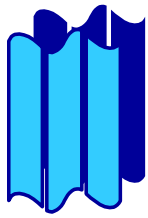

FICHIERS, HACHAGE ET INDEXATION

1. INTRODUCTION

Un SGBD inclut en son cœur une gestion de fichiers. Ce chapitre est consacré à l'étude des fichiers et de leurs méthodes d'accès. Historiquement, la notion de fichier a été introduite en informatique dans les années 50, afin de simplifier l'utilisation des mémoires secondaires des ordinateurs et de fournir des récipients de données plus manipulables aux programmes. Au début, un fichier était bien souvent l'image d'une portion nommée de bande magnétique. Puis, au fur et à mesure de la sophistication des systèmes informatiques, les fichiers sont devenus plus structurés. Des méthodes d'accès de plus en plus performantes ont été élaborées. Aujourd'hui, les fichiers sont à la base des grands systèmes d'information ; la gestion de fichiers est le premier niveau d'un SGBD.

Idéalement, un gestionnaire de fichiers doit permettre le traitement par l'informatique de la gestion complète d'une entreprise. Les données descriptives des objets gérés par l'entreprise, ainsi que les programmes spécifiant les traitements appliqués aux données, doivent pouvoir être stockés dans des fichiers gérés par le système informatique. Les traitements de ces données peuvent être exécutés par lots, en fin de mois par exemple, ou en fin de semaine, mais aussi (et c'est en général plus difficile) à l'unité dès que survient un événement dans le

monde réel : on parle alors de traitement transactionnel, mode de traitement privilégié par les bases de données.

Un gestionnaire de fichiers devrait par exemple permettre de traiter l'application comptabilité d'une société de livraison. Cette application est représentée figure III.1. Elle gère les comptes des clients et édite les factures correspondant aux livraisons. Pour calculer les montants à facturer et à déduire des comptes clients, un catalogue des prix est utilisé. Les traitements peuvent être effectués en traitement par lots, en fin de mois ; dans ce cas, les bordereaux de livraison du mois sont traités ensemble, par exemple à partir d'un fichier de saisie. Le traitement d'une livraison peut être effectué en transactionnel ; dans ce cas, la description de la livraison est tapée en direct sur un terminal et les traitements associés sont immédiatement exécutés.

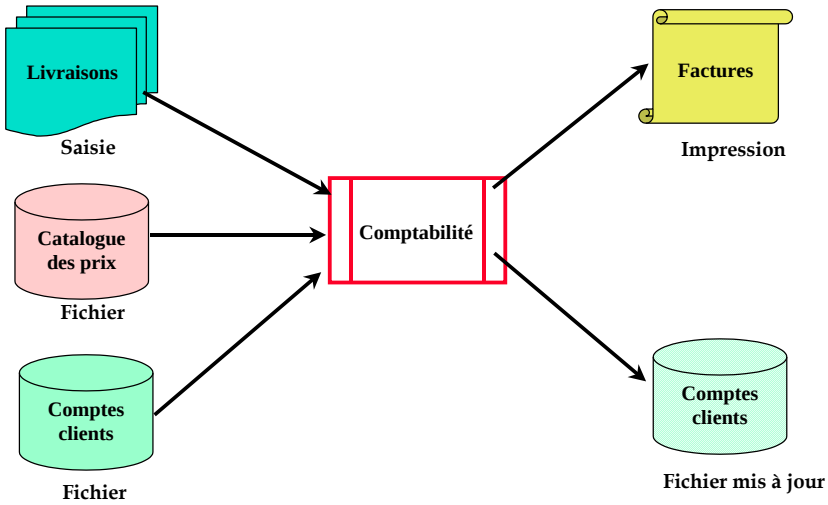

Figure III.1 — Un exemple d'application

Nous allons, dans ce chapitre, étudier plus spécifiquement la gestion des fichiers au cœur des SGBD. Un fichier est un récipient de données identifié par un nom et contenant des informations système ou utilisateur. La gestion des fichiers est une des fonctions essentielles offertes par les systèmes opératoires. C'est en effet grâce à cette fonction qu'il est possible de traiter et de conserver des quantités importantes de données, et de les partager entre plusieurs programmes. De plus, elle sert de base au niveau interne des Systèmes de Gestion de Bases de Données (SGBD) qui la complètent par des méthodes d'accès spécifiques ou la reprennent totalement.

Ce chapitre introduit tout d'abord les objectifs de la gestion des fichiers et les concepts de base associés, puis étudie les fonctions accomplies par le noyau d'un gestionnaire de fichiers. Enfin, et c'est son objet essentiel, il présente les principales organisations et méthodes d'accès de fichiers. Celles-ci sont groupées en deux classes : (i) les méthodes par hachage qui appliquent une fonction de hachage à l'identifiant des articles d'un fichier, appelé clé, afin de retrouver son emplacement dans le fichier ; (ii) les méthodes indexées qui gèrent une table des matières du fichier afin d'accéder aux articles.

2. OBJECTIFS ET NOTIONS DE BASE

Nous rappelons ici les notions fondamentales d'architecture d'ordinateur et de gestion de fichiers. Le lecteur informaticien pourra sauter cette section, voire la suivante.

2.1 Gestion des disques magnétiques

L'informatisation de la gestion d'une grande entreprise nécessite le stockage de volumes de données importants — bien souvent plusieurs milliards d'octets. Il n'est pas possible de stocker de tels volumes de données en mémoire centrale. On est ainsi conduit à introduire des **mémoires secondaires**.

> **Notion III.1 : Mémoire secondaire (*External Storage*)**
>
> Mémoire non directement adressable par les instructions du processeur central, mais par des instructions d'entrées-sorties spécialisées et dont les temps d'accès sont très supérieurs à ceux de la mémoire centrale.

Il existe différents types de mémoires secondaires : disques magnétiques (encore appelés disques durs), disques optiques numériques, bandes magnétiques, cassettes, mémoire électronique SSD, etc. Parmi les disques magnétiques, les plus utilisés sont les disques accédés par des têtes de lecture-écriture mobiles solidaires au bout d'un bras. La capacité des disques durs a augmenté beaucoup plus vite que le temps d'accès, limité par la mécanique assurant les déplacements de bras. Deux cas limites de disques magnétiques sont les minidisques des micro-ordinateurs de moins en moins utilisés et les disques optiques numériques. Ceux-ci constituent une technologie à bas prix permettant d'archiver plusieurs giga-octets sur un même disque. Ils ne sont malheureusement inscriptibles qu'une seule fois, bien que des disques optiques réinscriptibles existent mais sont lents en écriture.

Les disques durs courants des ordinateurs portables ou fixes sont de plus en plus petits en taille. Le diamètre habituel est de 3,5 pouces. Il peut même atteindre le pouce. Leur capacité s'est formidablement accrue depuis les années 1955 pour dépasser couramment le téraoctet aujourd'hui. La densité d'enregistrement est de plus en plus élevée. Les temps d'accès se sont également améliorés pour avoisiner 10 à 20 ms en moyenne. Notons qu'ils restent encore de l'ordre de 20 000 fois plus élevés que ceux des mémoires centrales. Ces disques sont organisés en piles montées sur un tourne-disque assurant la rotation des plateaux et la gestion des bras. Nous appellerons une pile de disques un **volume**.

> **Notion III.2 : Volume (*Disk Pack*)**
>
> Pile de disques constituant une unité de mémoire secondaire utilisable.

Un volume disque est associé à un tourne-disque. Les volumes peuvent être fixes ou amovibles. Si le volume est amovible, il est monté sur un tourne-disque pour utilisation puis démonté après. Les volumes amovibles sont montés et démontés par les opérateurs, en général sur un ordre du système. Un contrôleur de disque contrôle en général plusieurs tourne-

disques. La notion de volume s'applique également aux bandes magnétiques où un volume est une bande. Une unité peut alors comporter un ou plusieurs dérouleurs de bande.

Un volume et l'équipement de lecture-écriture associé à un tourne-disque sont représentés figure III.2. Un volume se compose de p disques (par exemple 9), ce qui correspond à 2p - 2 surfaces magnétisées, car les deux faces externes ne le sont pas. Les disques à têtes mobiles comportent une tête de lecture-écriture par surface. Cette tête est accrochée à un bras qui se déplace horizontalement de manière à couvrir toute la surface du disque. Un disque est divisé en pistes concentriques numérotées de 0 à n (par exemple n = 1024). Les bras permettant de lire/écrire les pistes d'un volume sont solidaires, ce qui force leur déplacement simultané. Les disques tournent continûment à une vitesse de quelques dizaines de tours par seconde. L'ensemble des pistes, décrit quand les bras sont positionnés, est appelé cylindre.

Chaque piste d'un disque supporte plusieurs enregistrements physiques appelés secteurs, de taille généralement constante, mais pouvant être variable pour certains types de disque. Le temps d'accès à un groupe de secteurs consécutifs est une des caractéristiques essentielles des disques. Il se compose :

1. du temps nécessaire au mouvement de bras pour sélectionner le bon cylindre (quelques millisecondes à des dizaines de millisecondes selon l'amplitude du déplacement du bras et le type de disque) ;

2. du temps de rotation du disque nécessaire pour que l'enregistrement physique désiré passe devant les têtes de lecture/écriture ; ce temps est appelé temps de latence ; il est de quelques millisecondes, selon la position de l'enregistrement par rapport aux têtes et selon la vitesse de rotation des disques ;

3. du temps de lecture/écriture du groupe de secteurs, appelé temps de transfert ; ce temps est égal au temps de rotation multiplié par la fraction de pistes lue, par exemple 1/16 de 10 ms pour lire un secteur sur une piste de 16 secteurs avec un disque tournant à 100 tours par seconde.

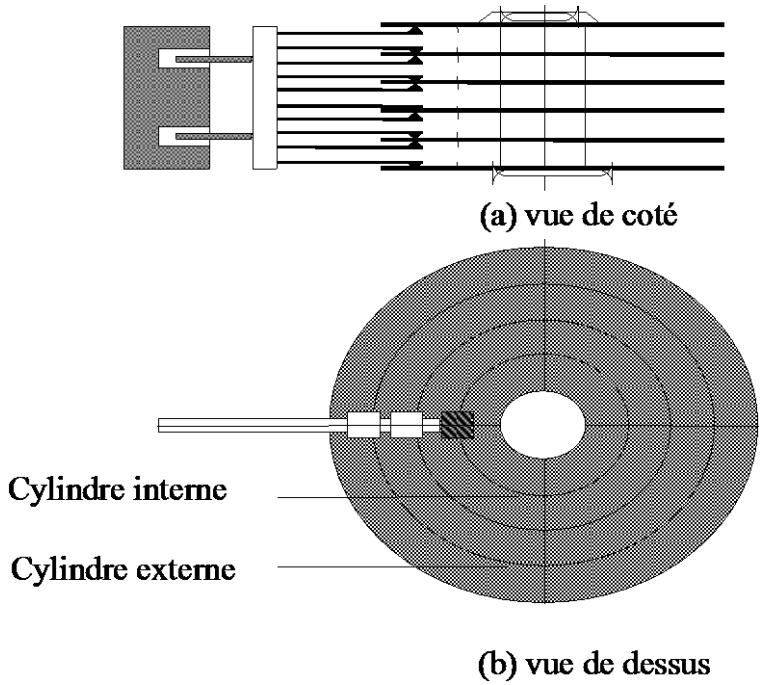

Figure III.2 — Volume amovible à têtes mobiles et équipement de lecture/écriture

Au total, les disques ont un temps d'accès variable selon la distance qui sépare l'enregistrement auquel accéder de la position des têtes de lecture/écriture. La tendance aujourd'hui est de réduire cette variance avec des moteurs à accélération constante pour commander les mouvements de bras, et avec des quantités de données enregistrées par cylindre de plus en plus importantes.

De plus en plus souvent, des volumes multiples organisés en **tableaux de disques** sont utilisés pour composer des unités fiables et de grande capacité (des dizaines de milliards d'octets). Ces systèmes de mémorisation portent le nom de RAID (*Redundant Array of Inexpensive Disks*). On distingue le RAID 0 sans redondance des RAID 1 à 6 qui gèrent des redondances. Avec ces derniers, différentes techniques de redondance sont utilisées lors des écritures et lectures afin d'assurer une grande fiabilité. Le RAID 1 groupe les disques du tableau par deux et effectuent les écritures sur les deux disques, l'un apparaissant donc comme le miroir de l'autre. En cas de panne d'un disque, le disque miroir peut toujours être utilisé. Les RAID 2, 3 et 4 gèrent un disque de parité, respectivement au niveau bit, octet ou bloc. En cas de panne d'un disque, celui-ci peut être reconstitué grâce à la parité. Les RAID 5 et 6 sont basés sur des redondances plus fortes, avec des codes cycliques (CRC). Ils permettent de résister à des doubles pannes. Les disques RAID permettent des performances en lecture et écriture élevées car de multiples contrôleurs d'entrées-sorties fonctionnent en parallèle.

2.2 Indépendance des programmes par rapport aux mémoires secondaires

Les disques magnétiques et plus généralement les mémoires secondaires doivent pouvoir être utilisés par plusieurs programmes d'application. En conséquence, il faut pouvoir partager

l'espace mémoire secondaire entre les données des diverses applications. Une gestion directe de cet espace par les programmes n'est pas souhaitable car elle interdirait de modifier l'emplacement des données sans modifier les programmes d'application. Les adresses utilisées par les programmes doivent être indépendantes de l'emplacement des données sur disques. Il faut donc introduire des couches systèmes intermédiaires permettant d'assurer l'indépendance des programmes vis-à-vis de l'emplacement des données sur les mémoires secondaires. Autrement dit, l'allocation de la mémoire secondaire doit être gérée par le système.

D'un autre côté, les progrès technologiques ne cessent d'améliorer le rapport performance/prix des mémoires secondaires. La densité des disques magnétiques (nombre de bits enregistrés/cm^2) double approximativement tous les deux ans, ce qui permet d'accroître les capacités de stockage et les performances. Un bon système doit permettre aux utilisateurs de profiter des avancées technologiques, par exemple en achetant des disques plus performants, et cela sans avoir à modifier les programmes. En effet, la reprogrammation coûte très cher en moyens humains. En résumé, il faut assurer l'**indépendance des programmes d'application par rapport aux mémoires secondaires**. Cette indépendance peut être définie comme suit :

> **Notion III.3 : Indépendance des programmes par rapport aux mémoires secondaires (*Program-Storage device independence*)**
>
> Possibilité de changer les données de localité sur les mémoires secondaires sans changer les programmes.

Afin de réaliser cette indépendance, on introduit des objets intermédiaires entre les programmes d'application et la mémoire secondaire. Ces objets sont appelés **fichiers**. Ainsi, les programmes d'application ne connaissent pas les mémoires secondaires, mais les fichiers qui peuvent être implantés sur diverses mémoires secondaires. Un fichier peut être défini comme suit :

> **Notion III.4 : Fichier (*File*)**
>
> Récipient d'information caractérisé par un nom, constituant une mémoire secondaire idéale, permettant d'écrire des programmes d'application indépendants des mémoires secondaires.

Un programme d'application ne manipule pas globalement un fichier mais lit/écrit et traite des portions successives de celui-ci, correspondant en général à un objet du monde réel, par exemple un client, un compte, une facture. Une telle portion est appelée **article**.

Notion III.5 : Article (*Record*)

Elément composant d'un fichier correspondant à l'unité de traitement par les programmes d'application.

Les articles sont stockés dans les récipients d'information que constituent les fichiers. Ils ne sont pas stockés n'importe comment, mais sont physiquement reliés entre eux pour composer le contenu d'un fichier. Les structures des liaisons constituent l'**organisation du fichier**.

Notion III.6 : Organisation de fichier (*File organization*)

Nature des liaisons entre les articles contenus dans un fichier.

Les programmes d'application peuvent choisir les articles dans un fichier de différentes manières, par exemple l'un après l'autre à partir du premier, ou en attribuant un nom à chaque article, selon la **méthode d'accès** choisie.

Notion III.7 : Méthode d'accès (*Acces Method*)

Méthode d'exploitation du fichier utilisée par les programmes d'application pour sélectionner des articles.

2.3 Utilisation de langages hôtes

Le système de gestion de fichiers doit être utilisable par un programme dont le code objet résulte d'une compilation d'un langage de haut niveau (COBOL, PL/1, PASCAL, C, etc.). De plus, il doit être possible d'utiliser les fichiers dans le langage choisi d'une manière aussi intégrée que possible, par exemple en décrivant les données du fichier comme des types dans le langage. On appelle **langage hôte** le langage de programmation qui intègre les verbes de manipulation de fichiers.

Notion III.8 : Langage hôte (*Host language*)

Langage de programmation accueillant les verbes de manipulation de fichiers et la définition des données des fichiers.

Il est utile de rappeler le cheminement d'un programme en machine. Celui-ci est donc écrit à l'aide d'un langage de programmation hôte et de verbes de manipulation de fichiers. Il est pris en charge par le compilateur du langage. Ce dernier génère du code machine translatable incluant des appels au système, en particulier au gestionnaire de fichiers. Le chargeur-éditeur de liens doit ensuite amener en bonne place en mémoire les différents modules composants du programme ; en particulier, les translations d'adresse doivent être effectuées à ce niveau. On obtient alors du code exécutable. La figure III.3 illustre ces étapes.

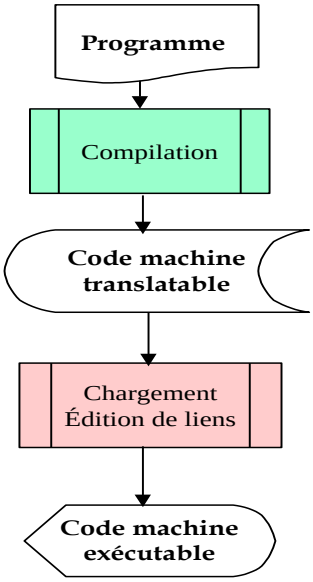

Figure III.3 — Cheminement d'un programme

2.4 Possibilités d'accès séquentiel et sélectif

Afin de caractériser le comportement des programmes d'application vis-à-vis des fichiers, il est possible d'utiliser deux mesures. Le taux de consultation (TC) est le quotient du nombre d'articles utilement lus par un programme sur le nombre d'articles total du fichier. Le taux de mouvement (TM) est le quotient du nombre d'articles modifiés par un programme sur le nombre d'articles total du fichier. On est ainsi conduit à introduire deux types de méthodes d'accès.

> **Notion III.9 : Méthode d'accès séquentielle (*Sequential Acces Method*)**
>
> Méthode d'accès consistant à lire successivement tous les articles d'un fichier, depuis le premier jusqu'à l'article désiré

> **Notion III.10 : Méthodes d'accès sélectives (*Key-based access methods*)**
>
> Ensemble de méthodes d'accès permettant de lire/écrire tout article au moyen de quelques accès disques (moins de 5, idéalement 1), y compris pour de très gros fichiers.

Une méthode d'accès séquentielle est adaptée si TC ou TC+TM sont de l'ordre de 1. Elle sera essentiellement utilisée en traitement par lots. Au contraire, une méthode d'accès sélective sera utilisée quand TC, ou TC+TM pour un programme modifiant, sera petit. Les méthodes d'accès sélectives sont donc particulièrement adaptées au transactionnel. Un gestionnaire de fichiers doit supporter des méthodes d'accès séquentielle et sélectives, cela afin de permettre à la fois les traitements par lots et le travail en transactionnel.

Afin de mettre en œuvre une méthode d'accès sélective, il faut pouvoir identifier de manière unique un article. En effet, une méthode d'accès sélective permet à partir de l'identifiant d'un article de déterminer l'adresse d'un article (adresse début) et de lire l'article en moins de 5 E/S. L'identifiant d'un article est appelé **clé** (que l'on qualifie parfois de primaire). La clé peut ou non figurer comme une donnée de l'article.

> **Notion III.11 : Clé d'article (Record Key)**
>
> Identifiant d'un article permettant de sélectionner un article unique dans un fichier.

Soit l'exemple d'un fichier décrivant des étudiants. Les articles comportent les champs suivants : numéro d'étudiant, nom, prénom, ville, date d'inscription et résultats. La clé est le numéro d'étudiant ; c'est une donnée de l'article. A partir de cette clé, c'est-à-dire d'un numéro d'étudiant, une méthode d'accès sélective doit permettre de déterminer l'adresse de l'article dans le fichier et d'accéder à l'article en principe en moins de 5 entrées/sorties disques.

Comme cela a été déjà dit, il existe différents types d'organisations sélectives (et méthodes d'accès associées). Nous allons ci-dessous étudier les principales. Elles peuvent être divisées en deux classes :

- Les méthodes d'accès par hachage utilisent des fonctions de calcul pour déterminer l'adresse d'un article dans un fichier à partir de sa clé ;

- Les méthodes d'accès par index utilisent des tables généralement stockées sur disques pour mémoriser l'association clé article-adresse article.

2.5 Possibilité d'utilisateurs multiples

Dans les machines modernes, l'exécution d'une instruction d'entrée-sortie ne bloque pas le processeur central : celui-ci peut continuer à exécuter des instructions machine en parallèle à l'exécution de l'entrée-sortie. Afin d'utiliser ces possibilités, le système exécute plusieurs programmes usagers simultanément, ce qui conduit à une **simultanéité inter-usagers**, c'est-à-dire entre différents programmes utilisateurs.

> **Notion III.12 : Simultanéité inter-usagers (*Inter-user parallelism*)**
>
> Type de simultanéité consistant à exécuter un programme d'application en processeur central pendant qu'un autre programme effectue des entrées-sorties.

Un bon système de fichiers doit permettre le partage des fichiers par différents programmes d'application sans que ceux-ci s'en aperçoivent. Le partage peut être simultané, mais aussi plus simplement réparti dans le temps. Nous étudierons plus en détail les problèmes de partage dans le contexte des bases de données où ils se posent avec plus d'acuité encore.

2.6 Sécurité et protection des fichiers

L'objectif sécurité des fichiers contre les accès mal intentionnés, ou plus simplement non autorisés, découle directement du besoin de partager les fichiers. En effet, lorsque les fichiers sont partagés, le propriétaire désire contrôler les accès, les autoriser à certains, les interdire à d'autres. C'est là une des fonctions que doit rendre un bon système de gestion de fichiers. Ces mécanismes sont généralement réalisés à l'aide de noms hiérarchiques et de clés de protection associés à chaque fichier, voire à chaque article. L'usager doit fournir ces noms et ces clés pour accéder au fichier ou à l'article. Nous étudierons les solutions dans le cadre des bases de données où le problème devient plus aigu.

Le gestionnaire de fichiers doit aussi garantir la conservation des fichiers en cas de panne du matériel ou du logiciel. En conséquence, il doit être prévu de pouvoir repartir après panne avec des fichiers corrects. On considère en général deux types de pannes : les pannes simples avec seulement perte du contenu de la mémoire secondaire, les pannes catastrophiques où le contenu de la mémoire secondaire peut être détruit. Ainsi, il est nécessaire d'incorporer des procédures de reprise après pannes simples et catastrophiques. Nous étudierons ces procédures dans le cadre des bases de données où elles sont généralement plus complètes, bien que souvent prises en compte au niveau de la gestion de fichiers.

3. FONCTIONS D'UN GÉRANT DE FICHIERS

3.1 Architecture d'un gestionnaire de fichiers

Un gestionnaire de fichiers est généralement structuré autour d'un noyau, appelé ici analyseur, qui assure les fonctions de base, à savoir la création/destruction des fichiers, l'allocation de la mémoire secondaire, la localisation et la recherche des fichiers sur les volumes et la gestion des zones de mémoires intermédiaires appelées tampons. Les méthodes d'accès sont des modules spécifiques qui constituent une couche plus externe et qui utilisent les fonctions du noyau. La figure III.4 représente les différents modules d'un gestionnaire de fichiers typique. Notons que ces modules sont organisés en trois couches de logiciel : les méthodes d'accès, le noyau et le gestionnaire d'entrées-sorties composé de modules plus ou moins spécifiques à chaque périphérique. Chaque couche constitue une machine abstraite qui accomplit un certain nombre de fonctions accessibles aux couches supérieures par des primitives (par exemple, lire ou écrire un article, ouvrir ou fermer un fichier) constituant l'interface de la couche.

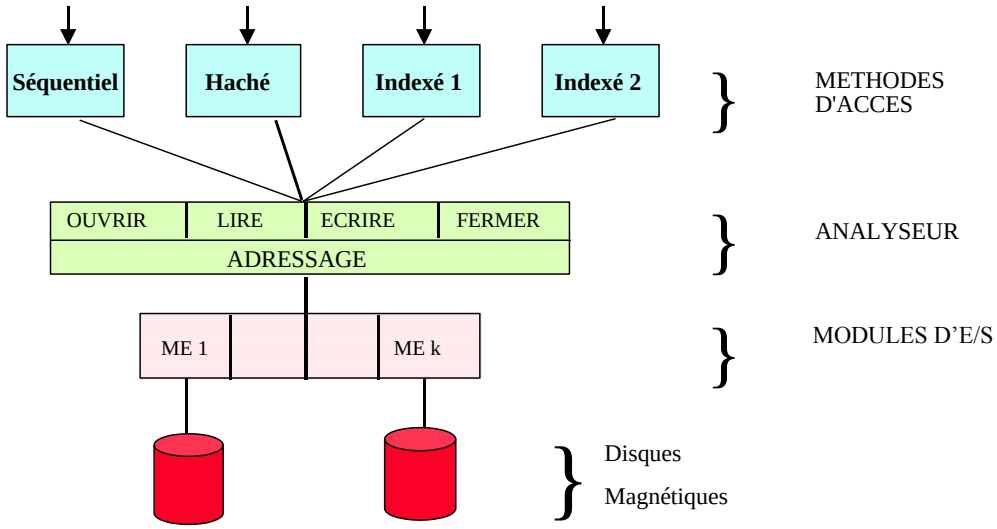

Figure III.4 — Architecture d'un gestionnaire de fichiers

Le noyau ou analyseur d'un gestionnaire de fichiers est chargé d'assurer la gestion des fichiers en temps que récipients non structurés. Il permet en général d'établir le lien entre un fichier et un programme (ouverture du fichier), de supprimer ce lien (fermeture), de lire ou écrire une suite d'octets à un certain emplacement dans un fichier. Il accède aux mémoires secondaires par l'intermédiaire du gestionnaire d'entrées-sorties. Celui-ci, qui n'appartient pas à proprement parler au gestionnaire de fichiers mais bien au système opératoire, gère les entrées-sorties physiques : il permet la lecture et l'écriture de blocs physiques de données sur tous les périphériques, gère les particularités de chacun, ainsi que les files d'attente d'entrées-sorties.

Chaque module méthode d'accès est à la fois chargé de l'organisation des articles dans le fichier et de la recherche des articles à partir de la clé. Un bon système de fichiers doit offrir une grande variété de méthodes d'accès. Il offrira bien sûr une méthode d'accès séquentielle, mais surtout plusieurs méthodes d'accès sélectives.

3.2 Fonctions du noyau d'un gestionnaire de fichiers

3.2.1 Manipulation des fichiers

Le programmeur travaillant au niveau du langage machine, ainsi que les modules méthodes d'accès accèdent au noyau du gestionnaire de fichiers à l'aide d'un ensemble d'instructions de manipulation de fichiers. Tout d'abord, deux instructions permettent de **créer** et de **détruire** un fichier. Puis, avant de **lire** ou d'**écrire** des données dans un fichier, il faut contrôler son identité et ouvrir un chemin pour les données entre le programme effectuant les lectures-écritures et la mémoire secondaire. Cette opération est généralement effectuée par une instruction d'ouverture : **ouvrir**. L'opération inverse est exécutée lorsque le programme se désintéresse du fichier par une instruction de fermeture : **fermer**.

3.2.2 Adressage relatif

Un fichier étant généralement discontinu sur mémoire secondaire, il est utile de pouvoir adresser son contenu à l'aide d'une adresse continue de 0 à n appelée **adresse relative**. Cela présente l'intérêt de disposer d'un repérage indépendant de la localisation du fichier sur mémoire secondaire (en cas de recopie du fichier, l'adresse relative ne change pas) et de pouvoir assurer que l'on travaille bien à l'intérieur d'un fichier sans risque d'atteindre un autre fichier (il suffit de contrôler que l'adresse relative ne dépasse pas la taille du fichier).

> **Notion III.13 : Adresse relative (*Relative address*)**
>
> Numéro d'unité d'adressage dans un fichier (autrement dit déplacement par rapport au début du fichier).

Pour réaliser l'adressage relatif, on divise généralement le fichier en **pages** (on trouve également selon les implantations les termes de bloc, groupe, intervalle) : une adresse relative octet se compose alors d'un numéro de page suivi d'un numéro d'octet dans la page. Pour éviter un nombre trop important d'entrées-sorties, la taille de la page est choisie de façon à contenir plusieurs enregistrements physiques et des tampons d'une page sont utilisés. La taille de la page, fixée dans le système ou lors de la création du fichier, est le plus souvent de l'ordre de quelques kilos (K) octets (par exemple, 4K). Elle dépend parfois de l'organisation du fichier. Celle d'un enregistrement physique dépasse rarement quelques centaines d'octets.

Ainsi, l'analyseur d'un gestionnaire de fichiers offre généralement la possibilité d'accéder à une ou plusieurs pages d'adresse relative (numéro de page) donnée dans un fichier. Il peut aussi permettre d'accéder directement aux octets, donc de lire une suite d'octets à partir d'une adresse relative en octets dans le fichier. L'analyseur se compose essentiellement d'algorithmes d'allocation de mémoire secondaire et de conversion d'adresse relative page en adresse réelle de l'enregistrement physique (secteur sur disque), et réciproquement. Finalement, il permet de banaliser toutes les mémoires secondaires en offrant un accès par adresse relative uniforme, avec quelques restrictions cependant : il est par exemple interdit d'accéder autrement qu'en séquentiel à une bande magnétique.

Les articles de l'usager sont alors implantés dans les pages par les méthodes d'accès selon l'organisation choisie. Si plusieurs articles sont implantés dans une même page, on dit qu'il y a **blocage** . Dans le cas où les articles sont implantés consécutivement sans trou à la suite les uns des autres, on dit qu'il y a **compactage** : aucune place n'est alors perdue sur la mémoire secondaire, mais des articles peuvent être à cheval sur plusieurs pages.

3.2.3 Allocation de la place sur mémoires secondaires

La taille d'un fichier est fixée soit de manière statique lors de sa création, soit de manière dynamique au fur et à mesure des besoins. Des solutions intermédiaires sont possibles avec une taille initiale extensible par paliers. Dans tous les cas, il est nécessaire de réserver des zones de mémoires secondaires continues pour le fichier. Ces zones sont appelées **régions**.

> **Notion III.14 : Région (*Allocation area*)**
>
> Ensemble de zones de mémoires secondaires (pistes) adjacentes allouées en une seule fois à un fichier.

Comme les fichiers vivent et sont de tailles différentes, les régions successivement allouées à un fichier ne sont généralement pas contiguës sur une mémoire secondaire. Le gestionnaire de fichiers doit alors pouvoir retrouver les régions composant un fichier. Pour cela, il peut soit garder la liste des régions allouées à un fichier dans une table, soit les chaîner, c'est-à-dire mettre dans une entrée d'une table correspondant à chaque région l'adresse de la région suivante.

La taille d'une région peut varier à partir d'un seuil minimal appelé **granule** (par exemple une piste, etc.). Il devient alors possible d'allouer des régions de taille variable à un fichier, mais toujours composées d'un nombre entier de granules consécutifs. Un granule est souvent choisi de taille égale à une piste où à une fraction de piste.

> **Notion III.15 : Granule d'allocation (*Allocation granule*)**
>
> Unité de mémoire secondaire allouable à un fichier.

Lorsqu'un fichier est détruit ou rétréci, les régions qui lui étaient allouées et qui ne sont plus utilisées sont libérées. Les granules composants sont alors libérés et introduits dans la liste des granules libres. Afin de maximiser la proximité des granules alloués à un fichier, plusieurs méthodes d'allocations ont été proposées. Nous allons étudier ci-dessous quelques algorithmes d'allocation des régions aux fichiers.

3.2.4 Localisation des fichiers sur les volumes

Il est nécessaire de pouvoir identifier un volume. En effet, lorsqu'un volume amovible n'est pas actif, il peut être enlevé. Il faut donc une intervention manuelle pour monter/démonter un volume sur un tourne-disque. Cette opération est exécutée par un opérateur. Celui-ci reconnaît un volume grâce à un numéro (ou nom) attribué à chaque volume. Pour pouvoir contrôler les opérateurs lors des montages/démontages de volume et éviter les erreurs, ce numéro est écrit sur le volume dans le **label** du volume.

> **Notion III.16 : Label de volume (*Label*)**
>
> Premier secteur d'un volume permettant d'identifier ce volume et contenant en particulier son numéro.

Lorsqu'on a retrouvé et monté le volume supportant un fichier, il faut retrouver le fichier sur le volume. Pour cela, chaque fichier possède un ensemble de données descriptives (nom, adresse début, localisation, etc.) regroupées dans un **descripteur du fichier**.

Notion III.17 : Descripteur de fichier (*Directory entry*)

Ensemble des informations permettant de retrouver les caractéristiques d'un fichier, contenant en particulier son nom, sa localisation sur disque, etc.

Il est important que l'ensemble des descripteurs de fichiers contenu sur un volume définisse tous les fichiers d'un volume. On obtient ainsi des volumes autodocumentés, donc portables d'une installation à une autre. Pour cela, les descripteurs de fichiers d'un volume sont regroupés dans une table des matières du volume appelée **catalogue**.

Notion III.18 : Catalogue (*Directory*)

Table (ou fichier) située sur un volume et contenant les descripteurs des fichiers du volume.

Le catalogue est soit localisé en un point conventionnel du volume (par exemple, à partir du secteur 1), soit écrit dans un fichier spécial de nom standard. Le descripteur de ce premier fichier peut alors être contenu dans le label du volume. En résumé, la figure III.5 illustre l'organisation des informations étudiées jusqu'à présent sur un volume.

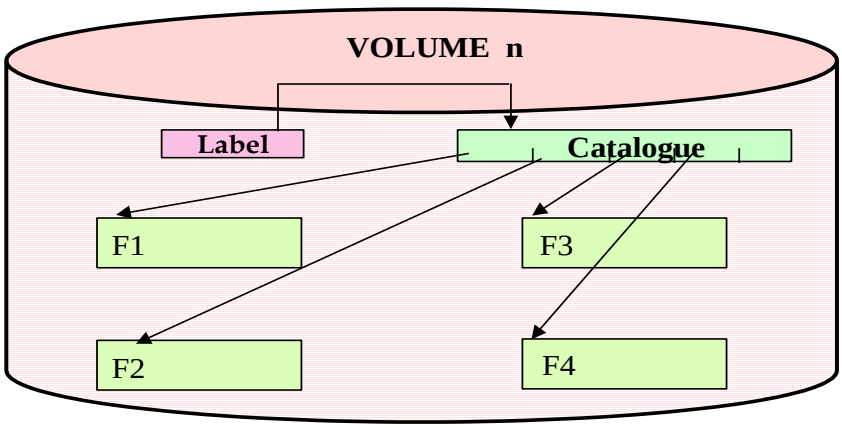

Figure III.5 — Schéma représentant l'organisation d'un volume

3.2.5 Classification des fichiers en hiérarchie

Quand le nombre de fichiers d'une installation devient élevé, on est conduit à classer les descripteurs de fichiers dans plusieurs catalogues, par exemple un par usager. Les descripteurs des fichiers catalogues peuvent alors être maintenus dans un catalogue de niveau plus élevé. On aboutit ainsi à des **catalogues hiérarchisés** qui sont implantés dans de nombreux systèmes [Daley65].

Notion III.19 : Catalogue hiérarchisé (Hierarchical directory)

Catalogue constitué d'une hiérarchie de fichiers, chaque fichier contenant les descripteurs des fichiers immédiatement inférieurs dans la hiérarchie.

Dans un système à catalogues hiérarchisés, chaque niveau de catalogue est généralement spécialisé. Par exemple, le niveau 1 contient un descripteur de fichier catalogue par usager. Pour chaque usager, le niveau 2 peut contenir un descripteur de fichier par application. Enfin, pour chaque couple <usager-application>, le niveau 3 peut contenir la liste des descripteurs de fichiers de données. La figure III.6 illustre un exemple de catalogue hiérarchisé. Le descripteur du catalogue de niveau 1 est appelé racine.

La présence de catalogue hiérarchisé conduit à introduire des noms de fichiers composés. Pour atteindre le descripteur d'un fichier, il faut en effet indiquer le nom du chemin qui mène à ce descripteur. Voici des noms de fichiers possibles avec le catalogue représenté figure III.6 :

- PIERRE
- PIERRE > BASES-DE-DONNEES
- PIERRE > BASES-DE-DONNES > MODELES

Afin d'éviter un cloisonnement trop strict des fichiers, les systèmes à catalogues hiérarchisés permettent en général l'introduction de *liens.* Un lien est simplement un descripteur qui contient un pointeur logique sur un autre descripteur, éventuellement dans une autre branche de la hiérarchie. Par exemple, le descripteur de nom > LIONEL > BASES-DE-DONNEES > LANGAGES pourra être un descripteur de type lien indiquant qu'il est un synonyme du descripteur > PIERRE > BASES-DE-DONNEES > LANGAGES. Les noms d'usagers étant généralement fournis par le système, des descripteurs de type lien permettent le partage des fichiers entre usagers.

Pour simplifier la nomination des fichiers, les systèmes à catalogues hiérarchisés gèrent bien souvent la notion de **catalogue de base**, encore appelé catalogue courant. Lors du début de session (login), le système positionne le catalogue courant par exemple sur les applications de l'usager. Celui-ci peut alors se déplacer par rapport à ce catalogue courant. Par exemple, Pierre passera au catalogue bases-de-données par le nom > Bases-de-données. Celui-ci deviendra alors son catalogue courant. L'accès au fichier modèles se fera alors par le nom > Modèles. Le retour au catalogue initial de Pierre s'effectuera par < < ; en effet, les noms de répertoires sont déterminés de manière unique quand on remonte la hiérarchie.

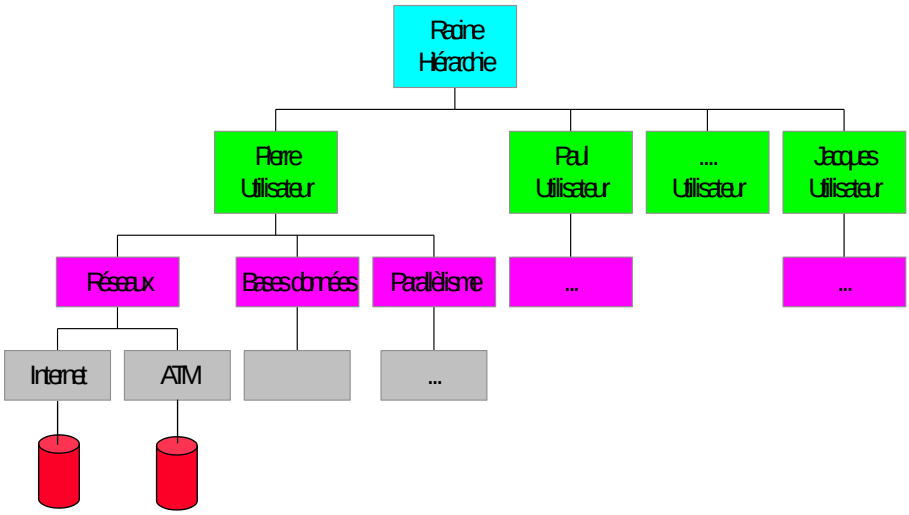

Figure III.6 — Exemple de catalogue hiérarchisé

3.2.6 Contrôle des fichiers

Le noyau d'un gestionnaire de fichiers inclut également des fonctions de contrôle des fichiers : partage des fichiers, résistances aux pannes, sécurité et confidentialité des données. Nous n'étudions pas ici ces problèmes, qui font l'objet de nombreux développements dans le contexte des bases de données, donc dans les chapitres suivants.

3.3 Stratégie d'allocation de la mémoire secondaire

3.3.1 Objectifs d'une stratégie

Il existe différentes stratégies d'allocation de la mémoire secondaire aux fichiers. Une bonne stratégie doit chercher :

1. à minimiser le nombre de régions à allouer à un fichier pour réduire d'une part les déplacements des bras des disques lors des lectures en séquentiel, d'autre part le nombre de descripteurs de régions associés à un fichier ;

2. à minimiser la distance qui sépare les régions successives d'un fichier, pour réduire les déplacements de bras en amplitude.

Les stratégies peuvent être plus ou moins complexes. La classe la plus simple de stratégies alloue des régions de taille fixe égale à un granule, si bien que les notions de granule et région sont confondues. Une classe plus performante alloue des régions de tailles variables, composées de plusieurs granules successifs. Nous allons approfondir ces deux classes ci-dessous.

Auparavant, il est nécessaire de préciser que toutes les méthodes conservent une table des granules libres ; une copie de cette table doit être stockée sur disque pour des raisons de fiabilité. La table elle-même peut être organisée selon différentes méthodes :

- liste des granules ou régions libres, ordonnée ou non ; l'allocation d'une région consiste alors à enlever la région choisie de cette liste pour l'associer au descripteur du fichier ;

- table de bits dans laquelle chaque bit correspond à un granule; l'allocation d'un granule consiste alors à trouver un bit à 0, le positionner à 1 et à adjoindre l'adresse du granule alloué au descripteur du fichier.

3.3.2 Stratégie par granule (à région fixe)

Ces stratégies confondent donc les notions de région et de granule. Elles sont simples et généralement implantées sur les petits systèmes. On peut distinguer :

- La **stratégie du premier trouvé** : le granule correspondant à la tête de liste de la liste des granules libres, ou au premier bit à 0 dans la table des granules libres, est choisi ;

- **La stratégie du meilleur choix** : le granule le plus proche (du point de vue déplacement de bras) du dernier granule alloué au fichier est retenu.

3.3.3 Stratégie par région (à région variable)

Ces stratégies permettent d'allouer des régions composées de plusieurs granules consécutifs, selon les besoins des fichiers. Le noyau du gestionnaire de fichiers reçoit alors des demandes d'allocation et libération de régions de tailles variables. Dans le cas où aucune région de la taille demandée ne peut être constituée par des granules consécutifs, la demande peut éventuellement être satisfaite par plusieurs régions. Parmi les stratégies par région, on peut distinguer :

- La **stratégie du plus proche choix**. Deux régions libres consécutives sont fusionnées en une seule. Lors d'une demande d'allocation, la liste des régions libres est parcourue jusqu'à trouver une région de la taille demandée ; si aucune région de la taille demandée n'est libre, la première région de taille supérieure est découpée en une région de la taille cherchée qui est allouée au fichier et une nouvelle région libre correspondant au reste.

- La **stratégie des frères siamois**. Elle offre la possibilité d'allouer des régions de 1, 2, 4, 8…2^K granules. Des listes séparées sont maintenues pour les régions libres de dimensions 2^0, 2^1, … 2^K granules. Lors d'une demande d'allocation, une région libre peut être extraite de la liste des régions libres de taille 2^{i+1} pour constituer deux régions libres de taille 2^i. Lors d'une libération, deux régions libres consécutives (deux siamoises) de taille 2^i sont fusionnées afin de constituer une région libre de taille 2^{i+1}. L'algorithme de recherche d'une région libre de taille 2^i consiste à chercher cette région dans la liste des régions de taille 2^i. Si cette liste est vide, on recherche alors une région de taille 2^{i+1} que l'on divise en deux. S'il n'y en a pas, on passe alors au niveau suivant 2^{i+2}, etc., jusqu'à atteindre le niveau k; c'est seulement dans le cas où les listes 2^i à 2^K sont vides que l'on ne peut satisfaire la demande par une seule région. Cet algorithme, qui est emprunté aux mécanismes d'allocation de segments dans les systèmes paginés [Lister84], est sans doute le plus efficace ; il est aussi très bien adapté à certaines méthodes d'accès.

En résumé, les stratégies par région de tailles variables sont en général plus efficaces du point de vue déplacement de bras et taille de la table des régions d'un fichier. Un problème commun à ces stratégies peut cependant survenir après des découpages trop nombreux s'il n'existe plus de régions de taille supérieure à un granule. Dans ce cas, l'espace disque doit être réorganisé (ramassage des miettes). Ce dernier point fait que les stratégies par granule restent les plus utilisées.

4. ORGANISATIONS ET MÉTHODES D'ACCÉS PAR HACHAGE

Les organisations et méthodes d'accès par hachage sont basées sur l'utilisation d'une fonction de calcul qui, appliquée à la clé, détermine l'adresse relative d'une zone appelée paquet (*bucket* en anglais) dans laquelle est placé l'article. On distingue les méthodes d'accès par hachage statique, dans lesquelles la taille du fichier est fixe, des méthodes par hachage dynamique, où le fichier peut grandir.

4.1 Organisation hachée statique

C'est la méthode la plus ancienne et la plus simple. Le fichier est de taille constante, fixée lors de sa création. Une fois pour toute, il est donc divisé en p paquets de taille fixe L. La clé permet de déterminer un numéro de paquet N dont l'adresse relative est obtenue par la formule AR = N * L. Nous définirons un **fichier haché statique** comme suit.

> **Notion III.20 : Fichier haché statique (*Static hashed file*)**
>
> Fichier de taille fixe dans lequel les articles sont placés dans des paquets dont l'adresse est calculée à l'aide d'une fonction de hachage fixe appliquée à la clé.

A l'intérieur d'un paquet, les articles sont rangés à la suite dans l'ordre d'arrivée. Ils sont retrouvés grâce à la donnée contenant la clé. La figure III.7 illustre un exemple de structure interne d'un paquet. En tête du paquet, on trouve l'adresse du premier octet libre dans le paquet. Ensuite, les articles successifs du paquet sont rangés avec leur longueur en tête, par exemple sur deux octets. A l'intérieur d'un tel paquet, on accède à un article par balayage séquentiel. Des structures de paquet plus sophistiquées permettent l'accès direct à un article de clé donnée à l'intérieur d'un paquet. De telles structures sont plus efficaces que la structure simple représentée figure III.7.

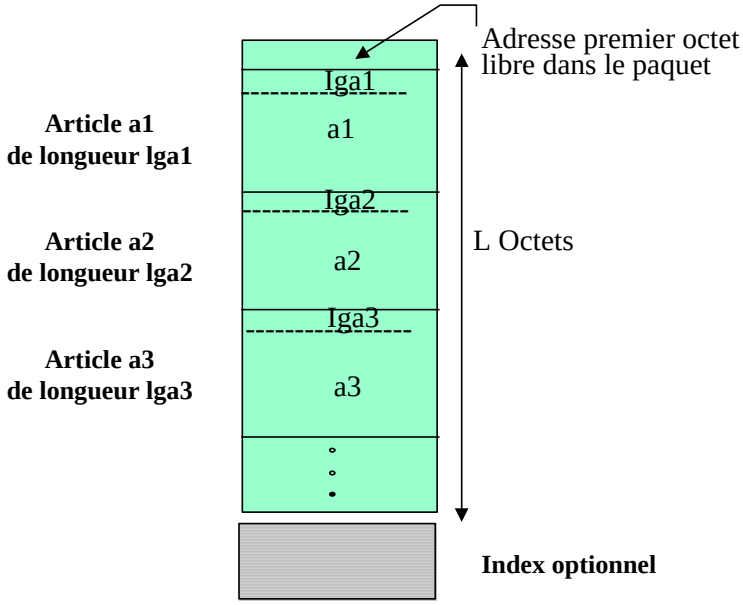

Figure III.7 — Structure interne d'un paquet

Lorsqu'un nouvel article est inséré dans un fichier, il est logé à la première place libre dans le paquet. S'il n'y a pas de place libre, on dit qu'il y a débordement. Il faut évidemment contrôler l'unicité de la clé d'un article lors des insertions. Cela nécessite de balayer tous les articles du paquet.

A partir de la clé d'un article, on calcule le numéro de paquet dans lequel l'article est placé à l'aide d'une fonction appelée **fonction de hachage** (Fig. III.8). Une fonction de hachage doit être choisie de façon à distribuer uniformément les articles dans les paquets. Plusieurs techniques sont possibles :

- le pliage, qui consiste à choisir et combiner des bits de la clé (par exemple par « ou exclusif ») ;
- les conversions de la clé en nombre entier ou flottant avec utilisation de la mantisse permettant d'obtenir également un numéro de paquet ;
- le modulo, sans doute la fonction la plus utilisée, qui consiste à prendre pour numéro de paquet le reste de la division de la clé par le nombre de paquets.

Ces techniques peuvent avantageusement être combinées.

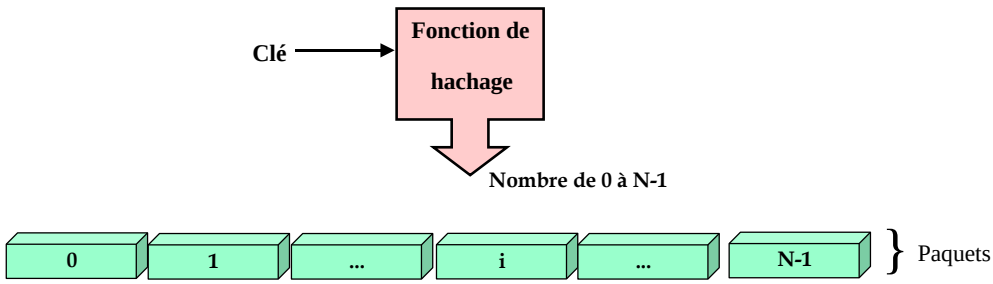

Figure III.8 — Illustration d'un fichier haché statique

Soit un fichier de 47 paquets et des articles de clé numérique. Le modulo 47 pourra alors être choisi comme fonction de hachage. Ainsi, l'article de clé 100 sera placé dans le paquet 6, l'article de clé 47 dans le paquet 0, celui de clé 123 dans le paquet 29, etc.

Le problème de débordement se pose lorsqu'un paquet est plein. Une première solution simple consiste à ne pas gérer de débordements et à répondre fichier saturé à l'utilisateur. Cela implique une mauvaise utilisation de la place occupée par le fichier, surtout si la distribution des articles dans les paquets est mauvaise. Des solutions plus satisfaisantes consistent à utiliser une technique de débordement parmi l'une des suivantes [Knuth73] :

- l'**adressage ouvert** consiste à placer l'article qui devrait aller dans un paquet plein dans le premier paquet suivant ayant de la place libre ; il faut alors mémoriser tous les paquets dans lequel un paquet plein a débordé ;

- le **chaînage** consiste à constituer un paquet logique par chaînage d'un paquet de débordement à un paquet plein ;

- le **rehachage** consiste à appliquer une deuxième fonction de hachage lorsqu'un paquet est plein; cette deuxième fonction conduit généralement à placer les articles dans des paquets de débordements.

Dans tous les cas, la gestion de débordements dégrade les performances et complique la gestion des fichiers hachés.

La méthode d'accès basée sur l'organisation hachée statique a plusieurs avantages. En particulier, elle s'adapte à des fichiers de clés quelconques, reste simple et donne d'excellentes performances tant qu'il n'y a pas de débordements : une lecture d'article s'effectue en une entrée-sortie (lecture du paquet) alors qu'une écriture en nécessite en général deux (lecture puis réécriture du paquet). Cependant, les débordements dégradent rapidement les performances. De plus, le taux d'occupation de la mémoire secondaire réellement utilisée peut rester assez éloigné de 1. Enfin, la taille d'un fichier doit être fixée a priori. Si le nombre d'articles d'un fichier devient plus important que prévu, le fichier doit être réorganisé.

4.2 Organisations hachées dynamiques

4.2.1 Principes du hachage dynamique

La première organisation **hachée dynamique** a été proposée pour des tables en mémoire [Knott71]. Puis plusieurs techniques fondées sur le même principe mais différentes ont été proposées pour étendre les possibilités du hachage à des fichiers dynamiques [Fagin79, Larson78, Litwin78, Larson80, Litwin80]. Le principe de base de ces différentes techniques est la digitalisation progressive de la fonction de hachage : la chaîne de bits résultat de l'application de la fonction de hachage à la clé est exploitée progressivement bit par bit au fur et à mesure des extensions du fichier.

Plus précisément, les méthodes dynamiques utilisent une fonction de hachage de la clé h(K) générant une chaîne de N bits, où N est grand (par exemple 32). La fonction est choisie de sorte qu'un bit quelconque de h(K) ait la même probabilité d'être à 1 ou à 0. Lors de la première implantation du fichier haché, seuls les M premiers bits de h(K) (avec M petit devant N) sont utilisés pour calculer le numéro de paquet dans lequel placer un article. Ensuite, lorsque le fichier est considéré comme saturé (par exemple, lorsqu'un premier paquet est plein), une partie du fichier (par exemple le paquet plein) est doublée : une nouvelle région est allouée pour cette partie et les articles de l'ancienne partie sont distribués entre l'ancienne partie et la nouvelle en utilisant le bit M+1 de la fonction de hachage. Ce processus d'éclatement est appliqué chaque fois que le fichier est saturé, de manière récursive. Ainsi, les bits (M+1), (M+2), (M+3) … de la fonction de hachage sont successivement utilisés et le fichier peut grandir jusqu'à 2^N paquets. Une telle taille est suffisante sous l'hypothèse que N soit assez grand.

Les méthodes de hachage dynamique diffèrent par les réponses qu'elles apportent aux questions suivantes :

(Q1) Quel est le critère retenu pour décider qu'un fichier haché est saturé ?

(Q2) Quelle partie du fichier faut-il doubler quand un fichier est saturé ?

(Q3) Comment retrouver les parties d'un fichier qui ont été doublées et combien de fois ont-elles été doublées ?

(Q4) Faut-il conserver une méthode de débordement, et si oui laquelle ?

Nous présentons ci-dessous deux méthodes qui nous paraissent des plus intéressantes: le hachage extensible [Fagin79] et le hachage linéaire [Litwin80]. Vous trouverez des méthodes sans doute plus élaborées dans [Larson80], [Lomet83], [Samet89] ainsi que des évaluations du hachage extensible dans [Scholl81] et du hachage linéaire dans [Larson82].

4.2.2 Le hachage extensible

Le **hachage extensible** [Fagin79] apporte les réponses suivantes aux questions précédentes :

(Q1) Le fichier est étendu dès qu'un paquet est plein ; dans ce cas un nouveau paquet est ajouté au fichier.

(Q2) Seul le paquet saturé est doublé lors d'une extension du fichier. Il éclate selon le bit suivant du résultat de la fonction de hachage appliquée à la clé h(K). Les articles ayant ce bit à 0 restent dans le paquet saturé, alors que ceux ayant ce bit à 1 partent dans le nouveau paquet.

(Q3) La fonction de hachage adresse un répertoire des adresses de paquets ; la taille du répertoire est 2**(M+P) où P est le niveau du paquet qui a éclaté le plus grand nombre de fois. Chaque entrée du répertoire donne l'adresse d'un paquet. Les 2**(P-Q) adresses correspondant à un paquet qui a éclaté Q fois sont identiques et pointent sur ce paquet. Ainsi, par l'indirection du répertoire, le système retrouve les paquets.

(Q4) La gestion de débordement n'est pas nécessaire.

Le hachage extensible associe donc à chaque fichier un répertoire des adresses de paquets. Au départ, M bits de la fonction de hachage sont utilisés pour adresser le répertoire. A la première saturation d'un paquet, le répertoire est doublé, et un nouveau paquet est alloué au fichier. Le paquet saturé est distribué entre l'ancien et le nouveau paquets, selon le bit suivant (M+1) de la fonction de hachage. Ensuite, tout paquet plein est éclaté en deux paquets, lui-même et un nouveau paquet alloué au fichier. L'entrée du répertoire correspondant au nouveau paquet est mise à jour avec l'adresse de ce nouveau paquet si elle pointait encore sur le paquet plein. Sinon, le répertoire est à nouveau doublé. En résumé, le hachage extensible peut être défini comme suit :

> **Notion III.21 : Hachage extensible (*Extensible hashing*)**
>
> Méthode de hachage dynamique consistant à éclater un paquet plein et à mémoriser l'adresse des paquets dans un répertoire adressé directement par les (M+P) premiers bits de la fonction de hachage, où P est le nombre d'éclatements maximal subi par les paquets.

Cette organisation est illustrée figure III.9. Nous montrons ici un répertoire adressé par 3 bits de la fonction de hachage. Le fichier avait été créé avec deux paquets et était adressé par le premier bit de la fonction de hachage (celui de droite). Puis le paquet 1 a éclaté et a été distribué entre le paquet 01 et 11. Le paquet 11 a éclaté à son tour et a été distribué entre le paquet 011 et le paquet 111. Le répertoire a donc été doublé deux fois (P = 2 alors que M = 1).

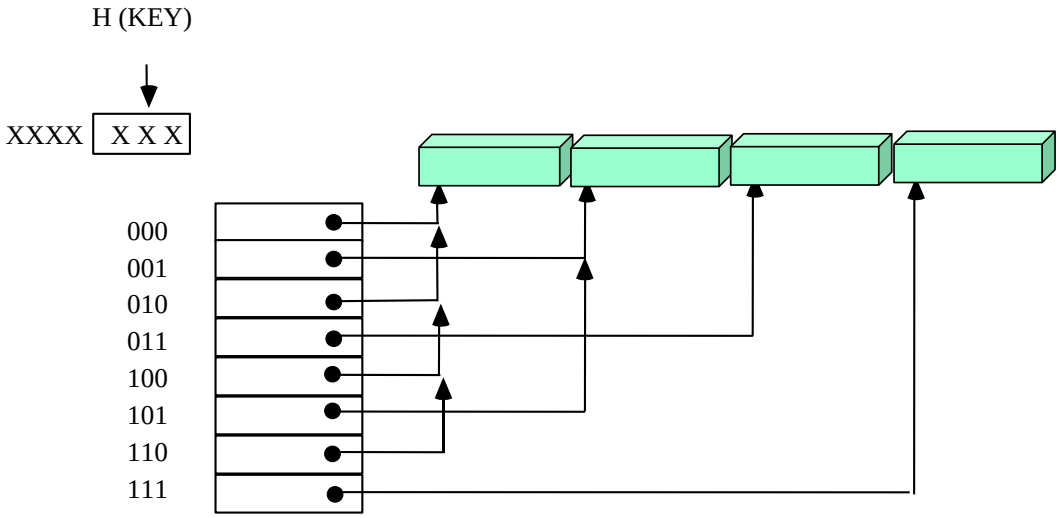

Figure III.9 — Répertoire et paquets d'un fichier haché extensible

Un fichier haché extensible est donc structuré en deux niveaux : le répertoire et les paquets. Soit P le niveau d'éclatement maximal du fichier. Le répertoire contient un en-tête qui indique la valeur de M+P, le nombre de bits de la fonction de hachage utilisés pour le paquet ayant le plus éclaté. Après l'en-tête figurent des pointeurs vers les paquets. Les M+P premiers bits de la fonction de hachage sont donc utilisés pour adresser le répertoire. Le premier pointeur correspond à la valeur 0 des (M+P) premiers bits de la fonction de hachage, alors que le dernier correspond à la valeur 2**(M+P) - 1, c'est-à-dire aux (M+P) premiers bits à 1. Soit Q le nombre d'éclatements subis par un paquet. A chaque paquet sont associés dans le répertoire 2**(P-Q) pointeurs qui indiquent son adresse. Le répertoire pointe ainsi plusieurs fois sur le même paquet, ce qui accroît sa taille.

L'insertion d'un article dans un fichier haché extensible nécessite tout d'abord l'accès au répertoire. Pour cela, les (M+P) bits de la clé sont utilisés. L'adresse du paquet dans lequel l'article doit être placé est ainsi lue dans l'entrée adressée du répertoire. Si le paquet est plein, alors celui-ci doit être doublé et son niveau d'éclatement Q augmenté de 1 ; un paquet frère au même niveau d'éclatement doit être créé ; les articles sont répartis dans les deux paquets selon la valeur du bit (M+Q+1) de la fonction de hachage. Le mécanisme d'éclatement de paquet est illustré figure III.10. Si le niveau du répertoire P est supérieur à Q, alors le répertoire doit simplement être mis à jour, 2**(P-Q+1) pointeurs étant forcés sur l'adresse du nouveau paquet. Si P est égal à Q, alors le répertoire doit être doublé.

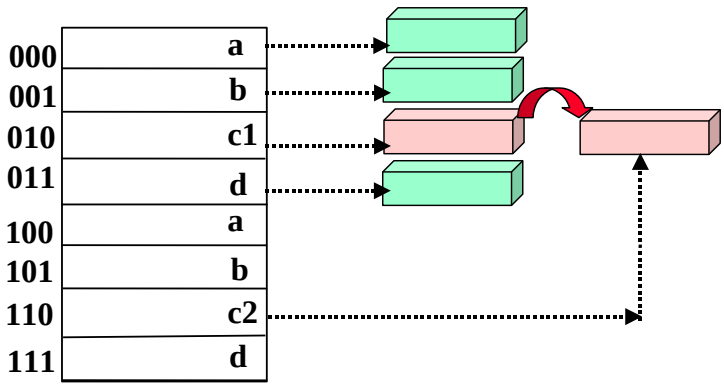

Figure III.10 — Éclatement de paquet dans un fichier haché extensible

En cas de suppression dans un paquet adressé par M+Q bits de la fonction de hachage, il est possible de tenter de regrouper ce paquet avec l'autre paquet adressé par M+Q bits s'il existe. Ainsi, la suppression d'un article dans un paquet peut théoriquement conduire à réorganiser le répertoire. En effet, si le paquet concerné est le seul paquet avec son frère au niveau d'éclatement le plus bas et si la suppression d'un article laisse assez de place libre pour fusionner les deux frères, la fusion peut être entreprise. Le niveau d'éclatement du répertoire doit alors être réduit de 1 et celui-ci doit être divisé par deux en fusionnant les blocs jumeaux.

4.2.3 Le hachage linéaire

Le **hachage linéaire** [Litwin80] apporte les réponses suivantes aux questions déterminantes d'une méthode de hachage dynamique :

(Q1) Le fichier est étendu dès qu'un paquet est plein, comme dans le hachage extensible ; un nouveau paquet est aussi ajouté au fichier à chaque extension ;

(Q2) Le paquet doublé n'est pas celui qui est saturé, mais un paquet pointé par un pointeur courant initialisé au premier paquet du fichier et incrémenté de 1 à chaque éclatement d'un paquet (donc à chaque saturation) ; lorsque ce pointeur atteint la fin de fichier, il est repositionné au début du fichier ;

(Q3) Un niveau d'éclatement P du fichier (initialisé à 0 et incrémenté lorsque le pointeur courant revient en début de fichier) est conservé dans le descripteur du fichier; pour un paquet situé avant le pointeur courant, (M+P+1) bits de la fonction de hachage doivent être utilisés alors que seulement (M+P) sont à utiliser pour adresser un paquet situé après le pointeur courant et avant le paquet 2**(M+P) ;

(Q4) Une gestion de débordement est nécessaire puisqu'un paquet plein n'est en général pas éclaté ; il le sera seulement quand le pointeur courant passera par son adresse. Une méthode de débordement quelconque peut être utilisée.

En résumé, il est possible de définir le hachage linéaire comme suit :

Notion III.22 : Hachage linéaire (*Linear hashing*)

Méthode de hachage dynamique nécessitant la gestion de débordement et consistant à : (1) éclater le paquet pointé par un pointeur courant quand un paquet est plein, (2) mémoriser le niveau d'éclatement du fichier afin de déterminer le nombre de bits de la fonction de hachage à appliquer avant et après le pointeur courant.

La figure III.11 illustre un fichier haché linéairement. Le pointeur courant est situé en début du 3e paquet (paquet 10). Les paquets 000 et 001, 100 et 101 (c'est-à-dire 0, 1, 4 et 5) sont adressés par les trois premiers bits de la fonction de hachage, alors que les paquets 10 et 11 (c'est-à-dire 2 et 3) sont seulement adressés par les deux premiers bits. Lors du prochain débordement, le paquet 10 (2) éclatera, quel que soit le paquet qui déborde.

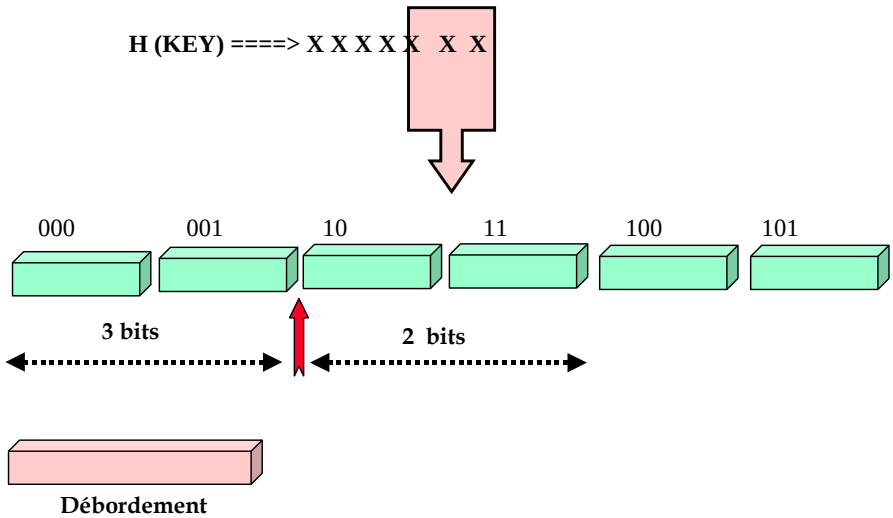

Figure III.11 — Fichier haché linéairement

Notons que le hachage linéaire peut aussi s'implémenter avec un répertoire. Dans ce cas, le pointeur courant est un pointeur sur le répertoire : il référence l'adresse du paquet suivant à éclater. L'avantage du hachage linéaire est alors la simplicité de l'algorithme d'adressage du répertoire ; on utilise tout d'abord M+P bits de la fonction de hachage ; si l'on est positionné avant le pointeur courant, on utilise un bit de plus, sinon on lit l'adresse du paquet dans le répertoire. A chaque éclatement, le répertoire s'accroît d'une seule entrée. L'inconvénient est bien sûr la nécessité de gérer des débordements.

L'insertion d'un article dans un fichier haché linéairement se fait très simplement : si P est le niveau d'éclatement du fichier, (M+P) bits de la clé sont tout d'abord pris en compte pour déterminer le numéro de paquet. Si le numéro obtenu est supérieur au pointeur courant, il est correct ; sinon, un bit supplémentaire de la fonction de hachage est utilisé pour déterminer le numéro de paquet. L'insertion s'effectue ensuite de manière classique, à ceci près que lorsqu'un paquet est saturé, le paquet pointé par le pointeur courant est éclaté ; ce pointeur courant est augmenté de 1 ; s'il atteint le paquet 2**(M+P) du fichier, il est ramené au début et le niveau d'éclatement du fichier est augmenté de un.

De manière analogue au répertoire du hachage extensible, la suppression d'article dans un paquet peut amener la fusion de deux paquets et donc le recul de 1 du pointeur courant. Attention : en général les paquets fusionnés n'incluent pas le paquet dans lequel a lieu la suppression ; la fusion peut amener des distributions d'articles en débordement.

Une variante de cette méthode consistant à changer la condition d'éclatement a été proposée dans [Larson80]. La condition retenue est l'atteinte d'un taux de remplissage maximal du fichier, le taux de remplissage étant le rapport de la place occupée par les articles sur la taille totale du fichier.

En résumé, les méthodes de hachage dynamique sont bien adaptées aux fichiers dynamiques, c'est-à-dire de taille variable, cependant pas trop gros pour éviter les saturations de paquets trop nombreuses et les accroissements de la taille du catalogue. Leurs limites sont encore mal connues. Le hachage extensible paraît plus robuste face aux mauvaises distributions de clés que le hachage linéaire. Par contre, la gestion d'un répertoire est plus lourde que celle d'un pointeur courant.

Le grand problème des méthodes par hachage reste l'absence de possibilités efficaces d'accès ordonné pour un tri total ou pour des questions sur des plages de valeur. Telle est la limite essentielle, qui nécessite l'emploi d'autres méthodes.

5. ORGANISATIONS ET MÉTHODES D'ACCÈS INDEXÉES

Dans cette section, nous étudions les principes de base des organisations avec index et les principales méthodes pratiques.

5.1 Principes des organisations indexées

5.1.1 Notion d'index

Le principe de base des organisations et méthodes d'accès indexées est d'associer à la clé d'un article son adresse relative dans le fichier à l'aide d'une « table des matières » du fichier. Ainsi, à partir de la clé de l'article, un accès rapide est possible par recherche de l'adresse relative dans la table des matières, puis par un accès en relatif à l'article dans le fichier. Les principales méthodes d'accès indexées se distinguent par le mode de placement des articles dans le fichier et par l'organisation de la table des matières, appelée **index**.

> **Notion III.23 : Index (*Index*)**
>
> Table (ou plusieurs tables) permettant d'associer à une clé d'article l'adresse relative de cet article.

La figure III.12 illustre cette notion d'index. Le fichier contient les articles a5, a2, a57, a3 et a10. L'index est rangé en fin de fichier comme le dernier article du fichier. Il contient une

entrée par article indiquant la clé de l'article et son adresse relative dans le fichier. L'index d'un fichier peut en général être rangé dans le fichier ou plus rarement dans un fichier spécial.

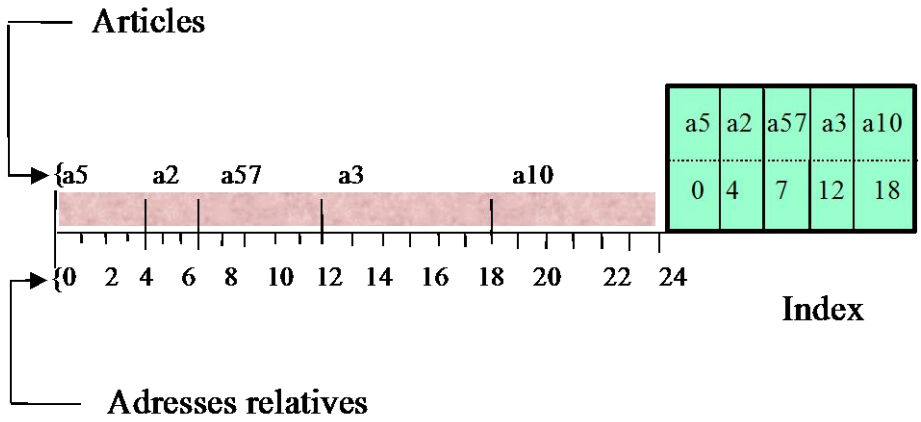

Figure III.12 — Exemple de fichier indexé

Les étapes successives exécutées pour l'accès à un article dans un fichier indexé sont les suivantes :

1. Accès à l'index qui est monté en mémoire dans un tampon.

2. Recherche de la clé de l'article désiré en mémoire afin d'obtenir l'adresse relative de l'article ou d'un paquet contenant l'article.

3. Conversion de l'adresse relative trouvée en adresse absolue par les couches internes du système de gestion de fichiers.

4. Accès à l'article (ou au paquet d'articles) sur disques magnétiques et transfert dans un tampon du système.

5. Transfert de l'article dans la zone du programme usager.

En général, l'accès à un article dans un fichier indexé nécessite une à trois entrées-sorties pour monter l'index en mémoire, puis une entrée sortie pour monter l'article en mémoire. Différentes variantes sont possibles, selon l'organisation des articles dans le fichier et de l'index.

5.1.2 Variantes possibles

Les variantes se distinguent tout d'abord par l'organisation de l'index. Celui-ci peut être trié ou non. Le fait que l'index soit trié autorise la recherche dichotomique. Ainsi, un index contenant n clés, divisé en blocs (d'une page) de b clés, nécessitera en moyenne n/2b accès pour retrouver une clé s'il n'est pas trié ; il suffira de $\log_2 n/b$ accès s'il est trié. Par exemple, avec $n = 10^6$, b = 100 clés, on obtient 10 accès si l'index est trié contre 5 000 accès sinon.

Un index d'un fichier indexé peut contenir toutes les clés (c'est-à-dire celles de tous les articles) ou seulement certaines. Un index qui contient toutes les clés est appelé index dense. Afin de différencier plus précisément les méthodes d'accès obtenues, il est possible d'introduire la notion de **densité d'un index**.

> **Notion III.24 : Densité d'un index (*Index key selectivity*)**
>
> Quotient du nombre de clés dans l'index sur le nombre d'articles du fichier.

La densité d'un index varie entre 0 et 1. Un index dense a donc une densité égale à 1. Dans le cas d'index non dense, toutes les clés ne figurent pas dans l'index. Aussi, les articles du fichier ainsi que l'index sont triés. Le fichier est divisé en paquets de taille fixe et chaque paquet correspond à une entrée en index contenant le doublet : <plus grande clé du paquet-adresse relative du paquet>. La figure III.13 illustre un index non dense et le fichier correspondant. Le paquet 1 contient les articles de clé 1, 3 et 7. La plus grande clé (7) figure dans l'index non dense, etc. L'index est composé ici d'un seul bloc contenant trois clés, la plus grande de chaque paquet.

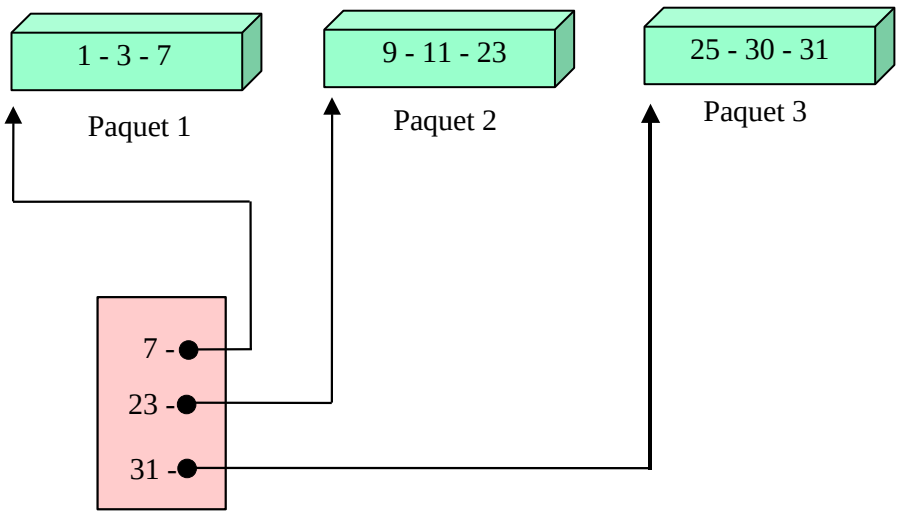

Figure III.13 — Exemple d'index non dense

Comme le fichier peut être trié ou non trié, et l'index dense ou non dense, trié ou non trié, diverses variantes sont théoriquement possibles. Deux méthodes sont particulièrement intéressantes : le fichier séquentiel non trié avec index trié dense, historiquement à la base de l'organisation IS3, et le fichier trié avec index non dense trié, sur lequel sont fondées des organisations telles ISAM, VSAM et UFAS. Il est impossible d'associer un index non dense à un fichier non trié.

5.1.3 Index hiérarchisé

Un index peut être vu comme un fichier de clés. Si l'index est grand (par exemple plus d'une page), la recherche d'une clé dans l'index peut devenir très longue. Il est alors souhaitable de créer un index de l'index vu comme un fichier de clés. Cela revient à gérer un index à plusieurs niveaux. Un tel index est appelé **index hiérarchisé**.

Notion III.25 : Index hiérarchisé (*Multilevel index*)

Index à n niveaux, le niveau k étant un index trié divisé en paquets, possédant lui-même un index de niveau k+1, la clé de chaque entrée de ce dernier étant la plus grande du paquet.

Un index hiérarchisé à un niveau est un index trié, généralement non dense, composé de paquets de clés. Un index hiérarchisé à n niveaux est un index hiéarchisé à n - 1 niveaux, possédant lui-même un index à un niveau. La figure III.14 illustre un index hiérarchisé à 3 niveaux. Le niveau 1 comporte trois paquets de clés. Le niveau 2 en comporte deux qui contiennent les plus grandes clés des paquets de niveau inférieur. Le niveau 3 est la racine et contient les plus grandes clés des deux paquets de niveau inférieur. La notion d'index hiérarchisé est indépendante du nombre de niveaux, qui peut grandir autant que nécessaire.

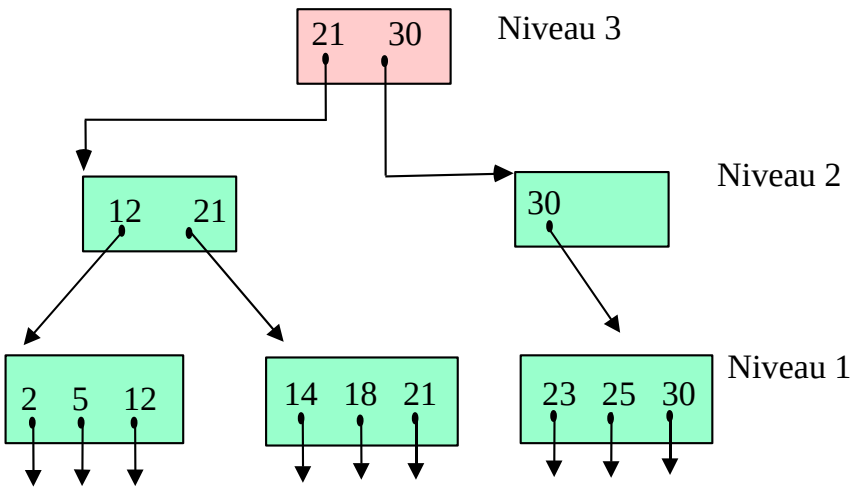

Figure III.14 — Exemple d'index hiérarchisé

5.1.4 Arbres B

Afin de mieux caractériser la notion d'index hiérarchisé et de la rendre indépendante des particularités d'implantation, on a été amené à introduire une structure d'arbre, avec un nombre variable de niveaux. Cette structure, appelée **arbre B** [Bayer72, Comer79], peut être introduite formellement comme suit :

Notion III.26: Arbre B (*B-tree*)

Un arbre B d'ordre m est un arbre au sens de la théorie des graphes tel que :

1°) Toutes les feuilles sont au même niveau ;

2°) Tout nœud non feuille a un nombre NF de fils tel que $m+1 \leq NF \leq 2m+1$, sauf la racine, qui a un nombre NFR de fils tel que $0 \leq NFR \leq 2m+1$.

La figure III.15 représente un arbre balancé d'ordre 2. La racine a deux fils. Les deux autres nœuds non feuilles ont trois fils.

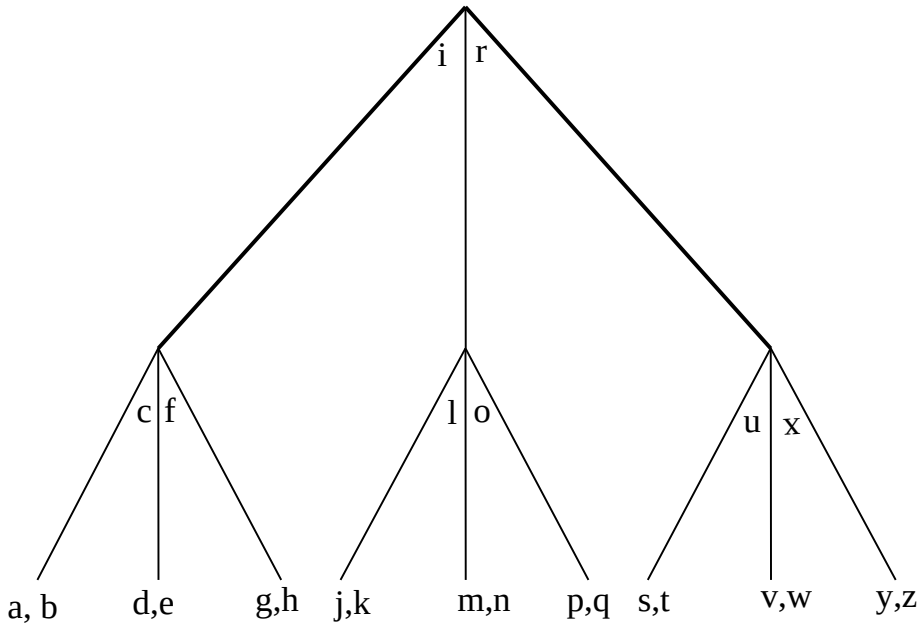

Figure III.15 — Arbre B d'ordre 2

Un arbre B peut être utilisé pour constituer un index hiérarchisé d'un fichier. Dans ce cas, les nœuds représentent des pages de l'index. Ils contiennent des clés triées par ordre croissant et des pointeurs de deux types : les pointeurs internes désignent des fils et permettent de définir les branches de l'arbre, alors que les pointeurs externes désignent des pages de données (en général, des adresses relatives d'articles). La figure III.16 précise la structure d'un nœud. Une clé xi d'un nœud interne sert de séparateur entre les deux branches internes adjacentes (Pi-1 et Pi). Un nœud contient entre m et 2m clés, à l'exception de la racine qui contient entre 1 et 2m clés.

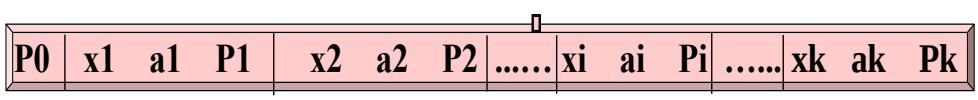

Figure III.16 — Structure d'un nœud d'un arbre B

De plus, l'ensemble des clés figurant dans l'arbre B doit être trié selon l'ordre post-fixé induit par l'arbre, cela afin de permettre les recherches en un nombre d'accès égal au nombre de niveaux. Plus précisément, en désignant par K(Pi) l'ensemble des clés figurant dans le sous-arbre dont la racine est pointée, on doit vérifier que :

1. (x1, x2…xK) est une suite croissante de clés ;
2. Toute clé y de K(P0) est inférieure à x1 ;
3. Toute clé y de K(P1) est comprise entre xi et xi+1 ;
4. Toute clé y de K(PK) est supérieure à xk.

La figure III.17 représente un exemple d'index sous forme d'arbre B d'ordre 2. Cet arbre contient les valeurs de clé de 1 à 26. Les flèches entre les nœuds représentent les pointeurs internes, les traits courts issus d'une clé les pointeurs externes vers les articles. Dans la suite, nous omettrons les pointeurs externes, qui seront donc implicites.

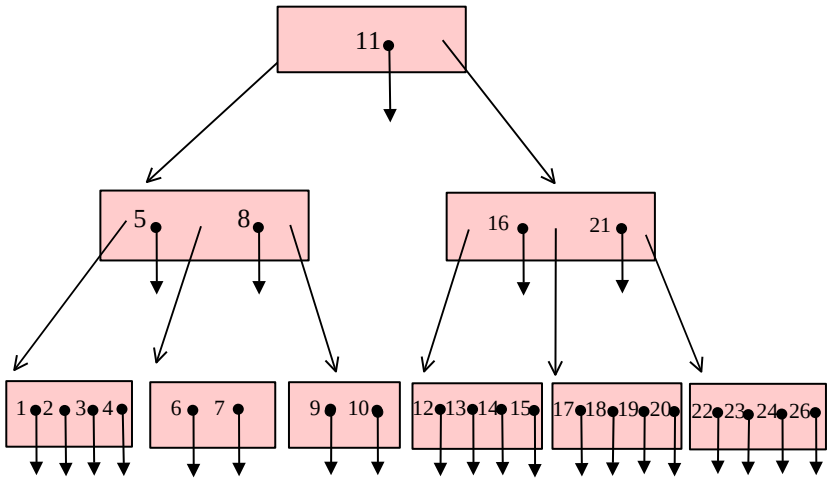

Figure III.17 — Exemple d'index sous forme d'arbre B

La recherche d'une clé dans un arbre B s'effectue en partant de la racine. En règle générale, les valeurs contenues dans un nœud partitionnent les valeurs possibles de clés en un nombre d'intervalles égal au nombre de branches. Ainsi, si la valeur cherchée est inférieure à la première clé du nœud, on choisit la première branche ; si elle est comprise entre la première clé et la deuxième clé, on choisit la deuxième branche, etc. Si une clé n'est pas trouvée après recherche dans un nœud terminal, c'est qu'elle n'existe pas.

Le nombre de niveaux d'un arbre B est déterminée par son degré et le nombre de clés contenues. Ainsi, dans le pire des cas, si l'arbre est rempli au minimum, il existe :

- une clé à la racine,
- deux branches en partent avec m clés,

- (m+1) branches partent de ces dernières avec m clés,
- etc.

Pour un arbre de niveaux h, le nombre de clés est donc :

$$N = 1 + 2\,m\,(1+ (m+1) + (m+1)^2 + \ldots + (m+1)^{h-2})$$

soit, par réduction du développement limité :

$$N = 1 + 2\,((m+1)^{h-1}-1).$$

D'où l'on déduit que pour stocker N clés, il faut :

$h = 1 + \log_{m+1}((N+1)/2)$ niveaux.

Par exemple, pour stocker 1 999 999 clés avec un arbre B de degré 99, $h = 1 + \log_{100}10^6 = 4$. Au maximum, quatre niveaux sont donc nécessaires. Cela implique qu'un article d'un fichier de deux millions d'articles avec un index hiérarchisé organisé comme un arbre B peut être cherché en quatre entrées-sorties.

L'insertion d'une clé dans un arbre B est une opération complexe. Elle peut être définie simplement de manière récursive comme suit :

a) Rechercher le nœud terminal qui devrait contenir la clé à insérer et l'y insérer en bonne place ;

b) Si le nombre de clés après insertion de la nouvelle clé est supérieur à 2 m, alors migrer la clé médiane au niveau supérieur, en répétant la procédure d'insertion dans le nœud supérieur.

A titre d'exemple, la figure III.18 représente les étapes nécessaires à l'insertion de la clé (25) dans l'arbre B représenté figure III.17. Tout d'abord, la place de la clé 25 est recherchée. Celle-ci doit être insérée dans le dernier nœud à droite (étape a). Cela provoque un éclatement du nœud qui a maintenant plus de 2 m clés, soit 4. La clé médiane 24 doit être remontée au niveau supérieur. Elle est alors insérée après 21 (étape b). Le nœud ayant trois clés, aucun autre éclatement n'est nécessaire.

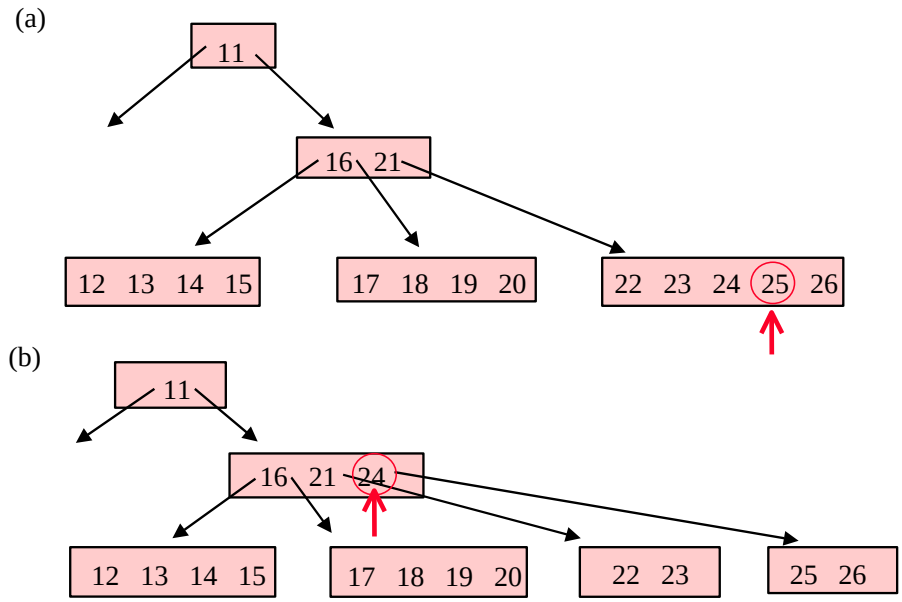

Figure III.18 — Insertion de la clé 25

La suppression d'une clé soulève également des problèmes. Tout d'abord, si l'on supprime une clé non terminale, il faut remonter la clé suivante pour garder le partitionnement. De plus, si un nœud a moins de m clés, il faut le regrouper avec le précédent de même niveau afin de respecter la définition et de conserver entre m et 2m clés dans un nœud non racine.

Une variante de l'arbre B tel que nous l'avons décrit pour réaliser des index est l'arbre B* [Knuth73, Comer79], dans lequel l'algorithme d'insertion essaie de redistribuer les clés dans un nœud voisin avant d'éclater. Ainsi, l'éclatement ne se produit que quand deux nœuds consécutifs sont pleins. Les deux nœuds éclatent alors en trois. Les pages des index de type arbre B* sont donc en général mieux remplies que celles des index de type arbre B.

5.1.5 Arbre B+

L'utilisation des arbres B pour réaliser des fichiers indexés tels que décrits ci-dessus conduit à des traitements séquentiels coûteux. En effet, l'accès selon l'ordre croissant des clés à l'index nécessite de nombreux passages des pages externes aux pages internes. Pour éviter cet inconvénient, il a été proposé de répéter les clés figurant dans les nœuds internes au niveau des nœuds externes. De plus, les pages correspondant aux feuilles sont chaînées entre elles. On obtient alors un **arbre B+.** L'arbre B+ correspondant à l'arbre B de la figure III.17 est représenté figure III.19. Les clés 11, 8 et 21 sont répétées aux niveaux inférieurs. Les pointeurs externes se trouvent seulement au niveau des feuilles.

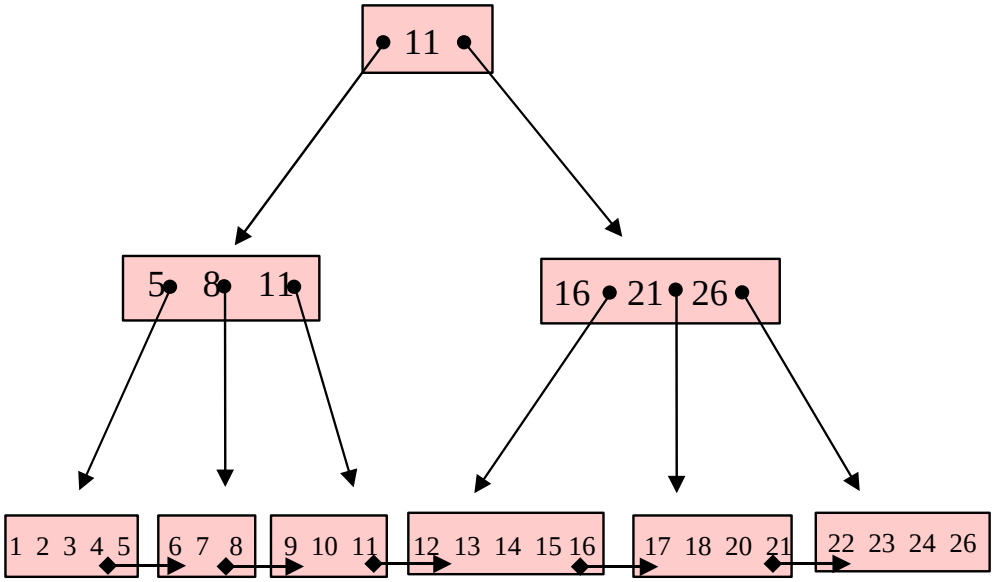

Figure III.19 — Exemple d'index sous forme d'arbre B+

Les arbres B+ peuvent être utilisés pour réaliser des fichiers à index hiérarchisés de deux manières au moins :

- L'arbre B+ peut être utilisé pour implémenter seulement les index. Autrement dit, les articles sont stockés dans un fichier séquentiel classique et l'arbre B+ contient toutes les clés ainsi que les adresses d'articles. Une telle organisation est proche de celle proposée par IBM sur les AS 400. Pour des raisons historiques, cette méthode s'appelle IS3.

- L'arbre B+ peut être utilisé pour implémenter fichiers et index. Dans ce cas, les pointeurs externes sont remplacés par le contenu des articles. Les articles sont donc triés. Seules les clés sont déplacées aux niveaux supérieurs qui constituent un index non dense. Cette méthode correspond à l'organisation séquentielle indexée régulière d'IBM sur MVS connue sous le nom de VSAM, et également à celle de BULL sur DPS7, connue sous le nom de UFAS.

5.2 Organisation indexée IS3

Cette organisation est voisine de celle développée tout d'abord sur les systèmes de la série 3 d'IBM. Les articles sont rangés en séquentiel dans un fichier dont l'index est dense et organisé sous forme d'un arbre B+.

> **Notion III.27 : Fichier indexé (*Indexed file*)**
>
> Fichier séquentiel non trié, d'index trié dense organisé sous la forme d'un arbre B+.

L'interprétation de la définition que constitue la notion III.27 soulève plusieurs problèmes. Tout d'abord, comment est défini l'ordre de l'arbre B+ qui constitue l'index ? La solution consiste à diviser l'index en pages (une page = 1 à p secteurs). Lors de la première écriture,

les pages ne sont pas complètement remplies. Lors d'une insertion, si une page est pleine elle est éclatée en deux pages à demi pleines. La clé médiane est remontée au niveau supérieur.

Un deuxième problème consiste à garder un index dense. En fait, celui-ci est dense au dernier niveau. Autrement dit, toutes les clés d'articles sont gardées au plus bas niveau. Ainsi, quand une page éclate, la clé médiane devient la plus grande clé de la page gauche résultant de l'éclatement. Cette clé est donc dupliquée au niveau supérieur de l'index. La figure III.20 illustre une insertion dans un fichier indexé IS3. L'insertion provoque l'éclatement de l'unique page index et la création d'une page index de niveau supérieur.

<voir Maîtriser les BD>

Figure III.20 — Insertion dans un fichier IS3

Un dernier problème est celui du stockage de l'index. Celui-ci peut être stocké en fin de fichier. Il est ainsi possible de lire la page supérieure de l'index en mémoire centrale lors du début d'un travail sur un fichier, puis de la réécrire en fin de travail. Il est aussi possible, avec une telle méthode d'enregistrement des index, de garder les versions historiques des index à condition que les nouveaux articles écrits le soient après le dernier index enregistré, c'est-à-dire en fin de fichier.

La méthode d'accès et l'organisation associée IS3 présentent plusieurs avantages : l'insertion des articles est simple puisqu'elle s'effectue en séquentiel dans le fichier ; il est possible de garder des versions historiques des index. Les performances de la méthode sont satisfaisantes. Si m est le nombre de clés par page d'index, du fait de l'organisation de l'index en arbre B+, le nombre d'entrées-sorties nécessaires pour lire un article dans un fichier de N articles reste inférieur ou égal à $2 + \log_{(m/2)} ((N+1)/2)$. Une écriture nécessite en général deux accès, sauf dans les cas d'éclatement de page index où il faut une lecture et deux écritures de plus par niveau éclaté. En pratique, et à titre d'exemple, un fichier de moins de 10^6 articles ne nécessitera pas plus de trois entrées-sorties en lecture.

Cette méthode présente cependant trois inconvénients sérieux :

- Du fait de la séparation des articles et de l'index, les mouvements de bras des disques sont en général importants.
- La lecture en séquentiel par clé croissante doit se faire par consultation de l'index et est en conséquence très coûteuse.
- L'index est dense et donc de grande taille si aucune technique de compactage de clés n'est mise en œuvre.

5.3 Organisation séquentielle indexée ISAM

5.3.1 Présentation générale

Il s'agit de l'organisation IBM utilisée dans les systèmes DOS, OS/VS, MVS et connue sous le nom ISAM (*Indexed Sequential Acces Method*) [IBM78]. Le fichier est organisé physiquement selon un découpage en pistes et cylindres. Cette méthode très ancienne reste populaire malgré son manque d'indépendance physique.

> **Notion III.28 : Fichier séquentiel indexé (*Indexed sequential file*)**
>
> Fichier trié d'index trié non dense composé d'une zone primaire et d'une zone de débordement ; une piste saturée déborde dans une extension logique constituée par une liste d'articles en débordement.

Un fichier ISAM comporte trois zones logiques :

- une zone primaire où l'on écrit les articles à la première écriture ;
- une zone de débordement où l'on transfère les articles lors des additions au fichier ;
- une zone d'index où l'on écrit les index.

Ci-dessous nous étudions successivement chacune des zones.

5.3.2 Étude de la zone primaire

La zone primaire se compose de cylindres successifs dont certaines pistes sont réservées pour l'index et les zones de débordement. En zone primaire, les articles sont enregistrés par ordre croissant des clés. Ils peuvent être bloqués ou non. Lors de la première écriture du fichier, les articles doivent être délivrés au système de fichiers par ordre croissant des clés. La figure III.21 illustre un fichier ISAM après une première écriture. Ce fichier est composé de deux cylindres. Chaque cylindre comporte deux pistes de données et une piste réservée pour les index.

<voir Maîtriser les BD>

Figure III.21 — Fichier ISAM après une première écriture des articles 1, 3, 5, 7, 9, 14, 17, 21, 23, 27, 29, 31 (dans l'ordre)

5.3.3 Étude de la zone de débordement

Il existe deux types de zones de débordement : la zone de débordement de cylindre composée de quelques pistes sur chaque cylindre et la zone de débordement indépendante, composée des derniers cylindres du fichier (voir figure III.22).

<voir Maîtriser les BD>

Figure III.22 — Zones de débordement d'un fichier ISAM.

En zone de débordement, les articles ne sont pas bloqués. Ils sont chaînés entre eux afin de reconstituer la piste qui a débordé de manière logique. Quand un article est inséré, on recherche sa séquence dans la piste logique. S'il est placé en zone primaire, les articles suivants sont déplacés et le dernier est écrit en zone de débordement. Les chaînages sont mis à jour. S'il vient en zone de débordement, il est écrit dans cette zone et est inséré en bonne place dans la chaîne des articles en débordement.

La zone de débordement de cylindre est tout d'abord utilisée. Lorsqu'elle est saturée, la zone de débordement indépendante est utilisée. Dans ce cas, comme le chaînage est effectué par ordre croissant des clés, il est possible qu'il parte de la zone de débordement de cylindre, pointe en zone de débordement indépendante, puis revienne en zone de débordement de cylindre, etc. Alors, la méthode devient particulièrement peu performante pour rechercher un article dans une piste ayant débordé, du fait des déplacements de bras.

5.3.4 Étude de la zone index

Il existe obligatoirement deux niveaux d'index et optionnellement trois : les index de pistes, les index de cylindres et le (ou les) index maître(s).

Le premier niveau d'index obligatoire est l'index de piste. Il en existe un par cylindre, en général contenu sur la première piste du cylindre. Chaque entrée correspond à une piste du cylindre et fournit la plus grande clé de la piste logique en zone de débordement ainsi que l'adresse du premier article en zone de débordement s'il existe. La figure III.23 illustre le format de l'index de piste. Chaque piste est décrite par une double entrée comportant, en plus des adresses, la plus grande clé en zone primaire et la plus grande clé en zone de débordement.

<voir Maîtriser les BD>

Figure III.23 — Format de l'index de piste

Le deuxième niveau de l'index obligatoire est l'index de cylindre. Il existe un index de cylindre par fichier. Cet index contient une entrée par cylindre comportant la plus grande clé du cylindre ainsi que l'adresse du cylindre. L'index de cylindre est généralement rangé dans une zone particulière, par exemple en début de zone de débordement indépendante. Cet index permet, à partir d'une clé d'article, de sélectionner le cylindre où doit se trouver l'article.

Le troisième niveau d'index est optionnel : c'est l'index maître. Il est utilisé quand l'index de cylindre est trop grand afin d'accélérer les recherches. Il existe une entrée en index maître par piste de l'index de cylindre. Cette entrée contient l'adresse de la piste et la valeur de la plus grande clé dans la piste.

5.3.5 Vue d'ensemble

La figure III.24 donne une vue d'ensemble d'un fichier ISAM après insertion d'articles, avec seulement deux niveaux d'index. Bien que seulement les chaînages du premier cylindre soient

représentés, on notera le grand nombre de chaînages. La première piste logique est constituée des articles a0, a1, a2 et a4 ; a0, a1, a2 sont en zone primaire ; a3 et a4 sont en zone de débordement de cylindre. Les articles a5, a6, a7, a8 et a9 sont rangés dans la deuxième piste logique ; a8 est en débordement de cylindre et a9 en zone de débordement indépendante.

<voir Maîtriser les BD>

Figure III.24 — Vue d'ensemble d'un fichier ISAM

(Les pointeurs issus du cylindre 1 ne sont pas représentés.)

Les avantages de la méthode sont que le fichier est trié, ce qui facilite l'accès en séquentiel trié, ainsi que les temps d'accès tant que le fichier n'a pas débordé (3 E/S pour lire un article). Les inconvénients sont essentiellement liés aux débordements. La gestion des débordements est complexe et dégrade les performances, de sorte qu'il est nécessaire de réorganiser périodiquement les fichiers ayant débordé. Le fait que la méthode d'accès ne soit pas indépendante des caractéristiques physiques du support (pistes, cylindres…) améliore les performances, mais rend le changement de support difficile. En fait, les performances dépendent des débordements. Si une piste comporte des articles en débordement, une lecture nécessitera approximativement 3+[d/2] entrées-sorties alors qu'une écriture demandera 2+[d/2]+4 entrées-sorties, cela afin de trouver la position de l'article, de l'écrire et de mettre à jour les chaînages. Cela peut devenir très coûteux.

5.4 Organisation séquentielle indexée régulière VSAM

5.4.1 Présentation générale

Il s'agit de l'organisation IBM utilisée dans les systèmes IBM et connue sous le nom de VSAM (*Virtual Sequential Acces Method*) [IBM87]. A la différence de ISAM, VSAM assure l'indépendance des fichiers au support physique et une réorganisation progressive du fichier, sans gestion de débordement. VSAM est une organisation basée sur les principes des arbres B+.

> **Notion III.29 : Fichier séquentiel indexé régulier (*Virtual sequential file*)**
>
> Fichier trié d'index trié non dense dont l'ensemble fichier plus index est organisé sous forme d'un arbre B+.

Afin d'assurer l'indépendance des fichiers aux supports physiques, des divisions logiques sont introduites. Le fichier est divisé en aires, une aire étant un ensemble de pistes du même cylindre ou de cylindres contigus. Chaque aire est elle-même divisée en intervalles, un intervalle étant généralement composé d'une partie de piste ou de plusieurs pistes consécutives lue(s) en une seule entrée-sortie. L'intervalle correspond à la feuille de l'arbre B+. Lorsqu'un intervalle est saturé, il éclate en deux intervalles ; lorsqu'une aire est saturée, elle éclate aussi en deux aires, si bien que le fichier se réorganise de lui-même par incréments.

Cette organisation résulte d'une étude critique de ISAM. Cette fois, le fichier est plus indépendant de la mémoire secondaire : la piste est remplacée par l'intervalle qui peut être une fraction de piste ou plusieurs pistes, le cylindre est remplacé par la notion d'aire. Les débordements ne perturbent plus l'organisation puisque le mécanisme de débordement est celui de l'arbre B+, c'est-à-dire qu'un intervalle ou une aire saturés éclatent en deux.

5.4.2 Étude de la zone des données

Cette zone qui contient les articles est donc divisée en intervalles et aires, comme sur la figure III.25. Lors de la première écriture, comme avec des fichiers séquentiels indexés ISAM, les articles sont enregistrés triés, c'est-à-dire que le programme doit les délivrer à la méthode d'accès dans l'ordre croissant des clés. Cette fois, les mises à jour sont prévues: de la place libre est laissée dans chaque intervalle et des intervalles libres sont laissés dans chaque aire afin de permettre les insertions de nouveaux articles.

A titre d'exemple, le fichier de la figure III.25 a été créé avec les paramètres suivants :

- pourcentage d'octets libres par intervalle = 25 ;
- pourcentage d'intervalles libres par aire = 25.

Lors de la première écriture, le programme créant le fichier a délivré au système les articles suivants (dans l'ordre) : 1, 5, 7, 9, 15, 20, 22, 27, 30, 31, 33, 37, 40, 43, 47, 51, 53.

<voir Maîtriser les BD>

Figure III.25 — Fichier séquentiel indexé régulier après première écriture

L'algorithme d'insertion d'un article consiste à déterminer, grâce aux index, l'intervalle qui doit contenir l'article. Ensuite, deux cas sont possibles :

a) Si l'intervalle n'est pas plein, alors l'article est rangé dans l'intervalle, en bonne place dans l'ordre lexicographique des clés. Par exemple, l'insertion de l'article de clé 10 conduira à la substitution d'intervalle représentée figure III.26.

<voir Maîtriser les BD>

Figure III.26 — Insertion de l'article de clé 10 dans le fichier précédent

b) Si l'intervalle est plein, alors il éclate en deux intervalles à demi pleins. Deux sous-cas sont encore possibles :

b1) S'il existe un intervalle libre dans l'aire, celui-ci est utilisé afin de stocker l'un des deux intervalles résultant de l'éclatement. Par exemple, l'insertion de l'article de clé 13 dans le fichier obtenu après insertion de l'article de clé 10 (Fig. III.26) conduit au fichier représenté figure III.27.

b2) S'il n'existe pas d'intervalle libre dans l'aire, on alloue alors une aire vide au fichier et on éclate l'aire saturée en deux à demi pleines ; pour cela, la moitié des intervalles contenant les articles de plus grandes clés sont recopiés dans l'aire nouvelle. Par exemple, l'insertion des articles de clés 11 et 12 dans le fichier de la figure III.27 conduit au fichier représenté figure III.28. Une nouvelle aire a été créée afin de dédoubler la première aire du fichier qui était saturée.

<voir Maîtriser les BD>

Figure III.27 — Fichier après insertion des articles de clés 10 et 13

<voir Maîtriser les BD>

Figure III.28 — Fichier après insertion des articles 11 et 12

5.4.3 Étude de la zone index

Il peut exister deux ou trois niveaux d'index. Le premier est appelé **index d'intervalles**. Il en existe un par aire. Chaque entrée contient la plus grande clé de l'intervalle correspondant ainsi que son adresse. Le deuxième niveau est appelé **index d'aires**. Il en existe un par fichier. Chaque entrée contient la plus grande clé de l'aire correspondante ainsi que l'adresse de l'aire. Il est également possible, dans le cas où l'index d'aire est trop grand (par exemple plus d'une piste), de faire gérer au système un index de troisième niveau appelé **index maître** et permettant d'accéder directement à la partie pertinente de l'index d'aire. A titre d'illustration, la figure III.29 représente les index d'intervalles et d'aires du fichier représenté figure III.28. Chaque entrée représente une clé suivie d'une adresse d'intervalle ou d'aire.

<voir Maîtriser les BD>

Figure III.29 — Index du fichier représenté figure III.28

5.4.4 Vue d'ensemble

La figure III.30 donne une vue d'ensemble d'un autre fichier VSAM composé de deux aires, avec seulement deux niveaux d'index. Chaque aire possède donc un index d'intervalles. L'index d'aires constitue ici la racine de l'arbre B+. Il indique que 17 est la plus grande clé de l'aire 0, alors que 32 est la plus grande de l'aire 1.

Les avantages de la méthode sont que le fichier est physiquement trié, ce qui facilite les accès en séquentiel trié, ainsi que les temps d'accès à un article : la lecture s'effectue en général en trois entrées-sorties. Les inconvénients sont peu nombreux. Cependant l'écriture peut parfois être très longue dans le cas d'éclatement d'intervalles ou, pire, dans le cas d'éclatement d'aires. En fait, les écritures ne s'effectuent pas en un temps constant : certaines sont très rapides (4 E/S lorsqu'il n'y a pas d'éclatement), d'autres sont très longues (lorsqu'il y a éclatement d'aire).

<voir Maîtriser les BD>

Figure III.30 — Vue d'ensemble d'un fichier séquentiel indexé régulier

6. MÉTHODES D'ACCÈS MULTI-ATTRIBUTS

Les méthodes d'accès étudiées jusque-là permettent de déterminer l'adresse d'un article dans un fichier à partir de la valeur d'une donnée unique de l'article, appelée clé. Nous allons maintenant présenter des méthodes qui permettent d'accéder rapidement à un groupe d'articles à partir des valeurs de plusieurs données, aucune d'entre elles n'étant en général discriminante.

6.1 Index secondaires

Le problème posé est de déterminer l'adresse d'un ou plusieurs articles à partir de la valeur de plusieurs données de cet article, aucune d'entre elles ne permettant en principe de déterminer à elle seule le ou les articles désirés. La solution la plus simple est d'utiliser plusieurs index, chacun d'eux n'étant pas discriminant. On appelle **index secondaire** un index non discriminant.

> **Notion III.30 : Index secondaire (*Secondary Index*)**
>
> Index sur une donnée non discriminante, donnant pour chaque valeur de la donnée la liste des adresses d'articles ayant cette valeur.

Par exemple, un index secondaire d'un fichier décrivant des vins pourra porter sur le champ cru. Les entrées correspondront aux différents crus (Chablis, Medoc, Volnay, etc.) et donneront pour chaque cru la liste des adresses d'articles correspondant à ce cru. Ainsi, en accédant l'entrée Volnay, on trouvera la liste des Volnay.

Un index secondaire est souvent organisé comme un arbre B, bien que d'autres organisations soient possibles. En particulier, un index secondaire peut être un fichier à part, servant d'index au fichier principal. On parle alors de **fichier inverse**, le fichier contenant l'index apparaissant comme une structure de données en quelque sorte renversée par rapport au fichier principal. Un fichier avec un index secondaire (ou un fichier inverse) est en général indexé ou haché sur une clé discriminante (par exemple, le numéro de vin). On parle alors de **clé primaire**. Par opposition, le champ sur lequel l'index secondaire est construit (par exemple, le cru) est appelé **clé secondaire**. Un fichier peut bien sûr avoir plusieurs clés secondaires, par exemple cru et région pour le fichier des vins.

La question qui se pose alors concerne la recherche d'un article dans le fichier. Si plusieurs clés secondaires sont spécifiées (par exemple, cru = "Volnay" et région= "Bourgogne"), on peut avoir intérêt à choisir un index (ici le cru), puis à lire les articles correspondant en vérifiant les autres critères (ici, que la région est bien Bourgogne). Dans certains cas où aucune donnée n'est a priori plus discriminante qu'une autre, on aura au contraire intérêt à lire les différentes entrées d'index sélectionnées, puis à faire l'intersection des listes d'adresses d'articles répondant aux différents critères. Finalement, un accès aux articles répondant à tous les critères (ceux dont l'adresse figure dans l'intersection) sera suffisant.

Un autre problème posé par la gestion d'index secondaires est celui de la mise à jour. Lors d'une mise à jour d'un article du fichier principal, il faut mettre à jour les index secondaires. Bien sûr, si la valeur d'indexation est changée, il faut enlever l'article de l'entrée correspondant à l'ancienne valeur, puis l'insérer dans l'entrée correspondant à la nouvelle valeur. Cette dernière doit être créée si elle n'existe pas. Plus coûteux, avec les méthodes d'accès utilisant l'éclatement de paquets, il faut aller modifier dans les index secondaires les adresses des articles déplacés lors des déplacements d'articles. On peut avoir alors intérêt à garder des références invariantes aux déplacements d'articles dans les index secondaires. Une solution consiste à garder les clés primaires à la place des adresses, mais cela coûte en général un accès via un arbre B.

En résumé, bien que présentant quelques difficultés de gestion, les index secondaires sont très utiles pour accélérer les recherches d'articles à partir de valeurs de données. Ils sont largement utilisés dans les SGBD. S'ils permettent bien d'améliorer les temps de réponse lors des interrogations, ils pénalisent les mises à jour. Il faut donc indexer les fichiers seulement sur les données souvent documentées lors de l'interrogation.

6.2 Hachage multi-attribut

Avec le hachage, il est possible de développer des méthodes d'accès portant sur de multiples champs (ou attributs) en utilisant plusieurs fonctions de hachage, chacune étant appliquée à un champ. En appliquant N fonctions de hachage à N attributs d'un article, on obtient un vecteur constituant une adresse dans un espace à N dimensions. Chaque coordonnée correspond au résultat d'une fonction de hachage. A partir de cette adresse, plusieurs méthodes sont possibles pour calculer le numéro de paquet où ranger l'article. Une telle approche est la base des méthodes dites de **placement multi-attribut**.

> **Notion III.31 : Placement multi-attribut (*Multiattribute clustering*)**
>
> Méthode d'accès multidimensionnelle dans laquelle l'adresse d'un paquet d'articles est déterminée en appliquant des fonctions de hachage à différents champs d'un article.

6.2.1 Hachage multi-attribut statique

Une méthode simple est le **placement multi-attribut statique**. La méthode consiste à concaténer les résultats des fonctions de hachage dans un ordre prédéfini. La chaîne binaire obtenue est alors directement utilisée comme adresse de paquet. On obtient donc un hachage statique à partir d'une fonction portant sur plusieurs attributs. Si A1, A2 … An sont les attributs de placement, chacun est transformé par application d'une fonction de hachage en bi bits de l'adresse du paquet. Si hi est la fonction de hachage appliquée à l'attribut bi, le numéro de paquet où placer un article est obtenu par a = h1(A1) || h2(A2) || … hn(An), où la double barre verticale représente l'opération de concaténation. Le nombre total de bits du numéro de paquet est B = b1+b2+… bn. Pour retrouver un article à partir de la valeur d'un attribut, disons Ai, le système devra seulement balayer les paquets d'adresse x || hi(Ai) ||y, où x représente toute séquence de b1 +… bi-1 bits et y toute séquence de bi+1+ … bn bits. Ainsi, le nombre de paquets à balayer sera réduit de 2**B à 2**(B-bi). Le nombre optimal de bits à allouer à l'attribut Ai dépend donc de la fréquence d'utilisation de cet attribut pour

l'interrogation. Plus cette fréquence sera élevée, plus ce nombre sera important ; l'attribut sera ainsi privilégié aux dépens des autres.

6.2.2 Hachages multi-attributs dynamiques

L'utilisation de techniques de hachage dynamique est aussi possible avec des méthodes de placement multi-attribut. L'approche la plus connue est appelée **fichier grille,** ou *grid file* en anglais [Nievergelt84]. La méthode peut être vue comme une extension du hachage extensible. Une fonction de hachage est associée à chaque attribut de placement. L'adresse de hachage multidimensionnelle est constituée d'un nombre suffisant de bits choisi en prélevant des bits de chacune des fonctions de hachage de manière circulaire. Cette adresse sert de pointeur dans le répertoire des adresses de paquets. Tout se passe donc comme avec le hachage extensible, mais avec une fonction de hachage multi-attribut mixant les bits des différentes fonctions composantes.

Afin de clarifier la méthode, une représentation par une grille multidimensionnelle du fichier est intéressante. Pour simplifier, supposons qu'il existe seulement deux attributs de placement A1 et A2. Au départ, le fichier est composé d'un seul paquet qui délimite la grille (voir figure III.31.a). Quand le fichier grossit au fur et à mesure des insertions, le paquet sature. Il est alors éclaté en deux paquets B0 et B1 selon la dimension A1 (voir figure III.31.b). Pour ce faire, le premier bit de la fonction de hachage appliquée à A1 (h1(A1)) est utilisé. Quand l'un des paquets, disons B0, est plein, il est à son tour éclaté en deux paquets B00 et B01, cette fois selon l'autre dimension. Le premier bit de la fonction de hachage appliquée à A2 (h2(A2)) est utilisé. Si l'un des paquets B00 ou B01 devient plein, il sera à son tour éclaté, mais cette fois selon la dimension A1. Le processus d'éclatement continue ainsi alternativement selon l'une ou l'autre des dimensions.

<voir Maîtriser les BD>

Figure III.31 — Éclatement de paquets dans un fichier grille

Pour retrouver un article dans un paquet à partir de valeurs d'attributs, il faut appliquer les fonctions de hachage et retrouver l'adresse du ou des paquets correspondants dans le répertoire des paquets. Pour cela, la méthode propose un répertoire organisé comme un tableau multidimensionnel. Chaque entrée dans le répertoire correspond à une région du fichier grille. Le répertoire est stocké continûment sur des pages disques, mais est logiquement organisé comme un tableau multidimensionnel. Par exemple, avec un placement bidimensionnel, l'adresse d'un paquet pour la région (i,j) sera déterminé par une transformation d'un tableau bidimensionnel à un répertoire linéaire : l'entrée 2**N*i + j sera accédée, N étant le nombre maximal d'éclatement dans la dimension A1. Ainsi, lors d'un éclatement à un niveau de profondeur supplémentaire d'une dimension, il faut doubler le répertoire selon cette dimension. Cela peut s'effectuer par recopie ou chaînage. En résumé, on aboutit à une gestion de répertoire complexe avec un répertoire qui grossit exponentiellement en fonction de la taille des données.

Pour améliorer la gestion du répertoire et obtenir une croissance linéaire avec la taille des données, plusieurs approches on été proposées. Une des plus efficaces est utilisée par les

arbres de prédicats [Gardarin83]. Avec cette méthode, l'ensemble des fonctions de hachage est ordonné selon les fréquences décroissantes d'accès (les attributs les plus souvent documentés lors des interrogations d'abord). Pour simplifier, supposons que l'ordre soit de A0 à An. Un arbre de placement permet d'illustrer les éclatement de paquets (voir figure III.32). Au premier niveau, les paquets éclatent progressivement selon les bits de la fonction de hachage h(A1). Quand tous les bits sont exploités, on utilise la fonction h2(A2), etc. Ainsi, chaque paquet est une feuille de l'arbre de placement caractérisée par une chaîne de bits plus ou moins longue correspondant aux éclatements successifs. Cette chaîne de bits est appelée signature du paquet.

<voir Maîtriser les BD>

Figure III.32 — Arbre de placement et répertoire associé

Le répertoire contient des doublets <signature de paquet, adresse de paquet>. Le répertoire est organisé lui-même comme un fichier placé par un arbre de prédicats correspondant aux bits de la signature (chaque bit est considéré comme un attribut haché par l'identité). Sa taille est donc linéaire en fonction du nombre de paquets du fichier de données. Pour rechercher un article à partir d'attributs connus, un profil de signature est élaboré. Les bits correspondant aux attributs connus sont calculés et les autres mis à la valeur inconnue. Ce profil est utilisé pour accéder au répertoire par hachage. Il permet de déterminer les paquets à balayer. Des évaluations et des expérimentations de la méthode ont démontré son efficacité en nombre d'entrées-sorties [Gardarin83]. D'autres méthodes similaires ont été proposées afin de réduire la taille du répertoire et de profiter d'un accès sélectif [Freeston87].

6.3 Index bitmap

L'utilisation d'index sous forme d'arbres B sur un champ ayant peu de valeur conduit à des difficultés. En effet, pour de gros fichiers les listes d'adresses relatives d'articles deviennent longues et lourdes à manipuler. Il est bien connu qu'un index sur le sexe d'un fichier de personnes est inutile : il alourdit fortement les mises à jour et n'aide pas en interrogation, car il conduit à accéder directement à la moitié des articles, ce qui est pire qu'un balayage. Les **index bitmap** ont été introduits dans le SGBD Model 204 [O'Neil87] pour faciliter l'indexation sur des attributs à nombre de valeurs limités. Ils ont été plus tard généralisés [O'Neil97, Chan98].

> **Notion : Index bitmap (*Bitmap index*)**
>
> Index sur une donnée A constitué par une matrice de bits indiquant par un bit à 1 en position (i, j) la présence de la j^e valeur possible de l'attribut indexé A pour l'article i du fichier, et son absence sinon.

Un index bitmap suppose l'existence d'une fonction permettant d'associer les N valeurs possibles de l'attribut indexé A de 0 à N-1. Cette fonction peut être l'ordre alphanumérique des valeurs. C'est alors une bijection. La figure III.33 donne un exemple d'index bitmap pour un fichier de sportifs sur l'attribut `sport`.

Données

Personne	Sport
Perrec	Athlétisme
Cantona	Foot
Virenque	Vélo
Leconte	Tennis
Barthez	Foot
Jalabert	Vélo
Pioline	Tennis

Index bitmap

Athlétisme	Foot	Tennis	Vélo
1	0	0	0
0	1	0	0
0	0	0	1
0	0	1	0
0	1	0	0
0	0	0	1
0	0	1	0

Figure III.33 — Exemple d'index bitmap simple

Les index bitmap sont particulièrement adaptés à la recherche. Par exemple, la recherche des sportifs pratiquant le vélo s'effectue par lecture de l'index, extraction de la colonne Vélo et accès à tous les articles correspondant à un bit à 1. On trouvera ainsi Virenque et Jalabert. L'accès à l'article s'effectue à partir du rang du bit, 3 pour Virenque, 6 pour Jalabert. Pour faciliter cet accès, le fichier doit être organisé en pages avec un nombre fixe p d'articles par page. Si j est le rang du bit trouvé, la page j/p doit être lue et l'article correspondant au j^e cherché dans cette page. Fixer le nombre d'article par page peut être difficile en cas d'articles de longueurs variables. Des pages de débordement peuvent alors être prévues.

Un index bitmap peut aussi permettre d'indexer des attributs possédant des valeurs continues en utilisant une fonction non injective, c'est-à-dire en faisant correspondre à une colonne de la bitmap plusieurs valeurs d'attributs. Une technique classique consiste à associer à chaque colonne une plage de valeurs de l'attribut indexé. Par exemple, la figure III.34 illustre un index bitmap de l'attribut Coût utilisant des plages de valeurs de 0 à 100. Lors d'une recherche sur valeur, on choisira la bonne plage, puis on accédera aux articles correspondant aux bits à 1, et on testera plus exactement le critère sur les données de l'article. Par exemple, la recherche d'un produit de coût 150 conduira à accéder à la 2^e colonne, puis aux articles 1, 3, et 4. Seul l'article 3 sera finalement retenu.

Enfin, un index bitmap peut être utilisé pour indexer des attributs multivalués, tel l'attribut `Produits` figure III.34. Pour chaque article, les bits de chacune des colonnes correspondant aux valeurs de l'attributs sont positionnés à 1.

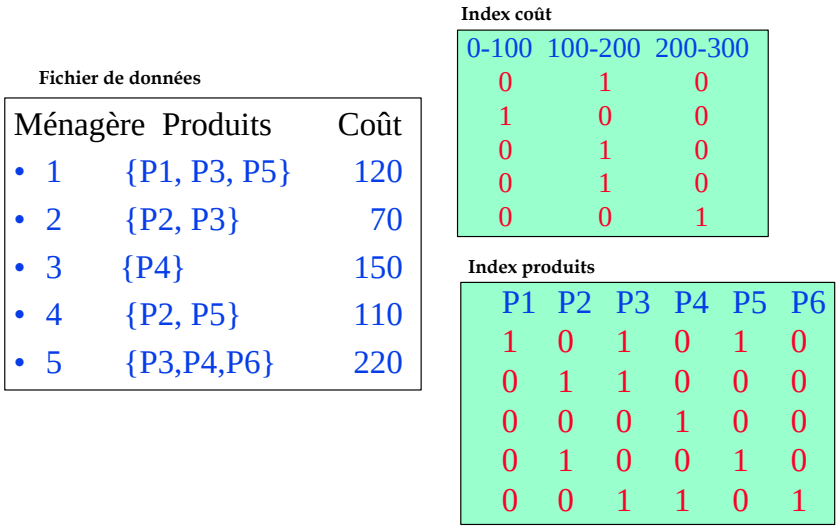

Figure III.34 — Exemple d'index bitmap plus complexes

Les index bitmap sont particulièrement utiles pour les accès sur des attributs multiples ou sur des valeurs multiples d'un même attribut. Il suffit pour cela d'effectuer des unions ou intersections de vecteurs de bits. Par exemple, la sélection des ménagères ayant acheté les produits P1 ou P2 avec un coût total supérieur à 100 s'effectuera par union des colonnes 1 et 2 de la bitmap index produits, puis intersection avec la colonne 3 unit à la colonne 2 de la bitmap index coût. On obtient alors un vecteur de bits qui désigne les ménagères 1 et 4. Les index bitmap sont donc très utile pour les accès multi-attributs. Ils sont aussi utilisés pour calculer des agrégats, voire extraire des règles des bases de données, comme nous le verrons plus tard. Les limites de cette technique développée récemment sont encore mal cernées.

7. CONCLUSION

Nous venons de passer en revue les fonctions essentielles et les techniques de base des gestionnaires de fichiers. D'autres études peuvent être trouvées, par exemple dans [Korth91] et [Widerhold83]. Il faut savoir qu'un gestionnaire de fichiers est de plus en plus souvent la base d'un système de gestion de bases de données. Pour ce faire, des niveaux de logiciels supérieurs sont généralement implantés pour assurer l'optimisation des recherches, la description centralisée des données des articles de fichiers, des interfaces de gestion de données variées avec les programmes d'application, etc.

La nécessité de pouvoir supporter un Système de Gestion de Bases de Données (SGBD) tend aujourd'hui à rendre le gestionnaire de fichiers de plus en plus élaboré. Il est par exemple possible de trouver des systèmes gérant plusieurs index secondaires pour un même fichier placé selon un hachage extensible éventuellement multi-attribut. En effet, pour permettre la consultation des données selon des critères variés, les SGBD nécessitent généralement plusieurs chemins d'accès à un même article. Nous reviendrons sur les problèmes de méthodes d'accès et d'organisation de fichiers pour les systèmes de gestion de bases de

données dans le chapitre spécifique aux modèles d'accès, puis plus tard pour les données multimédias.

8. BIBLIOGRAPHIE

[Bayer72] Bayer R., McCreight C., « Organization and Maintenance of Large Ordered Index », *Acta Informatica*, Vol. 1, N° 3, 1972, pp. 173-189.

Un des premiers articles introduisant l'organisation des index par arbres B et démontrant les avantages de cette organisation pour de gros fichiers.

[Chan98] Chan C-Y., Ioannidis Y.E., « Bitmap index Design and evaluation », *ACM SIGMOD Intl. Conf.*, SIGMOD Record V°27, N°2, Seattle, USA, 1998.

Cet article étudie la conception d'index bitmap pour traiter des requêtes complexes sur de grandes bases de données. En particulier, les techniques de mapping des valeurs d'attributs sur les indices de colonnes sont prises en compte et différents critères d'optimalité des choix sont étudiés.

[Comer79] Comer D., « The Ubiquitous B-Tree », *Computing Surveys*, Vol. 11, N° 2, juin 1979.

Une revue très complète des organisations basées sur les arbres B. Différentes variantes, dont les arbres B+, sont analysées.

[Daley65] Daley R.C., Neuman P.G., « A General Pupose File System for Secondary Storage », *Fall Joint Computer Conference*, 1965, pp. 213-229.

Une présentation de la première version du système de fichiers de MULTICS. Ce système introduit pour la première fois une organisation hiérarchique des catalogues de fichiers. L'intérêt de noms hiérarchiques et basés pour désigner un fichier est démontré. Le partage des fichiers par liens est introduit.

[Fagin79] Fagin R., Nivergelt J., Pippengar N., Strong H.R., « Extendible Hahing — A Fast Access Method for Dynamic Files », *ACM TODS*, Vol. 4, N° 3, septembre 1979, pp. 315-344.

L'article de base sur le hachage extensible. Il propose d'utiliser un répertoire d'adresses de paquets adressé par exploitation progressive des bits du résultat de la fonction de hachage. Les algorithmes de recherche et de mise à jour sont détaillés. Une évaluation démontre les avantages de la méthode aussi bien en temps d'accès qu'en taux de remplissage.

[Freeston87] Freeston M., « The BANG file — A New Kind of Grid File », *ACM SIGMOD*, mai 1987, San Fransisco, ACM Ed., pp. 260-269.

Cet article présente une variante du fichier grille, avec laquelle le répertoire reste linéaire en fonction de la taille des données. La technique est voisine de celle des signatures développées dans les arbres de prédicats, mais les attributs ne sont pas ordonnés et le répertoire des signatures dispose d'une organisation particulière. Le BANG file a été implémenté à l'ECRC — le centre de recherche commun BULL, ICL, SIEMENS à Munich — et a servi de base au système de bases de données déductives EKS.

[Gardarin83] Gardarin G., Valduriez P., Viémont Y., « Predicate Tree — An Integrated Approach to Multi-Dimensional Searching », *Rapport de recherche INRIA*, N° 203, avril 1983.

Cet article présente la méthode d'accès multi-attributs des arbres de prédicats. Cette méthode part d'une spécification d'une suite de collections de prédicats disjoints, chaque collection portant sur un attribut. La suite de collection permet d'affecter une signature à chaque tuple constituée par la concaténation des numéros de prédicats satisfaits. La signature est exploitée progressivement bit à bit pour placer les données sur disques. L' idée clé est de permettre un accès multicritères selon une hiérarchie de prédicats. Cette méthode a été mise en œuvre dans le SGBD SABRINA.

[IBM78] IBM Corporation, « Introduction to IBM Direct Access Storage Devices and Organization Methods », Student text, *Manual Form GC20-1649-10*.

Une description des supports de stockage et des méthodes d'accès supportées par IBM en 1978. Ce manuel contient en particulier une description très didactique de la méthode ISAM et une présentation de la méthode VSAM.

[Knott71] Knott G.D., « Expandable Open Addressing Hash Table Storage and Retrieval », *ACM SIGFIDET Workshop on Data Description, Access and Control*, 1971, ACM Ed., pp. 186-206.

Le premier article proposant un hachage dynamique, par extension progressive de la fonction de hachage en puissances de 2 successives. L'article s'intéresse seulement au cas de tables en mémoires.

[Knuth73] Knuth D.E., *The Art of Computer Programming*, Addisson-Wesley, 1973.

Le livre bien connu de Knuth. Une partie importante est consacrée aux structures de données, plus particulièrement à l'analyse de fonctions de hachage.

[Korth91] Korth H., Silberschatz A., *Database System Concepts*, Mc Graw-Hill Ed., 2e édition, 1991.

Un des livres les plus complets sur les bases de données. Deux chapitres sont plus particulièrement consacrés aux organisations de fichiers et aux techniques d'indexation.

[Larson78] Larson P., « Dynamic Hashing », *Journal BIT, N°18*, 1978, pp. 184-201.

Un des premiers schémas de hachage dynamique proposé, avec éclatement de paquets quand un taux de remplissage est atteint.

[Larson80] Larson P., « Linear Hashing with Partial Expansions », *6th Very Large Data Bases*, Montreal, octobre 1980, pp. 224-232.

Une variante du hachage linéaire ou les avancées du pointeur d'éclatement sont contrôlées par le taux de remplissage du fichier.

[Larson82] Larson P., « Performance Analysis of Linear Hashing with Partial Expansions », *ACM TODS*, Vol. 7, N° 4, décembre 1982, pp. 566-587.

Cet article résume la méthode présentée dans [Larson80] et démontre ses bonnes performances par une étude analytique.

[Lister84] Lister A.M., *Principes fondamentaux des systèmes d'exploitation*, Editions Eyrolles, 4e édition, 1984.

Cet excellent livre présente les couches successives constituant un système d'exploitation. Un chapitre est plus particulièrement consacré aux techniques d'allocation mémoire.

[Litwin78] Litwin W., « Virtual Hashing: A Dynamically Changing Hashing », *4th Very Lare Data Bases*, Berlin, septembre 1978, pp. 517-523.

Sans doute le premier schéma de hachage dynamique proposé pour des fichiers. Celui-ci est basé sur une table de bits pour marquer les paquets éclatés et sur un algorithme récursif de parcours de cette table pour retrouver la bonne fonction de hachage.

[Litwin80] Litwin W., « Linear Hashing — A New Tool for File and Table Addressing », *6th Very Large Data Bases*, Montreal, octobre 1980, pp. 224-232.

L'article introduisant le hachage linéaire. L'idée est de remplacer la table de bits complexe du hachage virtuel par un simple pointeur circulant sur les paquets du fichier. N bits de la fonction de hachage sont utilisés avant le pointeur, N+1 après. A chaque débordement, le paquet pointé par le pointeur est éclaté.

[Lomet83] Lomet D., « A High Performance, Universal Key Associative Access Method », *ACM SIGMOD Intl. Conf.*, San José, 1983.

Une méthode d'accès intermédiaire entre hachage et indexation, avec de très complètes évaluations de performances.

[Lomet89] Lomet D., Salzberg B., « Access Methods for Multiversion Data », *ACM SIGMOD Intl. Conf.*, Portland, 1989.

*Cet article discute les techniques d'archivage multiversion sur disques optiques. De tels disques sont inscriptibles une seule fois. Ils ne peuvent être corrigés, mais peuvent bien sûr être lus plusieurs fois (*Write Once Read Many times — WORM*). Ils nécessitent des organisations spécifiques étudiées dans cet article.*

[Nievergelt84] Nievergelt J., Hinterberger H., Sevcik K., « The Grid File: An Adaptable Symetric Multi-Key File Structure », *ACM TODS*, Vol. 9, N° 1, mars 1983.

*Cet article introduit le fichier grille (*Grid File*). L'idée est simplement d'étendre le hachage extensible en composant une fonction de hachage multi-attribut, à partir de différentes sous-fonctions portant sur un seul attribut. Une prise en compte successive des bits de chaque fonction garantie la symétrie. Différentes organisations de répertoires d'adresses sont possibles. Un tableau dynamique à n dimensions est proposé.*

[O'Neil87] O'Neil P. « Model 204 – Architecture and Performance », *Springer Verlag, LCNS* N°359, Proc. of 2nd Intl. Workshop on High Performance Transactions Systems, pp. 40-59, 1987.

Cet article décrit l'architecture du SGBD Model 204 de Computer Corporation of America. Il introduit pour la première fois les index bitmap.

[O'Neil97] O'Neil P., Quass D., « Improved Query Performance with Variant index », *ACM SIGMOD* Intl. Conf., SIGMOD Record V°26,N°2, Tucson, Arizona, USA, 1997.

Cet article présente les index bitmap comme une alternative aux index classiques à listes d'adresses. Il analyse les performances comparées de ces types d'index pour différents algorithmes de recherche.

[Samet89] Samet H., *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1989.

Ce livre présente des méthodes d'accès et organisations fondées sur les quadtrees *pour stocker et retrouver des données spatiales, telles que des points, des lignes, des régions, des surfaces et des volumes. Les* quadtrees *sont des structures de données hiérarchiques basées sur un découpage récursif de l'espace. Par exemple, une image plane en noir et blanc peut être découpée en deux rectangles, chaque rectangle étant à son tour découpé en deux jusqu'à obtention de rectangles d'une seule couleur. Les découpages successifs peuvent être représentés par un arbre dont les feuilles sont à 0 si le rectangle correspondant est blanc, à 1 s'il est noir. Une telle structure est un* quadtree. *Elle permet de constituer un index ou un arbre de placement utilisable pour une recherche efficace de motifs. Le livre de Samet étudie toutes les variantes des* quadtrees.

[Scholl81] Scholl M., « New File Organization Based on Dynamic Hashing », *ACM TODS*, Vol. 6, N° 1, mars 1981, pp. 194-211.

Une étude des performances des méthodes par hachage dynamique.

[Widerhold83] Widerhold G., *Database Design*, Mc Graw-Hill Ed., New York, 1983.

Ce livre, qui fut l'un des premiers sur les bases de données, consacre une partie importante aux disques magnétiques, aux fichiers et aux méthodes d'accès.