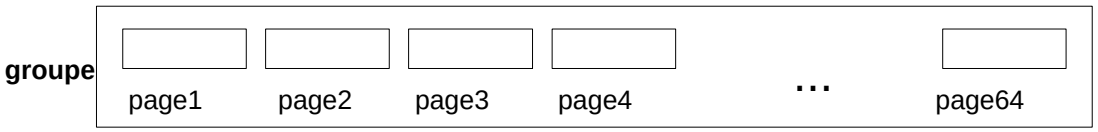
1.2) Structure des disques SSD

Les Disques à base de mémoires flash : '*Solid State Drive*' (**SSD**), représentent un nouveau type de ***MS***, de plus en plus utilisés pour leur performance et leur faible consommation d'énergie (relativement aux traditionnels disques magnétiques **HDD**)

C'est des dispositifs de stockage externe à base de mémoires flash (principalement de type NAND), ils sont aussi plus robustes que les disques magnétiques car ils ne contiennent pas de composants de mécanique de précision sensibles (comme les têtes de lectures/écritures du **HDD**) susceptibles de se détériorer en cas de chocs ou de mouvements brusques du périphérique. Ils sont à base, uniquement de semi-conducteurs.



La mémoire du **SSD** est divisée en **groupes** (ou bloc de pages, de capacité fixe, ex: 256K) et chaque groupe est composé d'un certain nombre de **pages** (ex. taille d'une page = 4K).



Il y a trois opérations possibles :
- Lire une page ⇒ très rapide (20 microsecondes, pour les disques actuels)
- Ecrire une page, à condition qu'elle soit dans un état effacé ⇒ rapide (100 à 200 microsecondes)
- Effacer toute les pages d'un groupe (un bloc de pages) ⇒ lente (quelques millisecondes)

Les groupes (ou blocs de pages) supportent un nombre fixe d'effacements, au delà duquel ils deviennent inutilisables. Il faut donc répartir de manière uniforme les effacements sur l'ensemble des groupes du **SSD** afin d'augmenter la durée de vie du disque ⇒ C'est la technique du ***Wear-Leveling***

Lors de la mise à jour du contenu d'une page, il est aussi préférable d'écrire le nouveau contenu directement dans une nouvelle page (déjà effacée). Cela évite l'attente (coûteuse) de l'effacement

d'un groupe au moment de l'écriture du nouveau contenu de la page. Les données stockées dans le disque, changent donc souvent d'emplacements physiques.
Pour rendre ces changements transparents, les utilisateurs manipulent les pages du **SSD** à travers des numéros logiques (qui restent constants et indépendants des localisations physiques des pages sur la mémoire flash). Le contrôleur du disque maintient alors une table d'association (c'est la ***FTL***) contenant les correspondances entre les numéros logiques et les adresses physiques des pages. A chaque écriture d'une page (dans un nouveau emplacement physique), la ***FTL*** est mise-à-jour en conséquence pour garder le lien entre le numéro logique de la page (qui reste inchangé) et le nouveau numéro physique utilisé. La ***FTL*** est chargée dans une mémoire vive interne au périphérique **SSD**.

2) Notion de fichier et système de fichiers

Pour faire abstraction de ces détails physiques (liés aux dispositifs utilisés : **HDD**, **SSD**, ...), on considère la mémoire secondaire comme une grande zone de stockage formée de **blocs physiques** de même taille fixe. Selon le type de la **MS**, le bloc physique, au niveau du système d'exploitation, peut être composé d'un ou de plusieurs secteurs d'un **HDD** ou alors d'une ou de plusieurs pages d'un **SSD**.

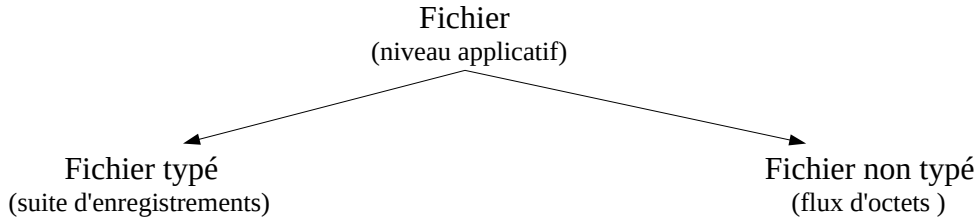
Chaque bloc physique représente donc un tableau d'octets pouvant être lu ou écrit en une seule opération d'accès (ou d'E/S physique) : **LireBloc(num, Buf)** et **EcrireBloc(num, Buf)**. L'accès à la mémoire secondaire ne peut se faire qu'à travers ces deux opérations de base.

L'opération **LireBloc(i ,buf)** effectue le transfert physique du contenu du bloc numéro *i* (de la **MS**) vers une zone en **MC** (la variable *buf*) ayant une taille suffisante pour stocker le contenu d'un bloc.

L'opération **EcrireBloc(i ,buf)** effectue le transfert physique inverse, du contenu de la variable *buf* (en **MC**) vers le bloc numéro *i* de la **MS**.

Ces deux opérations d'E/S physiques sont très coûteuses (en terme de temps d'exécution) par rapport aux accès à la mémoire centrale, car elles engendrent des transferts de données vers des dispositifs externes (les disques).

Un **fichier** est le concept à travers lequel, un programme ou une application stocke des données en mémoire externe. Les fichiers sont utilisés à différents niveaux d'abstraction avec des sémantiques différentes:

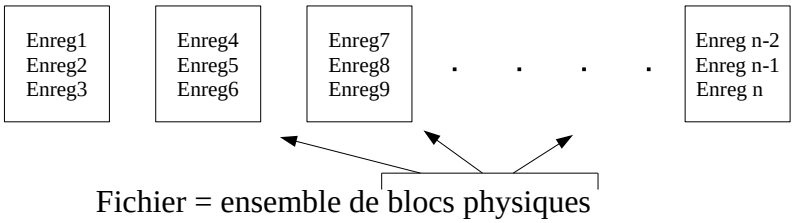


i) Au niveau 'application' (ou langage de programmation de haut niveau), un **fichier** est **une suite d'enregistrements** (ou articles) persistants. L'accès aux enregistrements, à travers le langage de programmation considéré, se fait via des opérations spéciales pour la manipulation des fichiers (ex. en langage *Pascal : read(f,e), write(f,e), reset(f), …*). On peut aussi définir un fichier comme étant

une suite d'octets, on parle alors de **flux** (ex : les *streams* de la librairie standard du langage *C : FILE **). C'est des fichiers « non typés ». Dans des systèmes d'exploitation de type Unix, ces flux (*streams*) généralisent aussi la notion d'E/S pour une manipulation uniforme et indépendante des types de périphérique utilisés (disques, imprimantes, écrans, clavier, réseaux, …).

Dans le cas des fichiers typés, les enregistrements (ou articles) sont formés par un ensemble de champs (ou attributs). Parmi ces champs, un ou plusieurs peuvent jouer le rôle de **clé de recherche** (*Search Key*). C'est le (les) champ(s) utilisé(s) généralement pour rechercher des enregistrements dans les applications (ex : le matricule d'un étudiant, le nom d'une personne, la date, le lieu et l'heure d'une mesure captée par une sonde … etc). N'importe quel champ (ou groupe de champs) peut jouer le rôle d'une clé de recherche. Pour faciliter certaines opérations sur les fichiers, on peut aussi stocker les enregistrements d'un fichier en ordre croissant ou décroissant selon les valeurs de la clé (on parle alors de **fichier ordonné**).

ii) Au niveau système (niveau physique), un fichier est un **ensemble de blocs en MS**, renfermant une suite d'octets non interprétés. Les données (par exemple les enregistrements) du fichier sont stockés à l'intérieur des blocs selon une certaine organisation. L'accès aux contenus des blocs se fait via les opérations d'E/S bufférisées (LireBloc et EcrireBloc).



Pour pouvoir manipuler correctement un fichier, on a besoin de connaître un certain nombre d'informations le concernant. Ce sont les **caractéristiques** du fichier. Cela peut être par exemple le nom du fichier, sa taille, les blocs qui le constituent, la date de la dernière mise-à-jour, celle de la dernière consultation, le propriétaire, les droit d'accès, … etc.

Toutes les opérations du niveau 'application' passent par le système qui les traduit en opérations de bas niveau pour accéder physiquement aux blocs d'E/S de la **MS**. Comme les opérations d'E/S sont coûteuses (en temps), le système maintient en **MC** une zone spéciale de taille limitée (la **zone tampon** ou "buffer cache") lui permettant de garder en **MC** les copies de quelques blocs physiques choisis selon certaines stratégies (par exemple les plus utilisés). Cette zone tampon est totalement transparentes aux programmes d'application qui utilisent les fichiers. Elle est gérée exclusivement par le système d'exploitation. Par contre elle influence beaucoup les performances des applications qui manipulent des fichiers.

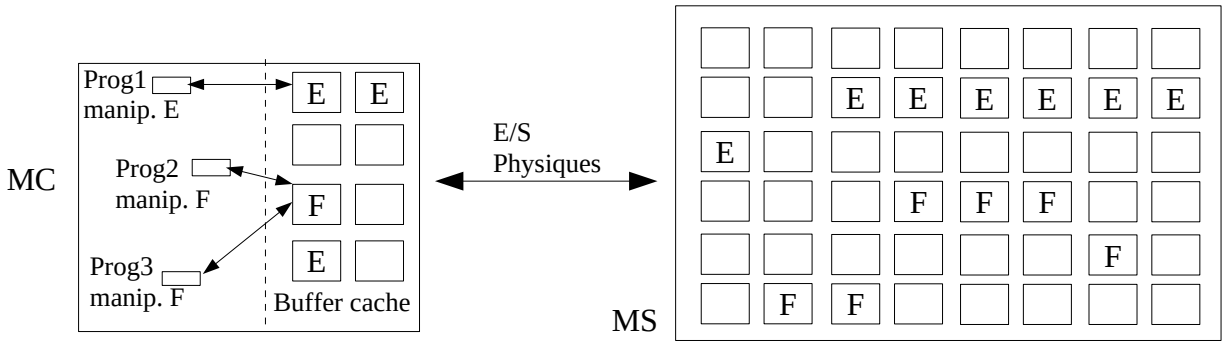
Ainsi, si par exemple, une application demande la lecture d'un enregistrement donné, le système vérifie d'abord si le bloc concerné ne se trouve pas déjà en **MC** (dans la zone tampon). S'il y est, l'enregistrement cherché, sera directement transmis à l'application, sans qu'il y ait de lecture physique en mémoire externe. Le temps d'accès à la donnée est dans ce cas très court. Sinon, si le bloc concerné n'est pas en **MC,** l'application est mise en attente et une opération de lecture physique de bloc est engagée afin de récupérer d'abord le bloc en **MC** et ensuite l'enregistrement cherché. Le coût d'accès dans ce cas est très élevé.

Lors de la lecture physique d'un bloc en **MC**, le système cherche un emplacement libre dans la zone

tampon pour y copier le contenu du bloc lu. S'il n'y a pas d'emplacement libre, le système choisit un des blocs déjà présents en zone tampon et le recopie, éventuellement, sur disque (EcrireBloc) avant de réutiliser son emplacement pour stocker le contenu du nouveau bloc lu. Le choix de l'emplacement à écraser dépend de la stratégie de remplacement adoptée par le système (par exemple la stratégie *LRU :* l'emplacement en zone tampon du bloc le moins récemment utilisé).

Lors de la mise à jour d'un enregistrement par une application, le système réalise l'opération directement dans sa zone tampon (en **MC**). L'écriture physique sur disque ne se fera que plus tard (lors d'un remplacement de bloc par exemple).

Périodiquement, le système synchronise son cache avec la **MS**. Tous les blocs qui ont été modifiés dans la zone tampon, seront alors physiquement écrits en **MS** lors de l'opération de synchronisation. En cas de panne provoquant la perte de la mémoire centrale (donc de la zone tampon aussi), les dernières modifications effectués sur les fichiers ouverts, risquent donc d'être perdues.



Pour les applications critiques, il existe des moyens pour **forcer l'écriture physique** (sans attendre la synchronisation périodique du cache). Dans les systèmes de type *Unix*, l'appel système '*fsync(f)*' permet d'écrire physiquement sur la **MS**, tous les blocs modifiés de la zone tampon et non encore synchronisés, relatifs au fichier f.

Caractéristiques et bloc d'en-tête

Pour que le système puisse gérer un fichier, il a besoin de connaître les informations sur ses caractéristiques : les blocs utilisés par le fichier, l'organisation du fichier, les droit d'accès associés, ...

Ces informations (ou certaines de ces informations) sont stockées à certains endroits réservé du disque. Par exemple, pour un système de fichier très simple, le bloc se trouvant à une adresse fixe du disque, pourrait être réservé par le système pour contenir une table où chaque ligne renseigne sur les caractéristiques d'un fichier (nom, taille, blocs utilisés, ...). Quand une application désire ouvrir un fichier de nom donné, le système récupère ses informations à partir de cette table.

Dans les **MS** séquentielles de type bandes magnétiques, ce type d'information (les caractéristiques) se trouvait au début de chaque fichier (d'où le nom de "bloc d'en-tête").

Certaines applications ont besoin aussi de gérer des caractéristiques particulières pour pouvoir manipuler correctement leurs fichiers de données. Ce type de caractéristiques peuvent aussi être stockées en début de fichier, avant les données à proprement parler.

Méthode d'accès

Une structure de fichier consiste à définir et concevoir des structures de données et les algorithmes associés, pour organiser les blocs d'un fichier sur **MS** et implémenter certaines opérations d'accès pour la manipulation des données du fichier.

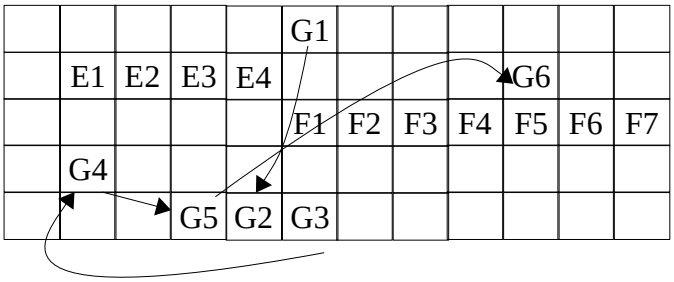
Cela inclus par exemple de définir :
- une manière d'organiser les blocs du fichier sur **MS**
- le placement des enregistrements à l'intérieur des blocs
- les caractéristiques et informations nécessaires pour manipuler le fichier
- le nombre de buffers à réserver en **MC** pour optimiser les accès
- l'implémentation des opérations d'accès (recherche, insertion, suppression, …)

3) Modèle pour les structures de fichiers

Afin d'étudier les méthodes d'accès aux fichiers dans un cadre général, on modélise la **MS** par une zone contiguë de blocs numérotés séquentiellement (ces **numéros** représentent les **adresses de blocs**).

Les blocs sont des zones contiguës de même taille, renfermant, entre autre, les données d'un fichier (enregistrements ou simple suite d'octets).

Dans le schéma suivant, la **MS** contient trois fichiers : *E, F* et *G*



Le fichier *E* est formé de 4 blocs contigus (*E1, E2, E3* et *E4*), le fichier *F* est formé de 7 blocs contigus et le fichier *G* est une liste de blocs.

Pour écrire des algorithmes sur les structures de fichiers on utilisera la **machine abstraite** définie par le modèle suivant:

{*Ouvrir, Fermer, LireDir, EcrireDir, Aff_Entete, Entete, Allocbloc* }

Dans ce modèle, on manipule des numéros de blocs relatifs au début de chaque ficher (c'est donc des numéros logiques). L'utilisation des adresses physiques n'est pas d'une utilité particulière à ce niveau de la présentation.

Dans ce modèle, un fichier est donc un ensemble de blocs numérotés logiquement (*1, 2, 3, ... n*). La fonction Allocbloc(F) ne fait alors que retourner le prochain numéro de bloc après le dernier.

Déclaration d'un fichier ***f*** et de ses zones tampons ***buf1, buf2*** (deux buffers) :

```
TypeBloc =   Structure        // le contenu des blocs
                  tab : Tableau [ b ] de TypeEnreg
                  NB : eniter
             fin
```

f : FICHIER de TypeBloc BUFFER ***buf1***, ***buf2*** ENTETE (type1, type2, ...typem);

Dans cette déclaration, il y a :

- La description du contenu des blocs (un tableau d'enregistrement et un entier)
- La variable ***f*** de type FICHIER
- Deux variables ***buf1*** et ***buf2***, de type 'TypeBloc' qui servent comme zones tampons pour lire et écrire les blocs du fichier.
- La structure de l'entête du fichier, formée par *m* caractéristiques dont les types sont spécifiés

Les opérations du modèle sont :

Ouvrir(f, nomfichier, mode)

Ouverture ou création d'un fichier de nom ***nomfichier***, selon la valeur de ***mode***

'A' : Ouverture d'un ancien fichier en lecture/écriture. Ses caractéristiques sont chargés en MC (par exemple à partir du bloc d'entête du fichier).

'N' : Création d'un nouveau fichier en lecture/écriture. Une zone en MC est allouée pour contenir ses caractéristiques, initialisées par des valeurs adéquates.

Fermer(f)

Fermeture du fichier f, en sauvegardant ses caractéristiques sur disque (par exemple dans le bloc d'entête) en cas de modifications.

LireDir(F, i, buf)

Lecture du bloc numéro ***i*** du fichier ***f*** dans la variable ***buf***.

EcrireDir(F, i, buf)

Ecriture du contenu de la variable ***buf*** dans le bloc numéro ***i*** du fichier ***f***.

Entete(f, i)

Cette fonction retourne la valeur de la $i^{ème}$ caractéristique de ***f***. Elle ne nécessite pas d'accès disque car les caractéristiques ont déjà été chargés en MC par l'opération d'ouverture.

Aff_entete(f, i, val)

Affecte la valeur ***val*** dans la $i^{ème}$ caractéristique de ***f***. Elle ne nécessite pas d'accès disque car l'affectation se fait en MC dans la zone déjà allouée par l'opération d'ouverture du fichier.

AllocBloc(f)

Allocation d'un nouveau bloc au fichier ***f***. Cette fonction retourne le numéro du bloc alloué.
Dans les fichiers formés par des blocs contigus, cette opération n'est pas nécessaire, car l'écriture dans un bloc juste après la fin du fichier est suffisante et est équivalente à une allocation de bloc.

4) Les fichiers en langage C

En langage ***C*** les fichiers sont vus comme une suite d'octets en **MS**. Dans la librairie standard (*libc*) il existe des fonctions portables manipulant les fichiers sous forme de flux à travers la structure *FILE* définie dans <*stdio.h*>. Les objets de type *FILE* seront alloués et libérés par les fonctions de la librairie de manière transparente au programme utilisateur. De ce fait on manipule cette structure qu'à travers des indirections (pointeurs).

La déclaration d'une variable fichier ***f*** (de type flux) est comme suit :

FILE *f;

L'ouverture d'un fichier se fait comme suit :

f = fopen(nomfichier, mode);
nomfichier une chaîne de caractères indiquant le nom du fichier et ***mode*** une chaîne de caractères indiquant le mode d'ouverture.

Pour un fichier texte, le mode peut être :

mode	explications
« r »	ouverture en lecture
« w »	création d'un nouveau fichier (écrasement de l'ancien s'il existe déjà)
« a »	ouverture en mode « ajout »
« r+ »	ouverture en lecture/écriture
« w+ »	création d'un nouveau fichier en lecture/écriture
« a+ »	ouverture en mode « ajout » en lecture/écriture

Pour un fichier binaire, le mode peut être :

mode	explications
« rb »	ouverture en lecture
« wb »	création d'un nouveau fichier (écrasement de l'ancien s'il existe déjà)
« ab »	ouverture en mode « ajout »
« rb+ » / « r+b »	ouverture en lecture/écriture
« wb+ » / « w+b »	création d'un nouveau fichier en lecture/écriture
« ab+ » / « a+b »	ouverture en mode « ajout » en lecture/écriture

Un fichier texte est formé d'un ensemble de lignes terminées chacune par une marque de fin de ligne. Dans les systèmes de type Unix (comme Linux par exemple), cette marque de fin de ligne est le caractère '*n*' (code ascii 10). Dans d'autres systèmes, la marque de fin de ligne peut être composée de deux caractères (comme '*r*' et '*n*'). Les fonctions de lecture et d'écriture sur les fichiers textes, permettent de s'abstraire de ces différences entre systèmes.

Un fichier binaire est simplement une suite d'octets sans aucune interprétation particulière faite par le système. Ces suites d'octets représentent les données telle qu'elles étaient représentées en **MC** avant leur transfert sur le fichier. De ce fait les données d'un fichier binaire sont très dépendantes du système où elles ont été produites (donc généralement peu portables).

La fermeture d'un fichier:

fclose(f);

Pour savoir si on a dépassé la fin de fichier (en mode lecture) :

feof(f)

Retourne vrai (entier différent de 0) si on a tenté de lire au delà de la fin de fichier.

Donc le **schéma général** pour **lire le contenu d'un fichier** dans ce cas est le suivant :

```
FILE *f ;
f = fopen(...) ;
// lecture des premières données
<< opération_de_lecture>>
while ( ! feof( f ) )  {
        // Traitement des données lus
        …
        // lecture des données suivantes
        << opération_de_lecture>>
}
fclose( f ) ;
```

a) Lecture / Ecriture en mode binaire

Pour lire des données d'un fichier binaire, on utilise généralement 'fread' :

nbelt_lus = fread(buf, taille_elt, nb_elt, f);

Demande de lecture d'un nombre d'éléments consécutifs égal à ***nb_elt***, chacun de taille ***taille_elt*** octets depuis le fichier ***f***. Le résultat de la lecture sera placé dans la zone pointée par ***buf*** et de taille au moins égale à ***taille_elt * nb_elt*** octets.

La fonction retourne le nombre d'éléments effectivement lus, qui peut donc être inférieur au nombre demandé (en cas de fin de fichier ou d'erreur de lecture).

Pour écrire des données dans un fichier binaire, on utilise généralement 'fwrite' :

nbelt_ecr = fwrite(buf, taille_elt, nb_elt, f);

Demande d'écriture d'un nombre d'éléments égal à ***nb_elt***, chacun de taille ***taille_elt*** octets dans le fichier ***f***. Les octets à écrire (au nombre de ***taille_elt*nb_elt***) sont à récupérer depuis la zone mémoire pointée par ***buf***.

La fonction retourne le nombre d'éléments effectivement écrits, qui peut donc être inférieur au nombre demandé (en cas d'erreur d'écriture).

A chaque opération de lecture ou d'écriture, la position courante dans le fichier avance d'un nombre d'octets égal à celui des données lus ou écrits.

Pour modifier la position courante dans un fichier, on peut utiliser 'fseek' :

fseek(f, deplacement, origine) ;

Déplace la position courante d'un nombre d'octets égal à ***deplacement***, relativement à :

- début du fichier, si ***origine*** vaut *SEEK_SET*
- position courante, si ***origine*** vaut *SEEK_CUR*
- fin du fichier, si ***origine*** vaut *SEEK_END*

b) Lecture / Ecriture en mode texte

Pour lire un caractère dans un fichier texte, on peut utiliser :

c = fgetc(f);

Lit et retourne (type *int*) le prochain caractère de ***f*** ou bien la constante *EOF* si la fin de fichier a été dépassée. En cas d'erreur *EOF* est aussi retournée.

Pour écrire un caractère dans un fichier texte, on peut utiliser :

fputc(c, f);

Ecrit le caractère ***c*** à la position courante du fichier ***f***.
En cas d'erreur, la fonction retourne (type *int*) la constante *EOF*.

Pour lire une ligne dans un fichier texte, on peut utiliser :

fgets(buf, n, f);

Lit dans la variable ***buf***, tous les caractères à partir de la position courante de ***f***, jusqu'à trouver une marque de fin de ligne '*n*' (qui est aussi lue dans ***buf***) ou alors jusqu'à ce que **n-1** caractères soient lus. Un caractère de fin de chaîne '*0*' est rajouté à la fin de ***buf***.
En cas d'erreur ou de dépassement de la fin de fichier, la fonction **fgets** retourne *NULL*.

Pour écrire une chaîne de caractères dans un fichier texte, on peut utiliser :

fputs(buf, f);

Ecrit tous les caractères contenus de ***buf*** (sauf le '*0*') dans le fichier ***f***, à partir de la position courante. Pour écrire une ligne, il faut prévoir un caractère '*n*' à la fin de ***buf*** (et avant le caractère de fin de chaîne '*0*').
En cas d'erreur, la fonction retourne *EOF*.

Pour effectuer une lecture formatée depuis un fichier texte, on peut utiliser :

fscanf(f, format, &var1, &var2, …)

Comme **scanf**, sauf que les données proviennent du fichier texte ***f*** (au lieu de la console)

Pour effectuer une écriture formatée dans un fichier texte, on peut utiliser :

fprintf(f, format, exp1, exp2, …)

Comme **printf**, sauf que les données iront dans le fichier texte ***f*** (au lieu de la console)

La position courante dans le fichier est automatiquement mise à jours après chaque opération de lecture ou d'écriture, pour se positionner sur les prochains octets à lire ou à écrire de manière séquentielle.
L'utilisation de fseek n'est pas conseillée avec les lectures/écriture en mode texte à cause des éventuelles transformations des caractères de fin de ligne effectuées dans certains systèmes d'exploitation. Dans ce cas le calcul des déplacements devient un peu difficile à faire.

c) Exemples de programmes C

Dans le programme suivant, on construit un fichier binaire contenant un certain nombre d'enregistrements pour une application simple de gestion d'agenda téléphonique :

```
/*
 * Construction d'un fichier binaire avec n enregistrements : < nom , telephone >
*/

#include <stdio.h>

int main()
{
  // variable enregistrement
  struct Tenreg {
        char nom[20];
        char tel[15];
  } e;
```

```
  // variable fichier
  FILE *f;

  char nomf[30];

  printf( "\nConstruction d'un fichier agenda telephonique\n\n");
  printf( "Donnez le nom du fichier à construire : ");
  // un espace avant %s permet de sauter tous les caractères blancs avant de lire le nom de fichier
  scanf( " %s", nomf );

  // creation du nouveau fichier en mode binaire
  f = fopen( nomf, "wb" );
  if ( f == NULL ) {
        printf( "erreur lors de l'ouverture du fichier %s en mode wb\n", nomf );
        return 0;
  }

  // lecture d'un enregistrement depuis la console
  printf( "donnez un nom et un tel (ou 0 0 pour terminer le programme) : " );
  // un espace avant %s permet de sauter tous les caractères blancs avant de lire les données e.nom et e.tel
  scanf( " %s %s", e.nom, e.tel );

  while ( e.nom[0] != '0' ) {
        // écriture dans le fichier
        fwrite( &e, sizeof(e), 1, f );

        // lecture d'un enregistrement depuis la console
        printf( "donnez un nom et un tel (ou 0 0 pour terminer le programme) : " );
        // un espace avant %s permet de sauter tous les caractères blancs avant de lire les données e.nom et e.tel
        scanf( " %s %s", e.nom, e.tel );
  }

  // fermeture du fichier
  fclose( f );
  return 0;

}
```

Voici un exemple de programme qui parcourt séquentiellement le fichier de données et affiche son contenu :

```
/*
 * Liste le contenu d'un fichier binaire avec enregistrements : < nom , telephone >
*/

#include <stdio.h>

int main()
{
  // variable enregistrement
  struct Tenreg {
        char nom[20];
        char tel[15];
  } e;

  FILE *f;      // variable fichier

  char nomf[30];
  int i;
```

```
  printf( "\nAffichage du contenu du fichier agenda telephonique\n\n" );
  printf( "Donnez le nom du fichier à lister : " );

  // un espace avant %s permet de sauter tous les caractères blancs avant de lire le nom de fichier
  scanf( " %s", nomf );

  // ouverture du fichier
  f = fopen( nomf, "rb" );
  if ( f == NULL ) {
        printf( "erreur lors de l'ouverture du fichier %s en mode rb\n", nomf );
        return  0;
  }

  // lecture des enregistrements depuis le fichier
  i = 1;
  fread( &e, sizeof(e), 1, f );
  while ( ! feof(f) ) {
        printf( "%3d nom : %s \t tel : %s\n", i++, e.nom, e.tel );
        fread( &e, sizeof(e), 1, f );
  }

  // fermeture du fichier
  fclose( f );
  return 0;

}
```

Le programme suivant montre un exemple d'accès direct (avec fseek) à un enregistrement de position donnée :

```
/*
 * Accès direct à un enregistrement de type : < nom , telephone >
*/

#include <stdio.h>

int main()
{
  // variable enregistrement
  struct Tenreg {
        char nom[20];
        char tel[15];
  } e;

  // variable fichier
  FILE *f;

  char nomf[30];
  int n, i;

  printf( "\nAccès direct au contenu d'un fichier 'agenda telephonique'\n\n" );
  printf( "Donnez le nom du fichier à manipuler : " );
  scanf( " %s", nomf );

  // ouverture du fichier
  f = fopen( nomf, "rb" );
  if ( f == NULL ) {
        printf( "erreur lors de l'ouverture du fichier %s en mode rb\n", nomf );
        return 0;
  }
```

```
  printf( "Donnez le numéro de l'enregistrement à lire : " );
  // un espace avant %d permet de sauter tous les caractères blancs avant de lire la donnée i
  scanf( " %d", &i );
  // lecture d'un enregistrement par un accès direct
  fseek( f, (i-1)*sizeof(e), SEEK_SET );
  n = fread( &e, sizeof(e), 1, f );
  if ( n == 1 )      // si le nombre d'enreg lus = 1
        printf( "enreg num:%3d \t nom : %s \t tel : %s\n", i, e.nom, e.tel );
  // fermeture du fichier
  fclose( f );
  return 0;

}
```

Si on avait opté pour un fichier texte, on aurait eu le programme suivant pour la construction :

```
/*
 * Construction d'un fichier texte avec n enregistrements <nom,telephone>
*/
#include <stdio.h>

int main()
{
  // variable enregistrement
  struct Tenreg {
        char nom[20];
        char tel[15];
  } e;
  // variable fichier
  FILE *f;
  char nomf[30];

  printf("\nConstruction d'un fichier agenda telephonique\n\n");
  printf("Donnez le nom du fichier à construire : ");
  // un espace avant %s permet de sauter tous les caractères blancs avant de lire le nom de fichier
  scanf(" %s", nomf);

  // creation du nouveau fichier en mode texte
  f = fopen( nomf, "w" );
  if ( f == NULL ) {
        printf( "erreur lors de l'ouverture du fichier %s en mode w\n", nomf );
        return 0;
  }
  // insertion des enregistrements lus à la console
  printf( "donnez un nom et un tel (ou 0 0 pour terminer le programme) : " );
  // un espace avant %s permet de sauter tous les caractères blancs avant de lire les données e.nom et e.tel
  scanf( " %s %s", e.nom, e.tel );

  while ( e.nom[0] != '0' ) {
        // écriture formatée dans le fichier
        fprintf( f, " %s , %s\n", e.nom, e.tel );          // un enregistrement par ligne

        printf( "donnez un nom et un tel (ou 0 0 pour terminer le programme) : " );
        scanf( " %s %s", e.nom, e.tel );
  }
  // fermeture du fichier
  fclose( f );
  return 0;

}
```

Voici un exemple de fichier texte produit par ce programme :

```
$ cat agenda1.txt

aaaaaa , 123456
bbbbb , 147258
ccccccc , 187456
ddd , 054963
eeeeee , 248269
fffff , 321654
```

Le programme suivant permet de lire séquentiellement le fichier ainsi construit et affiche les enregistrements à l'écran :

```
/*
 * Liste le contenu d'un fichier texte (enregistrements de type <nom,telephone>)
*/
#include <stdio.h>

int main()
{
  // variable enregistrement
  struct Tenreg {
        char nom[20];
        char tel[15];
  } e;

  FILE *f;      // variable fichier
  char nomf[30];
  int n, i;

  printf( "\nAffichage du contenu d'un fichier agenda telephonique\n\n" );
  printf( "Donnez le nom du fichier à lister : " );
  scanf( " %s", nomf );

  // ouverture du fichier texte en mode lecture
  f = fopen( nomf, "r" );
  if ( f == NULL ) {
        printf( "erreur lors de l'ouverture du fichier %s en mode r\n", nomf );
        return 0;
  }

  // lecture des enregistrements depuis le fichier
  i = 1;
  n = fscanf( f, "%s , %s", e.nom, e.tel );
  if ( n == 2 ) // si le nombre d'elements lus = 2 (le nom et le tel)
        printf( "%3d \t nom : %s \t tel : %s\n", i++, e.nom, e.tel );
  while ( ! feof(f) ) {
        n = fscanf( f, "%s , %s", e.nom, e.tel );
        if ( n == 2 ) // si le nombre d'elements lus = 2 (le nom et le tel)
              printf( "%3d \t nom : %s \t tel : %s\n", i++, e.nom, e.tel );
  }

  // fermeture du fichier
  fclose( f );
  return 0;

}
```

Voici un exemple d'affichage produit par ce programme :

```
Affichage du contenu d'un fichier agenda telephonique

Donnez le nom du fichier à lister : agenda1.txt
 1      nom : aaaaaa     tel : 123456
 2      nom : bbbbb      tel : 147258
 3      nom : ccccccc    tel : 187456
 4      nom : ddd        tel : 054963
 5      nom : eeeeee     tel : 248269
 6      nom : fffff      tel : 321654
```

Les écritures forcées en C

Les opérations de lecture/écriture du langage C utilisent des buffers additionnels comme une zone tampon entre le programme d'application et le buffer cache du système (voir la figure en page 7 de ce même chapitre « Généralités »). Lors des écritures effectuées par un programme (par exemple à l'aide de *fwrite*) les données sont d'abord envoyées vers ces buffers applicatifs jusqu'à devenir pleins auquel cas ils seront ensuite automatiquement vider vers les buffers de la zone tampon du système (toujours en **MC**). Au moment de la synchronisation du cache système, les données seront alors physiquement écrites sur **MS**.
Lors de la fermeture d'un fichier (*fclose*) les buffers applicatifs sont aussi vider vers la zone tampon du système. L'opération *fflush(**f**)* permet aussi de forcer le vidage du buffer applicatif de ***f*** vers la zone tampon système (en **MC**), même s'il n'était pas plein.

Donc pour réaliser une écriture forcée (écriture physique vers la **MS**), il faut (après l'écriture des données) d'abord vider le buffer applicatif (*fflush*) et ensuite synchroniser le cache système (*fsync*).

Ces écritures forcée sont quelquefois nécessaires pour mettre à l'abri des données critiques en les sauvegardant en **MS** avant l'occurrence d'une éventuelle panne qui pourrait effacer le contenu de la **MC** et donc des buffers non encore synchronisés par le système d'exploitation.

5) Conclusion

Les prochains chapitres concernent les différents types d'organisations de fichiers et l'étude de leur impacts sur les performances. Ce sont les "*méthodes d'accès*" aux fichiers ou alors "***structures de fichiers***".

Voici quelques exemples de structures de fichiers:

- blocs contigus, enregistrements à taille fixe, non ordonné
- liste de blocs, enregistrements à taille variable, ordonné
- ensemble de blocs avec table d'index
- arbre de blocs
- blocs contigus avec fonction de hachage
- … etc.

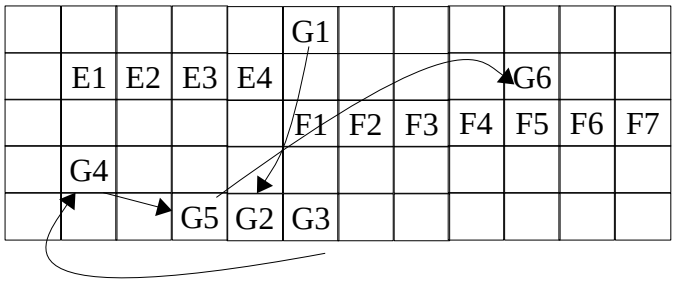
Certaines structures de fichiers sont bien adaptées pour des fichiers dont la taille reste relativement stable dans le temps (fichiers statiques) et d'autres peuvent supporter aussi des fichiers dynamiques et très volumineux (B-Arbres, Hachage dynamique).

Chapitre 2 Les Structures Séquentielles

1) Organisation globale des blocs

Dans un premier temps, on étudiera deux possibilités distinctes d'organiser les blocs au sein d'un fichier:
- soit le fichier est « *vu comme un tableau* » : tous les blocs qui le forment sont contigus
- soit le fichier est « *vu comme une liste* » : les blocs ne sont pas forcément contigus, mais sont chaînés entre eux.

Dans la figure ci-dessous, on a deux fichiers vus comme tableau (*E* et *F*) et un fichier vu comme une liste (*G*). Les blocs *F1, F2, … F7* sont contigus, de même pour les blocs *E1, E2, E3* et *E4*. Par contre les blocs *G1, G2, … G6* ne sont pas contigus, ils sont chaînés entre eux formant une liste de blocs.



Parmi les caractéristiques nécessaires pour manipuler un fichier vu comme tableau, on pourra avoir par exemple :
- Le numéro du premier bloc,
- Le numéro du dernier bloc (ou alors le nombre de blocs utilisés).

Dans le cas où on utilise des numéros logiques pour adresser les blocs, on n'aura pas alors besoin de la caractéristique 'numéro du premier bloc' car elle vaut toujours 1 dans ce cas.
C'est par exemple le cas avec la machine abstraite définie dans le premier chapitre.

Pour un fichier vu comme liste, il suffirait par contre de connaître le numéro du premier bloc (la tête de la liste), car dans chaque bloc, il y a le numéro du prochain bloc (comme le champ suivant dans une liste). Dans le dernier bloc, le numéro du prochain bloc pourra être mis à une valeur spéciale (par exemple -1) pour indiquer la fin de la liste.

2) Organisation interne des blocs

Les blocs contiennent les enregistrements d'un fichier. Ces derniers (les enregistrements) peuvent être de longueur *fixe* ou *variable*.

Dans le cas des **enregistrements de longueur fixe**, chaque bloc pourra alors contenir un tableau d'enregistrements de même type.

Exemple:

```
Type Tenreg = structure                  // structure d'un enregistrement du fichier
              matricule : chaine(10)     // matricule : taille 10 octets
              nom : chaine(20)           // nom : taille 20 octets
              age : entier               // age : taille 4 octets (pour simplifier l'exemple)
       fin

Type Tbloc = structure                   // structure d'un bloc du fichier
              tab : tableau[ b ] de Tenreg  // un tableau pouvant contenir (au max) b enreg.
              NB : entier                // nombre d'enreg insérés dans tab (entre 0 et b).
       fin
```

Dans le cas où la taille d'un bloc physique est de 4096 octets (c-a-d 4K), on pourra par exemple choisir la valeur 120 pour la constante *b* (la taille du champ 'tab') afin de minimiser l'espace perdu dans le bloc :

taille d'un enreg = 10+20+4 = 34 octets
avec b=120, le tableau 'tab' occupera alors 120*34 = 4080 octets
le champ 'NB' est sur 4 octets (taille d'un entier sur 32 bits)
donc la structure 'Tbloc' occupe une taille de : 4080+4 = 4084 octets
donc l'espace perdu dans le bloc est 4096 – 4084 = 12 octets

Si le champ 'age' dans la structure Tenreg et le champ 'NB' dans la structure Tbloc étaient de taille 1 octets (c-a-d des valeurs possibles entre 0 et 255 pour age et NB), on aurait pu mettre au maximum *b*=132 enregistrements par blocs (avec un espace perdu de 3 octets par bloc) :

taille d'un enreg = 10+20+1 = 31 octets
avec b=132, le tableau 'tab' occupera alors 132*31 = 4092 octets
le champ 'NB' est sur 1 octet
donc la structure 'Tbloc' occupe une taille de : 4092+1 = 4093 octets
donc l'espace perdu dans le bloc est 4096 – 4093 = 3 octets

Dans le cas des **enregistrements de tailles variables**, chaque enregistrement pourra par exemple, être vu comme étant une chaîne d'octets ou de caractères (de longueur variable).

Les enregistrements peuvent être de longueur variable pour différentes raisons. Par exemple, il y a un ou plusieurs champs de tailles variables dans la structure d'un enregistrement, ou alors le nombre de champs varie d'un enregistrement à un autre, … etc. On peut représenter les valeurs à taille variable par des espaces de taille fixe (au risque d'une perte importante de l'espace de stockage), mais ce n'est pas toujours possible de le faire, car dans certaines situations, on ne connaît pas la taille maximale que peut prendre une valeur donnée.

Pour séparer les champs entre eux (à l'intérieur de l'enregistrement), on pourra soit utiliser un caractère spécial, soit préfixer le début des champs par leur taille. Dans l'exemple ci-dessous, on utilise trois (03) positions pour indiquer la taille des valeurs des champs.

```
...|0|2|7|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|a|0|1|7|b|b|b|b|b|b|b|b|b|b|b|b|b|0|3|7|c|c|c|.....|c|c|.....
   <-----------------------------------------><-----------------------------><----------....------>
         un champ sur 27 caractères              un champ sur 17 caractères        …..
```

En utilisant un nombre fixe de positions pour représenter la taille, on a de ce fait limiter la taille maximale de la valeur. Dans l'exemple précédent, avec cette codification des nombres sur 3 positions, cela implique que la taille maximale d'un champ ne peut pas dépasser 999 caractères (ou octets). Si par contre la taille maximale n'est pas connue au moment de la conception de la méthode, on peut représenter la taille sur un nombre variable de positions terminée par un caractère spécial.

Dans le cas d'enregistrements à taille variable, le bloc ne peut pas être défini comme étant un tableau d'enregistrements, car les éléments d'un tableau doivent toujours être de même taille. Une solution possible sera alors de considérer le bloc comme étant (ou contenant) un grand tableau de caractères de taille fixe, renfermant les différents enregistrements (stockés caractère par caractère).

Pour séparer les enregistrements entre eux, on utilise les mêmes techniques que celles utilisées dans la séparation entre les champs d'un même enregistrement, soit avec un caractère spécial (ex. '$'), soit on préfixe chaque enregistrement par sa taille.

Voici un exemple de déclaration d'un type de bloc pouvant être utilisé dans la définition d'un fichier vu comme liste avec format (taille) variable des enregistrements.

```
Type Tbloc = structure
        tab : tableau[ b' ] de caractères      // tableau de caractères pour les enreg.
        suiv : entier                          // numéro du bloc suivant dans la liste
    fin
```

Remarque: Même si les enregistrements sont de longueurs variables, la taille des blocs reste toujours fixe.

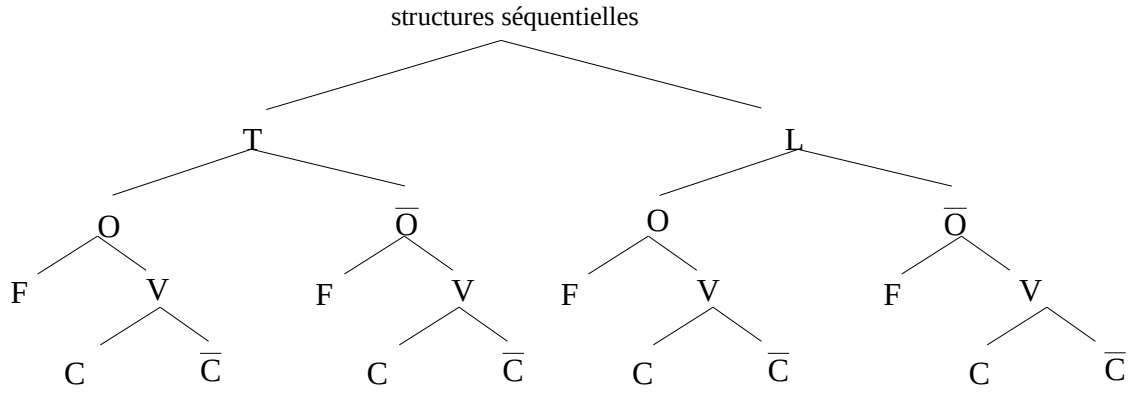
Pour minimiser l'espace perdu dans les blocs (dans le cas : format variable uniquement), on peut opter pour une organisation avec chevauchement entre deux ou plusieurs blocs:
Quand on veut insérer un nouvel enregistrement dans un bloc non encore plein et où l'espace vide restant n'est pas suffisant pour contenir entièrement cet enregistrement, celui-ci sera découpé en 2 parties de telle sorte à occuper tout l'espace vide du bloc en question par la 1ere partie, alors que le reste (la 2e partie) sera insérée dans un nouveau bloc alloué au fichier. On dit alors que l'enregistrement se trouve à cheval entre 2 blocs. On peut facilement généraliser cette approche pour supporter des enregistrements à cheval sur plusieurs blocs (c'est le cas des enregistrements de grandes tailles, éventuellement supérieure à celle d'un bloc physique)

3) Taxonomie des structures simples de fichiers

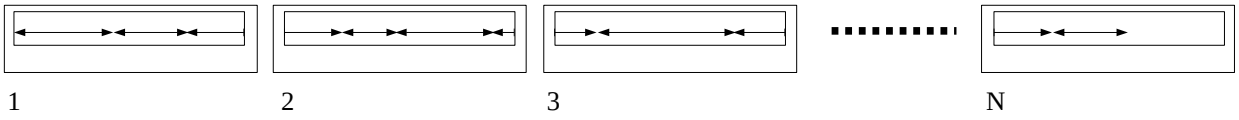
En combinant d'une part, entre l'organisation globale des fichiers (*vu comme tableau* ou *vu comme liste*) et celle interne aux blocs (*format fixe* ou *variable* des enregistrements) et d'autre part, en considérant la possibilité de maintenir le fichier *ordonné* ou non, on peut définir une classe de 12 méthodes d'accès (dites « séquentielle ») pour organiser des données sur disque.
Utilisons la notation suivante:

T : pour fichier *vu comme tableau*, L : pour fichier *vu comme liste*
O : pour fichier *ordonné*, $\overline{O}$: pour fichier *non ordonné*
F : pour format *fixe* des enregistrements, V : pour format *variable*
C : *avec chevauchement* des enregistrements entre blocs, $\overline{C}$: *sans chevauchement*

Les feuilles de l'arbre suivant, représentent les 12 méthodes d'accès séquentielles:

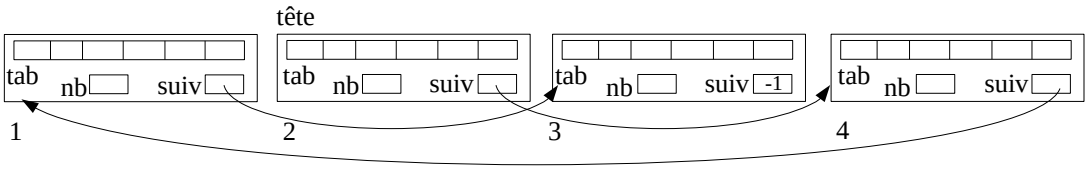


Par exemple la méthode **T $\overline{\mathbf{O}}$ V C** représente l'organisation d'un fichier *vu comme tableau* (**T**), *non ordonné* (**$\overline{\mathbf{O}}$**), avec des enregistrements de taille *variables* (**V**) et acceptant les *chevauchements* entre blocs (**C**) :



La recherche est séquentielle, l'insertion en fin de fichier et la suppression est logique.

Dans le cas d'un fichier **LOF** (fichier *vu comme liste*, *ordonné* avec enregistrements à taille *fixe*), chaque bloc pourra contenir par exemple, un tableau d'enregistrements (*tab*), un entier indiquant le nombre d'enregistrements dans le tableau (*nb*) et un entier pour garder la trace du bloc suivant dans la liste (*suiv*) :



La recherche est séquentielle, l'insertion provoque des décalages intra-blocs uniquement (pour garder l'ordre des enregistrements) et la suppression peut être logique ou physique.

4) Exemple complet: fichier de type « TOF »
(fichier *vu comme tableau*, *ordonné* avec enregistrements à taille *fixe*)

Nous ferons les choix suivants :

- La recherche d'un enregistrement est dichotomique (rapide).
- L'insertion peut provoquer des décalages intra et inter-blocs (coûteuse).
- La suppression peut être réalisée par des décalages inverses (suppression physique coûteuse) ou alors juste par un indicateur booléen (suppression logique beaucoup plus rapide). Optons pour cette dernière alternative.

- L'opération du chargement initial consiste à construire un fichier ordonné avec n enregistrements initiaux, en laissant un peut de vide dans chaque bloc. Ce qui permettra de minimiser les décalages pouvant être provoqués par les futures insertions.
- Avec le temps, le facteur de chargement du fichier (nombre d'insertions / nombre de places disponibles dans le fichier) augmente à cause des insertions futures, de plus les suppressions logiques ne libèrent pas de places. Donc les performances tendent à se dégrader avec le temps. Il est alors conseillé de réorganiser le fichier en procédant à un nouveau chargement initial. C'est l'opération de réorganisation périodique.

Déclaration du fichier:

```
Soit  b = 30          // capacité maximale des blocs (en nombre d'enregistrements)

// Les types utilisés :

    Tenreg = structure          // Structure d'un enregistrement :
        cle : typeqlq               // le champs utilisé comme clé de recherche
        champ2 : typeqlq            // les autres champs de l'enregistrement,
        champ3 : typeqlq            // … (sans importance pour la suite)
        …                           // …
    Fin

    Tbloc = structure               // Structure d'un bloc :
        tab : tableau[ b ] de Tenreg        // tableau d'enreg d'une capacité maximal = b
        eff : tableau[ b ] de booleen       // indicateurs pour la suppression logique
        NB : entier                         // nombre d'enreg dans tab ( ≤ b)
    Fin
```

// Les variables globales : *F* et *buf*

```
    F : Fichier de Tbloc Buffer buf  Entete (entier, entier)
    /* Description de l'entête du fichier F :
       L'entête contient deux caractéristiques de type entier.
          - la première sert à garder la trace du nombre de bloc utilisés (ou alors le
            numéro logique du dernier bloc du fichier)
          - la deuxième servira comme un compteur d'insertions pour pouvoir calculer
            rapidement le facteur de chargement, et donc voir s'il y a nécessité de
            réorganiser le fichier.
    */
```

Module de recherche: (dichotomique)

en entrée la clé (*c*) à chercher.
en sortie le booleen *Trouv*, le numéro de bloc (*i*) contenant la clé et le déplacement (*j*)

```
Rech( c : typeqlq , var Trouv : bool , var i , j :entier )
var
   bi, bs, inf, sup : entier
   trouv, stop : booleen
```

```
DEBUT
        /* on suppose que le fichier est déjà ouvert */
        bs ← entete( F,1 )          // la borne sup (le num du dernier bloc de F)
        bi ← 1                      // la borne inf  (le num du premier bloc de F)

        // boucle pour la recherche dichotomique externe (dans le fichier F)
        Trouv ← faux;   stop ← faux;   j ← 1
        TQ  ( bi ≤ bs  et  Non Trouv et Non stop )
          i ← (bi + bs) div 2       // le bloc du milieu entre bi et bs
          LireDir( F, i, buf )
          SI ( c ≥ buf.tab[1].cle  et  c ≤ buf.tab[ buf.NB ].cle )
                // boucle pour la recherche dichotomique interne dans le bloc i (dans buf)
                inf ← 1;  sup ← buf.NB
                TQ  (inf ≤ sup  et  Non Trouv)          // recherche interne
                        j ← (inf + sup) div 2
                        SI  (c  = buf.tab[ j ].cle)  Trouv ← vrai
                        SINON
                                SI (c  < buf.tab[ j ].cle)    sup ← j-1
                                SINON                         inf ← j+1
                                FSI
                        FSI
                FTQ
                SI ( inf > sup )
                        j ← inf
                SINON
                        SI ( buf.eff[ j ] )  Trouv ← faux FSI       // en cas d'effacement logique, retourner faux
                FSI
                // fin de la recherche interne.
                // j : la position où devrait se trouver c dans buf.tab
                stop ← vrai
          SINON  // non ( c ≥ buf.tab[1].cle  et  c ≤ buf.tab[ buf.NB ].cle )
                SI ( c < buf.tab[1].cle )
                        bs ← i-1
                SINON   // donc c > buf.tab[ buf.NB ].cle
                        bi ← i+1
                FSI
          FSI
        FTQ
        SI ( bi > bs )   i ← bi ;   j ← 1   FSI          // fin de la recherche externe.
                                                         // i : num du bloc où devrait se trouver c
FIN // Recherche
```

Le coût de l'opération de recherche est logarithmique car la recherche dichotomique effectue, dans le cas le plus défavorable, $\log_2 N$ lectures de blocs pour un fichier de taille N blocs. La complexité est $O(\log N)$.

Module d'insertion: (avec éventuellement des décalages intra et inter blocs)

Dans cette solution, on essayera d'abord de réutiliser l'emplacement d'un enreg effacé logiquement, avant de consommer un nouvel espace libre en fin de blocs (ou en fin de fichier).

Inserer(e:Tenreg , nomfich:chaine)

```
var
    trouv, continu, stop, reutilisation : booleen
    i,j,k : entier
    e,x : Tenreg
```

```
DEBUT
   Ouvrir( F,nomfich, 'A')
   // on commence par rechercher la clé e.cle avec le module précédent pour localiser l'emplacement (i,j)
   // où doit être insérer e dans le fichier.
   Rech( e.cle, nomfich, trouv, i, j )
   reutilisation ← faux      // indicateur de réutilisation d'un emplacement effacé logiquement
   SI  ( Non trouv )                  // e doit être inséré dans le bloc i à la position j
         continu ← vrai               // en décalant les enreg j, j+1, j+2, ... vers le bas
                                      // si i est plein, le dernier enreg de i doit être inséré dans i+1
         TQ ( continu )               // si le bloc i+1 est aussi plein son dernier enreg sera
                                      // inséré dans le bloc i+2, etc ... donc une boucle TQ.
               // faire les décalages internes, jusqu'à trouver un enreg effacé ou la fin du bloc ...
               stop ← faux
               TQ ( non stop et j ≤ buf.NB )
                     x ← buf.tab[ j ] ; buf.tab[ j ] ← e
                     SI ( buf.eff[ j ] )
                           stop ← vrai
                     SINON
                           e ← x ; j++
                     FSI
               FTQ
               SI ( stop )          // il y a un enreg effacé dans le bloc i
                     buf.eff[ j ] ← faux          // donc on utilise son emplacement j
                     reutilisation ← vrai
                     EcrireDir( F, i, buf )
                     continu ← faux
               SINON                // il n'y a pas d'enreg effacé dans le bloc i
                     SI ( buf.NB < b )            // mais il y a de la place libre
                           buf.tab[ j ] ← e ; buf.eff[ j ] ← faux ; buf.NB++
                           EcrireDir( F, i, buf )
                           continu ← faux
                     SINON // pas d'enreg effacé et pas de place libre dans le bloc i
                           EcrireDir( F, i, buf )
                           i++ ; j ← 1          // donc décalage vers le bloc i+1 (s'il existe)
                           SI ( i ≤ entete(F,1) )
                                 LireDir( F, i, buf )
                           SINON
                                 continu ← faux
                           FSI
                     FSI
               FSI
         FTQ  // continu
         // si on dépassé la fin de fichier, on rajoute un nouveau bloc contenant un seul enregistrement e
         SI i > entete( F, 1 )
               buf.tab[1] ← e ; buf.eff[1] ← faux ; buf.NB ← 1
               EcrireDir( F, i, buf )       // il suffit d'écrire un nouveau bloc à cet emplacement
               Aff-entete( F, 1, i )        // on sauvegarde le num du dernier bloc dans l'entete 1
         FSI
         // on incrémente le compteur d'insertions uniquement si on n'a pas réutilisé un emplacement effacé
         SI ( non reutilisation) Aff-entete( F, 2 , entete(F,2)+1 ) FSI
   FSI
   Fermer( F )
FIN // insertion
```

L'opération d'insertion peut nécessiter dans le cas le plus défavorable, des décalages inter-blocs qui se propagent jusqu'à la fin du fichier. Comme chaque décalage coûte une lecture et une écriture de bloc (c-a-d 2 accès disques), le coût total d'une insertion, en pire cas, est au voisinage de 2N accès disques pour un fichier formé de N blocs plus le coût de la recherche ($\log_2$ N lectures). La complexité est donc O(N).

La suppression logique consiste à rechercher l'enregistrement et positionner le champs 'effacé' à vrai :

```
Suppression( c:typeqlq; nomfich:chaine )
var
    trouv : booleen
    i,j : entier
DEBUT
   Ouvrir( F,nomfich, 'A')
   // on commence par rechercher la clé c pour localiser l'emplacement (i,j) de l'enreg à supprimer
   Rech( c, nomfich, trouv, i, j )
    // ensuite on supprime logiquement l'enregistrement
   SI  ( trouv )
         // Après Rech(...) buf contient déjà le contenu du bloc i (ce n'est donc pas la peine de le relire une 2e fois)
         buf.eff[j] ← VRAI
         EcrireDir( F, i, buf )
   FSI
   Fermer( F )
FIN // suppression
```

La suppression logique coûte une seule écriture de bloc, en plus du coût de la recherche ($\log_2$ N lectures) pour un fichier de N blocs. La complexité est donc celle de la recherche *O*(log N).

Le chargement initial d'un fichier ordonné consiste à construire un nouveau fichier contenant dès le départ n enregistrements. Ceci afin de laisser un peu de vide dans chaque bloc, qui pourrait être utilisé plus tard par les nouvelles insertions tout en évitant les décalages inter-blocs (très coûteux en accès disque) :

```
Chargement_Initial( nomfich : chaine; n : entier; u : reel )
// u est un réel dans ]0,1] et désigne le taux de chargement voulu au départ
var
    e : Tenreg
    i,j,k : entier
DEBUT
        Ouvrir( F, nomfich, 'N' )   // un nouveau fichier
        i ← 1 ; j ← 1      // i : num de bloc à remplir et j : num d'enreg dans le bloc
        ecrire( 'Donner les enregistrements en ordre croissant suivant la clé : ' )
        POUR k←1 , n
                lire( e )
                SI  ( j ≤ u*b )              // ex: si u=0.5, on remplira les bloc jusqu'à b/2 enreg
                        buf.tab[ j ] ← e ; buf.eff[ j ] ← faux ;  j ← j+1
                SINON                        // j > u*b : buf doit être écrit sur disque
                        buf.NB ← j-1
                        EcrireDir( F, i, buf )
                        buf.tab[1] ← e    // le kème enreg sera placé dans le prochain bloc, à la position 1
                        buf.eff[ 1 ] ← faux
                        i ← i+1 ;  j ← 2
                FSI
        FP
        // à la fin de la boucle, il reste des enreg dans buf qui n'ont pas été sauvegardés sur disque
        buf.NB ← j-1
        EcrireDir( F, i, buf )
        // mettre à jour l'entête (le num du dernier bloc et le compteur d'insertions)
        Aff-entete( F, 1, i )
        Aff-entete( F, 2, n )
        Fermer( F )
FIN // chargement-initial
```

Le chargement initial avec *n* enregistrements, nécessite la création d'un fichier formé de *n / (b*u)* blocs (donc *n / (b*u)* écritures) avec *b* la capacité maximale d'un bloc et *u* le facteur de chargement souhaité (un réel entre 0 et 1).

La réorganisation du fichier consiste à recopier les enregistrements vers un nouveau fichier de telle sorte à ce que les nouveaux blocs contiennent un peu de vide (1-u). Cette opération ressemble au chargement initial sauf que les enregistrements sont lus à partir de l'ancien fichier.

Fusion de 2 fichiers ordonnés (TOF)

On parcourt les 2 fichiers (*F1* et *F2*) ensemble avec 2 buffers (*buf1* et *buf2*) et on rempli un 3e buffer (*buf3*) pour construire un 3e fichier (*F3*) en ordre croissant.
Les déclarations sont celles utilisées dans les fichier TOF standards.

```
Fusion (nom1,nom2, nom3: chaine)
var     F1 : Fichier de Tbloc Buffer buf1 Entete( entier, entier)
        F2 : Fichier de Tbloc Buffer buf2 Entete( entier, entier)
        F3 : Fichier de Tbloc Buffer buf3 Entete( entier, entier)
        i1, i2, i3, j1, j2, j3, i, j, indic :  entier
        continu : booleen
        e, e1, e2 : Tenreg
        buf : Tbloc

Debut
  Ouvrir(F1, nom1, 'A' ) ; Ouvrir(F2, nom2, 'A' ) ; Ouvrir(F3, nom3, 'N' )
  i1 ← 1; i2 ← 1; i3 ← 1      // les num de blocs de F1, F2 et F3
  j1 ← 1; j2 ← 1; j3 ← 1      // les num d'enreg dans buf1, buf2 et buf3
  LireDir(F1, 1, buf1) ; LireDir(F2, 1, buf2) ;
  continu ← vrai

  TQ ( continu )              // tant que non fin de fichier dans F1 et F2 faire
     SI ( j1 ≤ buf1.NB  et  j2 ≤ buf2.NB )
        // choisir le plus petit enreg, dans buf1 et buf2
        e1 ← buf1.tab[ j1 ] ; e2 ← buf2.tab[ j2 ]
        SI ( e1.cle ≤ e2.cle ) e ← e1;   j1 ← j1 + 1 SINON   e ← e2;   j2 ← j2 + 1 FSI
        // et le mettre dans buf3
        SI ( j3 ≤ b )
              buf3.tab[ j3 ] ← e;   j3 ← j3 + 1
        SINON
              buf3.NB ← j3 – 1 ; EcrireDir(F3, i3, buf3 ); i3 ← i3 + 1 ; buf3.tab[1] ← e ; j3 ← 2
        FSI
     SINON      // c-a-d : non ( j1 ≤ buf1.NB  et  j2 ≤ buf2.NB )
        // si tous les enreg d'un des blocs (buf1 ou buf2) ont été traités, passer au prochain
        SI ( j1 > buf1.NB )
              SI ( i1 < entete(F1, 1) )
                 i1 ← i1 + 1 ; LireDir( F1, i1, buf1 ) ; j1 ← 1
              SINON // ( donc i1 ≥ entete(F1, 1) )
                 continu ← faux ; N ← entete(F2,1) ; buf ← buf2 ; Indic ← 2 ; i ← i2 ; j ← j2   // pour le 2e TQ
              FSI  // ( i1 < entete(F1, 1) )
        SINON // c-a-d ( j2 > buf2.NB )
              SI ( i2 < entete(F2, 1) )
                 i2 ← i2 + 1 ; LireDir( F2, i2, buf2 ); j2 ← 1
              SINON // ( donc i2 ≥ entete(F2, 1) )
                 continu ← faux ; N ← entete(F1,1) ; buf ← buf1 ; Indic ← 1 ;  i ← i1 ; j ← j1   // pour le 2e TQ
              FSI // ( i2 < entete(F2, 1) )
        FSI // ( j1 > buf1.NB )
     FSI  // ( j1 ≤ buf1.NB  et  j2 ≤ buf2.NB )

  FTQ  // continu
```

```
// Le 2e TQ pour continuer à recopier les enregistrements d'un seul fichier (i,j,buf) dans F3
continu ← vrai;
TQ ( continu )              // tant que non fin de fichier dans F1 ou F2 faire
    SI ( j ≤ buf.NB )
        SI ( j3 ≤ b )
                buf3.tab[ j3 ] ← buf.tab[ j ];   j3 ← j3 + 1
        SINON
                buf3.NB ← j3 – 1
                EcrireDir(F3, i3, buf3 )
                i3 ← i3 + 1
                buf3.tab[1] ← buf.tab[ j ]
                j3 ← 2
        FSI  // ( j3 ≤ b )
        j ← j + 1

    SINON // c-a-d non ( j ≤ buf.NB )
        SI ( i ≤ N )
                i ← i + 1
                SI ( Indic = 1 )
                        LireDir( F1, i, buf )
                SINON
                        LireDir( F2, i, buf )
                FSI
                j ← 1
        SINON
                continu ← faux
        FSI

    FSI // ( j ≤ buf.NB )

FTQ

// Le dernier buffer (buf3) n'a pas encore été ecrit sur disque ...
buf3.NB ← j3 – 1
EcrireDir( F3 , i3, buf3 )
Aff-entete( F3, 1, i3)                          // le nombre de blocs dand F3
Aff-entete( F3, 2, entete(F1,1) + entete(F2,1) )    // le nombre d'enregistrements dans F3

Fermer( F1 ) ;  Fermer( F2 ) ;  Fermer( F3 )
```

Fin

L'opération de fusion de 2 fichiers F_1 et F_2 de tailles respectives N_1 et N_2 blocs, nécessite le parcours complet de 2 fichiers (soit $N_1 + N_2$ lectures) et la création du fichier résultat (F_3) formé par $N_1 + N_2$ blocs (soit $N_1 + N_2$ écritures), en supposant un même facteur de chargement pour les trois fichiers. Le coût total de la fusion est donc $2N_1 + 2N_2$ accès disques.

Si les fichiers F_1 et F_2 était de même taille (N blocs chacun), le coût total de la fusion serait alors de 4N accès disques (2N lectures et 2N écritures).
La complexité de la fusion est donc en $O(N)$.

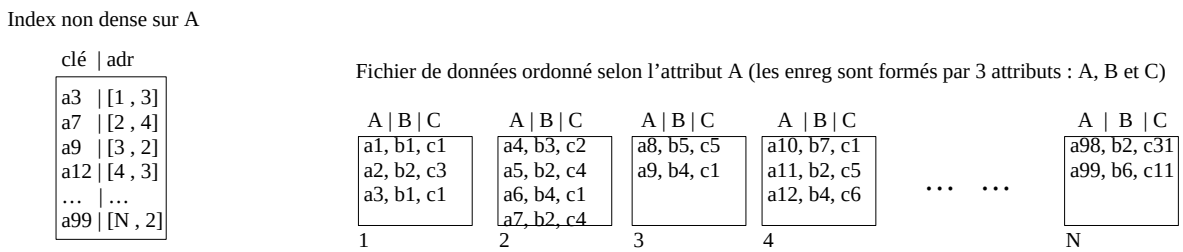
Chapitre 3 Fichiers avec index

1) Introduction

Avec les structures de fichiers séquentiels, quand le fichier de données devient volumineux, les opérations d'accès (recherche, insertion, ...) deviennent inefficaces.

Les méthodes d'index permettent d'améliorer, dans une certaine mesure, les performances en gérant une structure auxiliaire (table d'index) accélérant la recherche d'enregistrements.

Un index est (généralement) une table ordonnée en mémoire centrale (**MC**) contenant principalement des couples < *clé* , *adr* >, utilisée pour accélérer la recherche des enregistrements d'un fichier. Le champs '*clé*' représente un attribut (ou groupe d'attributs) d'un enregistrements du fichier et utilisé(s) pour la recherche. On appelle ce genre d'attribut une clé de recherche (*Search Key*). Cela peut être n'importe quel attribut, à valeurs uniques ou non. Le champs '*adr*' représente les informations de localisation d'un enregistrement dans le fichier (par exemple : <*numBloc*, *deplacement*>). On peut aussi mettre dans l'index d'autres informations qui pourraient être utiles pour la gestion du fichier, comme par exemple un booléen indiquant les effacements logiques, la taille occupé par l'enregistrement (surtout dans le cas avec 'format variable'), … etc.



Un index est dit « dense » (Non Clustered Index) s'il contient toutes les valeurs de l'attribut clé du fichier de données. Dans ce cas, on n'a pas besoin de garder le fichier ordonné sur cet attribut.

Un index est dit « non dense » (Clustered Index) s'il ne contient pas toutes les valeurs de l'attribut clé du fichier de données (par exemple, on ne garde dans l'index qu'une seule valeur par bloc ou groupe de blocs). Dans ce cas le fichier doit être ordonné sur l'attribut indexé. L'avantage d'un index non dense est sa taille (plus petite que celle d'un index dense pour le même fichier de données). Voir schéma ci-dessus.

Dans le fichier de données, les valeurs de l'attribut indexé (la clé de recherche) peuvent être uniques (toutes les valeurs sont distinctes), ou alors multiples (non uniques, c-a-d qu'il existe des enregistrements différents pouvant avoir la même valeur pour l'attribut indexé). Dans ce dernier cas, à chaque entrée dans la table d'index est associé une liste d'adresses. On peut aussi dupliquer la

valeur de l'attribut clé, dans la table d'index, en autant d'entrées que de valeurs multiples, chacune avec son adresse associée. Le schéma ci-dessous montre les deux possibilités de représentation des valeurs multiples (dans cet exemple la clé ***v*** se répète dans trois enregistrements localisés aux adresses ***a1***, ***a2*** et ***a3*** du fichier de données).

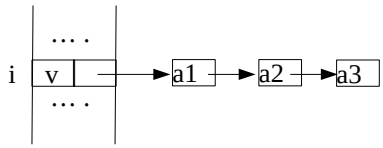


Table d'index avec une seule entrée par valeur multiple

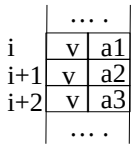


Table d'index avec plusieurs entrées par valeur multiple

2) Méthode avec index à un niveau

Pour accélérer les opérations d'accès sur un fichier de données ***F***, on peut maintenir en mémoire centrale (**MC**) une table d'index portant sur l'attribut (champs) utilisé comme clé de recherche (l'un des attributs des enregistrements de ***F***). S'il existe plusieurs attributs (*A1, A2, A3,* ...) pouvant être utilisés comme des clés de recherche, on peut aussi maintenir plusieurs index différents : *IndexA1, IndexA2, IndexA3,* … (un sur chaque attribut clé).

Si de plus le fichier ***F*** est ordonné selon un des attributs clés (par exemple sur *A1*), l'index associé (*IndexA1*) sera dit « primaire » ou alors « **Clustering Index** » et les autres (*IndexA2, IndexA3,* …) seront « secondaires » ou bien « **Non clustering indexes** ». Dans ce cas (***F*** est ordonné selon l'attribut *A1*), *IndexA1* sera donc non dense alors que *IndexA2, IndexA3,* … seront des index denses.

Il est à rappeler que dans le cas d'un fichier ordonné, l'opération d'insertion peut être coûteuse à cause des décalages inter-blocs générés. La suppression est généralement logique pour éviter les décalages inverses. Des réorganisations périodiques sont aussi à prévoir. On peut aussi maintenir une zone de débordement pour le fichier de données afin d'éviter les décalages inter-bloc lors des insertions, mais avec le nombre des insertions qui augmente dans le temps, les performances de la recherche se dégradent, malgré la présence de l'index, à cause du parcours séquentiel dans la zone de débordement, nécessitant donc aussi des réorganisations périodiques.

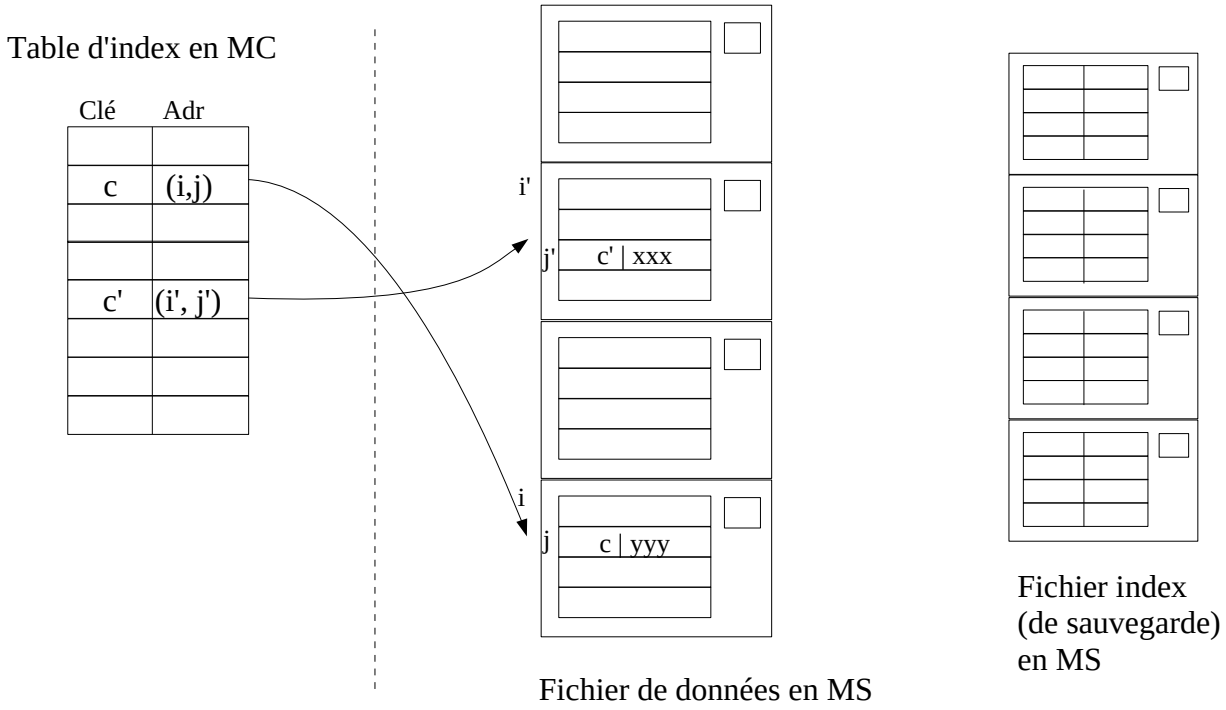
- Fichier d'index

Pour ne pas avoir à reconstruire la table d'index à chaque démarrage, on peut sauvegarder la table dans un fichier (dit « **fichier index** ») en fin de traitement par exemple. Celui-ci sera rechargé en **MC** dans une table lors de la prochaine session. Généralement le chargement de la table d'index depuis un fichier index est beaucoup plus rapide que de reconstruire la table à partir du fichier de données (surtout si ce dernier n'est pas ordonné).

Il est à noter que les opérations d'accès au fichier de données (recherche, insertion, suppression …) utilisent uniquement la table d'index en **MC** et le fichier de données en **MS**. Le fichier d'index en **MS** n'est ni consulté, ni mis-à-jour durant ces opérations.

Les fichiers de données et d'index peuvent être de n'importe quelle structure (blocs contigus, blocs chaînés, ...). De même que les enregistrements peuvent être à format fixe ou variable (avec ou sans chevauchement).

La figure ci-dessous montre les différentes structures utilisées dans un exemple de méthode d'accès avec un seul index.



Les opérations de base

a) Cas d'un fichier de données non ordonné (<u>index dense</u>) :

- La recherche d'un enregistrement consiste à faire une recherche dichotomique de sa clé dans la table d'index en **MC**, si elle existe, on récupère l'enregistrement à partir du fichier de donnée avec un seul accès disque.
=> coût de l'opération : 1 accès disque (ou moins, si la clé n'existe pas dans l'index).

- La requête à intervalle consiste à rechercher tous les enregistrements dont la valeur du champs clé appartient à un intervalle de valeurs donné : [***a,b***].

1. On commence par rechercher la plus petite clé ≥ ***a*** dans l'index (recherche dichotomique dans la table en **MC**).
2. Puis on continue séquentiellement dans la table (en **MC**) jusqu'à trouver une clé > ***b***.
3. Pour chaque clé, on accède au fichier de données pour récupérer le (ou les) enregistrement(s) associés. Ce dernier point peut être amélioré si on tri les numéros de blocs trouvés avant d'accéder au fichier de données (afin de ne pas lire, éventuellement, deux fois le même bloc).

=> coût de l'opération en nombre d'accès : le nombre d'enregistrements vérifiant la requête (dans le cas le plus défavorable).

- L'insertion d'un nouvel enregistrement se fait au niveau du fichier de données selon sa structure et organisation, ensuite la valeur de son attribut indexé (la clé de recherche) est utilisée pour mettre à jour la table d'index (en **MC**). Si la valeur de la nouvelle clé n'existe pas dans la table, une nouvelle entrée est alors ajoutée à la table (avec les décalages nécessaires pour maintenir l'ordre des valeurs de clé dans la table). Si par contre, la clé existe déjà dans la table (cas des valeurs multiples), la nouvelle localisation de l'enregistrement dans le fichier de données est alors rajoutée à la liste des adresses associée.

=> coût de l'opération : le nombre d'accès disques nécessaires pour l'insertion de l'enregistrement dans le fichier de donnée non ordonné (la m-à-j de la table d'index ne coûte aucun accès disque).

- La suppression s'effectue, éventuellement, au niveau du fichier de données (suppression physique ou logique), ensuite la table d'index est mise à jours soit en supprimant (par décalages) l'entrée associée à la valeur de la clé (lorsque la clé est à valeurs uniques), soit en supprimant un élément de la liste des adresses (lorsque la clé est valeurs multiples). Dans ce dernier cas, si la liste des adresses devient vide, l'entrée associée est aussi supprimée de la table par décalages.
On peut aussi opter pour positionner uniquement un indicateur d'effacement au niveau de la table d'index pour ne pas avoir à effectuer les décalages en MC.
=> coût : le nombre d'accès disques nécessaires pour la suppression de l'enregistrement dans le fichier de donnée non ordonné (la m-à-j de la table d'index ne coûte aucun accès disque).

b) Cas d'un fichier de données ordonné (donc généralement un <u>index non dense</u>) :

Si le fichier de données est ordonné, on peut garder dans la table d'index que certaines valeurs de clé. Par exemple une clé par bloc (ou groupe de blocs), on choisit soit la plus grande ou la plus petite de chaque groupe. C'est alors un index non dense (car il ne contient pas toutes les clés). Cela permet de diminuer considérablement la taille de la table d'index tout en gardant pratiquement les mêmes performances de recherche qu'une méthode d'index avec fichier non ordonné (si le nombre de blocs par groupe est petit, par exemple un bloc par groupe).

L'insertion risque par contre d'être coûteuse lorsque le fichier est formé par des blocs contigus (*vu comme tableau*), à cause des décalages dans le fichier de données s'il n'y a pas de zone de débordement.

- La recherche d'un enregistrement consiste à faire une recherche dichotomique dans la table d'index, puis on continue séquentiellement dans un des groupes de blocs du fichier de données.
=> coût de l'opération : 1 accès disque (si chaque groupe est formé d'un seul bloc) .

- Pour la requête à intervalle (clé de recherche ∈ [***a,b***]), on procède comme suit :
1- On commence par rechercher la plus petite clé ≥ ***a*** dans l'index (recherche dichotomique en **MC**).

2- Puis on accède au fichier de données à partir du numéro de bloc spécifié dans l'index et on continue séquentiellement dans le fichier jusqu'à trouver une clé > ***b***.
=> coût de l'opération en nombre d'accès se rapproche du nombre de clés vérifiant la requête divisé par la capacité moyenne d'un bloc (c'est un cas favorable).

- L'insertion d'un nouvel enregistrement se fait dans le fichier de données ou alors en zone de débordement (voir plus bas un exemple) en utilisant l'algorithme associé à la structure du fichier (vu comme tableau ou comme liste, format fixe ou variable ...) . La table d'index (en MC) est alors mise à jour à chaque fois que le représentant d'un groupe a changé suite aux déplacements d'enregistrements (les décalages, les éclatement ...) occasionnés par l'insertion du nouvel enregistrement dans le fichier.
=> coût de l'opération : celui de l'insertion dans le fichier de données associé

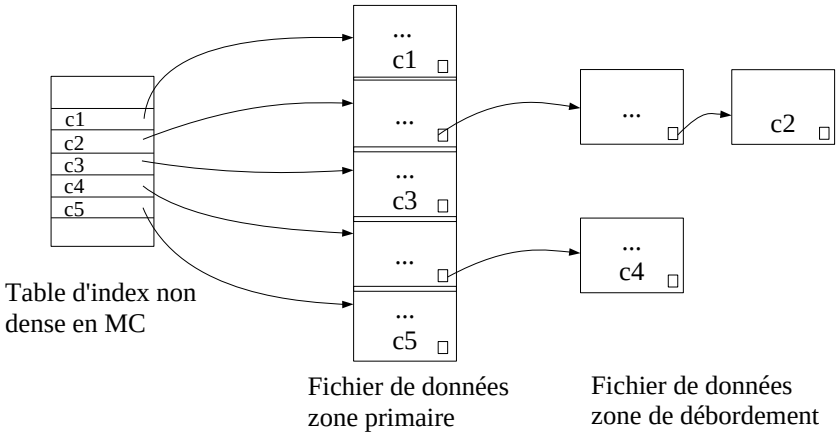
- La suppression est généralement logique dans le fichier de données, sauf dans le cas d'un fichier vu comme liste et sans chevauchement inter-blocs auquel cas la suppression physique devient envisageable (car peu coûteuse). La table d'index est rarement mise-à-jour.
=> coût : celui de la suppression dans le fichier de données associé

Gestion d'une <u>zone de débordement</u>

Dans le cas d'un fichier ordonné vu comme tableau, une alternatives aux décalages inter-blocs lors des insertions (et des suppressions physiques), serait de maintenir une zone de débordement pour y stocker, de manière simple, les enregistrements expulsés des blocs pleins du fichier de données.

A chaque insertion, si le bloc du fichier de données est plein, les enregistrements en surplus, seront insérés dans des blocs de débordements, chaînés au bloc principal pour maintenir l'ordre des clés. Au niveau de la table d'index, on met à jour la clé maximale (le représentant du groupe : la première ou la dernière clé du groupe) si elle vient à changer suite à l'insertion.

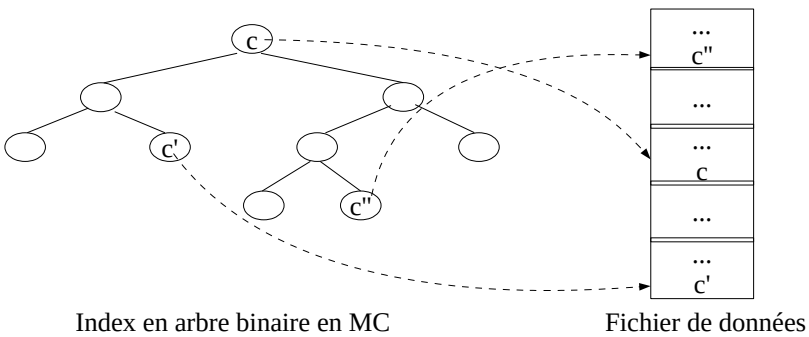
Les performances reste bonnes tant que la longueur des listes de blocs en débordement reste petite. Lorsque les groupes sont formés d'un seul bloc chacun, les coûts des opérations sont pratiquement les mêmes que dans le cas d'un fichier non ordonné avec index, plus un surcoût égal à la longueur moyenne des listes en débordement.



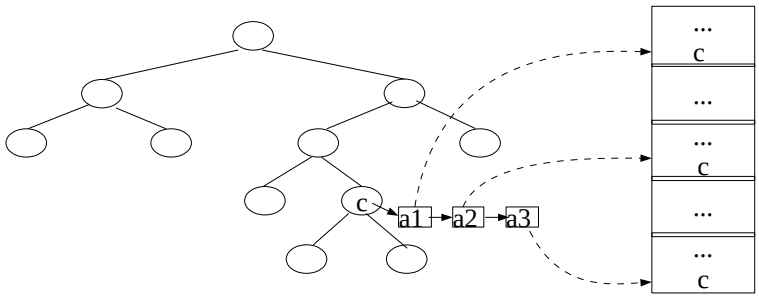
Si les blocs en débordement deviennent trop nombreux (les listes s'allongent), il faut réorganiser le fichier en créant un nouveau fichier principal, éventuellement plus grand s'il y a eu beaucoup d'insertions et peu de suppressions, ou alors plus petit dans le cas inverse. La nouvelle zone de débordement sera initialement vide.

Autre représentation de l'index en MC

On peut aussi représenter l'index en mémoire centrale sous forme d'un arbre de recherche binaire (*AVL, Red-Black,* …), au lieu d'une table ordonnée. L'avantage est d'éviter les décalages (en mémoire centrale) lors de l'insertion de nouvelles clés ou la suppression de clés dans l'index. L'inconvénient est l'encombrement mémoire plus important à cause des pointeurs (*fg, fd,*…) associés à chaque nœuds de l'arbre.



Pour la représentation des valeurs multiples des clés de recherche, on adapte assez facilement la structures des arbres binaire de recherche pour supporter les valeurs de clés identiques :
- Soit chaque nœud de l'arbre pointe une liste d'adresses d'enregistrements du fichier de données ayant comme valeur de clé l'information du nœud. ,Voir exemple dans la figure ci-dessous.



La valeur c apparaît dans 3 enregistrements

Fichier de données

- Soit relaxer la contrainte du maintient de l'ordre de l'arbre binaire : < en ≤ (ou respectivement : > en ≥) pour accepter les valeurs en double en les plaçant à gauche (respectivement à droite) du nœud courant.

Index de grande taille

Si l'index est trop grand pour résider en mémoire centrale (**MC**), on peut travailler directement sur le fichier index à la place de la table d'index. Dans ce cas le fichier index qui est de type TOF ou TOV$\overline{C}$ (pour pouvoir faire des recherches dichotomiques), est utilisé comme structure auxiliaire permettant d'accélérer les opérations d'accès au fichier de données (il n'y a plus de table d'index en **MC** dans ce cas particulier).

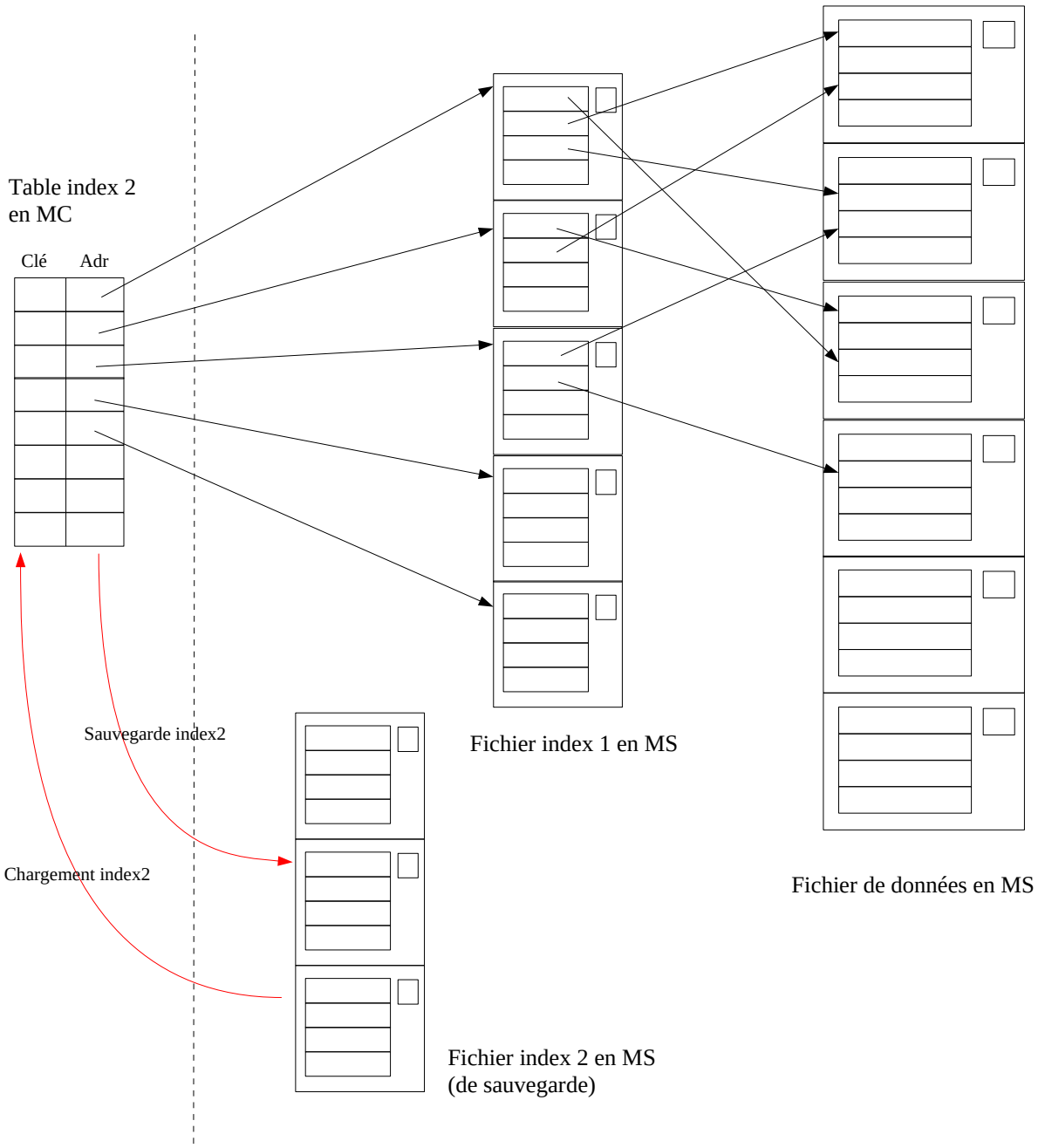
Par exemple une recherche d'enregistrement de clé ***v***, commence d'abord par une recherche de ***v*** dans le fichier index (par dichotomie) afin de récupérer les informations de localisation (numéros de blocs, déplacement, indicateur d'effacement logique, taille ...etc). Ensuite l'accès au fichier de données est effectuer pour récupérer le (ou les) enregistrement(s) associé(s). Les autres opérations (insertion, suppression, …) peuvent aussi être définies en utilisant le fichier index au lieu de la table d'index en **MC**.

3) Index multi-niveaux

Si le fichier index est assez volumineux, même la recherche dichotomique nécessitera un nombre de lectures de blocs relativement élevé (par exemple de l'ordre des dizaines d'accès disques pour des fichiers index de plus de mille blocs). On peut diminuer ce nombre en construisant un deuxième index sur le fichier index (ordonné). Dans ce cas, on choisira une seule clé pour chaque groupe de blocs du fichier index (index non dense) pour construire le deuxième index. On obtient ainsi un index à 2 niveaux.

Si le deuxième index est encore trop grand pour résider en **MC**, on le stocke sur disque (deuxième fichier index) et on construit un troisième index en choisissant une clé par groupe de blocs du deuxième fichier index. On peut répéter ce procédé jusqu'à obtenir un index pouvant résider en mémoire centrale (index à n niveaux).

La figure ci-dessous, montre les différents composants d'un index à 2 niveaux:



4) Accès multiclés

Pour améliorer les recherches basées sur plusieurs clé de recherche en même temps (requêtes multi-clés), on peut utiliser et combiner plusieurs index (un sur chaque attribut utilisé).

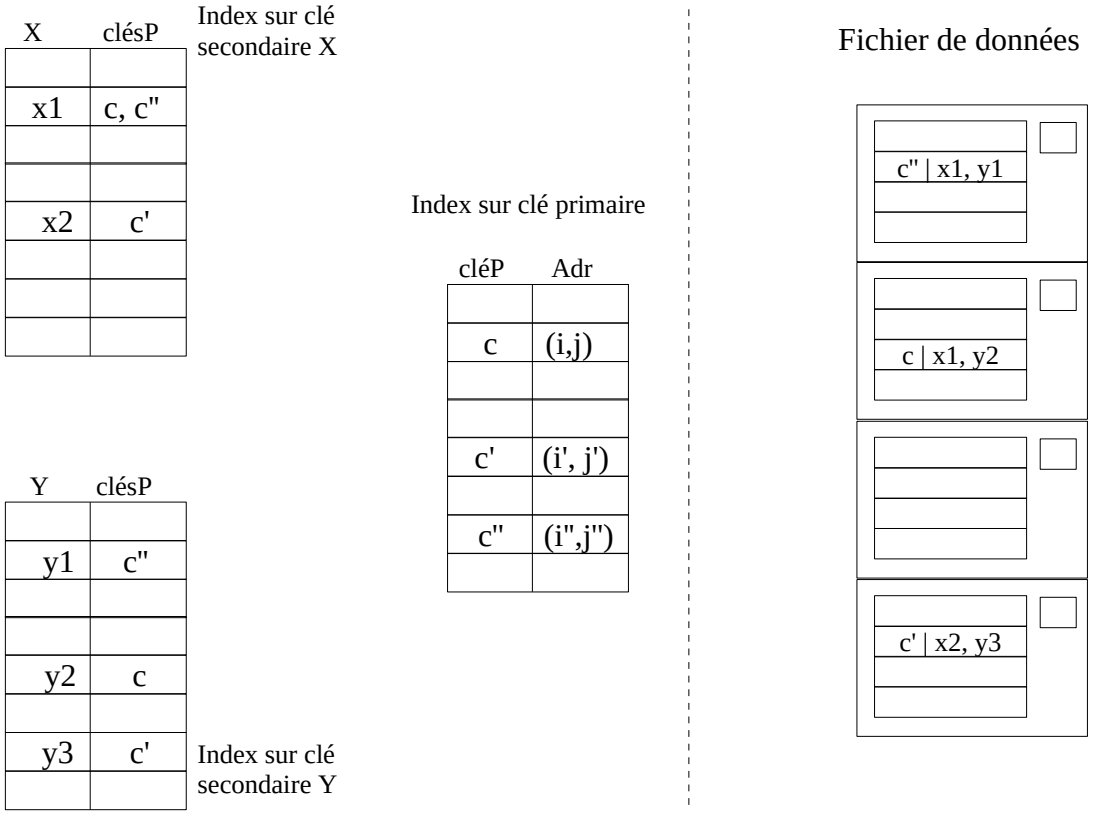
Une requête multi-clés est de la forme :
« trouver les enregistrement /^{que} P soit vrai »
avec P un prédicat de type conjonction ou disjonction de conditions élémentaire : $C_1 \curlywedge\!\curlyvee C_2 \ldots C_n$
chaque condition élémentaire C_k porte sur un seul attribut du fichier de données.
Par exemple si on dispose de 2 index : *IndA1* et *IndA2*, construits respectivement sur les attributs

A1 et *A2* du fichier de données, on peut localiser rapidement tous les enregistrements vérifiant une conditions de la forme $A1=v1 \wedge A2=v2$ (c'est une requête multi-clés portant sur *A1* et *A2*). Les étapes peuvent être résumées comme suit :

1. On recherche *v1* dans l'index *IndA1* produisant comme résultat une liste *L1* (éventuellement vide) d'adresses d'enregistrements.
2. On recherche *v2* dans l'index *IndA2* produisant comme résultat une liste *L2* (éventuellement vide) d'adresses d'enregistrements.
3. On effectue l'intersection des deux listes $L = L1 \cap L2$.
4. On trie *L* sur le numéro de bloc et on accède au fichier de données pour récupérer les enregistrements cherchés.

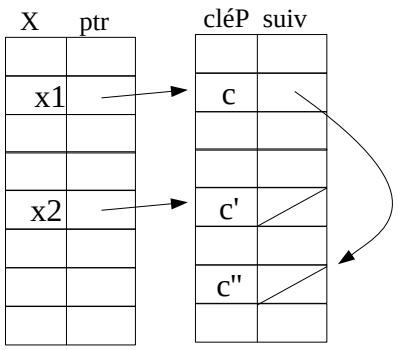
Cet algorithme (les 4 étapes ci-dessus) n'est pas forcément le meilleur (d'un point de vue performance). Tout dépend en fait du nombre d'enregistrements vérifiant la condition de la requête (c'est ce qui est appelé la sélectivité du prédicat *P*). Si ce nombre est important, un parcours séquentiel du fichier de données sera peut être plus avantageux que d'accéder par les index. On peut estimer à l'avance la sélectivité d'une requête si nous disposons de statistiques sur les valeurs des attributs dans le fichier de données. Par exemple lors des insertions/suppression on peut mettre à jours ce genre de statistiques.

Un cas particulier (et assez fréquent aussi) de l'utilisation de plusieurs index en même temps, est la méthode des **listes inversées**. Cette méthode est applicable lorsque l'un des attributs indexés est à valeurs uniques (en bases de données en parle alors de clé primaire).

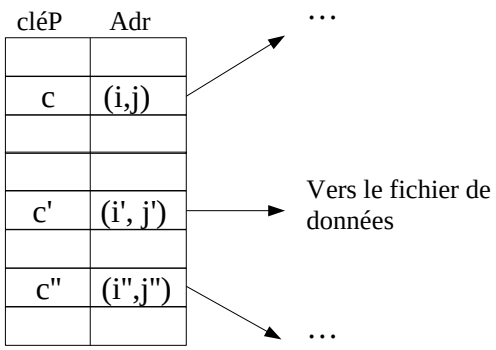


Par abus de langage, on désigne souvent l'index construit sur cet attribut comme étant « primaire » (même si le fichier de données n'est pas ordonné) et les autres index (sur les autres attributs) seront dits « secondaires » (cette terminologie : 'index primaire et index secondaire' est légèrement différente de celle utilisées généralement dans les structures de fichiers au niveau physique, comme cela a été présenté en début de chapitre et peut donc entraîner une certaine confusion chez le lecteur non averti). Dans la figure précédente, on dispose d'un index sur la clé primaire *P* (à valeurs uniques) et de deux index sur les clés secondaires *X* et *Y* (pouvant être à valeurs multiples).

Le problème avec les clés secondaires, est qu'il peut exister plusieurs enregistrements pour une même valeur du champ indexé. On implémente généralement cette multiplicité à travers des <u>listes de clés primaires</u> (au lieu de listes d'adresses).



Index secondaire avec ses listes de clés primaires



Index sur clé primaire

Quand on recherche les enregistrements suivant une clé secondaire (par exemple *X*=***a***), on utilise l'index sur la clé secondaire *X* pour récupérer la ou les clés primaires associées à la valeur cherchée (***a***). Pour chaque clé primaire trouvée, on utilise l'index sur la clé primaire *P* pour localiser l'enregistrement sur le fichier de données (numéro de bloc et déplacement).
Dans cette méthode (listes inversées) la recherche commence sur l'index de clé secondaire, puis passe par l'index de clé primaire avant d'atteindre le fichier de données.

Voici les étapes d'une requête multi-clés avec la méthode des listes inversées :
« trouver tous les enregistrements dont la valeur de *X* = ***vx*** *ET la valeur de Y* = ***vy*** *ET ...* »
avec *X, Y, ...* des clés secondaires.

1. En utilisant l'index secondaire *X*, trouver la liste **Lx** de <u>clés primaires</u> associées à la valeur de *X* (***vx***).
2. Refaire la même action pour chaque clé secondaire mentionnée dans la requête...
3. Faire <u>l'intersection des listes de clés primaires</u> **Lx**, **Ly**, ... pour trouver les <u>clés primaires</u> associées avec chaque valeur de clé secondaire mentionnée dans la requête.
4. Utiliser alors l'index primaire pour retrouver les enregistrements du fichier de données (en triant d'abord la séquence des numéros de blocs avant d'effectuer les transferts physiques)

Si la requête initiale était de forme disjonctive (c-a-d avec une condition de la forme « *X* = ***vx*** *OU Y* = ***vy*** *OU ...* »), on aurait utilisé l'opération d'union des listes *Lx, Ly, …* au lieu de l'intersection à l'étape 3 du pseudo algorithme précédent.

Pour insérer un enregistrement <***c,vx,vy,...***> avec **c** sa clé primaire et **vx**,**vy**,... ses clés secondaires, on procède comme suit :

1. Rechercher **c** dans l'index primaire pour vérifier qu'elle n'existe pas déjà et pour trouver l'indice *ip* où doit être insérée cette clé (recherche dichotomique).
2. Insérer l'enregistrement dans le fichier de données. Soit (***i,j***) son adresse. Si le fichier de données est ordonné, l'index sera probablement non dense. Soit ***c*** la clé du dernier enregistrement du groupe où à lieu l'insertion et soit (***i,j***) son adresse.
3. Dans le cas d'un index dense (par exemple un fichier non ordonné), insérer le couple <**c**, (***i,j***)> dans la table d'index primaire, à l'indice *ip* (en procédant par décalages).
   Dans le cas d'un index non dense (par exemple un fichier ordonné), il faut mettre à jour l'entrée *ip* de la table d'index pour qu'elle contienne <***c, (i,j)***> si l'ancienne valeur est différente.
4. Rechercher la valeur **vx** dans l'index secondaire *X*,
   si **vx** existe, rajouter **c** à la liste pointée par **vx**
   si **vx** n'existe pas, insérer **vx** (par décalages) dans la table *X*.
   La nouvelle entrée **vx**, pointera une liste formé par une seule clé primaire (***c***).
5. Refaire l'étape 4) pour chaque clé secondaire restante (**vy**, ...).

Pour supprimer logiquement un enregistrement de clé primaire ***c***, (c'est par exemple le cas d'un fichier ordonné avec blocs contigus ou bien un fichier avec format variable et chevauchement), il suffit de positionner un bit (ou caractère) d'effacement au niveau du fichier de données ou de la table d'index primaire, pour l'entrée ***c***.

Pour supprimer physiquement un enregistrement de clé primaire ***c***, (c'est par exemple le cas d'un fichier ordonné avec blocs chaînés et format fixe ou variable sans chevauchement ou bien alors un fichier non ordonné avec format fixe ou variable sans chevauchement), il faut d'abord supprimer physiquement l'enregistrement dans le fichier de données. Ensuite il faut mettre à jour la table d'index primaire soit en supprimant l'entrée relative à ***c*** (cas d'un index dense) ou alors modifier la clé et/ou l'adresse du représentant du groupe auquel appartient l'enregistrement supprimé (cas d'un index non dense).

Dans les deux types de suppression (logique ou physique), il n'est pas nécessaire de mettre à jour les autres index (associés aux clés secondaires).

Les index Bitmaps

Toujours pour l'accès multi-clés, lorsque le nombre de valeurs distinctes d'une clé de recherche est petit, l'utilisation d'une table d'index classique a peu d'avantage (une table contenant un nombre très petit d'entrées, et pour chaque entrée, on aura de très longues listes d'adresses à cause du grand nombre de valeurs qui se répètent).

Par contre on peut construire à la place d'un index classique (table ordonnée), un ensemble de chaînes de bits (appelées '**bitmap**'), à raison d'une chaîne par valeur d'attribut. La longueur de ces chaînes est égale au nombre d'enregistrements dans le fichier de données. Ainsi un index bitmap sur un attribut *A* ayant ***K*** valeurs distinctes (***v1, v2, ...vk***) sera composé de ***K*** chaînes de bits (*IndA_v1*, *IndA_v2*, … *IndA_vk*, une par valeur d'attribut) que l'on peut charger en mémoire indépendamment les unes des autres selon le besoin.

Le $i^{ième}$ bit de la chaîne bitmap associée à la valeur ***V*** de l'attribut indexé sera à 1 si le $i^{ième}$ enregistrement du fichier de données porte la valeur ***V*** dans l'attribut indexé. Sinon le bit associé sera à 0.

Il est impératif d'avoir un moyen rapide d'accéder aux enregistrements du fichier de données à partir de leur numéro d'ordre. Par exemple pour un fichier de type TOF, il facile de calculer le numéro de bloc et le déplacement dans le bloc pour localiser le $i^{ième}$ enregistrement :

[num_bloc = ((i-1) div b)+1] et [depl = i mod b ; SI (depl = 0) depl = b FSI]
b étant la capacité maximale d'un bloc.

S'il n'existe pas un tel moyen rapide pour localiser les enregistrements à partir de leur numéro d'ordre, on peut maintenir, en plus, un index classique (table ou arbre) associant chaque numéro d'enregistrement avec son adresse dans le fichier de données.

Exemple d'utilisation des index bitmap
Soit un fichier ***F*** contenant ***N*** enregistrements, et soient *A* et *B* deux attributs de ***F***.
Supposons que *A* peut avoir uniquement 3 valeurs possibles : ***v1***, ***v2*** ou ***v3***
(la cardinalité du domaine de *A* est donc 3).

Un index bitmap sur *A* est donc composé de 3 chaînes de ***N*** bits chacune (*IndA_v1*, *IndA_v2* et *IndA_v3*) :

```
        n° enr  :1  2 3 4  5  6 7 8 …   i   …   … N
                |--|--|--|--|--|--|--|--|----|--|----|--|--|--|
IndA_v1 :        1 0  0  0 1 1  0 1 ….  0  … 1  1  0   (N bits)   → la chaîne de bits associée à v1
IndA_v2 :        0 1  0  0 0 0  0 0 ….  1  … 0  0  0   (N bits)   → la chaîne de bits associée à v2
IndA_v3 :        0 0  1  1 0 0  1 0 ….  0  … 0  0  1   (N bits)   → la chaîne de bits associée à v3
                          (ex : la valeur de A dans l'enreg n°2  et n°i vaut v2)
```

De même supposons que pour l'attribut *B* les seules valeurs possibles sont ***w1***, ***w2***, ***w3*** ou ***w4*** (la cardinalité du domaine de *B* est donc 4).

Un index bitmap sur *B* est donc composé de 4 chaînes de ***N*** bits chacune (*IndB_w1*, *IndB_w2*, *IndB_w3* et *IndB_w4*) :

```
        n° enr : 1  2 3 4  5  6 7 8 …   i   …   … N
                 |--|--|--|--|--|--|--|--|----|--|----|--|--|--|
IndB_w1 :        0 0  1  0 0 0  0 1 ….  0  … 1  0  0   (N bits)
IndB_w2 :        1 1  0  0 0 1  0 0 ….  0  … 0  0  1   (N bits)
IndB_w3 :        0 0  0  1 1 0  0 0 ….  0  … 0  1  0   (N bits)
IndB_w4 :        0 0  1  0 0 0  1 0 ….  1  … 0  0  0   (N bits)
                          (ex : la valeur de B dans l'enreg n°3 , n°7 et n°i vaut w4)
```

Dans ce cas pour faire la recherche multi-clés suivante :
« trouver les enregistrements de ***F*** /^{que} *A=**v2** ET B=**w4*** »

On peut charger les 2 chaînes *IndA_v2* et *IndB_w4*, pour faire leur intersection (bit à bit). Le résultat est une chaîne de bit dont seules les positions relatives aux enregistrements vérifiant la condition [*A=**v2** ET B=**w4***] seront à 1. Dans cet exemple, l'enregistrement n°***i*** sera dans le résultat.

Les index bitmap sont très efficaces pour les requêtes multi-clés portant sur des attributs à faibles cardinalités. Ils sont aussi utilisés pour compter le nombre d'enregistrements vérifiant certaines conditions. Les différentes opérations élémentaires (le *ET*, le *OU*, le *NON* ...) sur les bitmaps sont effectuées sur des mots mémoire regroupant plusieurs bits à la fois (ex :8 bits, 16 bits, … 64 bits), ce qui diminue considérablement le nombre d'itérations nécessaires pour la résolution des requêtes. De plus même si une chaîne bitmap peut éventuellement être de grande taille, il est tout à fait envisageable de réaliser le traitement des requêtes en chargeant les différents bitmap, partie par

partie selon l’espace mémoire disponible.

Le principal inconvénient des index bitmap, est qu’en cas d’insertions ou de suppressions fréquentes ou alors en cas de simples mises-à-jours des enregistrements existants dans le fichier de données (par exemple modifier la valeur d’un attribut d’un enregistrement donné), les anciennes chaînes de bits doivent être corrigées ou même régénérées complètement dans certains cas, ce qui posent des problèmes de performance assez délicats. Les index bitmap sont de ce fait, utilisés principalement qu’avec des fichiers statiques (entrepôt de données, bases de données en lectures seules, …).

Structures de Fichiers

Chapitre 4 Les Arbres en Mémoire Secondaire (Fichiers arborescents)

1) Introduction

Les méthodes par tables d'index sont limitées à certains type de fichiers (petits fichiers ou fichiers statiques). Les méthodes basées sur les structures d'arbres sont mieux adaptées aux fichier volumineux et/ou dynamiques.

Afin de mieux occuper l'espace des blocs, on utilise des arbres m-aire où chaque nœud est représenté dans un bloc d'E/S.

2) Les Arbres de Recherche m-aire

Un arbre de recherche m-aire est la généralisation d'un arbre de recherche binaire.

Un arbre de recherche m-aire **d'ordre n** est un arbre où chaque nœud peut avoir au maximum ***n*** fils ($Fils_{1..n}$) et ***n-1*** valeurs ($Val_{1..n-1}$).

Le degré (***d***) d'un nœud représente le nombre de fils dans le nœud. Il est toujours égal au nombre de valeurs stockées dans le nœud plus un. Dans un arbre m-aire de recherche d'ordre ***n***, chaque nœud aura donc un degré $d \leq n$.

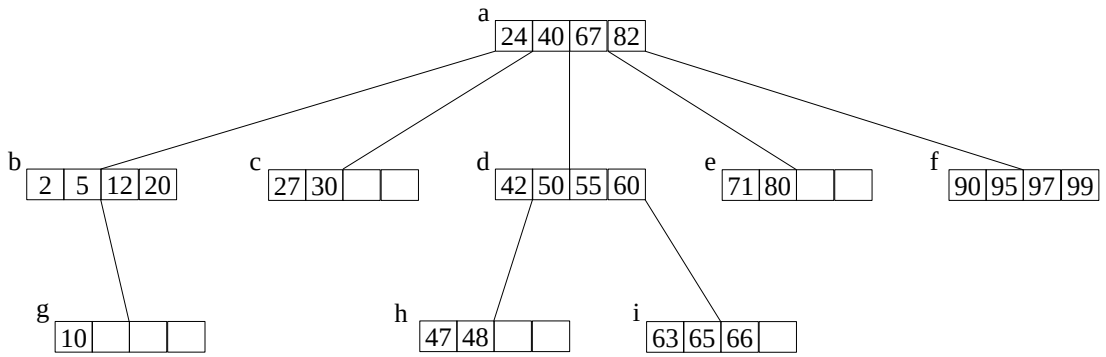
Les valeurs à l'intérieur d'un nœud sont ordonnées en ordre croissant. Les fils sont organisés en fonction des valeurs du nœuds, selon les règles suivantes :

i) Le $Fils_1$ pointe un sous-arbre contenant des valeurs $< Val_1$
ii) Le $Fils_i$, pour $i \in [2 , d-1]$, pointe un sous-arbre contenant des valeurs $\in]Val_{i-1} , Val_i[$
iii) Le $Fils_d$ pointe un sous-arbre contenant des valeurs $> Val_{d-1}$

La figure suivante montre un arbre de recherche m-aire d'ordre *5* de racine le nœud ***a***. Le nœud ***b*** est le $fils_1$ de ***a***, le nœud ***c*** est le $fils_2$ de ***a*** et ainsi de suite, jusqu'au nœud ***f*** qui est le $fils_5$ de ***a***.

Dans un nœud feuille, tous les fils sont à *nil*. Par contre dans un nœud interne, au moins un des fils doit être différent de *nil*. Le nœud qui n'a pas de père (et qui doit être unique) est appelé racine de l'arbre.

Par exemple, dans l'arbre de la figure, tous les fils de ***b*** à part le 3e fils, sont à *nil*.



La recherche d'une valeur ***C*** commence dans le nœud racine ***P*** et se poursuit le long d'une branche :
1- Si ***C*** existe dans ***P*** alors la recherche s'arrête avec succès
2- Si ***C*** n'existe pas dans ***P*** alors
soit ***k*** la position dans ***P*** où devrait être insérée ***C*** (pour que les valeurs restent ordonnées)
Si $Fils_k$ différent de *nil* alors ***P*** ← $Fils_k$; **aller à 1**
Sinon la recherche s'arrête avec échec.

Par exemple, la recherche de *48* dans l'arbre de la figure précédente se déroule comme suit:
- on commence la recherche dans le nœud ***a***. La valeur *48* n'existe pas et est comprise entre *40* et *67*. Donc le prochain nœud à visiter est le 3e fils (le nœud ***d***)
- la recherche se poursuit dans le nœud ***d***, et le prochain nœud à explorer est le 2e fils, car *48* est comprise entre *42* et *50*
- on continue la recherche dans le nœud ***h***. La valeur *48* existe (Val_2), on s'arrête donc avec succès.

Si on avait recherché la valeur *15*, on aurait visité d'abord le nœud ***a***, puis le nœud ***b*** ($Fils_1$ de ***a***, car *15* < *24*). Là, on se serait arrêté avec un échec, car *15* est comprise entre *12* et *20* et le 4e fils de ***b*** est à *nil*.

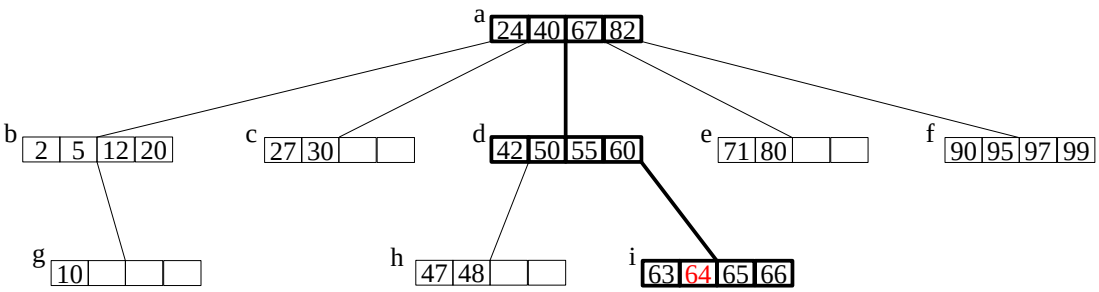
Pour insérer une nouvelle valeur ***V*** dans un arbre de recherche m-aire, on recherche d'abord la valeur, pour vérifier qu'elle n'existe pas déjà et pour localiser le nœud ***P*** où doit être insérée cette valeur (***P*** est le dernier nœud visité dans la recherche). La recherche retourne aussi l'indice ***k*** où devrait être insérée ***V*** s'il y a avait de l'espace dans ***P***.

Si ***P*** n'est pas plein,
- on insère ***V*** dans ***P***, par décalages afin de garder le tableau de valeurs ordonné.

Sinon (***P*** est plein) :
- on alloue un nouveau nœud ***Q*** contenant une seule valeur (Val_1 =***V***) et deux fils à *nil* ($Fils_1 = nil$ et $Fils_2 = nil$)
- on fait pointer le nouveau nœud ***Q*** par $Fils_k$ dans le nœud ***P*** (qui était, avant l'insertion, forcément à *nil*). L'indice ***k*** est celui retourné par la recherche.

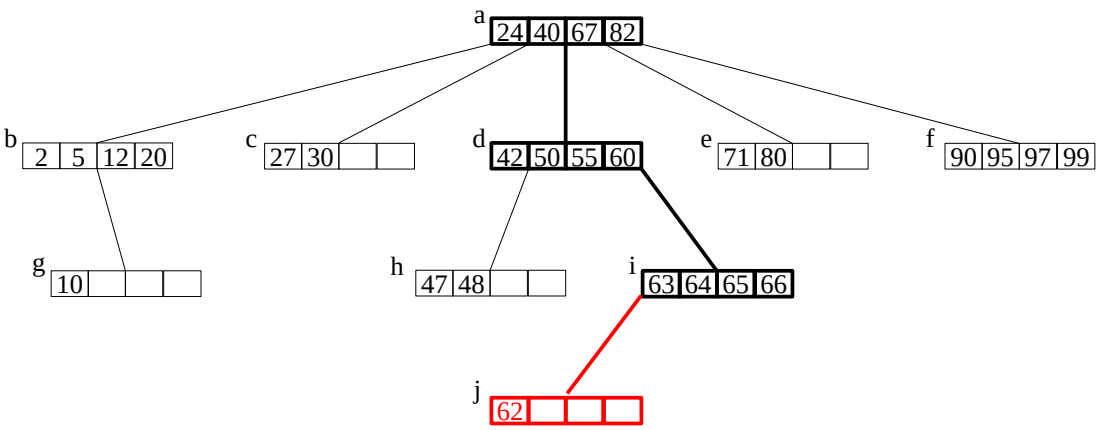
Par exemple, si on insère la valeur *64* dans l'arbre de la figure précédente, on procédera comme suit:
1- recherche de *64* → échec (le dernier nœud visité est ***i*** et la position où devrait être insérée *64* est ***k***=*2*)
2- comme le nœud ***i*** n'est pas plein, on peut alors insérer *64* à la position *2* en décalant à droite les valeurs > *64*

Le résultat de l'insertion est montré dans la figure ci-dessous :



Si ensuite, on insère la valeur *62*, on allouera alors un nouveau nœud (***j***) qui va contenir *62* et il sera pointé par $Fils_1$ au niveau du nœud ***i*** (***i*** étant le dernier nœud visité lors de la recherche de *62* et la position ou devrait être insérée *62*, s'il y avait de l'espace dans ***i***, était ***k*** = *1*).

La figure suivante montre le résultat d'une telle insertion:



Un arbre de recherche m-aire est dit « Top-Down » si tous ces nœuds internes sont remplis à 100%. L'algorithme d'insertion présenté ci-dessus, maintient cette propriété.

La suppression d'une valeur ***V*** peut être logique (ce qui favorise le maintient de la propriété top-down) ou bien physique en généralisant la suppression des arbres de recherche binaires (mais le maintient de la propriété top-down est légèrement plus difficile dans ce cas).

- En mémoire secondaire

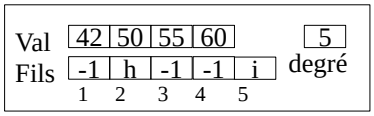
Chaque nœud de l'arbre est représenté par un bloc d'E/S.
La structure générale d'un bloc est alors comme suit :

```
type
    Tbloc = structure
        Val : tableau[N-1] de typeqlq;      // enregistrements ou (clés-adr)
        Fils : tableau[N] d'entier;         // numéros de blocs
        degré : entier;                     // nb d'elt dans le tableau Fils
    Fin;
```

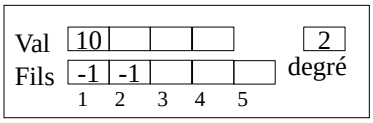
Les adresses de nœuds sont des entiers (des n° de blocs). Un entier spécial (ex -*1*) représente *nil.*

Parmi les caractéristiques d'un fichier vu comme arbre, il y a la racine. C'est le numéro du bloc contenant le nœud racine de l'arbre.

Par exemple, si l'arbre de la figure précédente était un fichier, le contenu du bloc ***d*** serait comme suit:



alors que le contenu du bloc ***g*** serait :



Le module suivant permet de réaliser une recherche dans un fichier organiser sous forme d'un arbre de recherche m-aire :

```
Rech( c:Typeqlq, var trouv:booleen, var i, j, prec : entier)
/*
 * recherche c et retourne en plus du booléen trouv, le num de bloc (i) qui contient c,
 * sa position (j) dans le tableau Val ainsi que le numéro du bloc qui précède i (prec).
 * Si c  n'existe pas, alors i sera positionnée à -1 et prec indique le numéro du dernier bloc visité.
 * la position ou devrait se trouver c dans le bloc prec est indiquée par j.
 */
        //on suppose que le fichier est déjà ouvert
        i ← Entete(F,1);  // le num du bloc racine
        prec ← -1;  j ← 1;  trouv ← FAUX

        TQ ( Non trouv et i <> -1 )
            LireDir( F, i, buf )

            // recherche interne dans le buffer en MC …
            Rech_interne( c, trouv , j )       // retourne : trouv et j

            Si  ( Non trouv )   prec ← i;   i ← buf.Fils[j]  Fsi
        FTQ
// fin du module de recherche.

// Recherche dichotomique interne de c dans buf. Résultats: trouv et j (la position de c)
Rech_interne( c : typeqlq , var trouv:bool , var j:entier )

        bi ← 1;  bs ← buf.degre-1 ;  trouv ← FAUX
        TQ  ( Non trouv et  bi ≤ bs )
                j ← (bi + bs) div 2
                Si ( c = buf.Val[j].cle )   trouv ← VRAI
                Sinon Si ( c < buf.Val[j].cle )  bs ← j-1 Sinon bi ← j+1  Fsi
                Fsi
        FTQ
        Si ( bi > bs ) j ← bi  Fsi
// fin de la recherche interne.
```

L'insertion d'une nouvelle valeur (enregistrement ou clé/adr) dans le fichier utilise la recherche:

Ins(e:Tenreg; nomf:chaine)

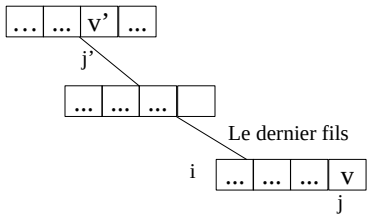
```
Ouvrir( F, nomf, 'A' )
Si ( Entete(F,1) = -1 )                                  // Si le fichier est vide
    i ← AllocBloc( F );   Aff_entete( F, 1, i )          // i est la nouvelle racine
   buf.degre ← 2;  buf.Val[1] ← e;   buf.Fils[1] ← -1;  buf.Fils[2] ← -1
   EcrireDir( F, i, buf )
Sinon
   Rech( e.cle, trouv, i, j, prec )
   Si ( Non trouv )
        // si le dernier bloc visité n'est pas plein, on y insère l'enreg e …
        Si ( buf.degre < N )
              // décalages interne à partir de j …
              Pour k ← buf.degre, j+1, -1                // boucle arrière avec pas = -1
                    buf.Val[k] ← buf.Val[k-1]
                    buf.Fils[k+1] ← buf.Fils[k]
              FinPour
              buf.Val[j] ← e                             // insérer e à la pos j
              buf.Fils[j+1] ← -1                         // et son « fils droit » à nil
              buf.degre ← buf.degre + 1                  // incrémenter le degré
              EcrireDir( F, prec, buf )

        Sinon  // si le bloc est plein, il faut allouer un nouveau …
              i ← AllocBloc( F )
              // et le chaîner avec le précédent
              buf.Fils[j] ← i
              EcrireDir( F, prec, buf )
              // insérer e à la pos 1 du nouveau bloc …
              buf.degre ← 1;  buf.Val[1] ← e;  buf.Fils[1] ← -1;  buf.Fils[2] ← -1
              EcrireDir( F, i, buf )
        Fsi // buf.degre < N
    Fsi  // Non trouv
Fsi  // Entete(F,1) = -1
Fermer(F)
```

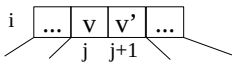
La **requête à intervalle** dans un fichier en arbre de recherche m-aire, s'effectue en recherchant d'abord la borne inférieure de l'intervalle, ensuite les prochaines valeurs sont retrouvées à l'aide des suivants-inordre.

Pour localiser le suivant-inordre d'une valeur ***v*** se trouvant dans le bloc numéro ***i*** à la position ***j***, on effectue les tests suivants :

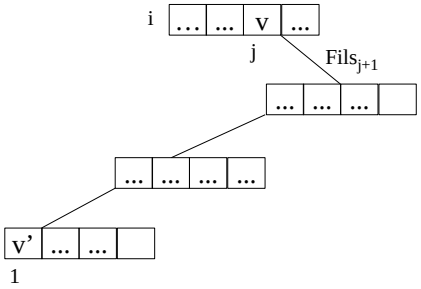
1- Si ***v*** se trouve à la dernière valeur du bloc ***i*** et le dernier fils est à *nil* (c-a-d : ***j*** = *buf.degre - 1 && $Fils_{degre}$ = -1*) le suivant ***v'*** se trouve dans un nœud ascendant de ***i***. Il faut remonter de bloc en bloc (à l'aide d'une pile par exemple) jusqu'à localiser le premier ascendant par lequel la remontée s'est effectuée par un fils autre que le dernier. Soit ***j'*** ce numéro de fils. Le suivant de ***v*** se trouve alors à la position ***j'*** de cet ascendant.
Voir schéma ci-dessous.



2- Si ***v*** n'est pas la dernière valeur du bloc ***i*** et son fils-droit ($Fils_{j+1}$) est à *nil* (c-a-d : ***j*** < *buf.degre - 1 && et* $Fils_{j+1}$ *= -1*), alors le suivant ***v'*** se trouve à la position ***j+1*** du même bloc ***i***.
Voir schéma ci-dessous.



3- Si ***v*** (qui se trouve à la position ***j*** du bloc ***i***) a un fils-droit différent de *nil* (c-a-d : $Fils_{j+1}$ *<> -1*) alors le suivant inordre ***v'*** se trouve à la 1^{ere} position du bloc le plus à gauche du sous-arbre $Fils_{j+1}$.
Pour localiser le bloc le plus à gauche, on descend de $Fils_1$ en $Fils_1$ jusqu'à trouver un bloc qui n'a pas de $Fils_1$ (c-a-d $Fils_1$ *= -1*). Voir schéma ci-dessous.



Le parcours inordre d'un arbre m-aire de recherche permet de visiter toutes les valeurs de l'arbre en ordre croissant.

```
Inordre( r:entier )              // en entrée r : le numéro du bloc racine de l'arbre
var   buf : Tbloc ;   k:entier   // variables locales : le buffer buf et l'indice k
SI ( r <> -1 )
        LireDir( F , r , buf )   // on suppose que la variable F est globale (le fichier déjà ouvert)
        POUR (k = 1 , buf.degre - 1 )
                Inordre( buf.Fils[k] )
                visiter( Val[k] )
        FP
        Inordre( buf.Fils[ buf.degre ] )
FSI
```

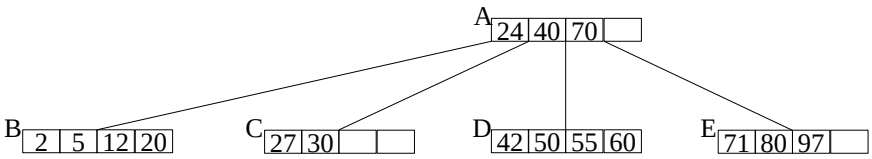
3) Les B-Arbres

Ce sont des arbres de recherche m-aire qui restent toujours équilibrés et sont donc très utilisés pour gérer des fichiers volumineux et dynamiques. Inventés par R. Bayer et E. Mc Creight en 1972, les B-Arbres sont devenus un standard de facto à cause de leurs performances remarquables et ce, jusqu'à nos jours.

Pour simplifier la présentation, on choisira un ordre impair (***N = 2d+1***)
Dans un B-Arbre d'ordre ***N*** :

- Tous les nœuds à part la racine, sont remplis au minimum à 50% (soit ***d*** valeurs)
- La racine peut contenir au minimum ***1*** valeur
- Toutes les branches ont la même longueur (arbre complètement équilibré)

La figure ci-dessous montre un B-Arbre d'ordre ***5*** (c-a-d au minimum ***d=2*** valeurs, au maximum ***2d=4*** valeurs par nœud)



La **recherche** dans un B-Arbre est similaire à la recherche dans un arbre de recherche m-aire.

La différence se situe dans l'algorithme d'insertion et l'algorithme de suppression.

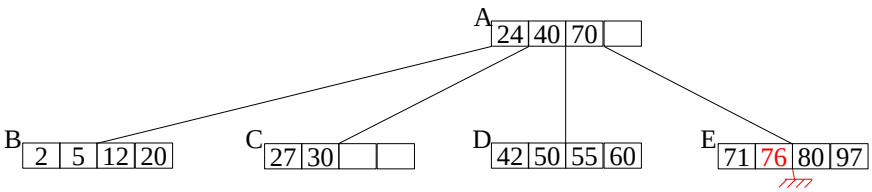
Pour **insérer** une valeur ***v*** dans un B-Arbre de racine ***R***, on procède comme suit :
On insère ***v*** et son fils-droit ***fd*** (initialement ***fd = -1***), c-a-d si ***v*** sera insérée à une position ***j*** dans un nœud, alors il faudra insérer aussi son fils-droit ***fd*** à la position ***j+1*** (dans le tableau des fils) :
1- Rechercher ***v***, pour vérifier qu'elle n'existe pas et trouver le dernier nœud feuille visité (***P***)
2- si ***P*** n'est pas plein, on insère (***v, fd***) dans ***P*** par décalages internes et on s'arrête.
3- si ***P*** est déjà plein, alors il va « *éclater* » en deux (***P*** et ***Q***, un nouveau nœud alloué) :

a) former la « séquence ordonnée » des valeurs de ***P*** incluant la valeur ***v*** (et son fils-droit ***fd***)
b) affecter les ***d*** premières valeurs avec les ***d+1*** premiers fils au nœud ***P***
c) affecter les ***d*** dernières valeurs avec les ***d+1*** derniers fils au nouveau nœud ***Q***
d) insérer la valeur du milieu (celle qui se trouve à la position ***d+1*** dans la séquence ordonnée) dans le nœud parent de ***P***, s'il existe. Le fils droit de cette valeur sera le nœud ***Q*** :
Si le parent de ***P*** existe, **Aller à 2** pour réaliser cette insertion de la valeur du milieu,
Sinon si le nœud parent de ***P*** n'existe pas (***P*** était la racine de l'arbre) : allouer une nouvelle racine ***R*** contenant uniquement la valeur du milieu avec comme fils ***1*** et ***2***, les nœuds ***P*** et ***Q*** et on s'arrête.

Par exemple, l'insertion de 76 dans le B-Arbre de la figure précédente, se déroule comme suit:

1- recherche de 76 → trouv =FAUX, le dernier nœud visité = ***E*** (c'est une feuille) et la position où doit être insérée 76 est 2
2- comme le nœud ***E*** n'est pas plein, on insère alors 76 avec son « fils-droit » -1 respectivement aux positions 2 dans le tableau *Val* et 3 dans le tableau *Fils* du nœud ***E***

On obtient alors le B-Arbre ci-dessous :



Si on insère maintenant la valeur 57, c'est le nœud ***D*** qui sera visité en dernier par la recherche.
La position de 57 dans ***D*** devrait alors être 4 (et son ***fd*** à la position 5)

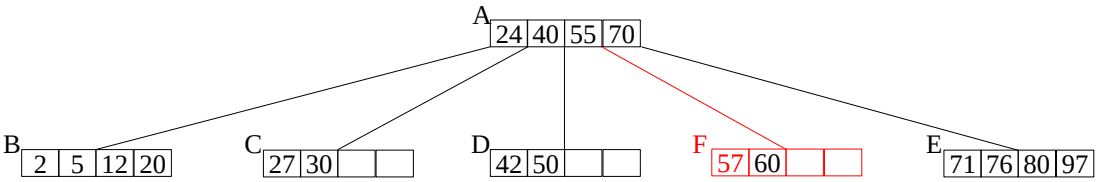
Comme le nœud ***D*** est déjà plein, il y aura alors un « éclatement » (cela veut dire, allocation d'un nouveau nœud, ***F*** par exemple).

a) formation de la « séquence ordonnée » incluant la nouvelle valeur 57:

Val : 42, 50, 55, 57, 60
Fils: -1, -1, -1, -1, -1, -1

b) les ***d*** premières valeurs (42, 50) et les ***d+1*** premiers fils (-1, -1, -1) seront affectés au nœud ***D***
les ***d*** dernières valeurs (57, 60) avec les ***d+1*** derniers fils (-1, -1, -1) seront affectés au nœud ***F***
la valeur du milieu (55) avec comme fils-droit (***F***) seront insérés dans le nœud parent (le nœud ***A***).
Comme le nœud ***A*** n'est pas plein, l'insertion de (55, ***F***) peut se faire par décalages internes

On obtient alors le B-Arbre suivant:



Si on continue l'insertion de 7, le nœud ***B*** va « éclater » (donc allocation d'un nouveau nœud ***G***)
la séquence ordonnée des valeurs de ***B*** + la valeur 7 est formée:

Val : 2 , 5 , 7 , 12 , 20
Fils: -1, -1, -1, -1 , -1 , -1

Les valeurs (2 et 5) avec les 3 premiers fils restent dans ***B***. Les valeurs (12 et 20) avec les 3 derniers fils seront affectés au nouveau nœud ***G***. La valeur du milieu (7) avec comme fils-droit le nœud ***G*** seront insérés dans le père (le nœud ***A***).

Comme le nœud ***A*** est lui aussi plein, il va alors « *éclater* » à son tour. Un nouveau nœud ***H*** est alors alloué et la séquence ordonnée est formée :

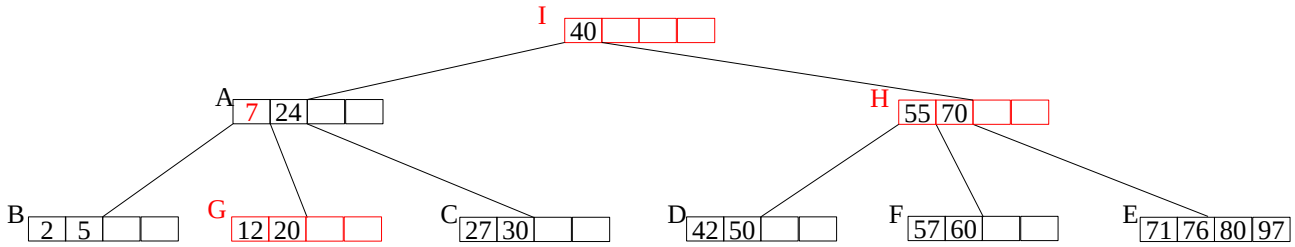
Val : 7 , 24 , 40, 55, 70
Fils: B , G , C , D , F , E

Les ***d*** premières valeurs (7 et 24) avec les 3 premiers fils (***B, G*** et ***C***) restent dans ***A***. Les ***d*** dernières valeurs (55 et 70) avec les 3 derniers fils (***D, F*** et ***E***) seront affectés au nouveau nœud ***H***. La valeur du milieu (40) avec comme fils-droit le nœud ***H*** seront insérés dans le père du nœud ***A***.

Comme le nœud ***A*** était la racine de l'arbre (il n'a donc pas de père), une nouvelle racine (***I***) est alors

allouée contenant uniquement la valeur 40 avec comme fils-gauche (*Fils[1]*), l'ancienne racine (***A***) et comme fils-droit (*Fils[2]*) le nœud ***H***.

La figure suivante, montre l'arbre obtenu :



Quand la racine d'un B-Arbre éclate (à cause d'une insertion), la profondeur de l'arbre augmente d'un niveau.

La **suppression** d'une valeur dans un B-Arbre, se déroule de manière symétrique à une opération d'insertion. Certaines suppressions entraînent une diminution de la profondeur de l'Arbre.

Pour supprimer une valeur ***v***, on procède comme suit:
1- rechercher v
les résultats → *p : le nœud qui contient v et*
k : la position de v dans le tableau Val de p

2- Si p est une feuille, Aller à 3
Sinon // p est un nœud interne
a- remplacer v par son successeur (v'), qui se trouve à la 1ere position de la feuille (p') la plus à gauche du sous-arbre droit de v (le sous-arbre de racine Fils[k+1])
b- supprimer v' dans p' comme suit :
v ← v'
p ← p'
Aller à 3

3- // suppression d'une valeur v dans un nœud p …
Ecraser v ainsi que son fils-droit en réalisant des décalages internes dans le bloc p
Si p contient encore au moins d valeurs alors Aller à 4 // fin de la suppression
Sinon // il reste moins de d valeurs (donc p devient sous-chargé)
Si un des frères (q) de p contient plus de d valeurs,
effectuer une redistribution entre p et q (voir ci-dessous)
Sinon // aucun des frères n'a plus de d valeurs
fusionner p avec l'un de ses frères (voir ci-dessous aussi)
et supprimer la valeur qui les séparait dans leur père comme suit :
v ← cette valeur
p ← le père de p
Aller à 3

4 - fin

La redistribution entre deux nœuds frères ***X*** et ***Y***, met en jeu leur père ***P*** afin d'ajouter une valeur à ***X*** et d'enlever une valeur à ***Y*** :

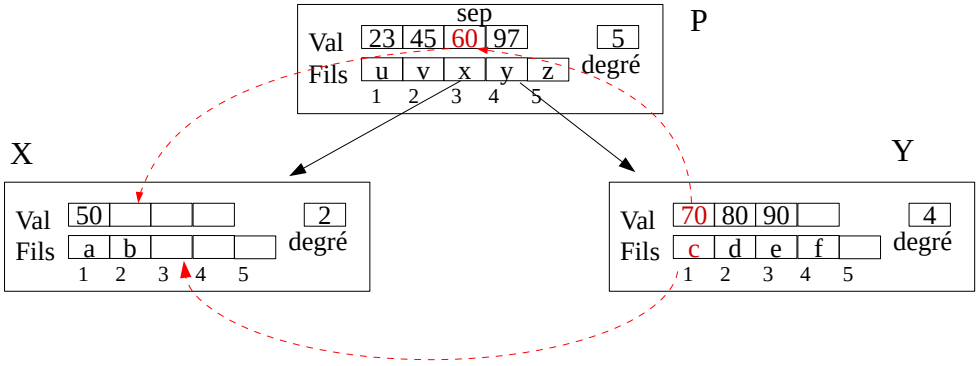
Soit ***sep*** la valeur qui séparait ***X*** et ***Y*** dans le père ***P***

Si ***Y*** est un frère droit de ***X***,
- ***sep*** est inséré comme dernière valeur de ***X***
- La 1ère valeur de ***Y*** monte vers ***P*** pour remplacer **sep**
- Le *Fils[1]* de ***Y*** est transmis à ***X*** à la dernière position
- Les valeurs et les fils sont décalés d'une position vers la gauche dans ***Y***
- Les degrés dans ***X*** et ***Y*** sont mis à jour en conséquence

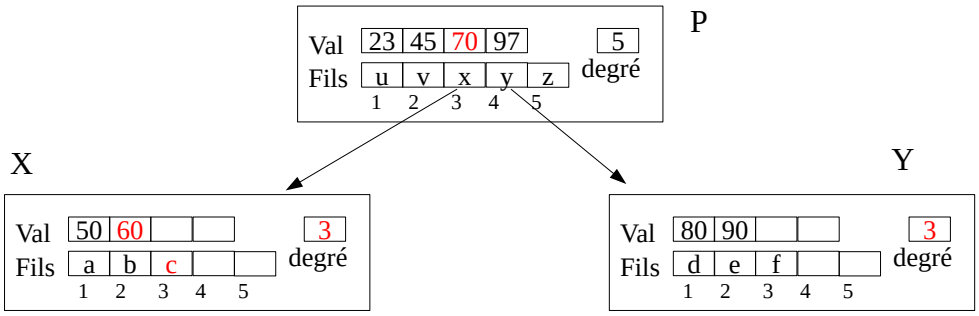
Si ***Y*** est un frère gauche de ***X***,
- Les valeurs et les fils sont décalés d'une position vers la droite dans ***X***
- ***sep*** est inséré comme première valeur de ***X*** (*Val[1]*)
- La dernière valeur de ***Y*** monte vers ***P*** pour remplacer ***sep***
- Le dernier fils de ***Y*** est transmis à ***X*** à la première position (*Fils[1]*)
- Les degrés dans ***X*** et ***Y*** sont mis à jour en conséquence

Les schémas suivants montrent une redistribution de ***Y*** vers ***X*** (avec ***Y*** un frère droit)

Avant la redistribution :



Après la redistribution :

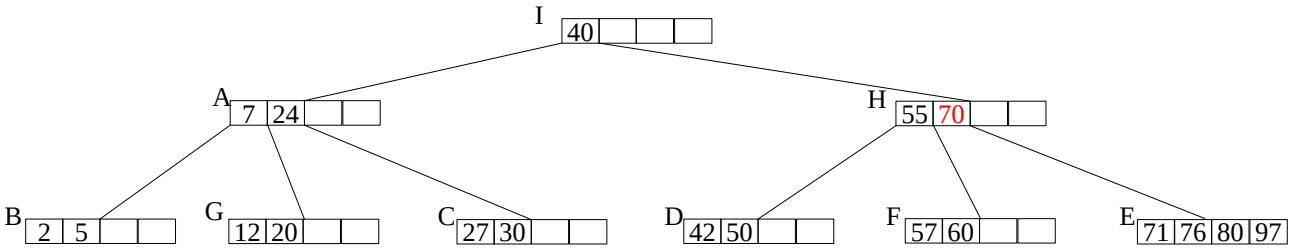


La fusion de 2 nœuds frères ***X*** et ***Y*** de père ***P***, consiste à rassembler dans un même nœud (***X*** ou ***Y***) tous les contenus des 2 nœuds (***X*** et ***Y***) en plus du séparateur qui se trouvait dans ***P***. L'un des 2 nœuds frères sera libéré.
Pour fusionner ***Y*** dans ***X***, il faut :

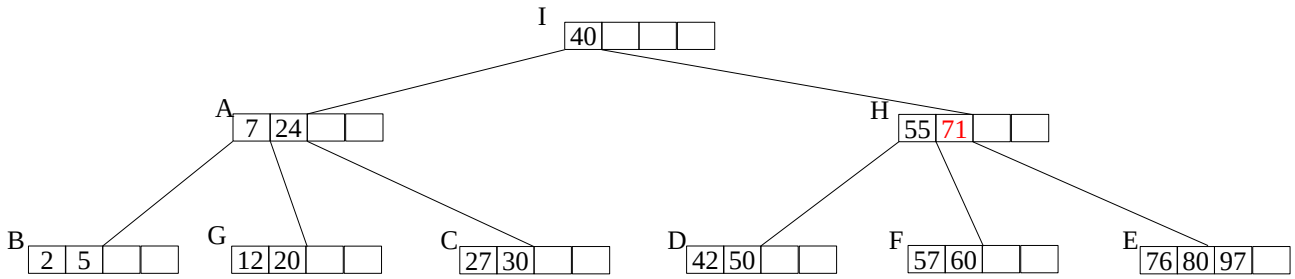
Faire descendre le séparateur de ***P*** vers ***X*** (en le supprimant de ***P***)
Transférer toutes les valeurs et tous les fils de ***Y*** vers ***X***
Libérer le nœud ***Y***

Exemples de suppressions

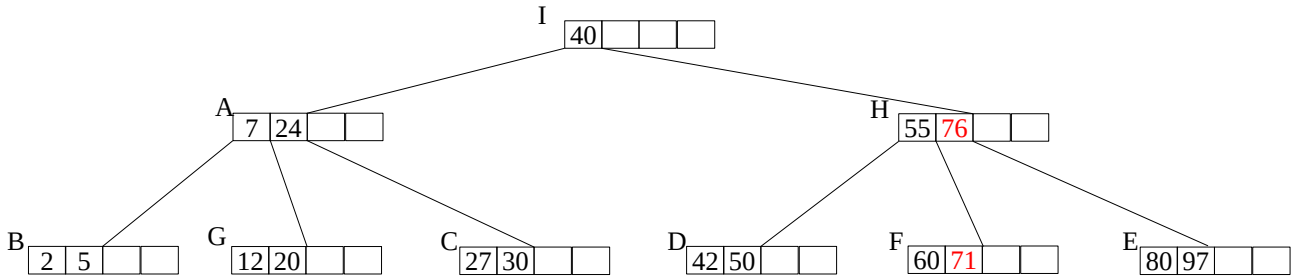
Si on veut supprimer la valeur 70 dans l'arbre ci-dessous :



comme 70 se trouve dans un nœud interne, on doit alors la remplacer par son successeur (71) qui se trouve à la 1ère position de la feuille ***E***
on doit ensuite supprimer 71 du nœud ***E***,
comme ***E*** n'est pas encore sous-chargé (il reste encore 3 valeurs), on s'arrête ici.



Si maintenant on supprime 57, le nœud ***F*** va devenir sous-chargé (nombre de valeurs restantes < ***d***)
comme l'un des frères de ***F*** (le nœud ***E***) contient plus de ***d*** valeurs, on peut alors faire une redistribution: 71 descend vers ***F*** et 76 monte vers ***H*** (le père des nœuds ***F*** et ***E***)



Si on continue à supprimer par exemple la valeur 40, il y aura :

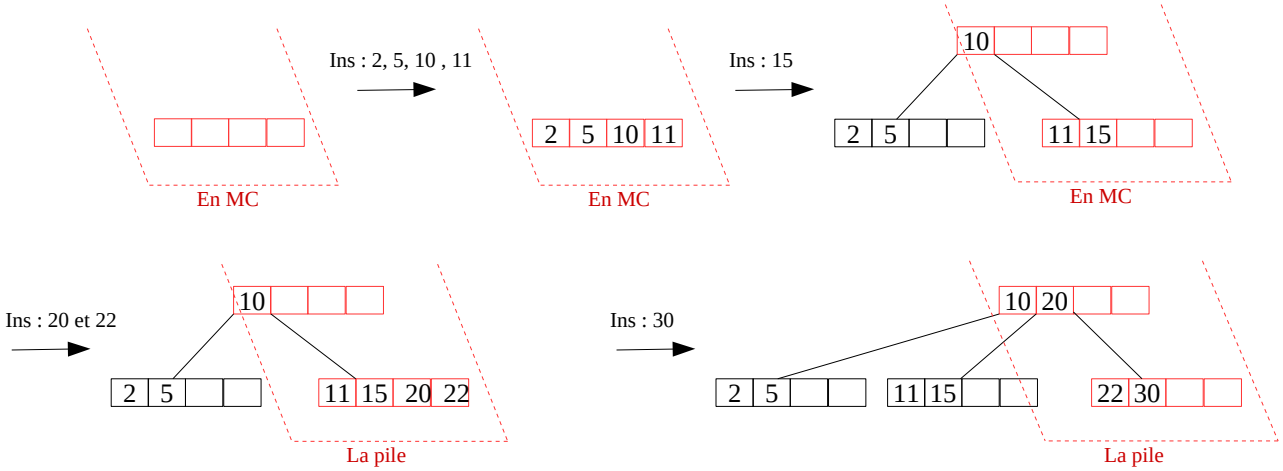
fusion des nœuds ***D*** et ***F*** (avec libération par exemple de ***F***)
fusion des nœuds ***A*** et ***H*** (avec libération par exemple de ***H***) et libération de la racine ***I*** (vide)
la profondeur de l'arbre décroît
→ comme exercice, retrouver les détails !

Opération de chargement d'un B-arbre

Souvent on a besoin d'insérer, de manière groupée, un grand nombre de données dans un B-arbre. Cela arrive par exemple quand on veut construire un fichier index (sous forme de B-arbre) à partir d'un fichier de données déjà existant (contenant déjà un nombre important d'enregistrements). Si on utilise pour cela l'algorithme de l'insertion classique, cela risque de prendre un temps assez grand (malgré que l'insertion dans un B-arbre d'une valeur donnée soit efficace ***O(log n)***, le problème provient du grand nombre de données à insérer nécessitant donc un nombre important de lectures et écritures physiques de blocs).

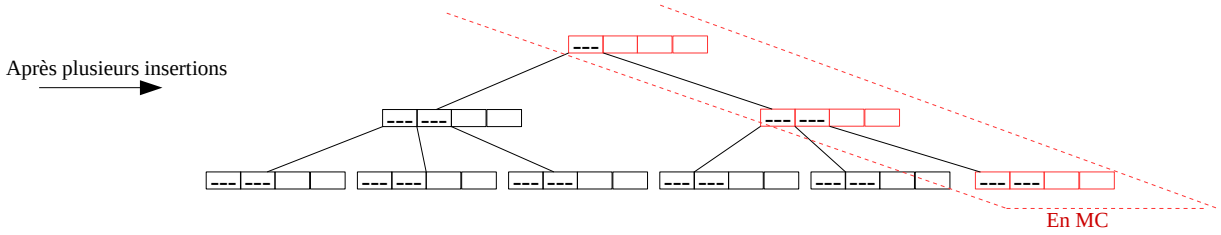
Une solution efficace, consiste à insérer les données en ordre croissant dans le B-arbre à construire, avec un algorithme particulier (*Bulk Insertions*). Cette approche consiste à remplir les feuilles de l'arbre de gauche à droite en réalisant les éclatements le long de la branche la plus à droite de l'arbre.

Les performances seront encore meilleurs en gardant en **MC** (dans un tableau de buffers pour exemple) tous les blocs de la branche la plus à droite de l'arbre en cours de construction. En effet, durant cette opération de chargement, toutes les insertions et éclatements ne vont concerner que les nœuds de la branche la plus à droite de l'arbre. A la fin, tous les buffers (représentant les blocs de la branche la plus à droite) seront écrits physiquement en **MS**. Dans le schéma ci-dessous, les nœuds en rouge sont les buffers en **MC**, les nœuds en noir sont les blocs déjà écrits physiquement en **MS** (disque).



L'idée principal de cet algorithme est d'insérer la prochaine valeur, directement dans le buffer associé à la feuille la plus à droite sans effectuer de recherche. Un grand nombre de valeurs pourront donc être insérées de cette manière sans faire aucun accès disque. Lorsque le buffer devient plein, on réalise un éclatement en écrivant sur disque, uniquement le bloc renfermant la moitié de gauche. La moitié de droite (le nouveau nœud le plus à droite) ainsi que la valeur du milieu (le nœud père sur la branche la plus à droite) restent en **MC** (jusqu'à ce qu'ils deviennent pleins à leur tour). De cette manière, le coût

d’un éclatement est ramené à une seule écriture seulement (au lieu de 3 écritures dans le cas classique).



Comme les données vont être insérées en ordre croissant, l’arbre ainsi obtenu sera faiblement chargé (au voisinage de 50 %). Pour éviter cela, on peut modifier légèrement l’opération d’éclatement utilisée lors du chargement pour remplir les nœuds de gauche avec un pourcentage contrôlé (supérieur à 50 %) et laisser les nœuds de droite temporairement sous-chargés (< 50%). A la fin de l’opération de chargement, on rééquilibre les nœuds de la branche la plus à droite (pouvant être sous-chargés à cause de cet éclatement particulier) en réalisant des redistributions avec leurs frères gauches respectifs avant de les écrire sur disque.

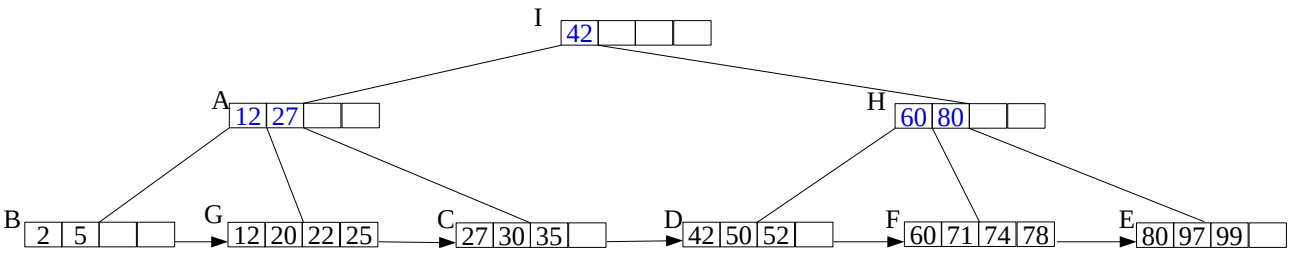
Quelques variantes

a) Les B$^+$-Arbres

La variante la plus utilisée de nos jours est sans aucun doute celle dite « **B$^+$-Arbre** » car elle est bien adaptée aux fichiers en séquentiel-indexé qui supportent de manière efficace aussi bien les accès aléatoires que les accès séquentiels (comme les requêtes à intervalle). De plus on peux gérer des fichiers dynamiques et volumineux avec pratiquement les mêmes algorithmes utilisés dans les B-Arbres de base.

L'idée est que le dernier niveau de l'arbre joue le rôle d'un fichier ordonné sous forme de liste de blocs, afin de favoriser l'opération « next » dans les parcours séquentiels. Les niveaux internes représentent un index pour guider la recherche aléatoire.

Lors des insertions, à chaque fois qu'un bloc feuille déborde, il éclate en deux et la première clé du nouveau bloc (la clé du milieu dans la séquence d'éclatement) et « recopiée » dans le bloc père (niveau interne) pour séparer les deux blocs issus de l'éclatement (« recopiée » veut dire : insérée dans le père sans être supprimée de la feuille). Si le nœud père éclate suite à cette recopie, la nouvelle clé du milieu est cette fois-ci déplacée (et non recopiée) vers le nœud parent (à ce niveau on reprend les mêmes techniques utilisées dans les B-Arbres de base).



Dans un B$^+$-arbre, les valeurs dans les nœuds internes sont les copies de certaines valeurs du niveau

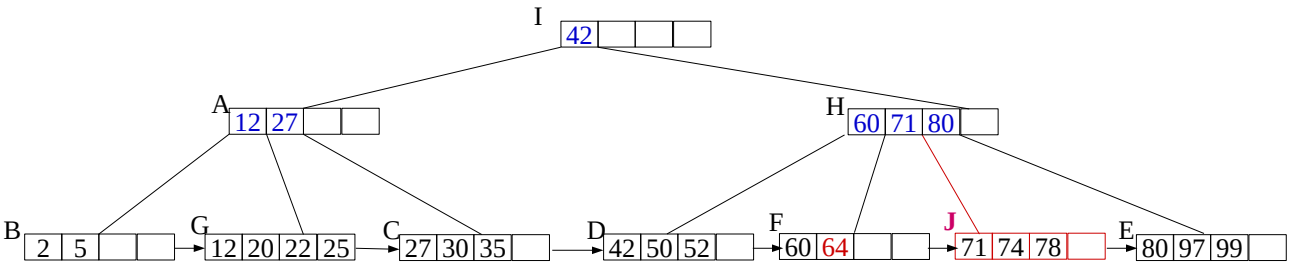
feuille. Ce sont des ‘séparateurs’ utilisés pour guider la recherche le long d’une branche. Le niveau feuille contient toutes les clés (et les informations associées).

Si le B⁺-arbre représente un fichier de données, le niveau feuille contient les enregistrements (clés comprises). Si le B⁺-arbre représente un fichier index, le niveau feuille contient les couples <clé, adr>. Dans les deux cas, les nœuds internes ne contiennent que quelques clés (sans le reste de l’information utile) pour guider la recherche vers les feuilles de l’arbre.

Comme les valeurs dans les nœuds internes sont des copies des clés qui existent déjà dans le niveau feuille, les niveaux internes de l’arbre décrivent donc des intervalles semi-ouverts (de la forme]a,b], au lieu d’intervalles ouverts]a,b[comme dans le cas des B-arbre de base). L’algorithme de recherche est légèrement modifié pour tenir compte de ce petit changement.

Voici un exemple d’insertion de la valeur 64 dans le B⁺-arbre de la figure précédente :
1- la recherche de 64 mène vers le bloc feuille ***F***
2- comme ***F*** est déjà plein, il y aura un éclatement au niveau feuille :
- Allocation d’un nouveau bloc feuille → le bloc ***J***
- La séquence ordonnée : [60,64,71,74,78] est divisée en 2 : [60,64] dans ***F*** et [71,74,78] dans ***J***
- la clé du milieu: 71 (la 1ere du nouveau bloc ***J***) sera recopiée dans le bloc parent ***H***
  (comme ***H*** n’est pas plein, l’insertion de 71 se fait juste par décalages)

On obtient l’arbre suivant :



b) Les B*-Arbres

C'est des B-Arbres où chaque nœud (autre que la racine) doit être rempli au minimum à 2/3 de sa capacité maximale (au lieu de 1/2 dans la version de base).

Lors d'une insertion, l'éclatement d'un nœud plein est retardé, en employant des redistributions avec les deux frères directs du nœud (celui à gauche et celui à droite), jusqu'à ce que ceux ci soient aussi remplis à leur capacité maximale. A ce moment là, deux de ces trois nœuds (celui subissant l'insertion et l'un des ses frères gauche ou droit) seront éclatés en trois, chacun rempli à 2/3.

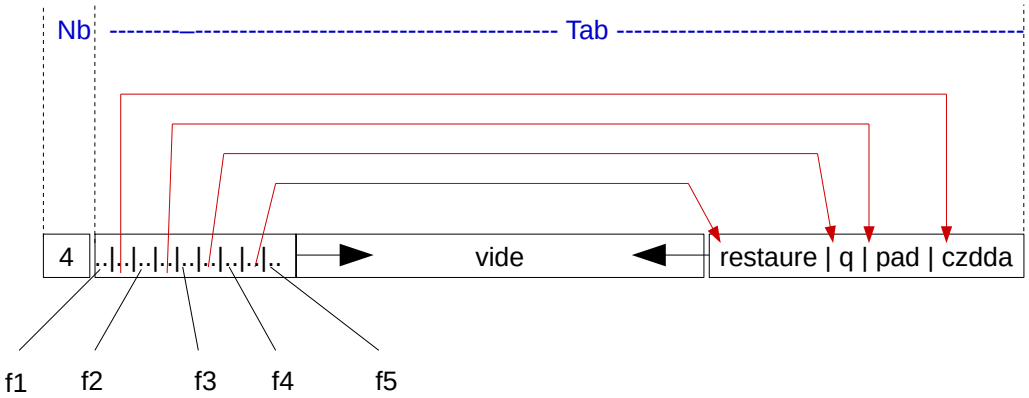
Avec cette variante, on peut avoir un remplissage minimal des nœuds proche de 66%. Ce qui a pour effet d'augmenter le facteur de branchement moyen, donc de diminuer la profondeur de l'arbre et donc, d'améliorer les coûts d'accès.

c) Les B⁺-Arbres préfixés

L'idée dans ce type de B⁺-Arbre, est de compacter au maximum la partie index de l'arbre (formée par les nœuds internes) en utilisant la plus petite partie de clé (et non pas la clé elle même) nécessaire pour séparer deux nœuds fils dans le tableau des séparateurs (*Val[]*). Ainsi les nœuds internes pourront contenir un nombre plus grand de séparateurs et par conséquent un nombre plus grand de fils aussi, ce qui permet de diminuer la profondeur de l'arbre et d'augmenter les performances d'accès.

Comme les préfixes ne sont pas forcément de même taille, le nombre maximum de valeurs (et donc de fils aussi) par nœud devient variable. Cela complique un peu plus les algorithmes d'accès.

La figure ci-dessous montre un exemple de structure utilisée pour représenter les nœuds internes d'un B⁺-arbre préfixé.



Ce nœud interne contient 4 séparateurs (czdda, pad, q et restaure) et 5 fils (f1, f2, … f5).
Le champ *Nb* représente le nombre de séparateurs
Le champ *Tab* est un tableau d'octets, contenant : la liste des numéros fils avec les indices vers les débuts de chaque séparateurs qui se trouvent à l'autre extrémité de *Tab*. Le vide se trouve donc au milieu.

Tab : |f1|96|f2|93|f3|92|f4|84|f5|……..|Restaure Q Pad Czdda|
84 92 93 96

Cette manière d'organiser de la structure d'un nœud interne (*Slotted page*), permet d'effectuer des recherches dichotomiques (dans les séparateurs) malgré le format variable des valeurs.

Chapitre 5 Méthodes de Hachage en MS

Cela concerne l'accès aux données d'un fichier en utilisant une fonction de hachage avec un temps constant. La fonction de hachage permet le calcul d'adresses (numéro de blocs) à partir de la clé. Les notions de fonctions de hachage et de méthodes de résolution de collision en mémoire centrale ont été présentés dans le cours ALSDD (Algorithmes et Structures de données).
Lorsqu'on veut utiliser les méthodes de hachage pour l'accès aux fichiers, il y a principalement trois possibilités :
1. Utiliser une table de hachage comme index, en mémoire centrale, pour accélérer les accès aux fichiers de données.
2. Utiliser un index en mémoire secondaire, géré par une méthode de hachage.
3. Gérer le fichier de données par une méthode de hachage.
Lorsqu'un fichier (index ou de données) est géré par une méthode de hachage (les deux dernières possibilités 2 et 3), le fichier sera formé par des blocs contigus (numérotés *0, 1, … N-1*). La fonction de hachage retourne alors un numéro de bloc (*h(**x**)* : représente le numéro de bloc où devrait se trouver la clé ***x***, c'est l'adresse primaire de ***x***).

5.1 Essai Linéaire en MS

Pour voir un exemple d'utilisation d'une des méthodes du hachage statique (vues en ALSDD) pour la manipulation des fichiers, considérons l'Essai-Linaire appliquée aux fichiers.
La fonction *h(**x**)* retourne le numéro de bloc où commence la recherche de ***x***. Si le bloc est plein et ***x*** n'existe pas dans ce bloc, on continue la séquence de tests : *i-1, i-2, … 0 , N-1 , N-2, ….* jusqu'à, soit trouver ***x***, soit trouver un bloc non plein (critère d'arrêt de la séquence de tests).

Algorithme de recherche

Voici un exemple de solution algorithmique pour la procédure de recherche avec cette méthode :

```
Rech(x:typeqlq, var trouv:booleen, var i:entier, var j:entier)
        // on suppose que le fichier F est déjà ouvert
        i ← h(x) ; trouv ← faux ; stop ← faux ;  N ← Entete(F,1)  // le nombre de blocs dans F
        TQ ( Non trouv && Non stop )
                LireDir( F, i, buf )
                // rech interne dans buf …
                j ← 1
                TQ ( j ≤ buf.NB && Non trouv )
                        SI ( x = buf.tab[ j ].cle )    trouv ← vrai    SINON   j ← j+1   FSI
                FTQ
                SI ( buf.NB < b )
                        // présence d'une case vide, donc arrêt de la séquence de tests
                        stop ← vrai
                SINON
                        SI ( Non trouv )
                                i ← i - 1 ;     SI ( i < 0 )   i ← N-1   FSI
                        FSI
                FSI
        FTQ
```

Mécanisme d'insertion

Pour insérer une nouvelle données ***e*** dans un fichier ***F*** géré par Essai-Linéaire, il suffit de rechercher la clé de ***e*** pour localiser l'emplacement où elle doit être insérée. Il faut aussi laisser au moins une case vide dans le fichier (c-a-d il doit toujours rester, au moins un bloc non plein dans ***F***), pour cela on peut utiliser un compteur d'insertions (comme caractéristique du fichier) et vérifier à chaque insertion qu'il reste inférieur au nombre de places disponibles (*NbIns* < ***N*** * ***b*** avec ***b*** la capacité maximale d'un bloc et ***N*** le nombre de blocs formant le fichier) :

```
Insertion (e:Tenreg)
    // on suppose que le fichier F est déjà ouvert
    N← Entete(F,1)          // le nombre de blocs dans F
    NbIns ← Entete(F,2)     // le compteur d'insertions
    SI ( NbIns < (N * b - 1) )
        Rech( e.clé , trouv , i , j )
        SI ( Non trouv )
            // buf contient déjà le dernier bloc visité (le bloc i) …
            buf.NB++
            buf.tab[ buf.NB ] = e      // on rajoute x dans le bloc i
            EcrireDir( F, i, buf )
            // incrémenter le compteurs d'insertions …
            Aff_Entete( F , 2 , NbIns+1 )
        FSI
    FSI
```

Mécanisme de Suppression

Le principe de la suppression physique d'un enregistrement de clé ***x*** dans un fichier géré par la méthode Essai-Linéaire, consiste à :

1. Rechercher ***x*** → bloc numéro : ***i*** et déplacement : ***j*** (où se trouve ***x*** dans le fichier)

2. Si le bloc ***i*** était déjà non plein, **aller à 3**.
Sinon Il faut vérifier qu'aucune autre donnée se trouvant dans les blocs précédents ***i*** (c-a-d : ***i-1***, ***i-2*** …), ne devienne inaccessible lorsque le bloc ***i*** sera non plein (c-a-d contiendra un vide).
Pour chaque donnée ***y*** devenant inaccessible et se trouvant dans un bloc ***k*** (***k*** = ***i-1***, ***i-2***, ….),
il faut déplacer ***y*** à la place de ***x*** dans le bloc ***i*** et continuer les tests à partir de ***k-1***,
en modifiant ***i*** et ***j*** : ***i*** ← ***k*** et ***j*** ← l'ancien emplacement de ***y*** dans le bloc ***k***.
La donnée à supprimer maintenant devient ***y*** (c-a-d : ***x*** ← ***y***).
Les vérifications s'arrêtent lorsque ***k*** atteint un bloc non plein et que toutes ses données auront été vérifiées.

3. Supprimer ***x*** dans le bloc ***i***. Cela à pour conséquence de rendre le bloc ***i*** non plein.

4. Fin de la suppression.

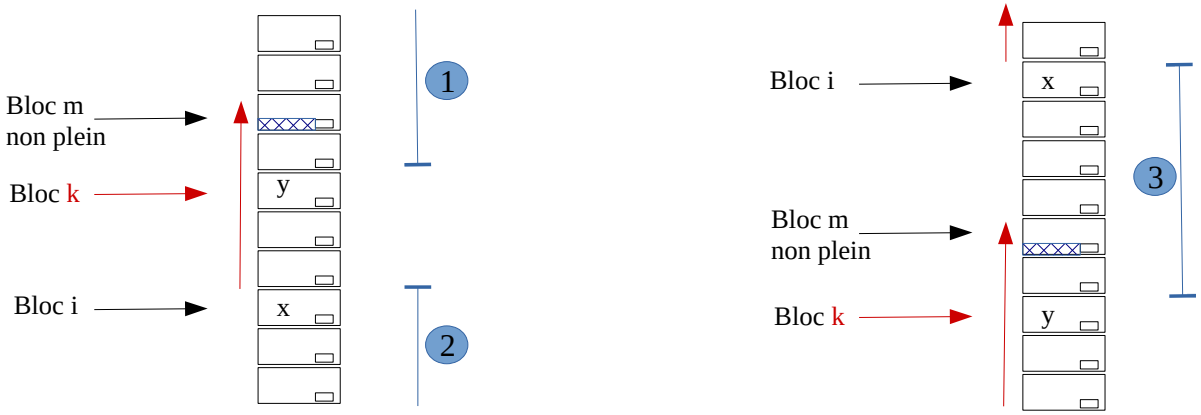
La suppression physique d'une donnée dans un bloc plein, le rend non plein. Cela aura pour conséquence de faire apparaître une nouvelle case vide. Cette nouvelle case vide va probablement briser une séquence de tests en deux. Il faut donc effectuer des tests dans la 2e sous-séquence pour voir si le vide introduit, perturbe ou pas les données existantes :

ancienne séquence : [a a-1 … i+1 i i-1 i-2 … *m*-1 ***m***]
tous les blocs entre a et m-1 étaient pleins avant la suppression (le bloc ***m*** n'est pas plein)
<u>après la suppression dans le bloc ***i***</u>, ce dernier va contenir <u>un vide</u>, on aura donc 2 séquence indépendantes :
nouvelles séquences : [a a-1 … i+1 ***i***] [i-1 i-2 … *m*-1 ***m***]
il faut donc tester les données dans les blocs ***k*** de la 2e séquence uniquement : [i-1, i-2, … m]

La séquence de blocs à tester [i-1, i-2, … m] peut éventuellement être coupée par 0, car la gestion des indices est circulaire (ex : [i-1, … 1 , 0, N-1, … m]). Ce dernier cas est représenté par le 3e intervalle de la figure ci-dessous.

Pour vérifier si une donnée ***y*** (se trouvant dans un bloc ***k***) va devenir inaccessible lorsque le bloc ***i*** va devenir non plein, il faut vérifier si son adresse primaire (***h(y)***) se trouve dans l'un des 3 intervalles de la figure ci-dessous :

Le 1er intervalle est caractérisé par la condition : **h(y) < k < i**
Le 2e intervalle est caractérisé par la condition : **k < i ≤ h(y)**
Le 3e intervalle est caractérisé par la condition : **i ≤ h(y) < k**



Dans le cas où le bloc ***i*** était plein, l'algorithme consiste donc à parcourir tous les blocs ***k***, précédents ***i***, jusqu'à trouver un bloc non plein (donc contenant au moins un vide). Si ***k*** devient ***< 0*** et on n'a pas encore trouvé un bloc non plein, on continue à partir de la fin du fichier (***N-1***) de manière circulaire.

Pour chaque bloc ***k***, on doit parcourir son contenu pour vérifier toutes les données ***y*** à l'intérieur de ***k*** pour voir s'il est nécessaire de les déplacer, ou non, avant de supprimer la donnée ***x*** dans ***i***.
A chaque fois qu'une donnée ***y*** du bloc ***k***, doit être déplacée dans ***i***, on positionne ***i*** à ***k*** pour tester les blocs restant par rapport au nouveau ***i***.

A la fin du parcours des blocs précédents ***i*** (c-a-d : lorsqu'on trouve un bloc non plein), on supprime physiquement ***x*** dans le bloc ***i*** en l'écrasant par la dernière donnée du bloc (*buf.tab[buf.nb]*).
Il est à noter qu'il existe toujours, au moins un bloc non plein avec la méthode Essai Linéaire.

L'algorithme pour supprimer la donnée ***x*** est comme suit :

```
Sup(x)
  // On suppose que le fichier est déjà ouvert
  Rech(x, trouv, i, j)
  SI ( trouv )
        SI ( buf.nb == b )          // si le bloc i est plein ...
            k ← i-1 ; SI ( k < 0 )  k ← N-1  FSI
            // boucle principale pour vérifier tous les blocs précédents i : i-1, i-2,… 0, N-1,…
            stop ← faux
            TQ ( Non stop )
                  LireDir( f, k, buf2 )
                  // boucle interne pour récupérer et tester les données y du bloc k ...
                  m ← 1 ;  stop2 ← faux
                  TQ ( m ≤ buf2.nb   et  Non stop2 )
                              y ← buf2.tab[ m ]
                              SI ( [ h(y) < k < i ] ou [ k < i ≤  h(y) ] ou [ i ≤  h(y) < k ] )
                                          // déplacement de y à la place de x …
                                          buf.tab[ j ] = y
                                          EcrireDir( f, i, buf )
                                          i ← k
                                          j ← m
                                          buf ← buf2
                                          stop2 ← vrai
                              SINON
                                          m++
                              FSI
                  FTQ     //fin de la boucle interne : les données y du bloc k
                  SI ( buf2.nb < b )
                              stop ← vrai
                  SINON
                              k ← k-1 ;   SI ( k < 0 )   k ← N-1   FSI
                  FSI
            FTQ   // fin de la boucle principale :vérification des blocs précédents i
            // suppression physique de x dans i …
            buf.tab[ j ] ← buf.tab[ buf.nb ]
            buf.nb--
            EcrireDir( f, i, buf )
        SINON
            // si le bloc i était au départ non plein …
            // suppression physique de x dans i …
            buf.tab[ j ] ← buf.tab[ buf.nb ]
            buf.nb--
            EcrireDir( f, i, buf )
        FSI  //  ( buf.nb == b )
  FSI  // ( trouv )
```

5.2 Hachage Linéaire (une méthode du hachage dynamique)

Dans ce qui suit nous allons présenter une des méthodes du hachage dynamique (la méthode *Hachage Linéaire*) applicable aux fichiers volumineux et dynamiques.

On dispose d'un fichier ***F*** formé par des blocs contigus numérotés: *0, 1, 2, …*
On maintient deux entiers (***i*** et ***n***) comme caractéristiques du fichier (ce sont des paramètres utilisées par cette méthode de hachage) :

n représente le numéro du prochain bloc qui subira l'opération d'éclatement.
Il est initialisé à *0* et est incrémenté à chaque éclatement de bloc.
(lors d'un éclatement, un nouveau bloc est rajouté à la fin du fichier)
Ce paramètre est réinitialisé à *0* à chaque fois que la taille du fichier double.

i représente le nombre de fois que le fichier a doublé de taille.
Il est initialisé à *0* et est incrémenté à chaque fois que ***n*** atteint 2^i.

Les enregistrements sont insérés dans ***F*** en utilisant deux fonctions de hachage : h_i et h_{i+1} définies comme suit : $h_i(x) = x \bmod 2^i$ et $h_{i+1}(x) = x \bmod 2^{i+1}$

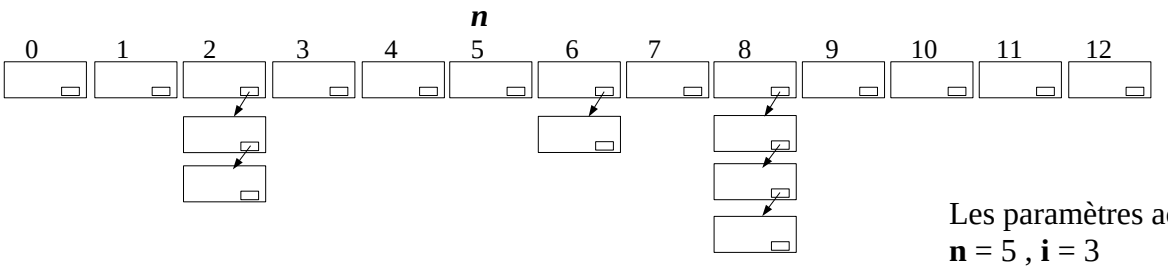
Lors de l'**insertion d'un enregistrement** de clé ***x***, le numéro de bloc à utiliser est calculer avec l'algorithme '**Adr**' suivant :

```
Adr(x)
  a ← hi(x) ;
  SI ( a < n )
        a ← hi+1(x) ;
  FSI
  return a ;        // a représente le num de bloc où devrait se trouver x
```

Si le bloc numéro ***a*** n'est pas plein, l'enregistrement de clé ***x*** est inséré dans ***a***.
Sinon (si le bloc ***a*** est déjà plein) l'enregistrement de clé ***x*** est inséré séquentiellement en zone de débordement associé au bloc ***a***.

Chaque bloc du fichier ***F*** peut alors avoir une liste de blocs de débordement contenant les enregistrements qui n'ont pas pu être insérés dans le bloc principal (car celui-ci était déjà plein).

La figure ci-dessous donne un aperçu du fichier ***F*** après un certain nombre d'insertions :



Les paramètres actuels :
n = 5 , **i** = 3

L'algorithme Adr(x) est aussi utilisé pour **rechercher** les enregistrements :

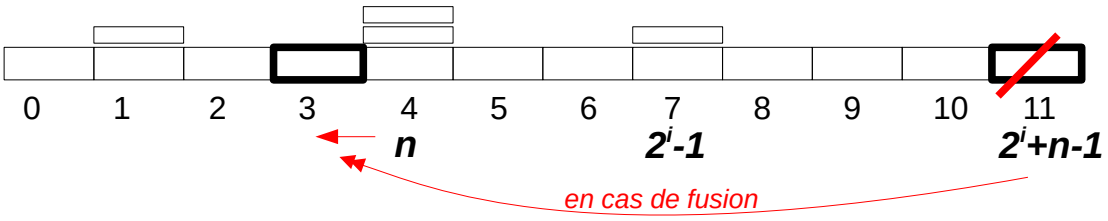
```
Rech(x)
  a = Adr(x)
  LireDir( F, a , buf )
  SI ( x est dans buf  )
        trouv ← vrai
  SINON
        // continuer séquentiellement dans la liste de blocs de débordement de a.
        i ← buf.lien ; trouv ← faux
        TQ ( Non trouv && i <> -1 )
              LireDir(F , i, buf )
              SI ( x est dans buf )   trouv ← vrai SINON i ← buf.lien FSI
        FTQ
  FSI
```

Après chaque nouvelle insertion (ou alors périodiquement) , on vérifie une certaine condition sur l'état du chargement du fichier (par exemple : vérifier si le facteur de chargement global du fichier dépasse un certain seuil S_{max} ou alors si le nombre de cases ayant subi un éclatement dépasse un certain seuil … etc).
Si la condition est vérifiée, on réalise alors l'opération d'**éclatement du bloc *n***, comme suit :

1. *Rajouter un nouveau bloc à la fin du fichier **F** (forcément le numéro du nouveau bloc sera 2^i+n)*
2. *Re-hacher avec la fonction h_{i+1} tous les enregistrements du bloc **n** et éventuellement, ceux de sa liste de débordement (si elle n'est pas vide). Cela a pour effet de déplacer, un certain nombre d'enregistrements vers le nouveau bloc (2^i+n). (ceux pour lesquels h_{i+1}(enreg.clé) a retourné 2^i+n)*
3. *Incrémenter **n***
4. *SI (**n** == 2^i)*
*n ← 0 ; // réinitialiser **n** à 0*
*i++ // à chaque incrémentation de **i**, la taille de **F** aura doublé*
FSI

Quand les données ***x*** d'une case ***c*** sont re-hachées avec h_{i+1}, il y aura que deux résultats possibles :
$h_{i+1}(x) = c$ ou alors $h_{i+1}(x) = c + 2^i$
Car pour toute donnée ***x*** se trouvant dans une case ***c*** (avant l'éclatement de ***c***) on a $h_i(x) = c$
x est donc de la forme $2^i k + c$ (avec ***k*** un entier naturel et $c < 2^i$).
Si ***k*** est pair ($k = 2k'$) alors $x = 2^i(2k') + c = 2^{i+1} k' + c$. Donc $h_{i+1}(x) = c$
Si ***k*** est impair ($k = 2k'+1$) alors $x = 2^i(2k'+1) + c = 2^{i+1} k' + 2^i + c$. Donc $h_{i+1}(x) = c + 2^i$

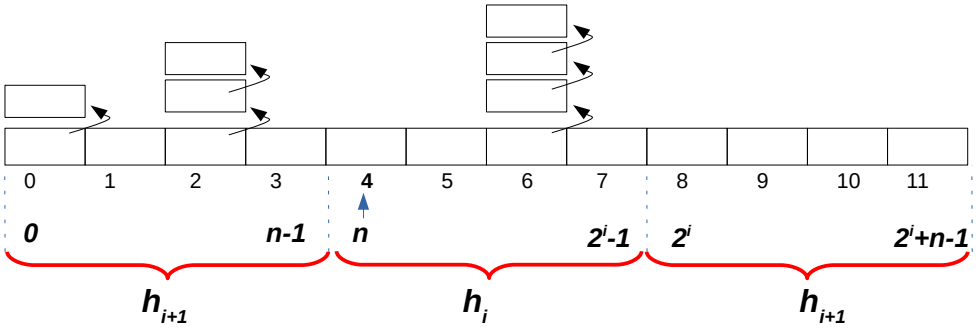
La suppression est réalisée de manière complètement symétrique à l'insertion :
quand une certaine condition est vérifiée (par exemple, quand le facteur de chargement global devient inférieur à un certain seuil S_{min}), on fusionne le bloc ***n-1*** avec le dernier bloc du fichier, on décrément ***n*** et si ***n*** devient < 0, on décrémente ***i*** et on remet ***n*** au dernier bloc de la zone adressable par h_i (2^i-1). A chaque fois que ***i*** est décrémenté, la taille de ***F*** aura été divisée par 2.




```
Sup( x )
  Rech(x) ⇒ (trouv , k )
  SI ( trouv )
        Supprimer x dans le bloc k ou dans sa zone de débordement
        SI ( facteur de charg. < Smin  && i > 0 )
              // on réalise une FUSION : l'inverse d'un éclatement...
              n--
              SI ( n < 0 )
                    i--
                    n ← 2^i - 1
              FSI
              - transférer le contenu du bloc 2^i + n (et éventuellement de sa liste de
                débordement) dans le bloc n (et sa liste de débordement s'il est plein)
              - libérer le bloc 2^i + n et sa liste de débordement si elle existe
        FSI
  FSI
```

Explications de l'algorithme de localisation Adr(x) utilisé dans les différentes opérations d'accès (recherche, insertion et suppression)

Pour bien comprendre la principe de l'algorithme **Adr(x)** permettant de localiser ***x***, il faut d'abord comprendre comment le fichier ***F*** est découpé en parties accessibles par les fonctions de hachage h_i et h_{i+1} comme dans la figure ci-dessous :



Dans cet exemple le fichier est formé par 12 blocs (0, 1, … 11), certains blocs ont débordés (0, 2 et 6). Les valeurs des paramètres actuels sont ***n*** = 4 (le prochain bloc à subir une opération d'éclatement) et ***i*** = 3 (le fichier a doublé de taille 3 fois).
Cela veut dire donc que ***n*** a déjà complètement parcouru le fichier (entre 0 et 2^i) 3 fois et on est actuellement dans le 4e passage.

Au début du 4e passage, ***n*** était à 0, ensuite il y a eu un éclatement (celui du bloc 0) et ***n*** est passé à 1, ensuite le bloc 1 a éclaté et ***n*** est passé à 2 … etc jusqu'à atteindre le bloc 4 (l'état actuel). Aux prochains éclatements (les blocs 4, 5, 6 et 7), ***n*** va continuer à avancer jusqu'à atteindre le bloc 8 (2^3), A ce moment, le fichier aura doublé de taille pour la 4e fois, le paramètre ***i*** devra être incrémenté et le paramètre ***n*** réinitialisé à 0 pour entamer un 5e passage (de 0 à 2^4 = 16) et ainsi de suite …

Donc à l'état actuel (***n*** = 4 et ***i*** = 3) on sait que les blocs 0, 1, 2 et 3 ont déjà éclatés (c-a-d leurs contenus ont été rehacher avec h_{i+1} et dans ce cas, si on doit par exemple rechercher un enregistrement qui se trouve dans l'un des ces blocs, il faut le localiser avec la fonction h_{i+1}). Alors que les blocs 4, 5, 6 et 7 n'ont pas encore éclatés et donc pour trouver les enregistrements qui y sont stockés, il faut utiliser la fonction h_i uniquement. Les blocs qui restent (c-a-d entre 8 et la fin du fichier : $2^i + n - 1 = 11$) ne sont que les résultats des éclatements qui se sont produits pour les blocs 0,1, 2 et 3. Donc leurs contenus peuvent être retrouvés à l'aide de la fonction h_{i+1}.

En résumé donc, quand on veut localiser un enregistrement ***x***, on calcule d'abord le numéro de bloc avec h_i (cela retourne forcément un numéro entre 0 et 2^i-1), si le numéro trouvé est inférieur à ***n***, on utilise alors la fonction h_{i+1} (car cela concerne un bloc qui a déjà éclaté). C'est le principe de l'algorithme **Adr(x)** vu au début de la section.

Chapitre 6 Opérations de haut niveau sur les fichiers

Les systèmes de bases de données utilisent des langages de requêtes de haut niveau pour la consultation de grands volumes de données.
Le langage SQL de consultation de bases de données, permet d'exprimer des requêtes simplement à un niveau d'abstraction élevé. Voici un exemple de requête en SQL :

Select *E.nom, E.prenom, N.moy, M.code*
From *Etudiant E , Notes N , Module M*
Where *E.mat = N.mat* ***and*** *N.code = M.code* ***and*** *M.annee = 2018*
Order by *N.moy* ***desc***

L'exécution de ce type de requêtes passe par une traduction en opérations de base (*scan*, *tri*, *jointure*…). Il existe un ou plusieurs algorithmes pour chacune de ces opérations de base.

L'implémentation efficace de ces opérations de base dépend souvent de la taille de la zone mémoire de travail allouée (en MC).

Nous allons voir dans ce chapitre l'implémentation de quelques opérations de base utilisées dans le traitement des requêtes de haut niveau.

1) Opération du tri séquentiel d'un fichier

Le tri d'un fichier de données est une opération très utilisée aussi bien dans les requêtes elles mêmes (les clauses *order by* et *unique*) que dans l'implémentation d'autres opérations de base aussi.

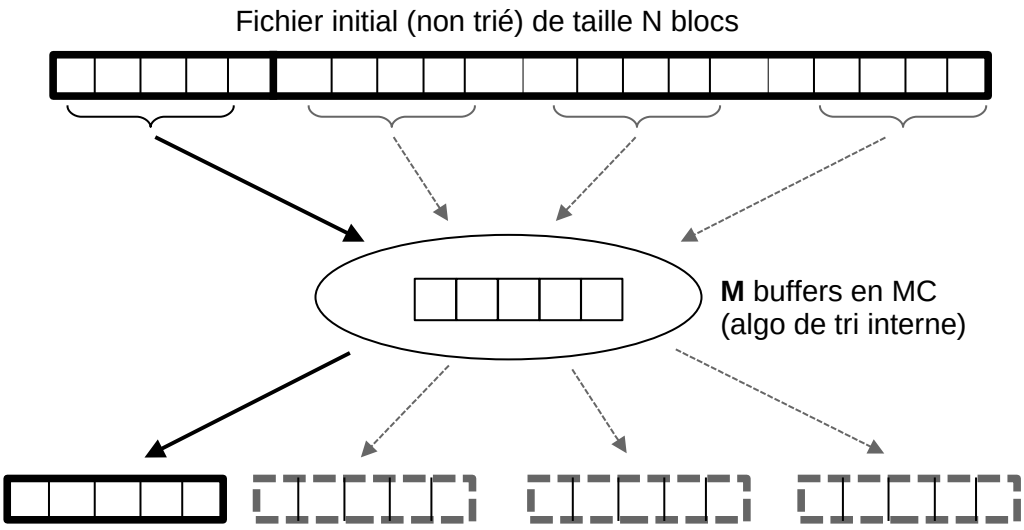
Si on dispose d'une zone en **MC** de ***M*** buffers, on peut alors trier un fichier formé de ***N*** blocs (souvent ***N >> M***) avec un coût = $\boldsymbol{2N\ (Log_{M-1}[N/M] + 1)}$.
Donc avec une complexité → ***O(N log(N))***.

Cet algorithme de ***tri externe*** se compose de deux étapes : *(i) Fragmentation* et *(ii) Fusion*.

(i) La première étape consiste à 'découper' le fichier à trier en ***[N/M]*** fragments ordonnées en **MS** (en utilisant un algorithme de tri interne pour ordonner chaque fragment en **MC**).

(ii) La deuxième étape consiste en une ou plusieurs phases de *multi-fusion* (fusions par groupe de ***M-1*** fragments). La fusion d'un groupe utilise en **MC**, ***M-1*** buffers d'entrée et ***1*** buffer de sortie.

A chaque phase de *multi-fusion*, le nombre de fragments est donc divisé par ***M-1***. Les phases de *multi-fusion* se répètent jusqu'à l'obtention d'un seul fragment (de taille ***N*** blocs) représentant le résultat final du tri externe.

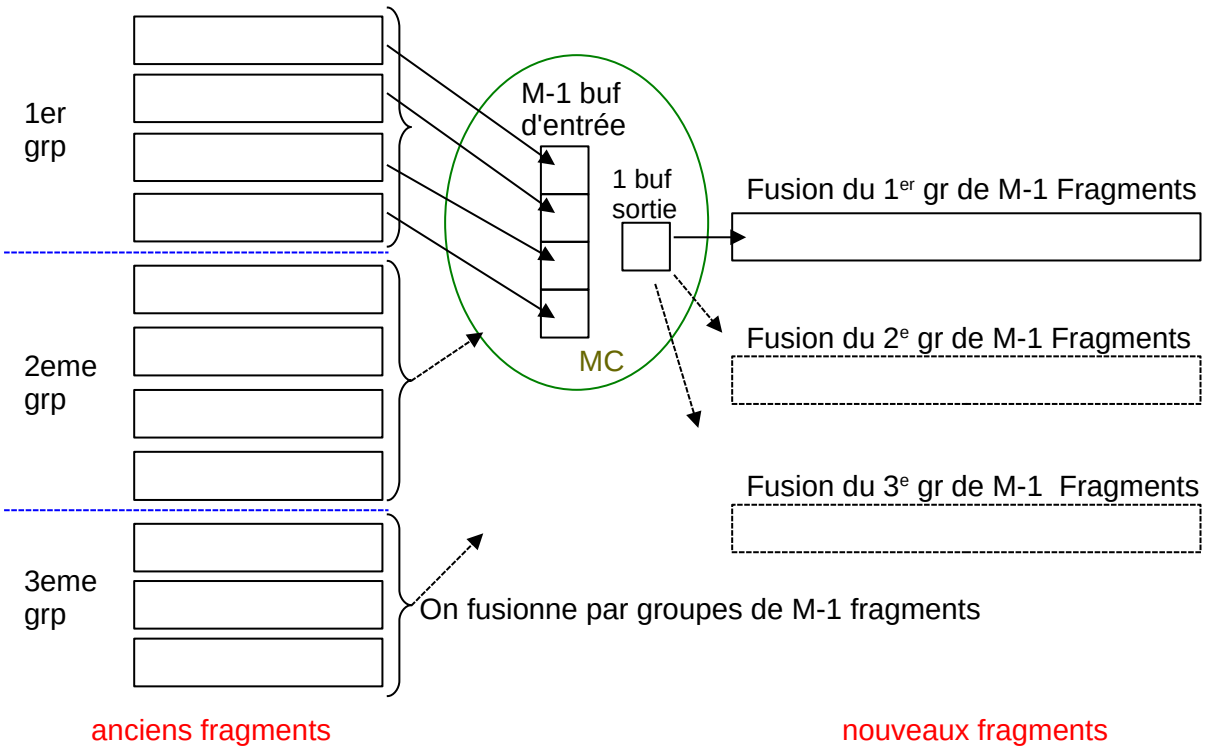


[N/M] Fragments initiaux triés, de taille M blocs chacun

Fragmentation du fichier en entrée

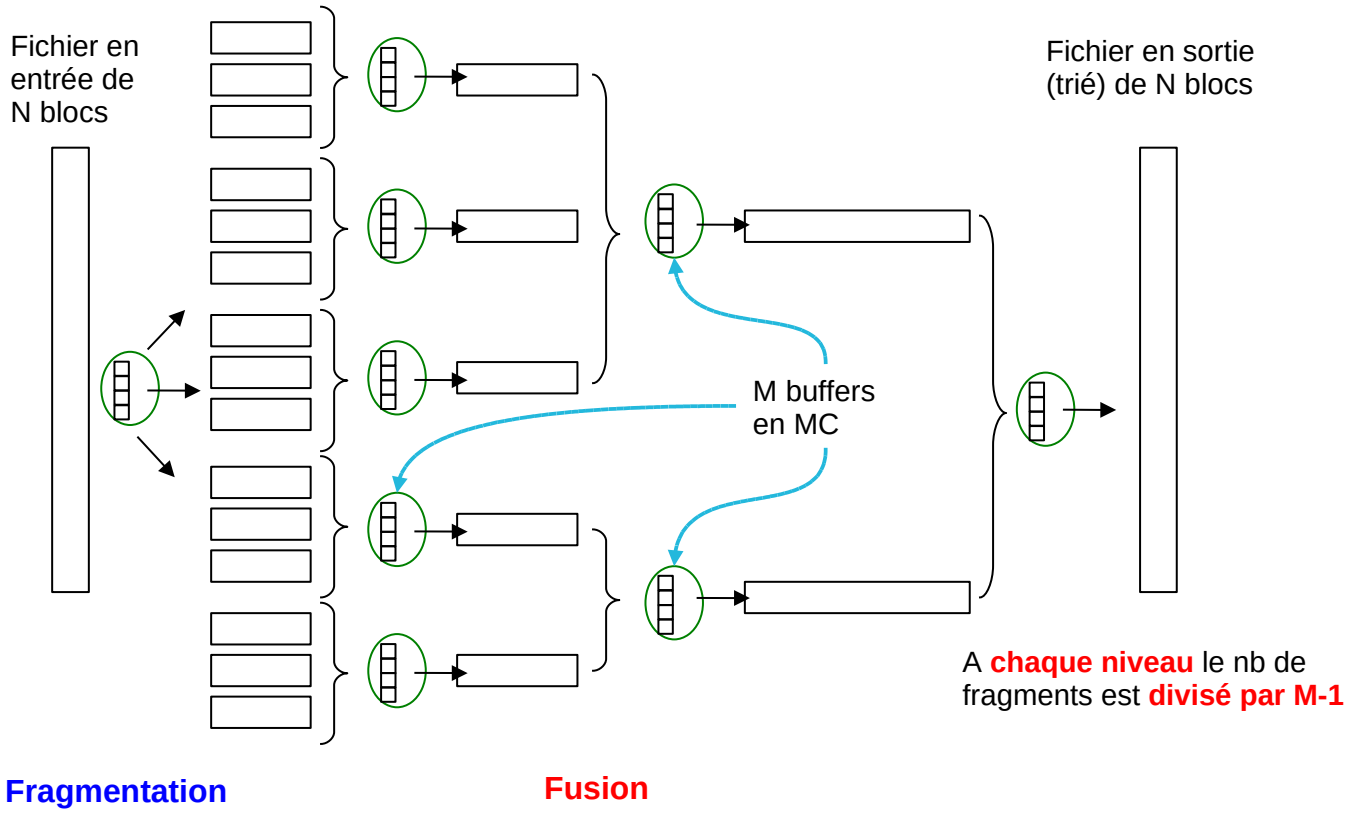
La première étape (fragmentation du fichier de ***N*** blocs en ***[N/M]*** fragments ordonnés) coûte ***2N*** opérations d'accès (***N*** lectures et ***N*** écritures), car tous les blocs du fichier en entrée sont lus une seule fois et réécrits une seule fois dans de nouveaux fichiers appelés 'Fragments'.

Dans la deuxième étape, chaque phase de *multi-fusion* nécessite la lecture de tous les blocs des fragments en entrées (soit ***N*** blocs) et l'écriture d'autant de blocs réparties sur moins de fragments mais plus longs (le nombre total de blocs reste donc ***N***). Chaque phase de multi-fusion coûte ainsi ***2N*** accès.



Une Phase de Multi-Fusion

Comme à chaque phase de *multi-fusion*, le nombre de fragments est divisé par ***M-1***, le nombre total de phases nécessaires pour avoir un seul fragment résultat est de l'ordre de $\boldsymbol{Log_{M-1}[N/M]}$.



Le **coût total du tri externe** d'un fichier de taille ***N*** blocs en utilisant ***M*** buffers en **MC** est donc :

$\boldsymbol{2N + 2N * Log_{M-1}[N/M]}$ accès disque.

Les fragments temporaires générés par l'algorithme du tri externe, peuvent être gérés comme dans une file d'attente (*FIFO*) de fragments. L'algorithme général du tri externe d'un fichier ***F*** formé par ***N*** blocs physiques aura alors la forme suivante :

```
/* Algorithme du Tri externe */
// F est le fichier à trier de taille N blocs
// buf est tableau de M buffers représentant l'espace MC alloué pour le tri

/* Etape 1 : fragmentation du fichier F en N/M fragments triés */
i = 1 ; j = 1 ; k = 1
TQ (i ≤ N )
        SI ( j ≤ M )
                LireBloc( F, i, buf[i] ) ; i++ ; j++
        SINON
                frag_k = Tri_interne( buf, 1, j-1 )      // tri interne de buf[1], buf[2], … buf[j-1]
                Enfiler( frag_k , FIFO )
                k++
                j = 1
        FSI
FTQ
```

// Ecriture du dernier fragment
$frag_k$ = Tri_interne(buf, 1, j-1) *// tri interne de buf[1], buf[2], … buf[j-1]*
Enfiler($frag_k$, FIFO)

/ **Etape 2 :** Phases de multi-fusions … jusqu'à l'obtention d'un seul fragment */*
***TQ** (la file fifo contient plus d'un élément)*
// retirer de la file les M-1 premiers fragments (ou moins, s'il n'y a pas assez)
Defiler_groupe(M-1, r_1, r_2, … r_j , FIFO) *// r_1, r_2, … r_j : les j fragments défilés*
k++
*$frag_k$ = **Multi_Fusion**(r_1, r_2, … r_j)*
Enfiler($frag_k$, FIFO) *// Enfiler le résultat $frag_k$ de la multi-fusion*
***FTQ** ;*

/ A la fin de la boucle la file FIFO contient un seul fragment : le résultat final */*
Defiler(resultat , FIFO)

/* Fin de l'algorithme du tri externe */

Cette manière de procéder (gestion des fragments temporaires en *FIFO*) permet d'optimiser le nombre d'itérations de l'étape de multi-fusion. A chaque itération on fusion ***M-1*** fragments temporaires, même si on est à la fin d'une phase de la multi-fusion et que le nombre de fragments restants dans cette phase est inférieur à ***M-1***. Le mécanisme de la file *FIFO* nous permet dans ce cas particulier de compléter le nombre de fragments à fusionner (pour atteindre ***M-1*** fragments) en prélevant des nouveaux fragments de la prochaine phase (qui se trouvent forcément en tête de file).

Ci-dessous les grandes lignes de **l'algorithme de la multi-fusion** d'un groupe de ***k*** fichiers ordonnés, en utilisant ***k+1*** buffers en mémoire centrale (**MC**) :

*// En entrée : **k** fichiers ordonnés **F1, F2, … Fk** de tailles respectives **N1, N2, ...Nk** blocs*
*// En sortie : un fichier ordonné **G** contenant tous les enregistrements des fichiers en entrée*
*// **buf** est un tableau de **k+1** buffers représentant l'espace **MC** alloué pour la multi-fusion*
*// **ind** est un tableau de **k+1** entiers représentants les indice de parcours des buffers associés*
*// **numBloc** est un tableau de **k+1** entiers représentants les numéros de blocs des **k+1** fichiers*

La fonction ***PlusPetit-existe(e)*** récupère dans ***e*** le plus petit enregistrement parmi :
buf[1].Tab[ind[1]] , buf[2].Tab[ind[2]] , … , buf[k].Tab[ind[k]]
L'indice associé à ***e*** (***ind[i]***) est incrémenté en conséquence, et s'il dépasse le nombre d'éléments dans le buffer associé, le prochain bloc du fichier d'entrée associé (***F[i]***) sera lu et l'indice associé (***ind[i]***) sera remis à ***1***.
Si le fichier associé (***F[i]***) a été complètement lu (c-a-d ***numBloc[i]*** > ***Ni***), il ne sera plus considéré.
Si tous les fichiers en entrée ont été complètement lus, la fonction PlusPetit-existe retournera ***faux***.

/* Algorithme de la multi-fusion */

/ Lecture des premiers blocs de chaque fragment et initialisations */*
***Pour** i = 1..k*
LireBloc(Fi , 1 , buf[i])
ind[i] = 1 *// en initialisant les indices de parcours et*
numBloc[i] = 1 *// les numéros de blocs (fichiers en entrée)*
***FP** ;*

```
ind[k+1] = 1 ;   numBloc[ k+1 ] = 1          // pour le fichier sortie aussi

/* Boucle principale de la multi-fusion : k buffers en entrée vers 1 buffer de sortie */
TQ  ( PlusPetit-existe( e ) )   // retourne dans e le plus petit enreg non encore considéré
                                // dans les k buffers d'entrée.
        buf[ k+1 ].Tab[ ind[k+1] ] = e       // mettre e dans le buffer de sortie
        ind[k+1]++
        SI ( ind[ k+1 ] > b )                // écrire le buffer de sortie s'il est plein
                EcrireBloc( G , numBloc[k+1], buf[k+1] )
                numBloc[k+1]++
                ind[k+1] = 1
        FSI
FTQ

/* Ecriture du dernier bloc de G */
SI (ind[ k+1 ] > 1 )   EcrireBloc( G, numBloc[k+1], buf[k+1] )  FSI

/* Ecritures des entêtes et Fermetures des fichiers */
…

/* Algorithme de la multi-fusion */
```

2) Opération de jointure séquentielle de deux fichiers

La jointure de deux fichiers est un genre de produit cartésien avec un filtre, sous forme de condition permettant de sélectionner les enregistrements à concaténer dans le résultat final.

Soient ***F1*** et ***F2*** deux fichiers de tailles respectives ***N1*** et ***N2*** blocs. La jointure de ***F1*** et ***F2*** selon une condition ***C***, consiste pour chaque couple d'enregistrements **<*e1,e2*>** de ***F1*** et ***F2***, à produire un nouvel enregistrement en concaténant ***e1*** et ***e2*** dans un nouveau fichier résultat, à condition que les attributs de ***e1*** et ***e2*** vérifient la condition donnée ***C***.

Dans le cas général, la condition ***C*** est formée d'opérateurs binaires faisant intervenir un opérande de chaque fichier : *attrF1* <*op*> *attrF2*, avec <*op*> un opérateur binaire : = , < , > , ≥ , ≤ …

Exemple : Soit à joindre deux fichiers ***F1*** et ***F2***.
F1*(mat, nom, prenom)* représente un fichier d'étudiants caractérisés par un matricule (*mat*), un *nom* et un *prénom*.
F2(*mat, code, moy*) représente un fichier de notes où chaque enregistrement représente la moyenne (*moy*) obtenue par un certain étudiant (*mat*) dans un certain module (*code*)

F1

mat	nom	prenom
100	aaaa	bbbb
200	cccc	dddd
300	eeee	fffff
400	gggg	mmm

F2

mat	code	moy
400	sfsd	12
400	poo	14
100	sfsd	15
300	stat	11

La jointure de ***F1*** et ***F2*** sur la condition : (*F1.mat = F2.mat)* donne le résultat suivant :

F1.mat	nom	prenom	F2.mat	code	moy
100	aaaa	bbbb	100	sfsd	15
300	eeee	fffff	300	stat	11
400	gggg	mmm	400	sfsd	12
400	gggg	mmm	400	poo	14

Pour implémenter l'opération de jointure de deux fichiers, nous supposerons par la suite que nous disposons d'un espace en **MC** de taille suffisante pour maintenir ***M*** buffers (généralement ***M*** << ***N1*** et ***M*** << ***N2***).

Il existe trois grandes classes d'algorithmes de jointure :
'*Boucles-Impriquées*', '*Tri-Fusion*' et '*Hachage*'.

- Algorithme de jointure par '*Boucles Imbriquées*' (Nested-Loop join algorithm)

C'est l'algorithme de base pour la jointure dans le cas général (quelque soit le type de la condition). Cet algorithme est composé de deux boucles imbriquées : une boucle externe et une boucle interne.
Dans la boucle externe, on parcourt ***F1*** en chargeant, à chaque itération, ***M-2*** blocs à la fois.
Pour chaque fragment de ***F1*** en **MC** (occupant ***M-2*** buffers), on parcourt dans la boucle interne le fichier ***F2*** bloc par bloc (***1*** buffer) et on réalise la jointure (en **MC**) entre les ***M-2*** buffers de ***F1*** et l'unique buffer de ***F2***. Un dernier buffer est utilisé pour l'écriture des résultats (buffer de sortie)

Ci-dessous un pseudo-algorithme de type 'boucles imbriquées' :

```
/* Boucle externe pour parcourir F1 */
Pour chaque fragment Fr de F1 (lecture en MC de M-2 blocs de F1)

    /* Boucle interne pour parcourir F2 */
    Pour chaque bloc B de F2 (lecture en MC d'un bloc de F2)

        /* jointure en MC de Fr avec B */
        Pour chaque enreg e1 dans Fr
            Pour chaque enreg e2 dans B
                Si C(e1,e2) rajouter <e1.e2> au buffer de sortie R
                Si le buffer R est plein, le vider sur disque
            FP
        FP

    FP /* fin boucle interne */

FP /* fin boucle externe */
```

Le fichier ***F1*** est lu une seule fois (soit un coût de ***N1*** lectures). A chaque itération de la boucle externe, un fragment de ***F1*** est lu (de taille ***M-2*** blocs).
Le nombre d'itérations de la boucle externe est donc : ***[N1 / (M-2)]***. (la parie entière supérieure)

Le fichier ***F2*** est lu entièrement (dans la boucle interne) pour chaque itération de la boucle externe.

Le coût en opération de lecture est alors = ***N1 + N2 * [N1 / (M-2)]*** lectures de blocs.

Le coût en écriture (α) dépend de la condition **C**. Dans le pire des cas (***C*** est vérifiée pour chaque couple d'enregistrements ***<e1,e2>*** de ***F1*** et ***F2***). Le coût en écriture serait alors ***(n1 * n2) / b***, avec ***n1*** et ***n2*** les nombres d'enregistrements des fichiers ***F1*** et ***F2*** et ***b*** la capacité maximale d'un bloc. Si par contre la condition ***C*** n'est vérifiée par aucun couple, α vaut ***0***. C'est le cas le plus favorable.

Le coût total de l'algorithme de jointure par '**boucles imbriquées**' est donc :

N1 + N2 * [N1/(M-2)] + α opérations d'E/S.

- Algorithme de jointure par '*Tri-Fusion*' (Sort-Merge join algorithm)

L'idée dans l'algorithme de jointure par *tri-fusion* est de trier d'abord les fichiers ***F1*** et ***F2***, puis parcourir les deux fichiers triés (en une seule passe) de la même manière que l'algorithme de *fusion*. Cette algorithme ne s'applique que si la condition de jointure est l'égalité des attributs (c-a-d de la forme *attrF1 = attrF2*). On parle alors **d'équi-jointure**.

L'algorithme nécessite, au minimum, ***3*** buffers en **MC**. Deux buffers pour parcourir les deux fichiers en entrée et un 3^{e} buffer pour la sortie (écriture du résultat dans le fichier de sortie), donc ***M = 3***. Le traitement des valeurs multiples nécessite cependant un espace supplémentaire en **MC** pour éviter les éventuels retours arrières générant des opérations de lectures physiques supplémentaires.

Ci-dessous un algorithme possible tenant compte du traitement (en **MC**) des valeurs multiples :

```
/* Algorithme de jointure : Tri-Fusion */
/* Déclaration des fichiers et zones tampons: */
        Tbloc1 = struct
                        tab[b1] de Tenreg1
                        NB : entier
                Fin
        Tbloc2 = struct
                        tab[b2] de Tenreg2
                        NB : entier
                Fin
        Tbloc3 = struct
                        tab[b3] de Tenreg3
                        NB : entier
                Fin
        F1 : Fichier de Tbloc1 buffer buf1,  entete(entier /* num du dernier bloc */ )
        F2 : Fichier de Tbloc2 buffer buf2, entete(entier /* num du dernier bloc */ )
        F3 : Fichier de Tbloc3 buffer buf3, entete(entier /* num du dernier bloc */ )
        E : Tableau[m] de Tenreg2  /* m représente le plus grand nombre d'occurrence des
                                      valeurs multiples dans F2 */

Ouvrir(F1, "...", 'A')
Ouvrir(F2, "...", 'A')
Ouvrir(F3, "...", 'N')
i1 = 1; i2 = 1; i3 = 1
j1 = 1; j2 = 1; j3 = 1
lireDir(F1, 1, buf1)
lireDir(F2, 1, buf2)
FinF1 = faux
```

```
FinF2 = faux
TQ ( Non FinF1 && Non FinF2 )

        ts = buf2.tab[j2];    k = 1;    E[k] = ts;    k++;    j2++
        Si (j2 > b2)
                i2++
                Si ( i2 ≤ entete(F2,1) )
                        lireDir(F2, i2, buf2)
                        j2 = 1
                Sinon
                        FinF2 = vrai
                Fsi
        Fsi

        stop = faux
        TQ ( non stop && non FinF2 )
                t = buf2.tab[j2]
                Si (t.attr == ts.attr)          /* attr : attribut de jointure */
                        E[k] = t
                        k++
                        j2++
                        Si (j2 > b2)
                                i2++
                                Si ( i2 ≤ entete(F2,1) )
                                        lireDir(F2, i2, buf2)
                                        j2 = 1
                                Sinon
                                        FinF2 = vrai
                                Fsi
                        Fsi
                Sinon
                        stop = vrai
                Fsi
        FTQ  // ( non stop && non FinF2 )

        tr = buf1.tab[j1]

        TQ ( non FinF1 && tr.attr < ts.attr )
                j1++
                Si (j1 > b1)
                        i1++
                        Si ( i1 ≤ entete(F1,1) )
                                lireDir(F1, i1, buf1)
                                j1 = 1
                        Sinon
                                FinF1 = vrai
                        Fsi
                Fsi
                tr = buf1.tab[j1]
        FTQ  // ( non FinF1 && tr.attr < ts.attr )
```

```
TQ ( non FinF1 && tr.attr == ts.attr )        // Test de la condition de jointure C
        POUR (i=1,k-1)
                ts = E[i]
                /* ajouter <tr.ts> dans le résultat */
                buf3[j3] = <tr.ts>  // concaténation de tr et ts
                j3++
                Si (j3 > b3)    buf3.NB = b3
                                ecrireDir(F3, i3, buf3)
                                buf3[1] = <tr.ts>
                                i3++
                                j3 = 2
                Fsi
        FP
        j1++
        Si (j1 > b1)
                i1++
                Si ( i1 ≤ entete(F1,1) )    lireDir(F1, i1, buf1);    j1 = 1
                Sinon  FinF1 = vrai
                Fsi
        Fsi
        tr = buf1.tab[j1]
FTQ  // ( non FinF1 && tr.attr == ts.attr )

FTQ  // ( Non FinF1 && Non FinF2 )

/* Dernière écriture dans F3 */
Si (j3 > 1)     buf3.NB = j3 - 1
                ecrireDir(F3, i3, buf3)
                i3++
Fsi

Aff_entete(F3, 1, i3 - 1)  // numero du dernier bloc de F3
Fermer(F1);   Fermer(F2);   Fermer(F3)
```

/* Fin de l'algorithme de jointure par tri-fusion */

Le coût de cet algorithme, en opérations de lectures, est celui du tri des deux fichiers en entrée plus celui de la fusion :

$N_1(1 + Log_{M-1}[N_1/M]) + N_2(1 + Log_{M-1}[N_2/M]) + (N_1 + N_2)$ lectures de blocs

–-- tri de F1 ---- –--tri de F2 ---- -Fusion-

Le coût en opérations d'écriture est celui associé au tri des 2 fichiers plus celui de la génération du résultat de la jointure lors de la phase de fusion (dépend de la condition ***C***). Appelons α ce dernier coût dépendant de la condition. On aura donc :

$N_1(1 + Log_{M-1}[N_1/M]) + N_2(1 + Log_{M-1}[N_2/M]) + \alpha$ écritures de blocs

–-- tri de F1 ---- –--tri de F2 ---- -Fusion-

Donc le coût total (en nombre de lectures et d'écritures) de **l'algorithme de jointure 'Tri-Fuision'**, en supposant que le fichier résultat occupera α blocs, est estimé à :

$2N_1(1 + Log_{M-1}[N_1/M]) + 2N_2(1 + Log_{M-1}[N_2/M]) + (N_1 + N_2 + \alpha)$ op. d'E/S.

- Jointure par Hachage (Hash join algorithm)

Le principe de cet algorithme **d'équi-jointure** (la condition ***C*** doit être de type « égalité » aussi : *F1.attr = F2.attr*) est de fragmenter les deux fichiers en entrée à l'aide d'une fonction de hachage ***h***, de sorte qu'un enregistrement ***e***, soit stocké dans le fragment d'indice ***h(e.attr)***. On obtient alors les fragments suivants :

$F1 = < r_0 , r_1 , r_2 , \ldots r_{n-1} >$

$F2 = < s_0 , s_1 , s_2 , \ldots s_{n-1} >$

La jointure de *F1* avec *F2* se réalise ensuite avec plusieurs petites jointures indépendantes :

$r_0 * s_0 , \quad r_1 * s_1 , \quad \ldots \quad r_{n-1} * s_{n-1}$

Chacune de ces petites jointures ($r_i * s_i$) est supposée être réalisable en **MC**. On doit donc disposer d'un espace en **MC** ayant une taille suffisante pour charger le plus grand fragment d'un des deux fichiers en entrée.

Ci-dessous un pseudo-algorithme de jointure par hachage :

// Pour le partitionnement on utilise n buffers de sortie en MC
Partitionner F1 en $r_0, r_1, \ldots r_{n-1}$;
Partitionner F2 en $s_0, s_1, \ldots s_{n-1}$;

POUR *i = 0 , n-1*
    / Construction d'un index par hachage du fragment 's_i' :* **étape du Build** **/*
    Lire bloc par bloc, le frag 's_i' et construire un index par hachage en MC
    (en utilisant une autre fonction de hachage, différente de h)

    / Jointure entre les enreg de 'r_i' et ceux de 's_i' :* **étape du Probe** **/*
    POUR *chaque enreg tr dans le frag 'r_i' (lire 'r_i' bloc par bloc)*
        - récupérer avec l'index en MC les enreg ts ayant la même val d'attr que tr (=condition de jointure C)
        *- **POUR** chaque ts récupéré*
            rajouter la concaténation <ts.tr> dans le résultat
        FP
    FP
FP

Dans la phase de partitionnement, les deux fichiers d'entrée (***F1*** et ***F2***) sont lus une seule fois (donc ***N1 + N2*** lectures de blocs). Le nombre de blocs utilisés pour former l'ensemble des fragments r_i et s_i est au minimum égal à ***N1 + N2*** blocs et au maximum ***N1 + N2 + 2n*** blocs (dans ce dernier cas, chaque fragment se termine par un bloc additionnel non plein). De ce fait, le coût en opérations de lecture et écriture de la phase de partitionnement est borné par : ***2(N1 + N2) + 2n*** op. d'E/S.

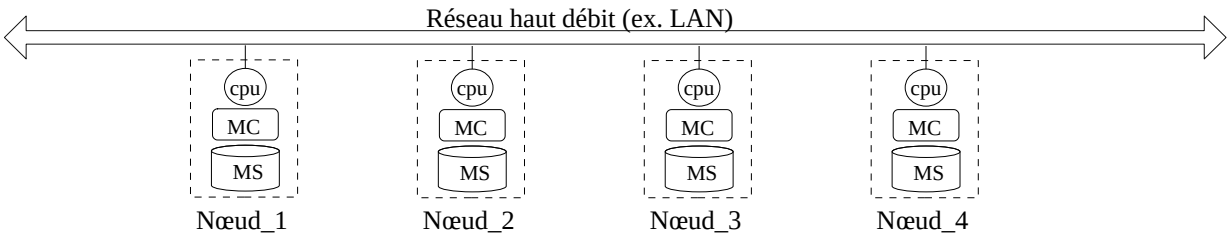
Dans la deuxième partie de l'algorithme, les étapes 'Build' et 'Probe' nécessitent la lecture de tous les fragments r_i et s_i une seule fois. Soit au maximum ***N1 + N2 + 2n*** opérations de lectures.

La taille (α) du fichier résultat dépend principalement de la condition ***C***. Ainsi le coût total de l'algorithme de jointure par hachage est : ***3(N1 + N2) + 4n + α*** opérations d'E/S.

3) Parallélisation des opérations sur les fichiers

Souvent les accès aux données externes (les fichiers) représentent le principal goulot d'étranglement pour les performances des applications de traitement de données massives. La répartition des données sur différents nœuds de calcul (architecture en cluster par exemple) permet souvent d'améliorer l'efficacité si la parallélisation est bien faite.

Nous nous intéressons ici à l'architecture dite 'à mémoire distribuée' (shared nothing) où chaque nœud du cluster renferme une unité de calcul (**CPU**), une mémoire centrale (**MC**) et une mémoire secondaire (**MS**). Les nœuds sont supposés être connectés par un réseau haut débit. La communication se fait exclusivement par *envoi de messages* (message passing model).



Cluster de 4 nœuds

Pour programmer des applications sur ce type d'architecture, on peut utiliser les API systèmes déjà existantes sur chaque nœud (comme par exemple la combinaison : Sockets *BSD* et Processus locales, sur un réseau *TCP/IP*). On peut aussi utiliser des librairies spécifiques au modèle 'Message Passing' (comme *MPI* ou *PVM*) ou alors des frameworks de parallélisation automatique (comme *MapReduce* ou *Spark*). Les deux premiers outils (Socket/Processus et Message-Passing) sont destinés à des programmeurs de haut niveau technique, maîtrisant les aspects théoriques de la programmation parallèle (Les ordres partiels, la synchronisation, la théorie de la sérialisabilité, le recouvrement …). Ils sont un peu difficiles à utiliser mais permettent généralement des solutions efficaces et très optimisées. Les frameworks de parallélisation automatiques sont beaucoup plus facile à utiliser et sont destinés principalement à des programmeurs n'ayant pas obligatoirement de telles connaissances théoriques sur la programmation parallèle. Leur but est d'utiliser le plus facilement et le plus rapidement possible un cluster à travers un langage séquentiel bien maîtrisé. La parallélisation (souvent simpliste mais à grande échelle) est en fait effectuée automatiquement par le framework d'exécution, déchargeant le programmeur des aspect complexes liés à la répartition de charges, de synchronisation, de recouvrement, ...etc.

Les solutions parallèles des opérations sur fichiers, peuvent améliorer très significativement les performances si les données sont bien réparties entre les différents nœuds du cluster. L'accélération obtenue peut ainsi être maximale lorsque les différentes ressources disponibles sont bien exploitées. Une mauvaise répartition des données peut facilement provoquer un déséquilibre de charge où certains nœuds auront beaucoup de travail à faire alors que d'autres seront sous-utilisés, induisant une utilisation non optimisée du cluster. Les effets d'un mauvais équilibrage de charge ou d'une forte contention sur les ressources partagées peuvent induire une très forte dégradation des performances.

Nous allons présenter dans ce qui suit, les approches de parallélisation des opérations de tri, de multi-fusion et de jointure. Nous terminerons cette section par un survole rapide des traitements parallèles en présence d'opérations de mises à jours tels que rencontré dans le cadre des transactions concurrentes et/ou réparties (*OLTP*) et un mot sur le contexte '*Big Data*' nécessitant les traitements

de consultation massivement parallèles.

- Le tri parallèle

Il y a deux grandes approches pour la parallélisation de l'algorithme du tri externe sur un cluster formé de ***k*** nœuds :

1- On peut *découper équitablement* (en nombre d'enregistrements) le fichier à trier en ***k*** fragments de tailles similaires. Chaque fragment $frag_i$ sera trié au niveau d'un nœuds N_i du cluster. Ainsi les différents fragments seront triés en parallèle. Les résultats des tris des différents fragments sont ensuite récoltés et ***fusionnés*** pour reconstituer le fichier final trié.

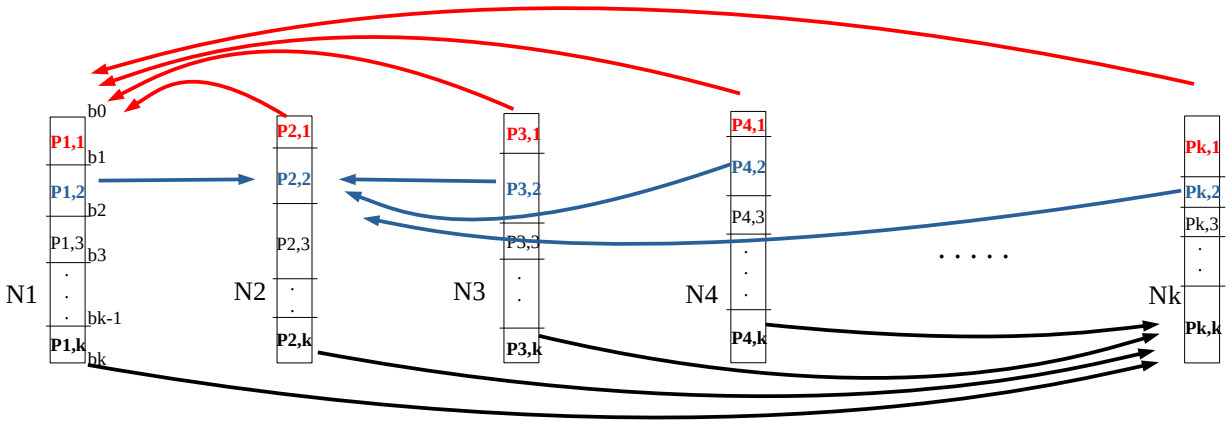
2- On peut aussi partitionner le domaine des valeurs de la clé en ***k*** intervalles ($Interv_1$, $Interv_2$ … $Interv_k$) et *fragmenter par valeurs* le fichier à trier, de telle sorte à former ***k*** fragments. Chaque fragment $frag_i$ ne contient que des valeurs appartenant à l'intervalle $Interv_i$ et sera trié au niveau du nœud N_i du cluster. A la fin, on récupère les résultats des différents tris pour les ***concaténer*** et former le fichier final trié.

Nous pouvons remarquer que dans la première approche, les différents nœuds auront une charge de travail équitable (car les fragments sont pratiquement de même taille), par contre l'opération de fusion (en $O(n)$) des résultats se déroulera en séquentiel sur un seul nœud, ce qui engendre au final un fort déséquilibre de charge et une sous-utilisation des ressources disponibles. Ce problème est évité dans la deuxième approche, car la concaténation des résultats est une opération très rapide (en $O(1)$). Par contre la fragmentation par valeurs (en intervalles) nécessaire pour le lancement de la deuxième approche détériore la qualité de la solution car elle doit s'effectuer (en $O(n)$) dans un seul nœud.

Ces deux solutions naïves peuvent être améliorées. Pour la première solution, l'opération de fusion finale peut se réaliser en parallèle sur plusieurs nœuds (voir fusion parallèle ci-dessous). Pour la deuxième solution, la fragmentation par intervalle peut se faire de manière incrémentale. Les données sont envoyés par petits blocs aux différents nœuds au fur et à mesure que le fichier en entrée est parcouru par le nœuds initial. Cela permet à chaque nœud de calcul d'entamer le processus de tri sans attendre que tout le fragment soit disponible dans sa mémoire locale.

Fusion parallèle (c'est une multi-fusion parallèle de ***k*** fragments triés) :

Chaque nœuds ***N1, N2, …. Nk*** dispose d'un fragment trié dans sa mémoire locale. Le résultat de la multi-fusion sera transmis à un nœud maître ***M*** (qui peut être l'un des nœuds ***N1, N2, ...Nk***).



Pour réaliser la fusion de ces ***k*** fragments en parallèle, l'**un des nœuds** (***N1*** par exemple) effectue ce qui suit :

1) Découpage du fragment trié en k parties égales :
$P_{1,1}$ =]-∞ – b_1] , $P_{1,2}$ =]b_1 – b_2] , … , $P_{1,k}$ =]b_{k-1} – +∞[
k-1 bornes sont ainsi déterminées (b_1, b_2, … b_{k-1}).

2) Diffusion des bornes (b_1, b_2, … b_{k-1}) à tous les autres nœuds (N2, N3, … Nk).

3) Envoi à chaque nœud Nj (j=2...k), la $j^{ème}$ partie ($P_{1,j}$ =]b_{j-1} .. b_j]) du fragment local.

4) Attendre des autres nœuds les 1^{eres} parties de leurs fragments respectifs (les $P_{i,1}$ avec i =2...k).

5) Effectuer une multi-fusion (séquentielle) entre les parties reçues et celle locale au nœud :
Res_1 = MultiFusion($P_{1,1}$, $P_{2,1}$, $P_{3,1}$, … $P_{k,1}$)

6) Envoi du résultat Res_1 au nœud maître M.

De son coté, **chacun des autres nœuds *Ni*** (i = 2...k) doit effectuer les actions suivantes :

1) Attendre la réception des bornes (b_1, b_2, … b_{k-1}) provenant de N1

2) Pour chaque borne b_j (j =1...k-1), effectuer une recherche dichotomique dans le fragment local pour localiser l'indice où elle devrait se positionner : ind_j = Rech_dicho($Frag_i$, b_j)

3) A l'aide des indices trouvés ind_1 , ind_2 … ind_{k-1}, découper le fragment local ($Frag_i$) en k parties :
$P_{i,1}$, $P_{i,2}$, … $P_{i,k}$. Puis envoyer à chaque nœud Nj (j=1 … k et j ≠ i) la partie qui le concerne : $P_{i,j}$.

4) Attendre des autres nœuds les $i^{èmes}$ parties de leurs fragments respectifs ($P_{j,i}$ avec i =1...k et j≠i).

5) Effectuer une multi-fusion (séquentielle) entre les parties reçues et celle locale au nœud :
Res_i = MultiFusion($P_{1,i}$, $P_{2,i}$, $P_{3,i}$, … $P_{k,i}$)

6) Envoi du résultat Resi au nœud maître M.

Le **nœud maître *M*** concatène (en *O*(1)) les différents résultats reçus (res_1 , res_2 … res_k) des autres nœuds pour former le résultat final trié.

- La jointure parallèle

Pour réaliser une jointure entre deux fichiers ***R*** et ***S***, on utilise toujours le principe de la répartition des données entre les différents nœuds de calcul disponibles, puis on effectue des jointures indépendantes en parallèle dans chaque nœud de calcul et on termine en combinant les résultats obtenus.

Une première solution consiste à découper un des fichiers (***R*** par exemple) en parties de tailles égales (*r1, r2, … rk*). Puis on envoie à chaque nœud de calcul ***Ni***, la ***$i^{ème}$*** partie *ri* de ***R*** ainsi que le contenu du deuxième fichier ***S*** en entier pour que ***Ni*** puisse calculer la jointure ***ri * S***.
Les résultats des différentes jointures indépendantes ***ri * S*** sont ensuite concaténées au niveau du

nœud maître pour former le résultat.
Cette solution découpe la charge de travail de manière équitable, mais la taille des sous-problèmes générés (***S*** n'est pas découpé en parties) reste éventuellement importante (surtout en présence de fichiers volumineux). Cette première solution s'applique pour des jointures avec une condition quelconque (ainsi que les opérations analogues : *Résolution d'Entités* par exemple dans le cas général).

L'idée pour une deuxième solution plus efficace (mais restreinte uniquement aux équi-jointures), est de s'inspirer de l'algorithme de jointure par hachage pour proposer une approche où les deux fichiers ***R*** et ***S***, sont fragmentés par valeurs.

Pour fragmenter un fichier par valeurs, en ***k*** fragments, il y a deux possibilités :
- Soit utiliser une fonction de hachage ***h*** (avec un ensemble d'arrivées de cardinalité ***k***) pour obtenir ***k*** fragments contenant chacun des données ayant la même image par la fonction ***h*** (des synonymes).
- Soit partitionner le domaine des valeurs de l'attribut de jointure en ***k*** intervalles et utiliser les bornes de la partition pour fragmenter le fichier par valeur en ***k*** intervalles.

Dans les deux cas, on obtiendra des fragment de ***R*** et ***S*** où les équi-jointures ne sont possibles qu'entre fragments ayant le même indice :
R = r1 , r2 , r3 , …. rk
S = s1 , s2 , s3 , …. sk

Au niveau de chaque nœud Ni, on effectue une des petites jointures : ri * si

Les résultats des k jointures effectuées en parallèle seront concaténés au niveau du nœud maître :
$R * S = (r1 * s1) \cup (r2 * s2) \ldots \cup (rk * sk)$

L'inconvénient de cette deuxième solution est que la fragmentation par valeurs n'assure pas dans tous les cas, un découpage équitable en termes de taille des fragments. D'où une possible exécution parallèle déséquilibrée.

Beaucoup de travaux de recherche ont abordé ce problème de déséquilibre de charge dû à la répartition des données (data skew).

- Les opérations de mises à jours

Dans un contexte d'exécution parallèles des opérations d'accès aux fichiers, les opérations de mises à jours (insertions, suppression et modification des enregistrements) doivent être exécutées en suivant rigoureusement des protocoles d'accès adéquats garantissant des exécutions concurrentes et correctes (sans incohérences).

Les incohérences pouvant être introduites dans les données, proviennent soit d'une exécution concurrente non conforme à l'ordre séquentiel des opérations d'accès, soit de pannes systèmes (par exemple arrêt brutal causé par une interruption de l'alimentation électrique) provoquant l'exécution incomplète de certaines tâche modifiants les données. Ces aspects liés à la correction des exécutions concurrentes sont étudiés dans le cadre de théorie de la sérialisabilité des opérations d'accès aux données et des techniques de recouvrement et reprise sur panne.

Les systèmes OLTP (Online Transaction Processing) où des bases de données sont manipulées par un grand nombre d'applications clientes concurrentes, pouvant effectuer des mises à jours en-ligne,

sont très utilisés pour la gestion des bases de production. L'une des métriques de performances les plus importantes dans ce type de système est le 'débit transactionnel' représentant le nombre de transactions validées (effectuées avec succès) par seconde. Parmi les protocoles de contrôle d'accès aux données les plus utilisés dans ces systèmes, nous pouvons citer : le verrouillage à 2 phases (2PL), l'ordonnancement par estampillage (TO) et les nouveaux schémas multi-versions sérialisables (comme SSI par exemple). Pour ce qui est de la reprise et de la persistance des modifications, les protocoles de journalisation (comme WAL : Write-Ahead Logging), en plus de celui de la validation atomique (ex : 2PC) dans le cas des systèmes répartis sur plusieurs nœuds de calcul, sont très utilisés dans les systèmes actuels.

- Le contexte 'Big Data'

De nos jours la quantité de données générées par les différentes applications est gigantesque (traces des internautes, fichiers logs des serveurs, bases de données semi-structurés des réseaux sociaux, données de mesures générés par des capteurs ou des dispositifs connectés de l'IoT …). De plus, les possibilités de stockage actuelles ont rendu possible la persistance de ces données à des fins d'analyses et d'études de tout genre (analyse de données, études comportementales, statistiques, extraction de connaissances, apprentissage, ...).

L'accès efficace à ces données serait, en théorie, possible à condition d'avoir le temps de les structurer (par exemple en B-arbres, Hachage dynamique … et en général vers des bases de données fortement structurées). Le problème est que la vitesse avec laquelle ces données sont produites est plus grande que celle nécessaire pour bien les structurer, car elles sont nombreuses, variées et générées en quasi-continu. Dans ce contexte, dit de '*Big Data*', où certaines applications doivent analyser rapidement l'information massive générée en peu de temps, l'utilisation des grands clusters de calcul offre une solution relativement acceptable. Le stockage est généralement réalisé en vrac sur des systèmes de fichiers répartis, qui fragmentent automatiquement les données sur un grand nombre de nœuds (de manière rapide et simpliste). Les programmes d'analyse sont massivement parallélisés (le plus souvent automatiquement avec des frameworks de calcul parallèle ou dans certains cas, manuellement avec des librairies de type Message-passing ou autres) pour espérer des temps de réponses raisonnables.