

CNAM Paris

Cours de bases de données

Aspects système

Philippe Rigaux

Philippe.Rigaux@cnam.fr

<http://deptinfo.cnam.fr/rigaux>

Table des matières

1	Techniques de stockage	7
1.1	Stockage de données	8
1.1.1	Supports	8
1.1.2	Fonctionnement d'un disque	9
1.1.3	Optimisations	11
1.1.4	Technologie RAID	15
1.2	Fichiers	17
1.2.1	Enregistrements	17
1.2.2	Blocs	19
1.2.3	Organisation d'un fichier	22
1.3	Oracle	25
1.3.1	Fichiers et blocs	26
1.3.2	Les <i>tablespaces</i>	28
1.3.3	Création des tables	31
1.4	Exercices	32
2	Indexation	35
2.1	Principes des index	36
2.2	Indexation de fichiers	37
2.2.1	Index non-dense	38
2.2.2	Index dense	39
2.2.3	Index multi-niveaux	41
2.3	L'arbre-B	41
2.3.1	Présentation intuitive	42
2.3.2	Recherches avec un arbre-B+	43
2.4	Hachage	46
2.4.1	Principes de base	46
2.4.2	Hachage extensible	49
2.5	Les index <i>bitmap</i>	51
2.6	Indexation de données géométriques	52
2.7	Indexation dans Oracle	61
2.7.1	Arbres B+	62
2.7.2	Arbres B	62
2.7.3	Indexation de documents	63
2.7.4	Tables de hachage	63
2.7.5	Index <i>bitmap</i>	64
2.8	Exercices	64
3	Organisation Physique	65

4	Évaluation de requêtes	71
4.1	Introduction à l'optimisation des performances	72
4.1.1	Opérations exprimées par SQL	72
4.1.2	Traitement d'une requête	72
4.1.3	Mesure de l'efficacité des opérations	73
4.1.4	Organisation du chapitre	74
4.1.5	Modèle d'exécution	75
4.2	Algorithmes de base	77
4.2.1	Recherche dans un fichier (sélection)	77
4.2.2	Quand doit-on utiliser un index ?	79
4.2.3	Le tri externe	82
4.3	Algorithmes de jointure	85
4.3.1	Jointure par boucles imbriquées	86
4.3.2	Jointure par tri-fusion	88
4.3.3	Jointure par hachage	90
4.3.4	Jointure avec un index	92
4.3.5	Jointure avec deux index	93
4.4	Compilation d'une requête et optimisation	94
4.4.1	Décomposition en bloc	94
4.4.2	Traduction et réécriture	96
4.4.3	Plans d'exécution	98
4.4.4	Modèles de coût	102
4.5	Oracle, optimisation et évaluation des requêtes	103
4.5.1	L'optimiseur d'ORACLE	104
4.5.2	Plans d'exécution ORACLE	105
4.6	Exercices	110
4.6.1	Opérateurs d'accès aux données	110
4.6.2	Plans d'exécution ORACLE	113
4.6.3	Utilisation de EXPLAIN et de TKPROF	117
4.6.4	Exercices d'application	119

Table des matières

Chapitre 1

Techniques de stockage

Sommaire

1.1	Stockage de données	8
1.1.1	Supports	8
1.1.2	Fonctionnement d'un disque	9
1.1.3	Optimisations	11
1.1.4	Technologie RAID	15
1.2	Fichiers	17
1.2.1	Enregistrements	17
1.2.2	Blocs	19
1.2.3	Organisation d'un fichier	22
1.3	Oracle	25
1.3.1	Fichiers et blocs	26
1.3.2	Les <i>tablespaces</i>	28
1.3.3	Création des tables	31
1.4	Exercices	32

Une base de données est constituée, matériellement, d'un ou plusieurs *fichiers* volumineux stockés sur un support non volatile. Le support le plus couramment employé est le disque magnétique (« disque dur») qui présente un bon compromis en termes de capacité de stockage, de prix et de performance. Il y a deux raisons principales à l'utilisation de fichiers. Tout d'abord il est courant d'avoir affaire à des bases de données dont la taille dépasse de loin celle de la mémoire principale. Ensuite – et c'est la justification principale du recours aux fichiers, même pour des bases de petite taille – une base de données doit survivre à l'arrêt de l'ordinateur qui l'héberge, que cet arrêt soit normal ou dû à un incident matériel.

L'accès à des données stockées sur un périphérique, par contraste avec les applications qui manipulent des données en mémoire centrale, est une des caractéristiques essentielles d'un SGBD. Elle implique notamment des problèmes potentiels de performance puisque le temps de lecture d'une information sur un disque est considérablement plus élevé qu'un accès en mémoire principale. L'organisation des données sur un disque, les structures d'indexation mises en œuvre et les algorithmes de recherche utilisés constituent donc des aspects très importants des SGBD. Un bon système se doit d'utiliser au mieux les techniques disponibles afin de minimiser les temps d'accès. Il doit aussi offrir à l'administrateur des outils de paramétrage et de contrôle qui vont lui permettre d'exploiter au mieux les ressources matérielles et logicielles d'un environnement donné.

Dans ce chapitre nous décrivons les techniques de stockage de données et leur transfert entre les différents niveaux de mémoire d'un ordinateur. Dans une première partie nous décrivons les aspects matériels liés au stockage des données par un ordinateur. Nous détaillons successivement les différents types de mémoire utilisées, en insistant particulièrement sur le fonctionnement des disques magnétiques. Nous abordons ensuite les mécanismes de transfert d'un niveau de mémoire à un autre, et leur optimisation.

La deuxième partie de ce chapitre est consacrée à l'organisation des données sur disque. Nous y abordons les notions d'*enregistrement*, de *bloc* et de *fichier*, ainsi que leur représentation physique.

1.1 Stockage de données

Un système informatique offre plusieurs mécanismes de stockage de l'information, ou *mémoires*. Ces mémoires se différencient par leur prix, leur rapidité, le mode d'accès aux données (séquentiel ou par adresse) et enfin leur durabilité. Les mémoires *volatiles* perdent leur contenu quand le système est interrompu, soit par un arrêt volontaire, soit à cause d'une panne. Les mémoires *non volatiles*, comme les disques, les CD ou les bandes magnétiques, préservent leur contenu même en l'absence d'alimentation électrique.

1.1.1 Supports

D'une manière générale, plus une mémoire est rapide, plus elle est chère et – conséquence directe – plus sa capacité est réduite. Les différentes mémoires utilisées par un ordinateur constituent donc une hiérarchie (figure 1.1), allant de la mémoire la plus petite mais la plus efficace à la mémoire la plus volumineuse mais la plus lente.

1. la *mémoire cache* est utilisée par le processeur pour stocker ses données et ses instructions ;
2. la *mémoire vive*, ou *mémoire principale* stocke les données et les processus constituant l'espace de travail de la machine ; toute donnée ou tout programme doit d'abord être chargé en mémoire principale avant de pouvoir être traité par un processeur ;
3. les *disques magnétiques* constituent le principal périphérique de type mémoire ; ils offrent une grande capacité de stockage tout en gardant des accès en lecture et en écriture relativement efficaces ;
4. enfin les CD ou les bandes magnétiques sont des supports très économiques mais leur lenteur les destine plutôt aux sauvegardes.

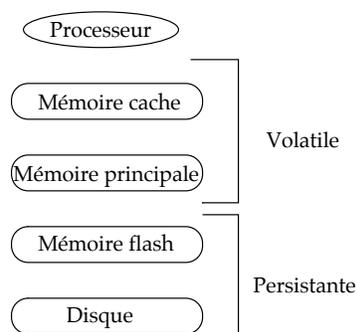


FIGURE 1.1 – Hiérarchie des mémoires

La mémoire vive et les disques sont les principaux niveaux à considérer pour des applications de bases de données. Une base de données est à peu près toujours stockée sur disque, pour les raisons de taille et de persistance déjà évoquées, mais les données doivent impérativement être placées en mémoire vive pour être traitées. Dans l'hypothèse (réaliste) où seule une petite fraction de la base peut résider en mémoire centrale, un SGBD doit donc en permanence effectuer des transferts entre mémoire principale et mémoire secondaire pour satisfaire les requêtes des utilisateurs. Le coût de ces transferts intervient de manière prépondérante dans les performances du système.

La technologie évoluant rapidement, il est délicat de donner des valeurs précises pour la taille et les temps d'accès des différentes mémoires. Le tableau 1.1 propose quelques ordres de grandeur. Un ordinateur

Mémoire	Taille (en Mo)	Temps d'accès (secondes)
cache	Quelques Mégas	$\approx 10^{-8}$ (10 nanosec.)
principale	$O(10^{12})$ (Gigas)	$\approx 10^{-8} - 10^{-7}$ (10-100 nanosec.)
Mémoire "flash"	$O(10^{12})$ (Gigas)	$\approx 10^{-4}$ (0,1 millisecc.)
disque magnétique	$O(10^{15})$ (Téras)	$\approx 10^{-2}$ (10 millisecc.)

TABLE 1.1 – Caractéristiques des différentes mémoires

est équipé de quelques dizaines de gigaoctets de mémoire vive et les disques stockent quelques téraoctets (typiquement 1 à 2 To). En contrepartie, le temps d'accès à une information en mémoire vive est de l'ordre de 10 nanosecondes (10^{-8}) tandis que le temps d'accès sur un disque est de l'ordre de 10 millisecondes (10^{-2}), ce qui représente un ratio approximatif de 1 000 000 entre les performances respectives de ces deux supports ! Il est clair dans ces conditions que le système doit tout faire pour limiter les accès au disque.

1.1.2 Fonctionnement d'un disque

Un disque est une surface circulaire magnétisée capable d'enregistrer des informations numériques. La surface magnétisée peut être située d'un seul côté (« simple face ») ou des deux côtés (« double face ») du disque.

Les disques sont divisés en *secteurs*, un secteur constituant la plus petite surface d'adressage. En d'autres termes, on sait lire ou écrire des zones débutant sur un secteur et couvrant un nombre entier de secteurs. La taille d'un secteur est le plus souvent de 512 octets.

Dispositif

La petite information stockée sur un disque est un bit qui peut valoir 0 ou 1. Les bits sont groupés par 8 pour former des octets, et une suite d'octets forme un cercle ou *piste* sur la surface du disque.

Un disque est entraîné dans un mouvement de rotation régulier par un axe. Une *tête de lecture* (deux si le disque est double-face) vient se positionner sur une des pistes du disque et y lit ou écrit les données. Le nombre minimal d'octets lus par une tête de lecture est physiquement défini par la taille d'un secteur (en général 512 octets). Cela étant le système d'exploitation peut choisir, au moment de l'*initialisation* du disque, de fixer une unité d'entrée/sortie supérieure à la taille d'un secteur, et multiple de cette dernière. On obtient des *blocs*, dont la taille est typiquement 512 octets (un secteur), 1024 octets (deux secteurs) ou 4096 octets (huit secteurs).

Chaque piste est donc divisée en *blocs* (ou *pages*) qui constituent l'unité d'échange entre le disque et la mémoire principale.

Toute lecture ou toute écriture sur les disques s'effectue par blocs. Même si la lecture ne concerne qu'une donnée occupant 4 octets, tout le bloc contenant ces 4 octets sera transmis en mémoire centrale. Cette caractéristique est fondamentale pour l'organisation des données sur le disque. Un des objectifs du SGBD est de faire en sorte que quand il est nécessaire de lire un bloc de 4096 octets pour accéder à un entier de 4 octets, les 4092 octets constituant le reste du bloc ont de grandes chances d'être utiles à court terme et se trouveront donc déjà chargée en mémoire centrale quand le système en aura besoin.

La tête de lecture n'est pas entraînée dans le mouvement de rotation. Elle se déplace dans un plan fixe qui lui permet de se rapprocher ou de s'éloigner de l'axe de rotation des disques, et d'accéder à une des pistes. Pour limiter le coût de l'ensemble de ce dispositif et augmenter la capacité de stockage, les disques sont empilés et partagent le même axe de rotation (voir figure 1.2). Il y a autant de têtes de lectures que de disques (deux fois plus si les disques sont à double face) et toutes les têtes sont positionnées solidairement dans leur plan de déplacement. À tout moment, les pistes accessibles par les têtes sont donc les mêmes pour tous les disques de la pile, ce qui constitue une contrainte dont il faut savoir tenir compte quand on cherche à optimiser le placement des données.

L'ensemble des pistes accessibles à un moment donné constitue le *cylindre*. La notion de cylindre correspond donc à toutes les données disponibles sans avoir besoin de déplacer les têtes de lecture.

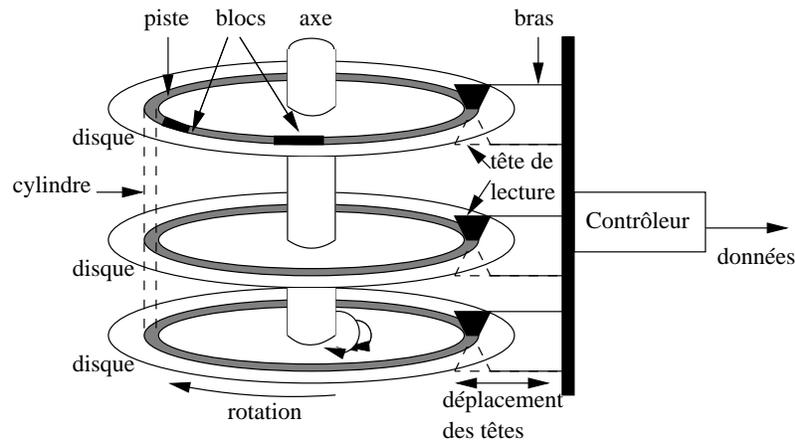


FIGURE 1.2 – Un disque magnétique

Enfin le dernier élément du dispositif est le *contrôleur* qui sert d'interface avec le système d'exploitation. Le contrôleur reçoit du système des demandes de lecture ou d'écriture, et les transforme en mouvements appropriés des têtes de lectures, comme expliqué ci-dessous.

Accès aux données

Un disque est une mémoire à accès direct. Contrairement à une bande magnétique par exemple, il est possible d'accéder à une information située n'importe où sur le disque sans avoir à parcourir séquentiellement tout le support. L'accès direct est fondé sur une adresse donnée à chaque bloc au moment de l'initialisation du disque par le système d'exploitation. Cette adresse est généralement composée des trois éléments suivants :

1. le numéro du disque dans la pile ou le numéro de la surface si les disques sont à double-face ;
2. le numéro de la piste ;
3. le numéro du bloc sur la piste.

La lecture d'un bloc, étant donné son adresse, se décompose en trois étapes :

- *positionnement de la tête de lecture* sur la piste contenant le bloc ;
- *rotation du disque* pour attendre que le bloc passe sous la tête de lecture (rappelons que les têtes sont fixe, c'est le disque qui tourne) ;
- *transfert du bloc*.

La durée d'une opération de lecture est donc la somme des temps consacrés à chacune des trois opérations, ces temps étant désignés respectivement par les termes *délai de positionnement*, *délai de latence* et *temps de transfert*. Le temps de transfert est négligeable pour un bloc, mais peu devenir important quand des milliers de blocs doivent être lus. Le mécanisme d'écriture est à peu près semblable à la lecture, mais peu prendre un peu plus de temps si le contrôleur vérifie que l'écriture s'est faite correctement.

Le tableau 1.2 donne les spécifications d'un disque en 2001, telles qu'on peut les trouver sur le site de n'importe quel constructeur (ici *Seagate*, www.seagate.com). Les chiffres donnent un ordre de grandeur pour les performances d'un disque, étant bien entendu que les disques destinés aux serveurs sont beaucoup plus performants que ceux destinés aux ordinateurs personnels. Le modèle donné en exemple dans le tableau 1.2 appartient au milieu de gamme.

Le disque comprend 17 783 438 secteurs de 512 octets chacun, la multiplication des deux chiffres donnant bien la capacité totale de 9,1 Go. Les secteurs étant répartis sur 3 disques double-face, il y a donc $17\,783\,438/6 = 2\,963\,906$ secteurs par surface.

Le nombre de secteurs par piste n'est pas constant, car les pistes situées près de l'axe sont bien entendu beaucoup plus petite que celles situées près du bord du disque. On ne peut, à partir des spécifications, que

calculer le nombre moyen de secteurs par piste, qui est égal à $2\,963\,906/9\,772 = 303$. On peut donc estimer qu'une piste stocke en moyenne $303 \times 512 = 155\,292$ octets. Ce chiffre donne le nombre d'octets qui peuvent être lus sans délai de latence ni délai de positionnement.

Caractéristique	Performance
Capacité	9,1 Go
Taux de transfert	80 Mo par seconde
Cache	1 Mo
Nbre de disques	3
Nbre de têtes	6
Nombre total secteurs (512 octets)	17 783 438
Nombre de cylindres	9 772
Vitesse de rotation	10 000 rpm (rotations par minute)
Délai de latence	En moyenne 3 ms
Temps de positionnement moyen	5.2 ms
Déplacement de piste à piste	0.6 ms

TABLE 1.2 – Spécifications du disque *Cheetah 18LP* (source www.seagate.com)

Ce qu'il faut surtout noter, c'est que les temps donnés pour le temps de latence et le délai de rotation ne sont que des moyennes. Dans le meilleur des cas, les têtes sont positionnées sur la bonne piste, et le bloc à lire est celui qui arrive sous la tête de lecture. Le bloc peut alors être lu directement, avec un délai réduit au temps de transfert.

Ce temps de transfert peut être considéré comme négligeable dans le cas d'un bloc unique, comme le montre le raisonnement qui suit, basé sur les performances du tableau 1.2. Le disque effectue 10000 rotations par minute, ce qui correspond à 166,66 rotations par seconde, soit une rotation toutes les 0,006 secondes (6 ms). C'est le temps requis pour lire une piste entièrement. Cela donne également le temps moyen de latence de 3 ms.

Pour lire un bloc sur une piste, il faudrait tenir compte du nombre exact de secteurs, qui varie en fonction de la position exacte. En prenant comme valeur moyenne 303 secteurs par piste, et une taille de bloc égale à 4 096 soit huit secteurs, on obtient le temps de transfert moyen pour un bloc :

$$\frac{6ms \times 8}{303} = 0,16ms$$

Le temps de transfert ne devient significatif que quand on lit plusieurs blocs consécutivement. Notez quand même que les valeurs obtenues restent beaucoup plus élevées que les temps d'accès en mémoire principale qui s'évaluent en nanosecondes.

Dans une situation moyenne, la tête n'est pas sur la bonne piste, et une fois la tête positionnée (temps moyen 5.2 ms), il faut attendre une rotation partielle pour obtenir le bloc (temps moyen 3 ms). Le temps de lecture est alors en moyenne de 8.2 ms, si on ignore le temps de transfert.

1.1.3 Optimisations

Maintenant que nous avons une idée précise du fonctionnement d'un disque, il est assez facile de montrer que pour un même volume de données, le temps de lecture peut varier considérablement en fonction de facteurs tels que le placement sur le disque, l'ordre des commandes d'entrées/sorties ou la présence des données dans une mémoire *cache*.

Toutes les techniques permettant de réduire le temps passé à accéder au disque sont utilisées intensivement par les SGBD qui, répétons-le, voient leurs performances en grande partie conditionnés par ces accès. Nous étudions dans cette section les principales techniques d'optimisation mises en œuvre dans une architecture simple comprenant un seul disque et un seul processeur. Nous verrons dans la partie suivante consacrée à la technologie RAID, comment on peut tirer parti de l'utilisation de plusieurs disques.

Regroupement

Prenons un exemple simple pour se persuader de l'importance d'un bon regroupement des données sur le disque : le SGBD doit lire 5 chaînes de caractères de 1000 octets chacune. Pour une taille de bloc égale à 4096 octets, deux blocs peuvent suffire. La figure 1.3 montre deux organisations sur le disque. Dans la première chaque chaîne est placée dans un bloc différent, et les blocs sont répartis aléatoirement sur les pistes du disque. Dans la seconde organisation, les chaînes sont rassemblées dans deux blocs qui sont consécutifs sur une même piste du disque.

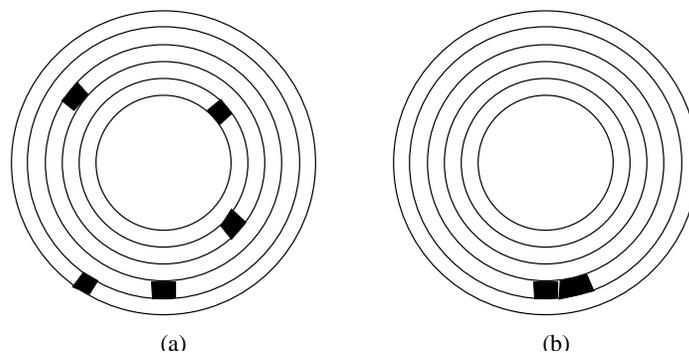


FIGURE 1.3 – Mauvaise et bonne organisation sur un disque

La lecture dans le premier cas implique 5 déplacements des têtes de lecture, et 5 délais de latence ce qui donne un temps de $5 \times (5.2 + 3) = 41$ ms. Dans le second cas, on aura un déplacement, et un délai de latence pour la lecture du premier bloc, mais le bloc suivant pourra être lu instantanément, pour un temps total de 8,2 ms.

Les performances obtenues sont dans un rapport de 1 à 5, le temps minimal s'obtenant en combinant deux optimisations : regroupement et contiguïté. Le regroupement consiste à placer dans le même bloc des données qui ont de grandes chances d'être lues au même moment. Les critères permettant de déterminer le regroupement des données constituent un des fondements des structures de données en mémoire secondaire qui seront étudiées par la suite. Le placement dans des blocs contigus est une extension directe du principe de regroupement. Il permet d'effectuer des *lectures séquentielles* qui, comme le montre l'exemple ci-dessus, sont beaucoup plus performantes que les lectures aléatoires car elles évitent des déplacements de têtes de lecture.

Plus généralement, le gain obtenu dans la lecture de deux données d_1 et d_2 est d'autant plus important que les données sont « proches », sur le disque, cette proximité étant définie comme suit, par ordre décroissant :

- la proximité maximale est obtenue quand d_1 et d_2 sont dans le même bloc : elles seront alors toujours lues ensemble ;
- le niveau de proximité suivant est obtenu quand les données sont placées dans deux blocs consécutifs ;
- quand les données sont dans deux blocs situés sur la même piste du même disque, elles peuvent être lues par la même tête de lecture, sans déplacement de cette dernière, et en une seule rotation du disque ;
- l'étape suivante est le placement des deux blocs dans un même cylindre, qui évite le déplacement des têtes de lecture ;
- enfin si les blocs sont dans deux cylindres distincts, la proximité est définie par la distance (en nombre de pistes) à parcourir.

Les SGBD essaient d'optimiser la proximité des données au moment de leur placement sur le disque. Une table par exemple devrait être stockée sur une même piste ou, dans le cas où elle occupe plus d'une piste, sur les pistes d'un même cylindre, afin de pouvoir effectuer efficacement un parcours séquentiel.

Pour que le SGBD puisse effectuer ces optimisations, il doit se voir confier, à la création de la base,

un espace important sur le disque dont il sera le seul à gérer l'organisation. Si le SGBD se contentait de demander au système d'exploitation de la place disque quand il en a besoin, le stockage physique obtenu risque d'être très fragmenté.

Séquencement

En théorie, si un fichier occupant n blocs est stocké contiguement sur une même piste, la lecture séquentielle de ce fichier sera – en ignorant le temps de transfert – approximativement n fois plus efficace que si tous les blocs sont répartis aléatoirement sur les pistes du disque.

Cet analyse doit cependant être relativisée car un système est souvent en situation de satisfaire simultanément plusieurs utilisateurs, et doit gérer leurs demandes concurrentes. Si un utilisateur A demande la lecture du fichier F_1 tandis que l'utilisateur B demande la lecture du fichier F_2 , le système alternera probablement les lectures des blocs des deux fichiers. Même s'ils sont tous les deux stockés séquentiellement, des déplacements de tête de lecture interviendront alors et minimiseront dans une certaine mesure cet avantage.

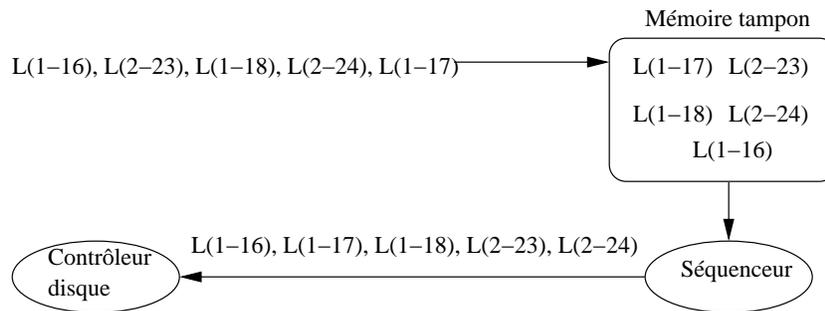


FIGURE 1.4 – Séquencement des entrées/sorties

Le système d'exploitation, ou le SGBD, peuvent réduire cet inconvénient en conservant temporairement les demandes d'entrées/sorties dans une zone tampon (*cache*) et en réorganisant (*séquencement*) l'ordre des accès. La figure 1.4 montre le fonctionnement d'un séquenceur. Un ensemble d'ordres de lectures est reçu, $L(1-16)$ désignant par exemple la demande de lecture du bloc 16 sur la piste 1. On peut supposer sur cet exemple que deux utilisateurs effectuent séparément des demandes d'entrée/sortie qui s'imbriquent quand elles sont transmises vers le contrôleur.

Pour éviter les accès aléatoires qui résultent de cette imbrication, les demandes d'accès sont stockées temporairement dans un *cache*. Le séquenceur les trie alors par piste, puis par bloc au sein de chaque piste, et transmet la liste ordonnée au contrôleur du disque. Dans notre exemple, on se place donc tout d'abord sur la piste 1, et on lit séquentiellement les blocs 16, 17 et 18. Puis on passe à la piste 2 et on lit les blocs 23 et 24. Nous laissons au lecteur, à titre d'exercice, le soin de déterminer le gain obtenu.

Une technique pour systématiser cette stratégie est celle dite « de l'ascenseur ». L'idée est que les têtes de lecture se déplacent régulièrement du bord de la surface du disque vers l'axe de rotation, puis reviennent de l'axe vers le bord. Le déplacement s'effectue piste par piste, et à chaque piste le séquenceur transmet au contrôleur les demandes d'entrées/sorties pour la piste courante.

Cet algorithme réduit au maximum de temps de déplacement des têtes puisque ce déplacement s'effectue systématiquement sur la piste adjacente. Il est particulièrement efficace pour des systèmes avec de très nombreux processus demandant chacun quelques blocs de données. En revanche il peut avoir des effets assez désagréables en présence de quelques processus gros consommateurs de données. Le processus qui demande des blocs sur la piste 1 alors que les têtes viennent juste de passer à la piste 2 devra attendre un temps respectable avant de voir sa requête satisfaite.

Mémoire tampon

La dernière optimisation, très largement utilisée dans tous les SGBD, est l'utilisation de mémoires tampon, ou *buffer*. Un buffer est un ensemble de blocs en mémoire principale qui sont des copies des blocs sur le disque. Quand le système demande à accéder à un bloc (*lecture logique*), une première inspection a lieu dans le buffer. Si le bloc s'y trouve déjà, une lecture a été évitée. Sinon on effectue la *lecture physique* sur le disque et on stocke la page dans le buffer.

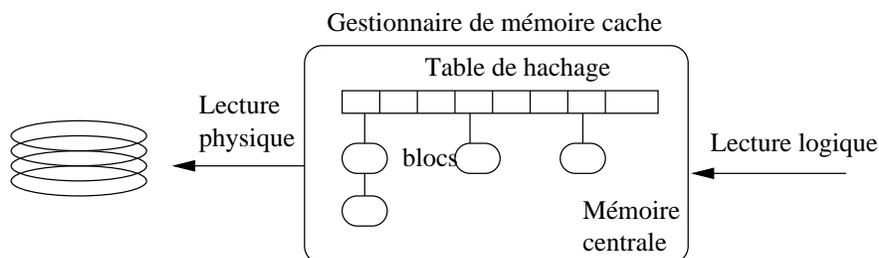


FIGURE 1.5 – Mémoire cache

L'idée est donc simplement de maintenir en mémoire principale une copie aussi large que possible du disque, même si une grande partie des blocs mis ainsi dans un buffer n'est pas directement utile. Le paramètre qui mesure l'efficacité d'une mémoire tampon est le *hit ratio*, défini comme suit :

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

Si toutes les lectures logiques (demande de bloc) aboutissent à une lecture physique (accès au disque), le *hit ratio* est 0 ; s'il n'y a aucune lecture physique, le *hit ratio* est de 1.

Noter que le *hit ratio* n'est pas simplement le rapport entre la taille de la mémoire cache et celle de la base. Ce serait si tous les blocs étaient lus avec une probabilité uniforme, mais le *hit ratio* représente justement la capacité du système à stocker dans le cache les pages les plus lues.

Plus la mémoire cache est importante, et plus il sera possible d'y conserver une partie significative de la base, avec un *hit ratio* élevé et des gains importants en terme de performance. Cela étant le *hit ratio* ne croît toujours pas linéairement avec l'augmentation de la taille de la mémoire cache. Si la disproportion entre la taille du cache et celle de la base est élevée, le *hit ratio* est conditionné par le pourcentage de accès à la base qui peuvent lire n'importe quelle page avec une probabilité uniforme. Prenons un exemple pour clarifier les choses.

Exemple 1. La base de données fait 2 GigaOctets, le buffer 30 MégaOctets. Dans 60 % des cas une lecture logique s'adresse à une partie limitée de la base, correspondant aux principales tables de l'application, dont la taille est, disons 200 Mo. Dans les 40 de cas suivants les accès se font avec une probabilité uniforme dans les 1,8 Go restant.

En augmentant la taille du buffer jusqu'à 333 Mo, on va améliorer régulièrement le *hit ratio* jusqu'à environ 0,6. En effet les 200 Mo correspondant à 60 % des lectures vont finir par se trouver placées en cache ($200 \text{ Mo} = 333 \times 0,6$), et n'en bougeront pratiquement plus. En revanche, les 40 % des autres accès accéderont à 1,8 Go avec seulement 133 Mo et le *hit ratio* restera très faible pour cette partie-là. □

Conclusion : si vous cherchez la meilleure taille pour un buffer sur une très grosse base, faites l'expérience d'augmenter régulièrement l'espace mémoire alloué jusqu'à ce que la courbe d'augmentation du *hit ratio* s'applatisse. En revanche sur une petite base où il est possible d'allouer un buffer de taille comparable à la base, on a la possibilité d'obtenir un *hit ratio* optimal de 1.

Quand la mémoire cache est pleine et qu'un nouveau bloc doit être lu sur le disque, un algorithme de remplacement doit être adopté pour retirer un des blocs de la mémoire et le replacer sur le disque (opérations de *flush*). L'algorithme le plus courant (utilisé dans MySQL) est dit *Least Recently Used* (LRU). Il consiste à choisir comme « victime » le bloc dont la dernière date de lecture logique est la plus ancienne.

Quand il reste de la place dans les buffers, on peut l'utiliser en effectuant des *lectures en avance* (*read ahead*, ou *prefetching*). Une application typique de ce principe est donnée par la lecture d'une table. Comme nous le verrons au moment de l'étude des algorithmes de jointure, il est fréquent d'avoir à lire une table séquentiellement, bloc à bloc. Il s'agit d'un cas où, même si à un moment donné on n'a besoin que d'un ou de quelques blocs, on sait que toute la table devra être parcourue. Il vaut mieux alors, au moment où on effectue une lecture sur une piste, charger en mémoire tous les blocs de la relation, y compris ceux qui ne serviront que dans quelques temps et peuvent être placés dans un buffer en attendant.

1.1.4 Technologie RAID

Le stockage sur disque est une des fonctionnalités les plus sensibles d'un SGBD, et ce aussi bien pour des questions de performances que pour des raisons de sécurité. Une défaillance en mémoire centrale n'a en effet qu'un impact limité : au pire les données modifiées mais non encore écrites sur disque seront perdues. Si un disque est endommagé en revanche, toutes les données sont perdues et il faut recourir à une sauvegarde. Cela représente un risque, une perte d'information (tout ce qui a été fait depuis la dernière sauvegarde) et une perte de temps due à l'indisponibilité du système pendant la récupération de la sauvegarde.

Le risque augmente statistiquement avec le nombre de disques utilisés. La durée de vie moyenne d'un disque (temps moyen avant une panne) étant de l'ordre d'une dizaine d'année, le risque de panne pour un disque parmi 100 peut être grossièrement estimé à $120/100=1,2$ mois. Si la défaillance d'un disque entraîne une perte de données, ce risque peut être considéré comme trop important.

La technologie RAID (pour *Redundant Array of Independent Disks*) a pour objectif principal de limiter les conséquences des pannes en répartissant les données sur un grand nombre de disques, et de manière à s'assurer que la défaillance de l'un des disques n'entraîne ni perte de données, ni même l'indisponibilité du système.

Il existe plusieurs niveaux RAID (de 0 à 6), chacun correspondant à une organisation particulière des données et donc à des caractéristiques différentes. Le niveau 0 est simplement celui du stockage sur un seul disque, avec les risques discutés précédemment. Nous présentons ci-dessous les caractéristiques des principaux niveaux.

Duplication (RAID 1)

Le RAID 1 applique une solution brutale : toutes les entrées/sorties s'effectuent en parallèle sur deux disques. Les écritures ne sont pas simultanées afin d'éviter qu'une panne électrique ne vienne interrompre les têtes de lecture au moment où elles écrivent le même bloc, qui serait alors perdu. L'écriture a donc d'abord lieu sur le disque principal, puis sur le second (dit « disque miroir »).

Le RAID 1 est coûteux puisqu'il nécessite deux fois plus d'espace que de données. Il permet certaines optimisations en lecture : par exemple la demande d'accès à un bloc peut être transmise au disque dont la tête de lecture est la plus proche de la piste contenant le bloc.

Les performances sont également améliorées en écriture car deux demandes de deux processus distincts peuvent être satisfaites en parallèle. En revanche il n'y a pas d'amélioration du taux de transfert puisque les données ne sont pas réparties sur les disques.

Répartition et parité (RAID 4)

Ce niveau combine deux techniques. La première consiste à traiter les n disques comme un seul très grand disque, et à répartir les données sur tous les disques. L'unité de répartition est le bloc. Si on a 4 disques et des données occupant 5 blocs, on écrit le premier bloc sur le premier disque, le second bloc sur le deuxième disque, et ainsi de suite. Le cinquième bloc est écrit sur le premier disque et le cycle recommence.

L'avantage de cette répartition est d'améliorer les performances en lecture. Si un seul bloc de données est demandé, une lecture sur un des disques suffira. Si en revanche les 2, 3 ou 4 premiers blocs de données sont demandés, il sera possible de combiner des lectures sur l'ensemble des disques. Le temps de réponse est alors celui d'une lecture d'un seul bloc. Plus généralement, quand de très larges volumes doivent être

lus, il est possible de répartir en parallèle la lecture sur les n disques, avec un temps de lecture divisé par n , et un débit multiplié par n .

L'autre aspect du RAID 4 est une gestion « intelligente » de la redondance en stockant non pas une duplication des données, mais un bit de parité. L'idée est la suivante : pour n disques de données, on va ajouter un *disque de contrôle* qui permettra de récupérer les données en cas de défaillance de l'un (un seul) des n disques. On suppose que les $n + 1$ disques ont la même taille et la même structure (taille de blocs, pistes, etc).

À chaque bit du disque de contrôle peuvent donc être associés les n bits des disques de données situés à la même position. Si il y a un nombre pair de 1 parmi ces n bit, le bit de parité vaudra 1, sinon il vaudra 0.

Exemple 2. Supposons qu'il y ait 3 disques de données, et que le contenu du premier octet de chaque disque soit le suivant :

D1 : 11110000
D2 : 10101010
D3 : 00110011

Alors il suffit de prendre chaque colonne et de compter le nombre p de 1 dans la colonne. La valeur du bit de parité est $p \bmod 2$. Pour la première colonne on a $p = 2$, et le bit de parité vaut 0. Voici le premier octet du disque de contrôle.

DC : 01101001

□

Si on considère les quatre disques dans l'exemple précédent, le nombre de bits à 1 pour chaque position est pair. Il y a deux 1 pour la première et la seconde position, 4 pour la troisième, etc. La reconstruction de l'un des n disques après une panne devient alors très facile puisqu'il suffit de rétablir la parité en se basant sur les informations des $n - 1$ autres disques et du disque de contrôle.

Supposons par exemple que le disque 2 tombe en panne. On dispose des informations suivantes :

D1 : 11110000
D3 : 00110011
DC : 01101001

On doit affecter des 0 et des 1 aux bits du disque 2 de manière à rétablir un nombre pair de 1 dans chaque colonne. Pour la première position, il faut mettre 1, pour la seconde 0, pour la troisième 1, etc. On reconstitue ainsi facilement la valeur initiale 10101010.

Les lectures s'effectuent de manière standard, sans tenir compte du disque de contrôle. Pour les écritures il faut mettre à jour le disque de contrôle pour tenir compte de la modification des valeurs des bits. Une solution peu élégante est de lire, pour toute écriture d'un bloc sur un disque, les valeurs des blocs correspondant sur les $n - 1$ autres disques, de recalculer la parité et de mettre à jour le disque de contrôle. Cette solution est tout à fait inefficace puisqu'elle nécessite $n + 1$ entrées/sorties.

Il est nettement préférable d'effectuer le calcul en tenant compte de la valeur du bloc *avant* la mise à jour. En calculant la parité des valeurs avant et après mise à jour, on obtient un 1 pour chaque bit dont la valeur a changé. Il suffit alors de reporter ce changement sur le disque de contrôle.

Exemple 3. Reprenons l'exemple précédent, avec trois disques D1, D2, D3, et le disque de contrôle DC.

D1 : 11110000
D2 : 10101010
D3 : 00110011
DC : 01101001

Supposons que D1 soit mis à jour et devienne 10011000. On doit calculer la parité des valeurs avant et après mise à jour :

avant : 11110000
après : 10011000

On obtient l'octet 01101000 qui indique que les positions 2, 3, et 5 ont été modifiées. Il suffit de reporter ces modifications sur le disque de contrôle en changeant les 0 en 1, et réciproquement, pour les positions 2, 3 et 5. On obtient finalement le résultat suivant.

D1 : 10011000
D2 : 10101010
D3 : 00110011
DC : 00000001

□

En résumé le RAID 4 offre un mécanisme de redondance très économique en espace, puisque un seul disque supplémentaire est nécessaire quel que soit le nombre de disques de données. En contrepartie il ne peut être utilisé dans le cas – improbable – où deux disques tombent simultanément en panne. Un autre inconvénient possible est la nécessité d'effectuer une E/S sur le disque de contrôle pour chaque écriture sur un disque de données, ce qui implique qu'il y a autant d'écritures sur ce disque que sur tous les autres réunis.

Répartition des données de parité (RAID 5)

Dans le RAID 4, le disque de contrôle a tendance à devenir le goulot d'étranglement du système puisque qu'il doit supporter n fois plus d'écritures que les autres. Le RAID 5 propose de remédier à ce problème en se basant sur une remarque simple : si c'est le disque de contrôle lui-même qui tombe en panne, il est possible de le reconstituer en fonction des autres disques. En d'autres termes, pour la reconstruction après une panne, la distinction disque de contrôle/disque de données n'est pas pertinente.

D'où l'idée de ne pas dédier un disque aux données de parité, mais de répartir les blocs de parité sur les $n + 1$ disques. La seule modification à apporter par rapport au RAID 4 est de savoir, quand on modifie un bloc sur un disque D_1 , quel est le disque D_2 qui contient les données de parité pour ce bloc. Il est possible par exemple de décider que pour le bloc i , c'est le disque $i \bmod n$ qui stocke le bloc de parité.

Défaillances simultanées (RAID 6)

Le dernier niveau de RAID prend en compte l'hypothèse d'une défaillance simultanée d'au moins deux disques. Dans un tel cas l'information sur la parité devient inutile pour reconstituer les disques : si la parité est 0, l'information perdue peut être soit 00, soit 11 ; si la parité est 1, l'information perdue peut être 01 ou 10.

Le RAID 6 s'appuie sur une codification plus puissante que la parité : les codes de Hamming ou les codes de Reed-Solomon. Nous ne présentons pas ces techniques ici : elles sont décrites par exemple dans [GUW00]. Ces codes permettent de reconstituer l'information même quand plusieurs disques subissent des défaillances, le prix à payer étant une taille plus importante que la simple parité, et donc l'utilisation de plus de disques de contrôles.

1.2 Fichiers

Il n'est jamais inutile de rappeler qu'une base de données n'est rien d'autre qu'un ensemble de données stockées sur un support persistant. La technique de très loin la plus répandue consiste à organiser le stockage des données sur un disque au moyen de *fichiers*. Leurs principes généraux sont décrits dans ce qui suit.

1.2.1 Enregistrements

Pour le système d'exploitation, un fichier est une suite d'octets répartis sur un ou plusieurs blocs. Les fichiers gérés par un SGBD sont un peu plus structurés. Ils sont constitués d'*enregistrements* (*records* en

Type	Taille en octets
SMALLINT	2
INTEGER	4
BIGINT	8
FLOAT	4
DOUBLE PRECISION	8
NUMERIC (M, D)	M, (D+2 si M < D)
DECIMAL (M, D)	M, (D+2 si M < D)
CHAR(M)	M
VARCHAR(M)	L+1 avec $L \leq M$
BIT VARYING	$< 2^8$
DATE	8
TIME	6
DATETIME	14

TABLE 1.3 – Types SQL et tailles (en octets)

anglais) qui représentent physiquement les « entités » du SGBD. Selon le modèle logique du SGBD, ces entités peuvent être des n-uplets dans une relation, ou des objets. Nous nous limiterons au premier cas dans ce qui suit.

Un n-uplet dans une table relationnelle est constitué d'une liste d'attributs, chacun ayant un type. À ce n-uplet correspond un enregistrement, constitué de *champs* (*field* en anglais). Chaque type d'attribut détermine la taille du champ nécessaire pour stocker une instance du type. Le tableau 1.3 donne la taille habituelle utilisée pour les principaux types de la norme SQL, étant entendu que les systèmes sont libres de choisir le mode de stockage.

La taille d'un n-uplet est, en première approximation, la somme des tailles des champs stockant ses attributs. En pratique les choses sont un peu plus compliquées. Les champs – et donc les enregistrements – peuvent être de taille variable par exemple. Si la taille de l'un de ces enregistrements de taille variable, augmente au cours d'une mise à jour, il faut pouvoir trouver un espace libre. Se pose également la question de la représentation des valeurs NULL. Nous discutons des principaux aspects de la représentation des enregistrements dans ce qui suit.

Champs de tailles fixe et variable

Comme l'indique le tableau 1.3, les types de la norme SQL peuvent être divisés en deux catégories : ceux qui peuvent être représentés par un champ une taille fixe, et ceux qui sont représentés par un champ de taille variable.

Les types numériques (entiers et flottants) sont stockés au format binaire sur 2, 4 ou 8 octets. Quand on utilise un type DECIMAL pour fixer la précision, les nombres sont en revanche stockés sous la forme d'une chaîne de caractères. Par exemple un champ de type DECIMAL(12, 2) sera stocké sur 12 octets, les deux derniers correspondant aux deux décimales. Chaque octet contient un caractère représentant un chiffre.

Les types DATE et TIME peuvent être simplement représentés sous la forme de chaînes de caractères, aux formats respectifs 'AAAAMMJJ' et 'HHMMSS'.

Le type CHAR est particulier : il indique une chaîne de taille fixe, et un CHAR(5) sera donc stocké sur 5 octets. Se pose alors la question : comment est représentée la valeur 'Bou' ? Il y a deux solutions :

- on complète les deux derniers caractères avec des blancs ;
- on complète les deux derniers caractères avec un caractère conventionnel.

La convention adoptée influe sur les comparaisons puisque dans un cas on a stocké 'Bou ' (avec deux blancs), et dans l'autre 'Bou' sans caractères de terminaison. Si on utilise le type CHAR il est important d'étudier la convention adoptée par le SGBD.

On utilise beaucoup plus souvent le type VARCHAR(n) qui permet de stocker des chaînes de longueur variable. Il existe (au moins) deux possibilités :

- le champ est de longueur $n + 1$, le premier octet contenant un entier indiquant la longueur exacte de la chaîne ; si on stocke 'Bou' dans un VARCHAR(10), on aura donc '3Bou', le premier octet stockant un 3 au format binaire, les trois octets suivants des caractères 'B', 'o' et 'u', et les 7 octets suivants restant inutilisés ;
- le champ est de longueur $l + 1$, avec $l < n$; ici on ne stocke pas les octets inutilisés, ce qui permet d'économiser de l'espace.

Noter qu'en représentant un entier sur un octet, on limite la taille maximale d'un VARCHAR à 255. Une variante qui peut lever cette limite consiste à remplacer l'octet initial contenant la taille par un caractère de terminaison de la chaîne (comme en C).

Le type BIT VARYING peut être représenté comme un VARCHAR, mais comme l'information stockée ne contient pas que des caractères codés en ASCII, on ne peut pas utiliser de caractère de terminaison puisqu'on ne saurait par le distinguer des caractères de la valeur stockée. On préfixe donc le champ par la taille utile, sur 2, 4 ou 8 octets selon la taille maximale autorisée pour ce type.

On peut utiliser un stockage optimisé dans le cas d'un type énuméré dont les instances ne peuvent prendre leur (unique) valeur que dans un ensemble explicitement spécifié (par exemple avec une clause CHECK). Prenons l'exemple de l'ensemble de valeurs suivant :

$$\{ 'valeur1', 'valeur2', \dots 'valeurN' \}$$

Le SGBD doit contrôler, au moment de l'affectation d'une valeur à un attribut de ce type, qu'elle appartient bien à l'ensemble énuméré $\{ 'valeur1', 'valeur2', \dots 'valeurN' \}$. On peut alors stocker l'indice de la valeur, sur 1 ou 2 octets selon la taille de l'ensemble énuméré (au maximum 65535 valeurs pour 2 octets). Cela représente un gain d'espace, notamment si les valeurs consistent en chaînes de caractères.

En-tête d'enregistrement

De même que l'on préfixe un champ de longueur variable par sa taille utile, il est souvent nécessaire de stocker quelques informations complémentaires sur un enregistrement dans un en-tête. Ces informations peuvent être :

- la taille de l'enregistrement, s'il est de taille variable ;
- un pointeur vers le schéma de la table, pour savoir quel est le type de l'enregistrement ;
- la date de dernière mise à jour ;
- etc.

On peut également utiliser cet en-tête pour les valeurs NULL. L'absence de valeur pour un des attributs est en effet délicate à gérer : si on ne stocke rien, on risque de perturber le découpage du champ, tandis que si on stocke une valeur conventionnelle, on perd de l'espace. Une solution possible consiste à créer un masque de bits, un pour chaque champ de l'enregistrement, et à donner à chaque bit la valeur 0 si le champ est NULL, et 1 sinon. Ce masque peut être stocké dans l'en-tête de l'enregistrement, et on peut alors se permettre de ne pas utiliser d'espace pour une valeur NULL, tout en restant en mesure de décoder correctement la chaîne d'octets constituant l'enregistrement.

Exemple 4. Prenons l'exemple d'une table *Film* avec les attributs *id* de type INTEGER, *titre* de type VARCHAR(50) et *annee* de type INTEGER. Regardons la représentation de l'enregistrement (123, 'Vertigo', NULL) (donc l'année est inconnue).

L'identifiant est stocké sur 4 octets, et le titre sur 8 octets, dont un pour la longueur. L'en-tête de l'enregistrement contient un pointeur vers le schéma de la table, sa longueur totale (soit 4 + 8), et un masque de bits 110 indiquant que le troisième champ est à NULL. La figure 1.6 montre cet enregistrement : notez qu'en lisant l'en-tête, on sait calculer l'adresse de l'enregistrement suivant.

□

1.2.2 Blocs

Le stockage des enregistrements dans un fichier doit tenir compte du découpage en blocs de ce fichier. En général il est possible de placer plusieurs enregistrements dans un bloc, et on veut éviter qu'un enregis-

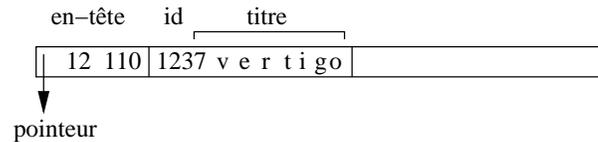


FIGURE 1.6 – Exemple d’un enregistrement avec en-tête

trement chevauche deux blocs. Le nombre maximal d’enregistrements de taille E pour un bloc de taille B est donné par $\lfloor B/E \rfloor$ où la notation $\lfloor x \rfloor$ désigne le plus grand entier inférieur à x .

Prenons l’exemple d’un fichier stockant une table qui ne contient pas d’attributs de longueur variable – en d’autres termes, elle n’utilise pas les types VARCHAR ou BIT VARYING. Les enregistrements ont alors une taille fixe obtenue en effectuant la somme des tailles de chaque attribut. Supposons que cette taille soit en l’occurrence 84 octets, et que la taille de bloc soit 4096 octets. On va de plus considérer que chaque bloc contient un en-tête de 100 octets pour stocker des informations comme l’espace libre disponible dans le bloc, un chaînage avec d’autres blocs, etc. On peut donc placer $\lfloor \frac{4096-100}{84} \rfloor = 47$ enregistrements dans un bloc. Notons qu’il reste dans chaque bloc $3996 - (47 \times 84) = 48$ octets inutilisés dans chaque bloc.

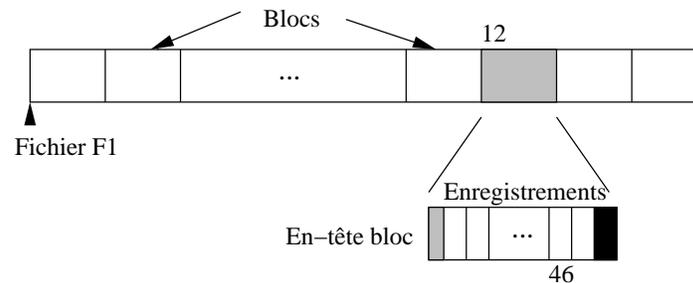


FIGURE 1.7 – Fichier avec blocs et enregistrements

Le transfert en mémoire de l’enregistrement 563 de ce fichier est simplement effectué en déterminant dans quel bloc il se trouve (soit $\lfloor 563/47 \rfloor + 1 = 12$), en chargeant le douzième bloc en mémoire centrale et en prenant dans ce bloc l’enregistrement. Le premier enregistrement du bloc 12 a le numéro $11 \times 47 + 1 = 517$, et le dernier enregistrement le numéro $12 \times 47 = 564$. L’enregistrement 563 est donc l’avant-dernier du bloc, avec pour numéro interne le 46 (voir figure 1.7).

Le petit calcul qui précède montre comment on peut localiser physiquement un enregistrement : par son fichier, puis par le bloc, puis par la position dans le bloc. En supposant que le fichier est codé par 'F1', l’adresse de l’enregistrement peut être représentée par 'F1.12.46'.

Il y a beaucoup d’autres modes d’adressage possibles. L’inconvénient d’utiliser une adresse physique par exemple est que l’on ne peut pas changer un enregistrement de place sans rendre du même coup invalides les pointeurs sur cet enregistrement (dans les index par exemple).

Pour permettre le déplacement des enregistrements on peut combiner une *adresse logique* qui identifie un enregistrement indépendamment de sa localisation. Une table de correspondance permet de gérer l’association entre l’adresse physique et l’adresse logique (voir figure 1.8). Ce mécanisme d’indirection permet beaucoup de souplesse dans l’organisation et la réorganisation d’une base puisqu’il suffit de référencer systématiquement un enregistrement par son adresse logique, et de modifier l’adresse physique dans la table quand un déplacement est effectué. En revanche il entraîne un coût additionnel puisqu’il faut systématiquement inspecter la table de correspondance pour accéder aux données.

Une solution intermédiaire combine adressages physique et logique. Pour localiser un enregistrement on donne l’adresse physique de son bloc, puis, dans le bloc lui-même, on gère une table donnant la locali-

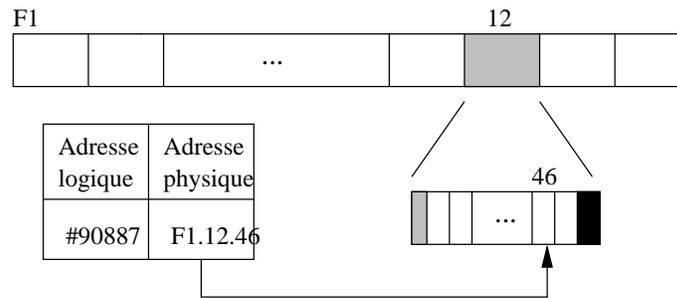


FIGURE 1.8 – Adressage avec indirection

sation au sein du bloc ou, éventuellement, dans un autre bloc.

Reprenons l'exemple de l'enregistrement F1.12.46. Ici F1.12 indique bien le bloc 12 du fichier F1. En revanche 46 est une identification logique de l'enregistrement, gérée au sein du bloc. La figure 1.9 montre cet adressage à deux niveaux : dans le bloc F1.12, l'enregistrement 46 correspond à un emplacement au sein du bloc, tandis que l'enregistrement 57 a été déplacé dans un autre bloc.

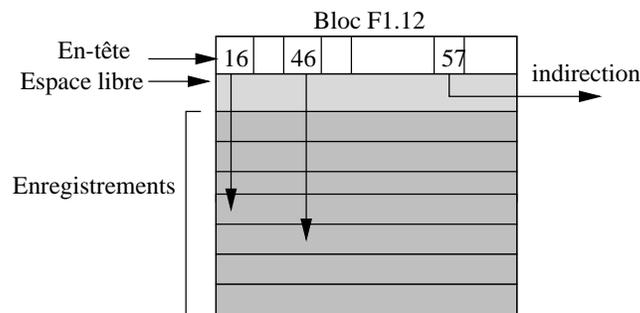


FIGURE 1.9 – Combinaison adresse logique/adresse physique

Noter que l'espace libre dans le bloc est situé entre l'en-tête du bloc et les enregistrements eux-mêmes. Cela permet d'augmenter simultanément ces deux composantes au moment d'une insertion par exemple, sans avoir à effectuer de réorganisation interne du bloc.

Ce mode d'identification offre beaucoup d'avantages, et est utilisé par ORACLE par exemple. Il permet de réorganiser simplement l'espace interne à un bloc.

Enregistrements de taille variable

Une table qui contient des attributs VARCHAR ou BIT VARYING est représentée par des enregistrements de taille variable. Quand un enregistrement est inséré dans le fichier, on calcule sa taille non pas d'après le type des attributs, mais d'après le nombre réel d'octets nécessaires pour représenter les valeurs des attributs. Cette taille doit de plus être stockée au début de l'emplacement pour que le SGBD puisse déterminer le début de l'enregistrement suivant.

Il peut arriver que l'enregistrement soit mis à jour, soit pour compléter la valeur d'un attribut, soit pour donner une valeur à un attribut qui était initialement à NULL. Dans un tel cas il est possible que la place initialement réservée soit insuffisante pour contenir les nouvelles informations qui doivent être stockées dans un autre emplacement du même fichier. Il faut alors créer un chaînage entre l'enregistrement initial et les parties complémentaires qui ont dû être créées. Considérons par exemple le scénario suivant, illustré dans la figure 1.10 :

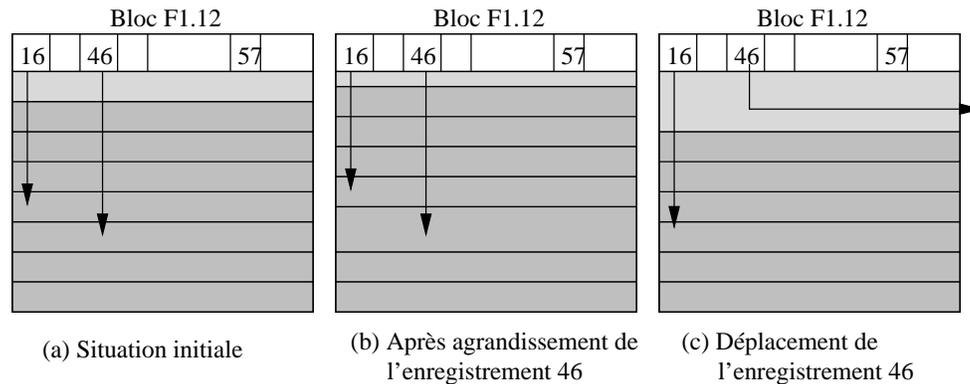


FIGURE 1.10 – Mises à jour d'un enregistrement de taille variable

1. on insère dans la table *Film* un film « Marnie », sans résumé ; l'enregistrement correspondant est stocké dans le bloc F1.12, et prend le numéro 46 ;
2. on insère un autre film, stocké à l'emplacement 47 du bloc F1.12 ;
3. on s'aperçoit alors que le titre exact est « Pas de printemps pour Marnie », ce qui peut se corriger avec un ordre UPDATE : si l'espace libre restant dans le bloc est suffisant, il suffit d'effectuer une réorganisation interne pendant que le bloc est en mémoire centrale, réorganisation qui a un coût nul en terme d'entrées/sorties ;
4. enfin on met à nouveau l'enregistrement à jour pour stocker le résumé qui était resté à NULL : cette fois il ne reste plus assez de place libre dans le bloc, et l'enregistrement doit être déplacé dans un autre bloc, tout en gardant la même adresse.

Au lieu de déplacer l'enregistrement entièrement (solution adoptée par Oracle par exemple), on pourrait le fragmenter en stockant le résumé dans un autre bloc, avec un chaînage au niveau de l'enregistrement (solution adoptée par MySQL). Le déplacement (ou la fragmentation) des enregistrements de taille variable est évidemment pénalisante pour les performances. Il faut effectuer autant de lectures sur le disque qu'il y a d'indirections (ou de fragments), et on peut donc assimiler le coût d'une lecture d'un enregistrement en n parties, à n fois le coût d'un enregistrement compact. Comme nous le verrons plus loin, un SGBD comme Oracle permet de réserver un espace disponible dans chaque bloc pour l'agrandissement des enregistrements afin d'éviter de telles réorganisations.

Les enregistrements de taille variable sont un peu plus compliqués à gérer pour le SGBD que ceux de taille fixe. Les modules accédant au fichier doivent prendre en compte les en-têtes de bloc ou d'enregistrement pour savoir où commence et où finit un enregistrement donné.

En contrepartie, un fichier contenant des enregistrements de taille variable utilise souvent mieux l'espace qui lui est attribué. Si on définissait par exemple le titre d'un film et les autres attributs de taille variable comme des CHAR et pas comme des VARCHAR, tous les enregistrements seraient de taille fixe, au prix de beaucoup d'espace perdu puisque la taille choisie correspond souvent à des cas extrêmes rarement rencontrés – un titre de film va rarement jusqu'à 50 octets.

1.2.3 Organisation d'un fichier

Les systèmes d'exploitation organisent les fichiers qu'ils gèrent dans une arborescence de *répertoires*. Chaque répertoire contient un ensemble de fichiers identifiés de manière unique (au sein du répertoire) par un nom. Il faut bien distinguer l'emplacement *physique* du fichier sur le disque et son emplacement *logique* dans l'arbre des répertoires du système. Ces deux aspects sont indépendants : il est possible de changer le nom d'un fichier ou de modifier son répertoire sans que cela affecte ni son emplacement physique ni son contenu.

Qu'est-ce qu'une organisation de fichier ?

Du point de vue du SGBD, un fichier est une liste de blocs, regroupés sur certaines pistes ou répartis aléatoirement sur l'ensemble du disque et chaînés entre eux. La première solution est bien entendu préférable pour obtenir de bonnes performances, et les SGBD tentent dans la mesure du possible de gérer des fichiers constitués de blocs consécutifs. Quand il n'est pas possible de stocker un fichier sur un seul espace contigu (par exemple un seul cylindre du disque), une solution intermédiaire est de chaîner entre eux de tels espaces.

Le terme *d'organisation* pour un fichier désigne la structure utilisée pour stocker les enregistrements du fichier. Une bonne organisation a pour but de limiter les ressources en espace et en temps consacrées à la gestion du fichier.

- **Espace.** La situation optimale est celle où la taille d'un fichier est la somme des tailles des enregistrements du fichier. Cela implique qu'il y ait peu, ou pas, d'espace inutilisé dans le fichier.
- **Temps.** Une bonne organisation doit favoriser les opérations sur un fichier. En pratique, on s'intéresse plus particulièrement à la recherche d'un enregistrement, notamment parce que cette opération conditionne l'efficacité de la mise à jour et de la destruction. Il ne faut pas pour autant négliger le coût des insertions.

L'efficacité en espace peut être mesurée comme le rapport entre le nombre de blocs utilisés et le nombre minimal de blocs nécessaire. Si, par exemple, il est possible de stocker 4 enregistrements dans un bloc, un stockage optimal de 1000 enregistrements occupera 250 blocs. Dans une mauvaise organisation il n'y aura qu'un enregistrement par bloc et 1000 blocs seront nécessaires. Dans le pire des cas l'organisation autorise des blocs vides et la taille du fichier devient indépendante du nombre d'enregistrements.

Il est difficile de garantir une utilisation optimale de l'espace à tout moment à cause des destructions et modifications. Une bonne gestion de fichier doit avoir pour but – entre autres – de réorganiser dynamiquement le fichier afin de préserver une utilisation satisfaisante de l'espace.

L'efficacité en temps d'une organisation de fichier se définit en fonction d'une opération donnée (par exemple l'insertion, ou la recherche) et se mesure par le rapport entre le nombre de blocs lus et la taille totale du fichier. Pour une recherche par exemple, il faut dans le pire des cas lire tous les blocs du fichier pour trouver un enregistrement, ce qui donne une complexité linéaire. Certaines organisations permettent d'effectuer des recherches en temps sous-linéaire : arbres-B (temps logarithmique) et hachage (temps constant).

Une bonne organisation doit réaliser un bon compromis pour les quatre principaux types d'opérations :

1. insertion d'un enregistrement ;
2. recherche d'un enregistrement ;
3. mise à jour d'un enregistrement ;
4. destruction d'un enregistrement.

Dans ce qui suit nous discutons de ces quatre opérations sur la structure la plus simple qui soit, le *fichier séquentiel* (non ordonné). Le chapitre suivant est consacré aux techniques d'indexation et montrera comment on peut optimiser les opérations d'accès à un fichier séquentiel.

Dans un fichier séquentiel (*sequential file* ou *heap file*), les enregistrements sont stockés dans l'ordre d'insertion, et à la première place disponible. Il n'existe en particulier aucun ordre sur les enregistrements qui pourrait faciliter une recherche. En fait, dans cette organisation, on recherche plutôt une bonne utilisation de l'espace et de bonnes performances pour les opérations de mise à jour.

Recherche

La recherche consiste à trouver le ou les enregistrements satisfaisant un ou plusieurs critères. On peut rechercher par exemple tous les films parus en 2001, ou bien ceux qui sont parus en 2001 et dont le titre commence par 'V', ou encore n'importe quelle combinaison booléenne de tels critères.

La complexité des critères de sélection n'influe pas sur le coût de la recherche dans un fichier séquentiel. Dans tous les cas on doit partir du début du fichier, lire un par un tous les enregistrements en mémoire centrale, et effectuer à ce moment-là le test sur les critères de sélection. Ce test s'effectuant en mémoire

centrale, sa complexité peut être considérée comme négligeable par rapport au temps de chargement de tous les blocs du fichier.

Quand on ne sait par a priori combien d'enregistrements on va trouver, il faut systématiquement parcourir tout le fichier. En revanche, si on fait une recherche par clé unique, on peut s'arrêter dès que l'enregistrement est trouvé. Le coût moyen est dans ce cas égal à $\frac{n}{2}$, n étant le nombre de blocs.

Si le fichier est trié sur le champ servant de critère de recherche, il est possible d'effectuer une recherche par dichotomie qui est beaucoup plus efficace. Prenons l'exemple de la recherche du film *Scream*. L'algorithme est simple :

1. prendre le bloc au milieu du fichier ;
2. si on y trouve *Scream* la recherche est terminée ;
3. sinon, soit les films contenus dans le bloc précédent *Scream* dans l'ordre lexicographique, et la recherche doit continuer dans la partie droite, du fichier, soit la recherche doit continuer dans la partie gauche ;
4. on recommence à l'étape (1), en prenant pour espace de recherche la moitié droite ou gauche du fichier, selon le résultat de l'étape 2.

L'algorithme est récursif et permet de diminuer par deux, à chaque étape, la taille de l'espace de recherche. Si cette taille, initialement, est de n blocs, elle passe à $\frac{n}{2}$ à l'étape 1, à $\frac{n}{2^2}$ à l'étape 2, et plus généralement à $\frac{n}{2^k}$ à l'étape k .

Au pire, la recherche se termine quand il n'y a plus qu'un seul bloc à explorer, autrement dit quand k est tel que $n < 2^k$. On en déduit le nombre maximal d'étapes : c'est le plus petit k tel que $n < 2^k$, soit $\log_2(n) < k$, soit $k = \lceil \log_2(n) \rceil$.

Pour un fichier de 100 Mo, un parcours séquentiel implique la lecture des 25 000 blocs, alors qu'une recherche par dichotomie ne demande que $\lceil \log_2(25000) \rceil = 15$ lectures de blocs !! Le gain est considérable.

L'algorithme décrit ci-dessus se heurte cependant en pratique à deux obstacles.

1. en premier lieu il suppose que le fichier est organisé d'un seul tenant, et qu'il est possible à chaque étape de calculer le bloc du milieu ; en pratique cette hypothèse est très difficile à satisfaire ;
2. en second lieu, le maintien de l'ordre dans un fichier soumis à des insertions, suppressions et mises à jour est très difficile à obtenir.

Cette idée de se baser sur un tri pour effectuer des recherches efficaces est à la source de très nombreuses structures d'index qui seront étudiées dans le chapitre suivant. L'arbre-B, en particulier, peut être vu comme une structure résolvant les deux problèmes ci-dessus. D'une part il se base sur un système de pointeurs décrivant, à chaque étape de la recherche, l'emplacement de la partie du fichier qui reste à explorer, et d'autre part il utilise une algorithmique qui lui permet de se réorganiser dynamiquement sans perte de performance.

Mises à jour

Au moment où on doit insérer un nouvel enregistrement dans un fichier, le problème est de trouver un bloc avec un espace libre suffisant. Il est hors de question de parcourir tous les blocs, et on ne peut pas se permettre d'insérer toujours à la fin du fichier car il faut réutiliser les espaces rendus disponibles par les destructions. La seule solution est de garder une structure annexe qui distingue les blocs pleins des autres, et permette de trouver rapidement un bloc avec de l'espace disponible. Nous présentons deux structures possibles.

La première est une liste doublement chaînée des blocs libres (voir figure 1.11). Quand de l'espace se libère dans un bloc plein, on l'insère à la fin de la liste chaînée. Inversement, quand un bloc libre devient plein, on le supprime de la liste. Dans l'exemple de la figure 1.11, en imaginant que le bloc 8 devienne plein, on chaînera ensemble les blocs 3 et 7 par un jeu classique de modification des adresses. Cette solution nécessite deux adresses (bloc précédent et bloc suivant) dans l'en-tête de chaque bloc, et l'adresse du premier bloc de la liste dans l'en-tête du fichier.

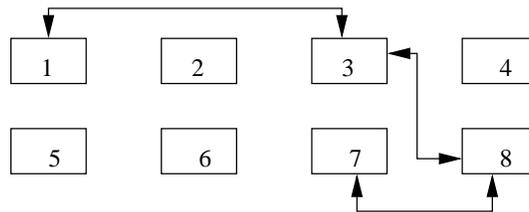


FIGURE 1.11 – Gestion des blocs libres avec liste chaînée

Un inconvénient de cette structure est qu’elle ne donne pas d’indication sur la quantité d’espace disponible dans les blocs. Quand on veut insérer un enregistrement de taille volumineuse, on risque d’avoir à parcourir une partie de la liste – et donc de lire plusieurs blocs – avant de trouver celui qui dispose d’un espace suffisant.

La seconde solution repose sur une structure séparée des blocs du fichier. Il s’agit d’un répertoire qui donne, pour chaque page, un indicateur O/N indiquant s’il reste de l’espace, et un champ donnant le nombre d’octets (figure 1.12). Pour trouver un bloc avec une quantité d’espace libre donnée, il suffit de parcourir ce répertoire.

libre ?	espace	adresse
O	123	1
N		2
		...
O	1089	7

FIGURE 1.12 – Gestion des blocs libres avec répertoire

Le répertoire doit lui-même être stocké dans une ou plusieurs pages associées au fichier. Dans la mesure où l’on n’y stocke que très peu d’informations par bloc, sa taille sera toujours considérablement moins élevée que celle du fichier lui-même, et on peut considérer que le temps d’accès au répertoire est négligeable comparé aux autres opérations.

1.3 Oracle

Le système de représentation physique d’Oracle est assez riche et repose sur une terminologie qui porte facilement à confusion. En particulier les termes de « représentation physique » et « représentation logique » ne sont pas employés dans le sens que nous avons adopté jusqu’à présent. Pour des raisons de clarté, nous adaptons quand c’est nécessaire la terminologie d’Oracle pour rester cohérent avec celle que nous avons employée précédemment.

Un système Oracle (une *instance* dans la documentation) stocke les données dans un ou plusieurs *fichiers*. Ces fichiers sont entièrement attribués au SGBD. Ils sont divisés en *blocs* dont la taille – paramétrable – peut varier de 1K à 8K. Au sein d’un fichier des blocs consécutifs peuvent être regroupés pour former des *extensions* (*extent*). Enfin un ensemble d’extensions permettant de stocker un des objets physiques de la base (une table, un index) constitue un *segment*.

Il est possible de paramétrer, pour un ou plusieurs fichiers, le mode de stockage des données. Ce paramétrage comprend, entre autres, la taille des extensions, le nombre maximal d’extensions formant un

segment, le pourcentage d'espace libre laissé dans les blocs, etc. Ces paramètres, et les fichiers auxquels ils s'appliquent, portent le nom de *tablespace*.

Nous revenons maintenant en détail sur ces concepts.

1.3.1 Fichiers et blocs

Au moment de la création d'une base de données, il faut attribuer à Oracle au moins un fichier sur un disque. Ce fichier constitue l'espace de stockage initial qui contiendra, au départ, le dictionnaire de données.

La taille de ce fichier est choisie par le DBA, et dépend de l'organisation physique qui a été choisie. On peut allouer un seul gros fichier et y placer toutes les données et tous les index, ou bien restreindre ce fichier initial au stockage du dictionnaire et ajouter d'autres fichiers, un pour les index, un pour les données, etc. Le deuxième type de solution est sans doute préférable, bien qu'un peu plus complexe. Il permet notamment, en plaçant les fichiers sur plusieurs disques, de bien répartir la charge des contrôleurs de disque. Une pratique courante – et recommandée par Oracle – est de placer un fichier de données sur un disque et un fichier d'index sur un autre. La répartition sur plusieurs disques permet en outre, grâce au paramétrage des *tablespaces* qui sera étudié plus loin, de régler finement l'utilisation de l'espace en fonction de la nature des informations – données ou index – qui y sont stockées.

Les blocs ORACLE

Le bloc est la plus petite unité de stockage gérée par ORACLE. La taille d'un bloc peut être choisie au moment de l'initialisation d'une base, et correspond obligatoirement à un multiple de la taille des blocs du système d'exploitation. À titre d'exemple, un bloc dans un système comme Linux occupe 1024 octets, et un bloc ORACLE occupe typiquement 4 096 ou 8 092 octets.

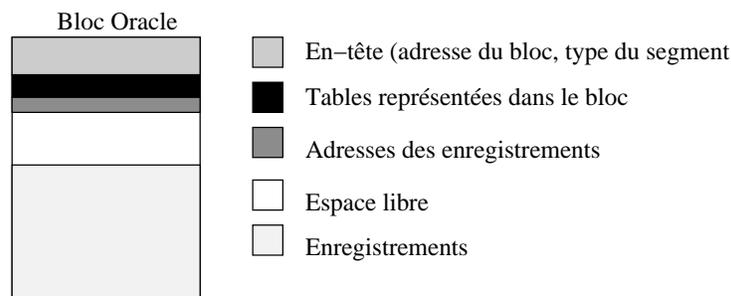


FIGURE 1.13 – Structure d'un bloc Oracle

La structure d'un bloc est identique quel que soit le type d'information qui y est stocké. Elle est constituée des cinq parties suivantes (voir figure 1.13) :

- l'*entête (header)* contient l'adresse du bloc, et son type (données, index, etc) ;
- la *répertoire des tables* donne la liste des tables pour lesquelles des informations sont stockées dans le bloc ;
- la *répertoire des enregistrements* contient les adresses des enregistrements du bloc ;
- un *espace libre* est laissé pour faciliter l'insertion de nouveaux enregistrements, ou l'agrandissement des enregistrements du bloc (par exemple un attribut à NULL auquel on donne une valeur par un UPDATE).
- enfin l'*espace des données* contient les enregistrements.

Les trois premières parties constituent un espace de stockage qui n'est pas directement dédié aux données (ORACLE le nomme l'*overhead*). Cet espace « perdu » occupe environ 100 octets. Le reste permet de stocker les données des enregistrements.

Les paramètres PCTFREE et PCTUSED

La quantité d'espace libre laissée dans un bloc peut être spécifiée grâce au paramètre PCTFREE, au moment de la création d'une table ou d'un index. Par exemple une valeur de 30% indique que les insertions se feront dans le bloc jusqu'à ce que 70% du bloc soit occupé, les 30% restant étant réservés aux éventuels agrandissements des enregistrements. Une fois que cet espace disponible de 70% est rempli, ORACLE considère qu'aucune nouvelle insertion ne peut se faire dans ce bloc.

Notez qu'il peut arriver, pour reprendre l'exemple précédent, que des modifications sur les enregistrements (mise à NULL de certains attributs par exemple) fassent baisser le taux d'occupation du bloc. Quand ce taux baisse en dessous d'une valeur donnée par le paramètre PCTUSED, ORACLE considère que le bloc est à nouveau disponible pour des insertions.

En résumé, PCTFREE indique le taux d'utilisation maximal au-delà duquel les insertions deviennent interdites, et PCTUSED indique le taux d'utilisation minimal en-deçà duquel ces insertions sont à nouveau possibles. Les valeurs de ces paramètres dépendent de l'application, ou plus précisément des caractéristiques des données stockées dans une table particulière. Une petite valeur pour PCTFREE permet aux insertions de remplir plus complètement le bloc, et peut donc mieux exploiter l'espace disque. Ce choix peut être valable pour des données qui sont rarement modifiées. En contrepartie une valeur plus importante de PCTFREE va occuper plus de blocs pour les mêmes données, mais offre plus de flexibilité pour des mises à jour fréquentes.

Voici deux scénarios possibles pour spécifier PCTUSED et PCTFREE. Dans le premier, PCTFREE vaut 30%, et PCTUSED 40% (notez que la somme de ces deux valeurs ne peut jamais excéder 100%). Les insertions dans un bloc peuvent donc s'effectuer jusqu'à ce que 70% du bloc soit occupé. Le bloc est alors retiré de la liste des blocs disponibles pour des insertions, et seules des mises à jour (destructions ou modifications) peuvent affecter son contenu. Si, à la suite de ces mises à jour, l'espace occupé tombe en-dessous de 40%, le bloc est à nouveau marqué comme étant disponible pour des insertions.

Dans ce premier scénario, on accepte d'avoir beaucoup d'espace inoccupé, au pire 60%. L'avantage est que le coût de maintenance de la liste des blocs disponibles pour l'insertion est limité pour ORACLE.

Dans le second scénario, PCTFREE vaut 10% (ce qui est d'ailleurs la valeur par défaut), et PCTUSED 80%. Quand le bloc est plein à 90%, les insertions s'arrêtent, mais elles reprennent dès que le taux d'occupation tombe sous 80%. On est assuré d'une bonne utilisation de l'espace, *mais* le travail du SGBD est plus important (et donc pénalisé) puisque la gestion des blocs disponibles/indisponibles devient plus intensive. De plus, en ne laissant que 10% de marge de manœuvre pour d'éventuelles extensions des enregistrements, on s'expose éventuellement à la nécessité de chaîner les enregistrements sur plusieurs blocs.

Enregistrements

Un enregistrement est une suite de données stockés, à quelques variantes près, comme décrit dans le tableau 1.3, page 18. Par exemple les données de type CHAR(*n*) sont stockées dans un tableau d'octets de longueur *n* + 1. Le premier octet indique la taille de la chaîne, qui doit donc être comprise entre 1 et 255. Les *n* octets suivants stockent les caractères de la chaînes, complétés par des blancs si la longueur de cette dernière est inférieure à la taille maximale. Pour les données de type VARCHAR(*n*) en revanche, seuls les octets utiles pour la représentation de la chaîne sont stockés. C'est un cas où une mise à jour élargissant la chaîne entraîne une réorganisation du bloc.

Chaque attribut est précédé de la longueur de stockage. Dans Oracle les valeurs NULL sont simplement représentées par une longueur de 0. Cependant, si les *n* derniers attributs d'un enregistrement sont NULL, ORACLE se contente de placer une marque de fin d'enregistrement, ce qui permet d'économiser de l'espace.

Chaque enregistrement est identifié par un ROWID, comprenant trois parties :

1. le numéro du bloc au sein du fichier ;
2. le numéro de l'enregistrement au sein du bloc ;
3. enfin l'identifiant du fichier.

Un enregistrement peut occuper plus d'un bloc, notamment s'il contient les attributs de type LONG. Dans ce cas ORACLE utilise un *chaînage* vers un autre bloc. Une situation comparable est celle de l'agran-

dissement d'un enregistrement qui va au-delà de l'espace libre disponible. Dans ce cas ORACLE effectue une *migration* : l'enregistrement est déplacé en totalité dans un autre bloc, et un pointeur est laissé dans le bloc d'origine pour ne pas avoir à modifier l'adresse de l'enregistrement (*ROWID*). Cette adresse peut en effet être utilisée par des index, et une réorganisation totale serait trop coûteuse. Migration et chaînage sont bien entendu pénalisants pour les performances.

Extensions et segments

Une extension est une suite contiguë (au sens de l'emplacement sur le disque) de blocs. En général une extension est affectée à un seul type de données (par exemple les enregistrements d'une table). Comme nous l'avons vu en détail, cette contiguïté est un facteur essentiel pour l'efficacité de l'accès aux données, puisqu'elle évite les déplacements des têtes de lecture, ainsi que le délai de rotation.

Le nombre de blocs dans une extension peut être spécifié par l'administrateur. Bien entendu des extensions de tailles importantes favorisent de bonnes performances, mais il existe des contreparties :

1. si une table ne contient que quelques enregistrements, il est inutile de lui allouer une extension contenant des milliers de blocs ;
2. l'utilisation et la réorganisation de l'espace de stockage peuvent être plus difficiles pour des extensions de grande taille.

Les extensions sont l'unité de stockage constituant les segments. Si on a par exemple indiqué que la taille des extensions est de 50 blocs, un segment (de données ou d'index) consistera en n extensions de 50 blocs chacune. Il existe quatre types de segments :

1. les segments de données contiennent les enregistrements des tables, avec un segment de ce type par table ;
2. les segments d'index contiennent les enregistrements des index ; il y a un segment par index ;
3. les segments temporaires sont utilisés pour stocker des données pendant l'exécution des requêtes (par exemple pour les tris) ;
4. enfin les segments *rollbacks* contiennent les informations permettant d'effectuer une reprise sur panne ou l'annulation d'une transaction ; il s'agit typiquement des données avant modification, dans une transaction qui n'a pas encore été validée.

Une extension initiale est allouée à la création d'un segment. De nouvelles extensions sont allouées dynamiquement (autrement dit, sans intervention de l'administrateur) au segment au fur et à mesure des insertions : rien ne peut garantir qu'une nouvelle extension est contiguë avec les précédentes. En revanche une fois qu'une extension est affectée à un segment, il faut une commande explicite de l'administrateur, ou une destruction de la table ou de l'index, pour que cette extension redevienne libre.

Quand ORACLE doit créer une nouvelle extension et se trouve dans l'incapacité de constituer un espace libre suffisant, une erreur survient. C'est alors à l'administrateur d'affecter un nouveau fichier à la base, ou de réorganiser l'espace dans les fichiers existants.

1.3.2 Les *tablespaces*

Un *tablespace* est un espace physique constitué de un (au moins) ou plusieurs fichiers. Une base de données ORACLE est donc organisée sous la forme d'un ensemble de *tablespace*, sachant qu'il en existe toujours un, créé au moment de l'initialisation de la base, et nommé SYSTEM. Ce *tablespace* contient le dictionnaire de données, y compris les procédures stockées, les *triggers*, etc.

L'organisation du stockage au sein d'un *tablespace* est décrite par de nombreux paramètres (taille des extensions, nombre maximal d'extensions, etc) qui sont donnés à la création du *tablespace*, et peuvent être modifiés par la suite. C'est donc au niveau de *tablespace* (et pas au niveau du fichier) que l'administrateur de la base peut décrire le mode de stockage des données. La création de plusieurs *tablespaces*, avec des paramètres de stockage individualisés, offre de nombreuses possibilités :

1. adaptation du mode de stockage en fonction d'un type de données particulier ;
 2. affectation d'un espace disque limité aux utilisateurs ;
-

3. contrôle sur la disponibilité de parties de la base, par mise hors service d'un ou plusieurs *tablespaces* ;
4. enfin – et surtout – répartition des données sur plusieurs disques afin d'optimiser les performances.

Un exemple typique est la séparation des données et des index, si possible sur des disques différents, afin d'optimiser la charge des contrôleurs de disque. Il est également possible de créer des *tablespaces* dédiées aux données temporaires ce qui évite de mélanger les enregistrements des tables et ceux temporairement créés au cours d'une opération de tri. Enfin un *tablespace* peut être placé en mode de lecture, les écritures étant interdites. Toutes ces possibilités donnent beaucoup de flexibilité pour la gestion des données, aussi bien dans un but d'améliorer les performances que pour la sécurité des accès.

Au moment de la création d'un *tablespace*, on indique les paramètres de stockage par défaut des tables ou index qui seront stockés dans ce *tablespace*. L'expression « par défaut » signifie qu'il est possible, lors de la création d'une table particulière, de donner des paramètres spécifiques à cette table, mais que les paramètres du *tablespace* s'appliquent si on ne le fait pas. Les principaux paramètres de stockage sont :

1. la taille de l'extension initiale (par défaut 5 blocs) ;
2. la taille de chaque nouvelle extension (par défaut 5 blocs également) ;
3. le nombre maximal d'extensions, ce qui donne donc, avec la taille des extensions, le nombre maximal de blocs alloués à une table ou index ;
4. la taille des extensions peut croître progressivement, selon un ratio indiqué par `PCTINCREASE` ; une valeur de 50% pour ce paramètre indique par exemple que chaque nouvelle extension a une taille supérieure de 50% à la précédente.

Voici un exemple de création de *tablespace*.

```
CREATE TABLESPACE TB1
  DATAFILE 'fichierTB1.dat' SIZE 50M
  DEFAULT STORAGE (
    INITIAL 100K
    NEXT 40K
    MAXEXTENTS 20,
    PCTINCREASE 20);
```

La commande crée un *tablespace*, nommé TB1, et lui affecte un premier fichier de 50 mégaoctets. Les paramètres de la partie `DEFAULT STORAGE` indiquent, dans l'ordre :

1. la taille de la première extension allouée à une table (ou un index) ;
2. la taille de la prochaine extension, si l'espace alloué à la table doit être agrandi ;
3. le nombre maximal d'extensions, ici 20 ;
4. enfin chaque nouvelle extension est 20% plus grande que la précédente.

En supposant que la taille d'un bloc est 4K, on obtient une première extension de 25 blocs, une seconde de 10 blocs, une troisième de $10 \times 1,2 = 12$ blocs, etc.

Le fait d'indiquer une taille maximale permet de contrôler que l'espace ne sera pas utilisé sans limite, et sans contrôle de l'administrateur. En contrepartie, ce dernier doit être prêt à prendre des mesures pour répondre aux demandes des utilisateurs quand des messages sont produits par ORACLE indiquant qu'une table a atteint sa taille limite.

Voici un exemple de *tablespace* défini avec un paramétrage plus souple : d'une part il n'y a pas de limite au nombre d'extensions d'une table, d'autre part le fichier est en mode « auto-extension », ce qui signifie qu'il s'étend automatiquement, par tranches de 5 mégaoctets, au fur et à mesure que les besoins en espace augmentent. La taille du fichier est elle-même limitée à 500 mégaoctets.

```
CREATE TABLESPACE TB2
  DATAFILE 'fichierTB2.dat' SIZE 2M
  AUTOEXTEND ON NEXT 5M MAXSIZE 500M
  DEFAULT STORAGE (INITIAL 128K NEXT 128K MAXEXTENTS UNLIMITED);
```

Il est possible, après la création d'un *tablespace*, de modifier ses paramètres, étant entendu que la modification ne s'applique pas aux tables existantes mais à celles qui vont être créées. Par exemple on peut modifier le *tablespace* TB1 pour que les extensions soient de 100K, et le nombre maximal d'extensions porté à 200.

```
ALTER TABLESPACE TB1
  DEFAULT STORAGE (
    NEXT 100K
    MAXEXTENTS 200);
```

Voici quelque-unes des différentes actions disponibles sur un *tablespace* :

1. On peut mettre un *tablespace* hors-service, soit pour effectuer une sauvegarde d'une partie de la base, soit pour rendre cette partie de la base indisponible.

```
ALTER TABLESPACE TB1 OFFLINE;
```

Cette commande permet en quelque sorte de traiter un *tablespace* comme une sous-base de données.

2. On peut mettre un *tablespace* en lecture seule.

```
ALTER TABLESPACE TB1 READ ONLY;
```

3. Enfin on peut ajouter un nouveau fichier à un *tablespace* afin d'augmenter sa capacité de stockage.

```
ALTER TABLESPACE ADD DATAFILE 'fichierTB1-2.dat' SIZE 300 M;
```

Au moment de la création d'une base, on doit donner la taille et l'emplacement d'un premier fichier qui sera affecté au *tablespace* SYSTEM. À chaque création d'un nouveau *tablespace* par la suite, il faudra créer un fichier qui servira d'espace de stockage initial pour les données qui doivent y être stockées. Il faut bien noter qu'un fichier n'appartient qu'à un seul *tablespace*, et que, dès le moment où ce fichier est créé, son contenu est exclusivement géré par ORACLE, même si une partie seulement est utilisée. En d'autres termes il ne faut pas affecter un fichier de 1 Go à un *tablespace* destiné seulement à contenir 100 Mo de données, car les 900 Mo restant ne servent alors à rien.

ORACLE utilise l'espace disponible dans un fichier pour y créer de nouvelles extensions quand la taille des données augmente, ou de nouveaux segments quand des tables ou index sont créés. Quand un fichier est plein – ou, pour dire les choses plus précisément, quand ORACLE ne trouve pas assez d'espace disponible pour créer un nouveau segment ou une nouvelle extension –, un message d'erreur avertit l'administrateur qui dispose alors de plusieurs solutions :

- créer un nouveau fichier, et l'affecter au *tablespace* (voir la commande ci-dessus) ;
- modifier la taille d'un fichier existant ;
- enfin, permettre à un ou plusieurs fichiers de croître dynamiquement en fonction des besoins, ce qui peut simplifier la gestion de l'espace.

Comment inspecter les *tablespaces*

ORACLE fournit un certain nombre de vues dans son dictionnaire de données pour consulter l'organisation physique d'une base, et l'utilisation de l'espace.

- La vue *DBA_EXTENTS* donne la liste des extensions ;
- La vue *DBA_SEGMENTS* donne la liste des segments ;
- La vue *DBA_FREE_SPACE* permet de mesurer l'espace libre ;
- La vue *DBA_TABLESPACES* donne la liste des *tablespaces* ;
- La vue *DBA_DATA_FILES* donne la liste des fichiers.

Ces vues sont gérées sous le compte utilisateur SYS qui est réservé à l'administrateur de la base. Voici quelques exemples de requêtes permettant d'inspecter une base. On suppose que la base contient deux *tablespace*, SYSTEM avec un fichier de 50M, et TB1 avec deux fichiers dont les tailles respectives sont 100M et 200M.

La première requête affiche les principales informations sur les *tablespaces*.

```
SELECT tablespace_name "TABLESPACE",
       initial_extent "INITIAL_EXT",
       next_extent "NEXT_EXT",
       max_extents "MAX_EXT"
FROM sys.dba_tablespaces;
```

TABLESPACE	INITIAL_EXT	NEXT_EXT	MAX_EXT
SYSTEM	10240000	10240000	99
TB1	102400	50000	200

On peut obtenir la liste des fichiers d'une base, avec le *tablespace* auquel ils sont affectés :

```
SELECT file_name, bytes, tablespace_name
FROM sys.dba_data_files;
```

FILE_NAME	BYTES	TABLESPACE_NAME
fichier1	5120000	SYSTEM
fichier2	10240000	TB1
fichier3	20480000	TB1

Enfin on peut obtenir l'espace disponible dans chaque *tablespace*. Voici par exemple la requête qui donne des informations statistiques sur les espaces libres du *tablespace* SYSTEM.

```
SELECT tablespace_name, file_id,
       COUNT(*) "PIECES",
       MAX(blocks) "MAXIMUM",
       MIN(blocks) "MINIMUM",
       AVG(blocks) "AVERAGE",
       SUM(blocks) "TOTAL"
FROM sys.dba_free_space
WHERE tablespace_name = 'SYSTEM'
GROUP BY tablespace_name, file_id;
```

TABLESPACE	FILE_ID	PIECES	MAXIMUM	MINIMUM	AVERAGE	SUM
SYSTEM	1	2	2928	115	1521.5	3043

SUM donne le nombre total de blocs libres, PIECES montre la fragmentation de l'espace libre, et MAXIMUM donne l'espace contigu maximal. Ces informations sont utiles pour savoir s'il est possible de créer des tables volumineuses pour lesquelles on souhaite réserver dès le départ une extension de taille suffisante.

1.3.3 Création des tables

Tout utilisateur ORACLE ayant les droits suffisants peut créer des tables. Notons que sous ORACLE la notion d'utilisateur et celle de base de données sont liées : un utilisateur (avec des droits appropriés) dispose d'un espace permettant de stocker des tables, et tout ordre CREATE TABLE effectué par cet utilisateur crée une table et des index qui appartiennent à cet utilisateur.

Il est possible, au moment où on spécifie le profil d'un utilisateur, d'indiquer dans quels *tablespaces* il a le droit de placer des tables, de quel espace total il dispose sur chacun de ces *tablespaces*, et quel est le *tablespace* par défaut pour cet utilisateur.

Il devient alors possible d'inclure dans la commande CREATE TABLE des paramètres de stockage. Voici un exemple :

```

CREATE TABLE Film (... )
PCTFREE 10
PCTUSED 40
TABLESPACE TB1
STORAGE ( INITIAL 50K
          NEXT 50K
          MAXEXTENTS 10
          PCTINCREASE 25 );

```

On indique donc que la table doit être stockée dans le *tablespace* TB1, et on remplace les paramètres de stockage de ce *tablespace* par des paramètres spécifiques à la table *Film*.

Par défaut une table est organisée séquentiellement sur une ou plusieurs extensions. Les index sur la table sont stockés dans un autre segment, et font référence aux enregistrements grâce au ROWID.

Il est également possible d'organiser sous forme d'un arbre, d'une table de hachage ou d'un regroupement *cluster* avec d'autres tables. Ces structures seront décrites dans le chapitre consacré à l'indexation.

1.4 Exercices

Caractéristique	Performance
Taille d'un secteur	512 octets
Nbre de plateaux	5
Nbre de têtes	10
Nombre de cylindres	100 000
Nombre de secteurs par piste	4 000
Temps de positionnement moyen	10 ms
Vitesse de rotation	7 400 rpm
Déplacement de piste à piste	0,5 ms
Taux de transfert max.	120 Mo/s

TABLE 1.4 – Spécifications d'un disque magnétique

Exercice 1. Le tableau 1.4 donne les spécifications partielles d'un disque magnétique. Répondez aux questions suivantes.

1. Quelle est la capacité d'une piste ?, d'un cylindre ? d'une surface ? du disque ?
2. Quel est le temps de latence maximal ? Quel est le temps de latence moyen ?
3. Quel serait le taux de transfert (en Mo/sec) nécessaire pour pouvoir transmettre le contenu d'une piste en une seule rotation ? Est-ce possible ?

Exercice 2. Étant donné un disque contenant C cylindres, un temps de déplacement entre deux pistes adjacentes de s ms, donner la formule exprimant le temps de positionnement moyen. On décrira une demande de déplacement par une paire $[d, a]$ où d est la piste de départ et a la piste d'arrivée, et on supposera que toutes les demandes possibles sont équiprobables.

Attention, on ne peut pas considérer que les têtes se déplacent en moyenne de $C/2$ pistes. C'est vrai si la tête de lecture est au bord des plateaux ou de l'axe, mais pas si elle est au milieu du plateau. Il faut commencer par exprimer le temps de positionnement moyen en fonction d'une position de départ donnée, puis généraliser à l'ensemble des positions de départ possibles.

On pourra utiliser les deux formules suivantes :

1. $\sum_{i=1}^n (i) = \frac{n \times (n+1)}{2}$
2. $\sum_{i=1}^n (i^2) = \frac{n \times (n+1) \times (2n+1)}{6}$

et commencer par exprimer le nombre moyen de déplacements en supposant que les têtes de lecture sont en position p . Il restera alors à effectuer la moyenne de l'expression obtenue, pour l'ensemble des valeurs possibles de p .

Exercice 3. Soit un disque de 5 000 cylindres tournant à 12 000 rpm, avec un temps de déplacement entre deux pistes adjacentes égal à 0,2 ms et 500 secteurs de 512 octets par piste. Quel est le temps moyen de lecture d'un bloc de 4 096 ?

Exercice 4. On considère un fichier de 1 Go et un buffer de 100 Mo.

- quel est le hit ratio en supposant que la probabilité de lire les blocs est uniforme ?
- même question, en supposant que 80 % des lectures concernent 200 Mo, les 20 % restant étant répartis uniformément sur 800 Mo ?
- avec l'hypothèse précédente, jusqu'à quelle taille de buffer peut-on espérer une amélioration significative du hit ratio ?

Exercice 5. Soit l'ordre SQL suivant :

```
DELETE FROM Film
WHERE condition
```

La table est stockée dans un fichier de taille n blocs, sans index. Donner le nombre moyen de blocs à lire, en fonction de n , pour l'exécution de l'ordre SQL dans les cas suivants :

- aucun enregistrement ne satisfait la condition ;
- la condition porte sur une clé unique et affecte donc un seul enregistrement ;
- la condition ne porte pas sur une clé unique (et on ne connaît donc pas à priori le nombre d'enregistrements concernés).

Exercice 6. Soit la table de schéma suivant :

```
CREATE TABLE Personne (id      INT NOT NULL,
                        nom     VARCHAR(40) NOT NULL,
                        prenom  VARCHAR(40) NOT NULL,
                        adresse  VARCHAR(70),
                        dateNaissance DATE)
```

Cette table contient 300 000 enregistrements, stockés dans des blocs de taille 4 096 octets. Un enregistrement ne peut pas chevaucher deux blocs, et chaque bloc comprend un entête de 200 octets.

1. Donner la taille maximale et la taille minimale d'un enregistrement. On suppose par la suite que tous les enregistrements ont une taille maximale.
2. Quel est le nombre maximal d'enregistrements par bloc ?
3. Quelle est la taille du fichier ?
4. En supposant que le fichier est stocké sur le disque de l'exercice 1, combien de cylindres faut-il pour stocker le fichier ?
5. Quel est le temps moyen de recherche d'un enregistrement dans les deux cas suivants : (a) les blocs sont stockés le plus contiguement possible et (b) les blocs sont distribués au hasard sur le disque.
6. On suppose que le fichier est trié sur le nom. Quel est le temps d'une recherche dichotomique pour chercher une personne avec un nom donné ?

Exercice 7. On considère un disque composé de C cylindres qui doit satisfaire un flux de demandes d'entrées-sorties de la forme $[t, a]$ où t est l'instant de soumission de la demande et a l'adresse. On suppose que le SGBD applique de deux techniques de séquençement des entrées-sorties. La première, nommée PDPS (premier demandé premier servi) est classique. La seconde, balayage se décrit comme suit :

- les demandes d'entrées-sorties sont stockées au fur et à mesure de leur arrivée dans un tableau T_{es} à C entrées ; chaque entrée est une liste chaînée stockant, dans l'ordre d'arrivée, les demandes concernant le cylindre courant ;
- les entrées du tableau T_{es} sont balayées séquentiellement de 1 à C , puis de C à 1, et ainsi de suite ; quand on arrive sur une entrée $T_{io}[c]$, on place les têtes de lecture sur le cylindre c , et on effectue les entrées/sorties en attente dans l'ordre de la liste chaînée.

On supposera qu'on traite les demandes connues au moment où on arrive sur le cylindre,

L'idée de balayage est bien entendu de limiter les déplacements importants des têtes de lecture. On suppose que le disque a 250 cylindres, effectue une rotation en 6 ms, et qu'un déplacement entre deux pistes adjacentes prend 0,2 ms.

1. On suppose que le flux des entrées/sorties est le suivant :

$$d_1 = [1, 100], d_2 = [15, 130], d_3 = [17, 130], d_4 = [25, 15],$$

$$d_5 = [42, 130], d_6 = [27, 155], d_7 = [29, 155], d_8 = [70, 250]$$

Indiquez à quel moment chaque entrée/sortie est effectuée (attention : une E/S ne peut être traitée avant d'être soumise !). L'adresse est ici simplement le numéro de cylindre et les instants sont exprimés en valeur absolue et en millisecondes. On suppose que les têtes de lectures sont à l'instant 1 sur le cylindre 1.

2. Donnez les instants où sont effectuées les entrées/sorties avec l'algorithme classique PDPS (contrairement à balayage, on traite donc les demandes dans l'ordre où elles sont soumises).
3. Mêmes questions avec la liste suivante :

$$[1, 100], [10, 130], [15, 10], [20, 180], [25, 50], [30, 250]$$

4. Soit la liste d'E/S suivante :

$$[1, c_1], [2, c_2], [3, c_3], \dots, [100, c_{100}]$$

Supposons que le disque vienne de traiter la demande $d_1 = [1, c_1]$. Donner le temps maximal pour que la demande $d_2 = [2, c_2]$ soit traitée, avec les deux approches (balayage et premier demandé-premier servi).

5. Soit $n > 250$ le nombre de demandes en attente. On suppose qu'elles sont également réparties sur tous les 250 cylindres. Quel est le temps moyen d'accès à un bloc avec l'algorithme de balayage, exprimé en fonction de n ?

Chapitre 2

Indexation

Sommaire

2.1	Principes des index	36
2.2	Indexation de fichiers	37
2.2.1	Index non-dense	38
2.2.2	Index dense	39
2.2.3	Index multi-niveaux	41
2.3	L'arbre-B	41
2.3.1	Présentation intuitive	42
2.3.2	Recherches avec un arbre-B+	43
2.4	Hachage	46
2.4.1	Principes de base	46
2.4.2	Hachage extensible	49
2.5	Les index <i>bitmap</i>	51
2.6	Indexation de données géométriques	52
2.7	Indexation dans Oracle	61
2.7.1	Arbres B+	62
2.7.2	Arbres B	62
2.7.3	Indexation de documents	63
2.7.4	Tables de hachage	63
2.7.5	Index bitmap	64
2.8	Exercices	64

Quand une table est volumineuse, un parcours séquentiel est une opération relativement lente et pénalisante pour l'exécution des requêtes, notamment dans le cas des jointures où ce parcours séquentiel doit parfois être effectué répétitivement. La création d'un *index* permet d'améliorer considérablement les temps de réponse en créant des chemins d'accès aux enregistrements beaucoup plus directs. Un index permet de satisfaire certaines requêtes (mais pas toutes) portant sur un ou plusieurs attributs (mais pas tous). Il ne s'agit donc jamais d'une méthode universelle qui permettrait d'améliorer indistinctement tous les types d'accès à une table.

L'index peut exister indépendamment de l'organisation du fichier de données, ce qui permet d'en créer plusieurs si on veut être en mesure d'optimiser plusieurs types de requêtes. En contrepartie la création sans discernement d'un nombre important d'index peut être pénalisante pour le SGBD qui doit gérer, pour chaque opération de mise à jour sur une table, la répercussion de cette mise à jour sur tous les index de la table. Un choix judicieux des index, ni trop ni trop peu, est donc un des facteurs essentiels de la performance d'un système.

Ce chapitre présente les structures d'index les plus classiques utilisées dans les systèmes relationnels. Après une introduction présentant les principes de base des index, nous décrivons en détail une structure de données appelée *arbre-B* qui est à la fois simple, très performante et propre à optimiser plusieurs types de

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

TABLE 2.1 – La table à indexer

requêtes : recherche par clé, recherche par intervalle, et recherche avec un préfixe de la clé. Le « B » vient de *balanced* en anglais, et signifie que l'arbre est équilibré : tous les chemins partant de la racine vers une feuille ont la même longueur. L'arbre B est utilisé dans tous les SGBD relationnels.

Une structure concurrente de l'arbre B est le *hachage* qui offre, dans certains cas, des performances supérieures, mais ne couvre pas autant d'opérations. Nous verrons également dans ce chapitre des structures d'index dédiées à des données non-traditionnelles pour lesquels l'arbre B n'est pas adapté. Ce sont les *index bitmap* pour les entrepôts de données, et les *index spatiaux* pour les données géographiques.

Pour illustrer les techniques d'indexation d'une table nous prendrons deux exemples.

Exemple 5. Le premier est destiné à illustrer les structures et les algorithmes sur un tout petit ensemble de données, celui de la table *Film*, avec les 16 lignes du tableau 2.1. Nous ne donnons que les deux attributs *titre* et *année* qui seront utilisés pour l'indexation. □

Exemple 6. Le deuxième exemple est destiné à montrer, avec des ordres de grandeur réalistes, l'amélioration obtenue par des structures d'index, et les caractéristiques, en espace et en temps, de ces structures. Nous supposons que la table contient 1 000 000 films, la taille de chaque enregistrement étant de 120 octets. Pour une taille de bloc de 4 096 octets, on aura donc au mieux 34 enregistrements par bloc. Il faut donc 29 412 blocs ($\lceil 1000000/34 \rceil$) occupant un peu plus de 120 Mo (120 471 552 octets, le surplus étant imputable à l'espace perdu dans chaque bloc). C'est sur ce fichier de 120 Mo que nous allons construire nos index. □

2.1 Principes des index

Prenez n'importe quel livre d'informatique (ou autre sujet technique) : il contient un index à la fin. Cet index présente une liste de termes importants, classés en ordre alphabétique, et associées aux numéros des pages où on trouve un développement consacré à ce terme. Les index dans un SGBD suivent exactement le même principe. On sélectionne dans une table une liste des attributs (au moins un), puis on concatène les valeurs de ces attributs dans l'ordre choisi : on obtient l'équivalent des termes indexant le livre. On trie alors cette liste selon l'ordre alphanumérique. Finalement on associe à chaque valeur dans la liste triée un ou plusieurs pointeur(s) vers le (ou les enregistrements) correspondant à cette valeur : c'est l'équivalent des numéros de page. On obtient l'index.

Pour bien utiliser l'index, il faut être en mesure de trouver rapidement le terme qui nous intéresse. Dans un livre, une pratique spontanée consiste à prendre une page de l'index au hasard et à déterminer, en

fonction de la lettre courante, s'il faut regarder avant ou après pour trouver le terme qui nous intéresse. On peut recommencer la même opération sur la partie qui suit ou qui précède, et converger ainsi très rapidement vers la page contenant le terme (recherche « par dichotomie »). Les index des SGBD sont organisés pour appliquer exactement la même technique.

Continuons l'analogie avec les index plaçant (*clustering*) ou non :

1. l'index *plaçant* détermine la position d'un enregistrement ;
2. l'index non plaçant est une structure indépendante du fichier de données, qui se contente de « pointer » vers les enregistrements ;

La différence entre les deux structures est la même que celle entre un dictionnaire et un livre classique. Dans un dictionnaire, les mots sont placés dans l'ordre, alors que dans un livre classique ils apparaissent sans ordre prédéfini. Le dictionnaire correspond à l'index plaçant, l'index d'un livre classique (comme celui à la fin du présent ouvrage) à l'index non plaçant. Notez que :

- le dictionnaire est considérablement plus gros qu'un index sur des termes ; en contrepartie il n'a pas le désavantage de l'indirection (chercher d'abord dans l'index, puis chercher la page par son numéro) ;
 - il peut y avoir plusieurs index non plaçants dans un livre (l'index des termes, des figures, etc.) mais il ne peut y avoir qu'un seul index plaçant (autrement dit, un seul critère d'organisation du livre).
- Ces principes posés, passons aux détails techniques.

2.2 Indexation de fichiers

Le principe de base d'un index est de construire une structure permettant d'optimiser les *recherches par clé* sur un fichier. Le terme de « clé » doit être compris ici au sens de « critère de recherche », ce qui diffère de la notion de clé primaire d'une table. Les recherches par clé sont typiquement les sélections de lignes pour lesquelles la clé a une certaine valeur. Par exemple :

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

La clé est ici le titre du film, que rien n'empêche par ailleurs d'être également la clé primaire de la table. En pratique, la clé primaire est un critère de recherche très utilisé.

Outre les recherches par valeur, illustrées ci-dessus, on veut fréquemment optimiser des recherches par intervalle. Par exemple :

```
SELECT *
FROM Film
WHERE annee BETWEEN 1995 AND 2002
```

Ici la clé de recherche est l'année du film, et l'existence d'un index basé sur le titre ne sera d'aucune utilité. Enfin les clés peuvent être composées de plusieurs attributs, comme, par exemple, les nom et prénom des artistes.

```
SELECT *
FROM Artiste
WHERE nom = 'Alfred' AND prenom='Hitchcock'
```

Pour toutes ces requêtes, en l'absence d'un index approprié, il n'existe qu'une solution possible : parcourir séquentiellement le fichier (la table) en testant chaque enregistrement. Sur notre exemple, cela revient à lire les 30 000 blocs du fichier, pour un coût qui peut être de l'ordre de 30 secondes si le fichier est très mal organisé sur le disque (chaque lecture comptant alors pour environ 10 ms).

Un index permet d'éviter ce parcours séquentiel. La recherche par index d'effectue en deux étapes :

1. le parcours de l'index doit fournir l'adresse de l'enregistrement ;

2. par accès direct au fichier en utilisant l'adresse obtenue précédemment, on obtient l'enregistrement (ou le bloc contenant l'enregistrement, ce qui revient au même en terme de coût).

Il existe des variantes à ce schéma, notamment quand l'index est *plaçant* et influence l'organisation du fichier, mais globalement nous retrouverons ces deux étapes dans la plupart des structures.

2.2.1 Index non-dense

Nous commençons par considérer le cas d'un fichier trié sur la clé primaire (il n'y a donc qu'un seul enregistrement pour une valeur de clé). Dans ce cas restreint il est possible, comme nous l'avons vu dans le chapitre 1, d'effectuer une recherche par dichotomie qui s'appuie sur une division récursive du fichier, avec des performances théoriques très satisfaisantes. En pratique la recherche par dichotomie suppose que le fichier est constitué d'une seule séquence de blocs, ce qui permet à chaque étape de la récursion de trouver le bloc situé au milieu de l'espace de recherche.

Si cette condition est facile à satisfaire pour un tableau en mémoire, elle l'est beaucoup moins pour un fichier occupant plusieurs dizaines, voire centaines de mégaoctets. La première structure que nous étudions permet d'effectuer des recherches sur un fichier triés, même si ce fichier est fragmenté.

L'index est lui-même un fichier, contenant des enregistrements [valeur, Adr] où valeur désigne une valeur de la clé de recherche, et Adr l'adresse d'un bloc. On utilise parfois le terme *d'entrée* pour désigner un enregistrement dans un index.

Toutes les valeurs de clé existant dans le fichier de données ne sont pas représentées dans l'index : on dit que l'index est *non-dense*. On tire parti du fait que le fichier est trié sur la clé pour ne faire figurer dans l'index que les valeurs de clé des premiers enregistrements de chaque bloc. Comme nous allons le voir, cette information est suffisante pour trouver n'importe quel enregistrement.

La figure 2.1 montre un index non-dense sur le fichier des 16 films, la clé étant le titre du film. On suppose que chaque bloc du fichier de données contient 4 enregistrements, ce qui donne un minimum de 4 blocs. Il suffit alors de quatre paires [titre, Adr] pour indexer le fichier. Les titres utilisés sont ceux des premiers enregistrements de chaque bloc, soit respectivement *Annie Hall*, *Greystoke*, *Metropolis* et *Smoke*.

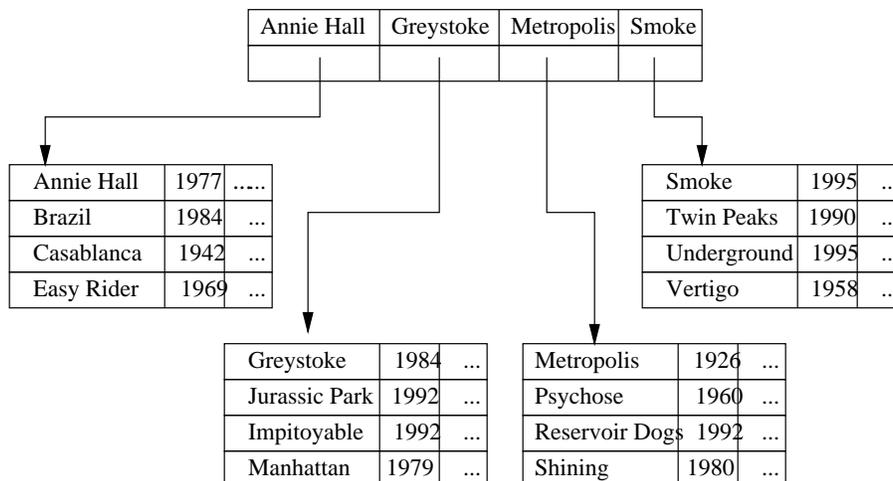


FIGURE 2.1 – Un index non dense

Si on désigne par $\{c_1, c_2, \dots, c_n\}$ la liste ordonnée des clés dans l'index, il est facile de constater que'un enregistrement dont la valeur de clé est c est stocké dans le bloc associé à la clé c_i telle que $c_i \leq c < c_{i+1}$. Supposons que l'on recherche le film *Shining*. En consultant l'index on constate que ce titre est compris

entre *Metropolis* et *Smoke*. On en déduit donc que *Shining* se trouve dans le même bloc que *Metropolis*. Il suffit de lire ce bloc et d'y rechercher l'enregistrement. Le même algorithme s'applique aux recherches basées sur un préfixe de la clé (par exemple tous les films dont le titre commence par 'V').

Le coût d'une recherche dans l'index est considérablement plus réduit que celui d'une recherche dans le fichier principal. D'une part les enregistrements dans l'index sont beaucoup plus petits que ceux du fichier de données puisque seule la clé (et un pointeur) y figurent. D'autre part l'index ne comprend qu'un enregistrement par bloc.

Exemple 7. Considérons l'exemple 6 de notre fichier contenant un million de films (voir page 36). Il est constitué de 29 142 blocs. Supposons qu'un titre de films occupe 20 octets en moyenne, et l'adresse d'un bloc 8 octets. La taille de l'index est donc $29142 * (20 + 8) = 815976$ octets, à comparer aux 120 Mo du fichier de données. □

Le fichier d'index étant trié, il est bien entendu possible de recourir à une recherche par dichotomie pour trouver l'adresse du bloc contenant un enregistrement. Une seule lecture suffit alors pour trouver l'enregistrement lui-même.

Dans le cas d'une recherche par intervalle, l'algorithme est très semblable : on recherche dans l'index l'adresse de l'enregistrement correspondant à la borne inférieure de l'intervalle. On accède alors au fichier grâce à cette adresse et il suffit de partir de cet emplacement et d'effectuer un parcours séquentiel pour obtenir tous les enregistrements cherchés. La recherche s'arrête quand on trouve un enregistrement donc la clé est supérieure à la borne supérieure de l'intervalle.

Exemple 8. Supposons que l'on recherche tous les films dont le titre commence par une lettre entre 'J' et 'P'. On procède comme suit :

1. on recherche dans l'index la plus grande valeur strictement inférieure à 'J' : pour l'index de la figure 2.1 c'est *Greystoke* ;
2. on accède au bloc du fichier de données, et on y trouve le premier enregistrement avec un titre commençant par 'J', soit *Jurassic Park* ;
3. on parcourt la suite du fichier jusqu'à trouver *Reservoir Dogs* qui est au-delà de l'intervalle de recherche, : tous les enregistrements trouvés durant ce parcours constituent le résultat de la requête. □

Le coût d'une recherche par intervalle peut être assimilé, si néglige la recherche dans l'index, au parcours de la partie du fichier qui contient le résultat, soit $\frac{r}{b}$, où r désigne le nombre d'enregistrements du résultat, et b le nombre d'enregistrements dans un bloc. Ce coût est optimal.

Un index dense est extrêmement efficace pour les opérations de recherche. Bien entendu le problème est de maintenir l'ordre du fichier au cours des opérations d'insertions et de destructions, problème encore compliqué par la nécessité de garder une étroite correspondance entre l'ordre du fichier de données et l'ordre du fichier d'index. Ces difficultés expliquent que ce type d'index soit peu utilisé par les SGBD, au profit de l'arbre-B qui offre des performances comparables pour les recherches par clé, mais se réorganise dynamiquement.

2.2.2 Index dense

Que se passe-t-il quand on veut indexer un fichier qui n'est pas trié sur la clé de recherche ? On ne peut tirer parti de l'ordre des enregistrements pour introduire seulement dans l'index la valeur de clé du premier élément de chaque bloc. Il faut donc baser l'index sur *toutes* les valeurs de clé existant dans le fichier, et les associer à l'adresse d'un enregistrement, et pas à l'adresse d'un bloc. Un tel index est *dense*.

La figure 2.2 montre le même fichier contenant seize films, trié sur le titre, et indexé maintenant sur l'année de parution des films. On constate d'une part que toutes les années du fichier de données sont reportées dans l'index, ce qui accroît considérablement la taille de ce dernier, et d'autre part que la même année apparaît autant qu'il y a de films parus cette année là.

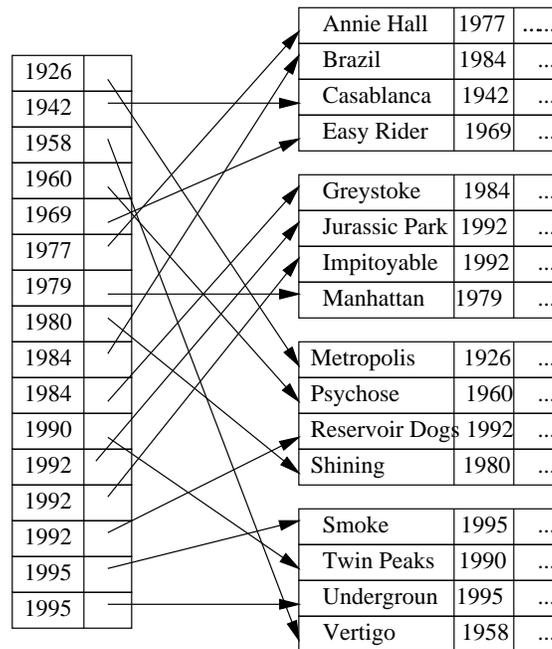


FIGURE 2.2 – Un index dense

Exemple 9. Considérons l'exemple 6 de notre fichier contenant un million de films (voir page 36). Il faut créer une entrée d'index pour chaque film. Une année occupe 4 octets, et l'adresse d'un bloc 8 octets. La taille de l'index est donc $1000000 * (4 + 8) = 12\ 000\ 000$ octets, soit seulement dix fois moins que les 120 Mo du fichier de données. □

Un index dense peut coexister avec un index non-dense. Comme le suggèrent les deux exemples qui précèdent, on peut envisager de trier un fichier sur la clé primaire et de créer un index non-dense, puis d'ajouter des index dense pour les attributs qui servent fréquemment de critère de recherche. On parle alors parfois d'*index primaire* et d'*index secondaire*, bien que ces termes soient moins précis.

Il est possible en fait de créer autant d'index denses que l'on veut puisqu'ils sont indépendants du fichier de données. Cette remarque n'est plus vraie dans le cas d'un index non-dense puisqu'il s'appuie sur le tri du fichier et qu'un fichier ne peut être trié que d'une seule manière.

La recherche par clé ou par préfixe avec un index dense est similaire à celle déjà présentée pour un index non-dense. Si la clé n'est pas unique (cas des années de parution des films), il faut prendre garde à lire dans l'index *toutes* les entrées correspondant au critère de recherche. Par exemple, pour rechercher tous les films parus en 1992 dans l'index de la figure 2.2, on trouve d'abord la première occurrence de 1992, pointant sur *Jurassic Park*, puis on lit en séquence les entrées suivantes dans l'index pour accéder successivement à *Impitoyable* puis *Reservoir Dogs*. La recherche s'arrête quand on trouve l'entrée 1995 : l'index étant trié, aucun film paru en 1992 ne peut être trouvé en continuant.

Notez que rien ne garantit que les films parus en 1992 sont situés dans le même bloc : on dit que l'index est *non-plaçant*. Cette remarque a surtout un impact sur les recherches par intervalle, comme le montre l'exemple suivant.

Exemple 10. Voici l'algorithme qui recherche tous les films parus dans l'intervalle [1950, 1979].

1. on recherche dans l'index la première valeur comprise dans l'intervalle : pour l'index de la figure 2.2 c'est 1958 ;
2. on accède au bloc du fichier de données pour y prendre l'enregistrement *Vertigo* : notez que cet enregistrement est placé dans le dernier bloc du fichier ;

3. on parcourt la suite de l'index, en accédant à chaque fois à l'enregistrement correspondant dans le fichier de données, jusqu'à trouver une année supérieure à 1979 : on trouve successivement *Psychose* (troisième bloc), *Easy Rider*, *Annie Hall* (premier bloc) et *Manhattan* (deuxième bloc). □

Pour trouver 5 enregistrements, on a dû accéder aux quatre blocs. Le coût d'une recherche par intervalle est, dans le pire des cas, égale à r , où r désigne le nombre d'enregistrements du résultat (soit une lecture de bloc par enregistrement). Il est intéressant de le comparer avec le coût $\frac{r}{b}$ d'une recherche par intervalle avec un index non-dense : on a perdu le facteur de blocage obtenu par un regroupement des enregistrements dans un bloc. Retenons également que la recherche dans l'index peut être estimée comme tout à fait négligeable comparée aux nombreux accès aléatoires au fichier de données pour lire les enregistrements.

2.2.3 Index multi-niveaux

Il peut arriver que la taille du fichier d'index devienne elle-même si grande que les recherches dans l'index en soit pénalisées. La solution naturelle est alors d'indexer le fichier d'index lui-même. Rappelons qu'un index est un fichier constitué d'enregistrements $[clé, adr]$, trié sur la clé : ce tri nous permet d'utiliser, dès le deuxième niveau d'indexation, un index non-dense.

Reprenons l'exemple de l'indexation des films sur l'année de parution. Nous avons vu que la taille du fichier était seulement dix fois moindre que celle du fichier de données. Même s'il est possible d'effectuer une recherche par dichotomie, cette taille est pénalisante pour les opérations de recherche.

On peut alors créer un deuxième niveau d'index, comme illustré sur la figure 2.3. On a supposé, pour la clarté de l'illustration, qu'un bloc de l'index de premier niveau ne contient que quatre entrées $[date, adr]$. Il faut donc quatre blocs (marqués par des traits gras) pour cet index.

L'index de second niveau est construit sur les valeurs de clés des premiers enregistrements des quatre blocs. Il suffit donc d'un bloc pour ce second niveau. S'il y avait deux blocs (par exemple parce que les blocs ne sont pas complètement pleins) on pourrait envisager de créer un troisième niveau, avec un seul bloc contenant deux entrées pointant vers les deux blocs du second niveau, etc.

Tout l'intérêt d'un index multi-niveaux est de pouvoir passer, dès le second niveau, d'une structure dense à une structure non-dense. Si ce n'était pas le cas on n'y gagnerait rien puisque tous les niveaux auraient la même taille que le premier.

Une recherche, par clé ou par intervalle, part toujours du niveau le plus élevé, et reproduit d'un niveau à l'autre les procédures de recherches présentées précédemment. Pour une recherche par clé, le coût est égal au nombre de niveaux de l'arbre.

Exemple 11. On recherche le ou les films parus en 1990. Partant du second niveau d'index, on sélectionne la troisième entrée correspondant à la clé 1984. L'entrée suivante a en effet pour valeur 1992, et ne pointe donc que sur des films antérieurs à cette date.

La troisième entrée mène au troisième bloc de l'index de premier niveau. On y trouve une entrée avec la valeur 1990 (il pourrait y en avoir plusieurs). Il reste à accéder à l'enregistrement. □

Les index multi-niveaux sont très efficaces en recherche, et ce même pour des jeux de données de très grande taille. Le problème est, comme toujours, la difficulté de maintenir des fichiers triés sans dégradation des performances. L'arbre-B, étudié dans la section qui suit, représente l'aboutissement des idées présentées jusqu'ici, puisqu'à des performances équivalentes à celles des index en recherche, il ajoute des algorithmes de réorganisation dynamique qui garantissent que la structure reste valide quelles que soient les séquences d'insertion et suppression subies par les données.

2.3 L'arbre-B

L'arbre-B est une structure d'index qui offre un excellent compromis pour les opérations de recherche par clé et par intervalle, ainsi que pour les mises à jour. C'est une structure arborescente qui a les propriétés suivantes :

- l'arbre est *équilibré*, tous les chemins de la racine vers les feuilles ont la même longueur ;

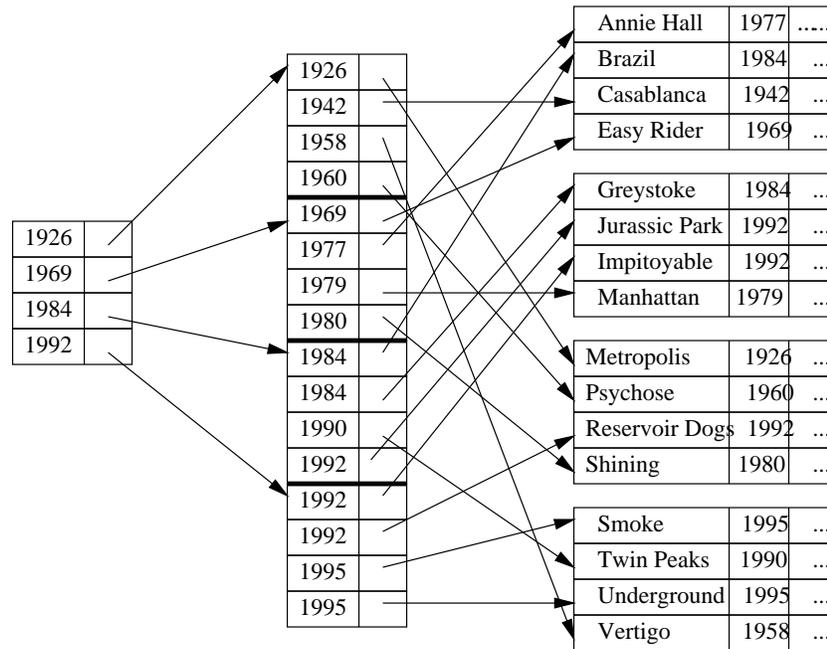


FIGURE 2.3 – Index multi-niveaux

- chaque nœud (sauf la racine) est un bloc occupé au moins à 50 % par des entrées de l'index ;
- une recherche s'effectue par une simple traversée en profondeur de l'arbre, de la racine vers les feuilles.

Ces qualités expliquent qu'il soit systématiquement utilisé par tous les SGBD, notamment pour indexer la clé primaire des tables relationnelles. Le fait que les blocs soient pleins au moins à 50 % (en pratique on constate un remplissage de 70 % en moyenne) garantit notamment une utilisation satisfaisante de l'espace alloué au fichier d'index. On parle d'un arbre-B d'ordre n pour indiquer que n est le nombre minimal d'entrée dans un bloc. Le nombre maximal d'entrée d'un arbre d'ordre n est donc $2n$.

Dans ce qui suit nous supposons que le fichier des données stocke séquentiellement les enregistrements dans l'ordre de leur création et donc indépendamment de tout ordre lexicographique ou numérique sur l'un des attributs. Notre présentation est essentiellement consacrée à la variante de l'arbre-B dite « l'arbre-B+ ».

2.3.1 Présentation intuitive

La figure 2.4 montre un arbre-B+ indexant notre collection de 16 films. L'index est organisé en blocs de taille égale, ce qui ajoute une souplesse supplémentaire à l'organisation en niveaux étudiées précédemment. En pratique un bloc peut contenir un assez grand nombre de titres de films, mais pour la clarté de l'illustration nous supposons que l'on peut stocker au plus 4 titres dans un bloc. Notons qu'au sein d'un même bloc, les titres sont triés par ordre lexicographique.

Les blocs sont chaînés entre eux de manière à créer une structure arborescente qui, dans notre exemple, comprend deux niveaux. Le premier niveau consiste en un seul bloc, la racine de l'arbre. Il contient 3 titres et 4 chaînages vers 4 blocs du second niveau. L'arbre est construit de telle manière que les titres des films dans ces 4 blocs sont organisés de la manière suivante.

1. dans le bloc situé à gauche de *Easy Rider*, on ne trouve que des titres inférieurs ou égaux, selon l'ordre lexicographique, à *Easy Rider* ;
2. dans le bloc situé entre *Easy Rider* et *Manhattan*, on ne trouve que des titres strictement supérieurs à *Easy Rider* et inférieurs ou égaux à *Manhattan* ;

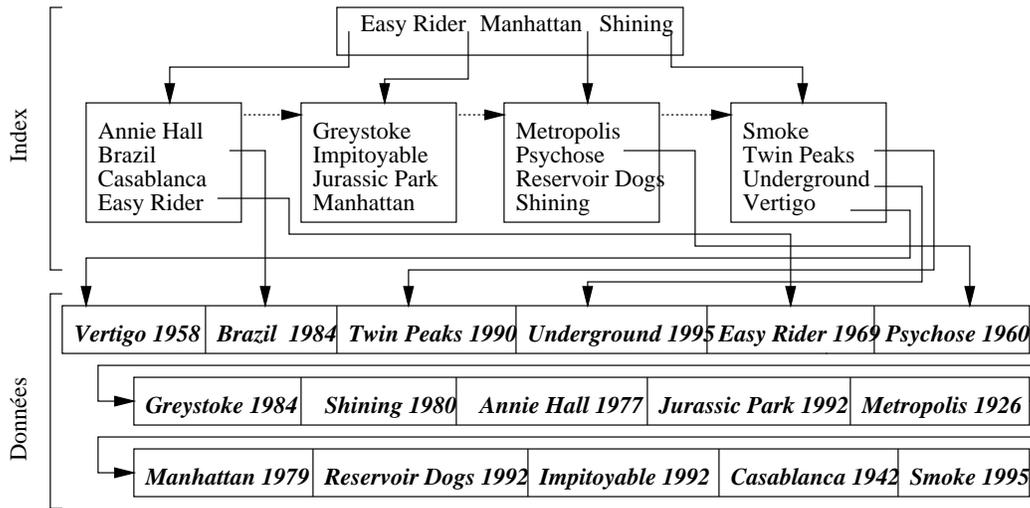


FIGURE 2.4 – Le fichier des films, avec un index unique sur le titre.

3. et ainsi de suite : le dernier bloc contient des titres strictement supérieurs à *Shining*.

Le dernier niveau (le second dans notre exemple) est celui des feuilles de l'arbre. Il constitue un index *dense* alors que les niveaux supérieurs sont non-denses. À ce niveau on associe à chaque titre l'adresse du film dans le fichier des données. Étant donné cette adresse, on peut accéder directement au film sans avoir besoin d'effectuer un parcours séquentiel sur le fichier de données. Dans la figure 2.4, nous ne montrons que quelques-uns de ces chaînages (*index, données*).

Il existe un deuxième chaînage dans un arbre-B+ : les feuilles sont liées entre elles en suivant l'ordre lexicographique des valeurs qu'elles contiennent. Ce second chaînage – représenté en pointillés dans la figure 2.4 – permet de répondre aux recherches par intervalle.

L'attribut *titre* est la clé unique de *Film*. Il n'y a donc qu'une seule adresse associée à chaque film. On peut créer d'autre index sur le même fichier de données. Si ces autres index ne sont pas construits sur des attributs formant une clé unique, on aura plusieurs adresses associés à une même valeur.

La figure 2.5 montre un index construit sur l'année de parution des films. Cet index est naturellement non-unique puisque plusieurs films paraissent la même année. On constate par exemple que la valeur 1995 est associée à deux films, *Smoke* et *Underground*. Ce deuxième index permet d'optimiser des requêtes utilisant l'année comme critère de recherche.

Quand un arbre-B+ est créé sur une table, soit directement par la commande `CREATE INDEX`, soit indirectement avec l'option `PRIMARY KEY`, un SGBD relationnel effectue automatiquement les opérations nécessaires au maintien de la structure : insertions, destructions, mises à jour. Quand on insère un film, il y a donc également insertion d'une nouvelle valeur dans l'index des titres et dans l'index des années. Ces opérations peuvent être assez coûteuses, et la création d'un index, si elle optimise des opérations de recherche, est en contrepartie pénalisante pour les mises à jour.

2.3.2 Recherches avec un arbre-B+

L'arbre-B+ supporte des opérations de recherche par clé, par préfixe de la clé et par intervalle.

Recherche par clé

Prenons l'exemple suivant :

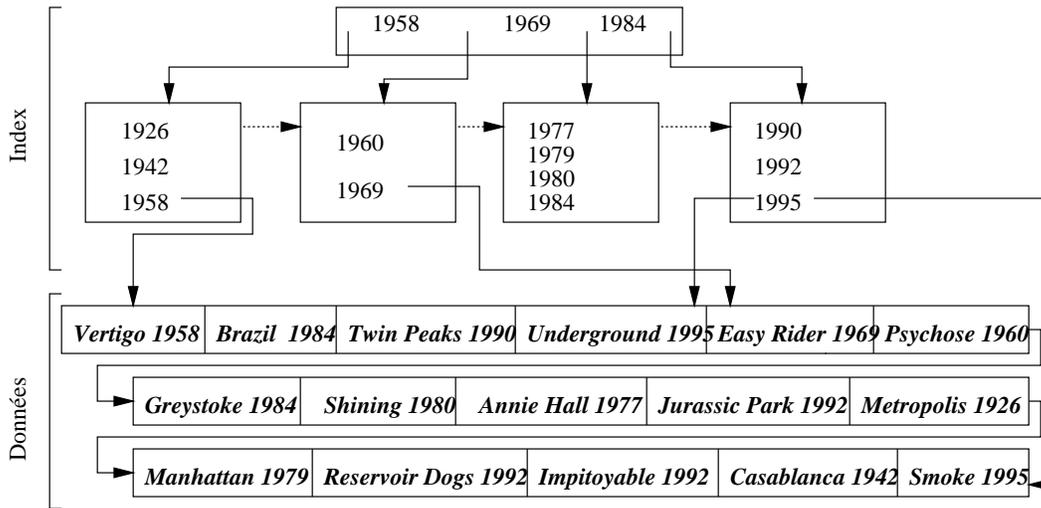


FIGURE 2.5 – Le fichier des films, avec un index unique.

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

En l'absence d'index, la seule solution est de parcourir le fichier. Dans l'exemple de la figure 2.4, cela implique de lire inutilement 13 films avant de trouver *Impitoyable* qui est en quatorzième position. L'index permet de trouver l'enregistrement beaucoup plus rapidement.

- on lit la racine de l'arbre : *Impitoyable* étant situé dans l'ordre lexicographique entre *Easy Rider* et *Manhattan*, on doit suivre le chaînage situé entre ces deux titres ;
- on lit le bloc feuille dans lequel on trouve le titre *Impitoyable* associé à l'adresse de l'enregistrement dans le fichier des données ;
- il reste à lire l'enregistrement.

Donc trois lectures sont suffisantes. Plus généralement, le nombre d'accès disques nécessaires pour une recherche par clé est égal au nombre de niveaux de l'arbre, plus une lecture pour accéder au fichier de données. Dans des conditions réalistes, le nombre de niveaux d'un index est très faible, même pour de grands ensembles de données. Cette propriété est illustrée par l'exemple suivant.

Exemple 12. Reprenons l'exemple 6 de notre fichier contenant un million de films. Une entrée d'index occupe 28 octets. On place donc $\lfloor \frac{4096}{28} \rfloor = 146$ entrées (au maximum) dans un bloc. Il faut $\lceil \frac{1000000}{146} \rceil = 6850$ blocs pour le premier niveau de l'arbre-B+.

Le deuxième niveau comprend 6850 entrées, une pour chaque bloc du premier niveau. Il faut donc $\lfloor \frac{6850}{146} \rfloor = 47$ blocs. Finalement, un troisième niveau, constitué d'un bloc avec 47 entrées suffit pour compléter l'arbre-B+. □

Quatre accès disques (trois pour l'index, un pour l'enregistrement) suffisent pour une recherche par clé, alors qu'il faudrait parcourir les 30 000 blocs d'un fichier en l'absence d'index.

Le gain est d'autant plus spectaculaire que le nombre d'enregistrements est élevé. Voici une petite extrapolation montrant le nombre de films indexés en fonction du nombre de niveaux dans l'arbre¹.

1. avec un niveau d'index (la racine seulement) on peut donc référencer 146 films ;
2. avec deux niveaux on indexe 146 blocs de 146 films chacun, soit $146^2 = 21\,316$ films ;

1. Pour être plus précis le calcul qui suit devrait tenir compte du fait qu'un bloc n'est pas toujours plein.

3. avec trois niveaux on indexe $146^3 = 3\,112\,136$ films ;
4. enfin avec quatre niveaux on index plus de 450 millions de films.

Il y a donc une croissance très rapide – exponentielle – du nombre de films indexés en fonction du nombre de niveaux et, réciproquement, une croissance très faible – logarithmique – du nombre de niveaux en fonction du nombre d’enregistrements. Le coût d’une recherche par clé étant proportionnel au nombre de niveaux et pas au nombre d’enregistrements, l’indexation permet d’améliorer les temps de recherche de manière vraiment considérable.

L’efficacité d’un arbre-B+ dépend entre autres de la taille de la clé : plus celle-ci est petite, et plus l’index sera petit et efficace. Si on indexait les films avec une clé numérique sur 4 octets (un entier), on pourrait référencer $\lceil \frac{4096}{4+8} \rceil = 341$ films dans un bloc, et un index avec trois niveaux permettrait d’indexer $341^3 = 39\,651\,821$ films ! Du point de vue des performances, le choix d’une chaîne de caractères assez longue comme clé des enregistrements est donc assez défavorable.

Recherche par intervalle

Un arbre-B+ permet également d’effectuer des recherches par intervalle. Le principe est simple : on effectue une recherche par clé pour la borne inférieure de l’intervalle. On obtient la feuille contenant cette borne inférieure. Il reste à parcourir les feuilles de l’arbre, grâce au chaînage des feuilles, jusqu’à ce que la borne supérieure ait été rencontrée ou dépassée. Voici une recherche par intervalle :

```
SELECT *
FROM Film
WHERE annee BETWEEN 1960 AND 1975
```

On peut utiliser l’index sur les clés pour répondre à cette requête. Tout d’abord on fait une recherche par clé pour l’année 1960. On accède alors à la seconde feuille (voir figure 2.5) dans laquelle on trouve la valeur 1960 associée à l’adresse du film correspondant (*Psychose*) dans le fichier des données.

On parcourt ensuite les feuilles en suivant le chaînage indiqué en pointillés dans la figure 2.5. On accède ainsi successivement aux valeurs 1969, 1977 (dans la troisième feuille) puis 1979. Arrivé à ce point, on sait que toutes les valeurs suivantes seront supérieures à 1979 et qu’il n’existe donc pas de film paru en 1975 dans la base de données. Toutes les adresses des films constituant le résultat de la requête ont été récupérées : il reste à lire les enregistrements dans le fichier des données.

C’est ici que les choses se gâtent : jusqu’à présent chaque lecture d’un bloc de l’index ramenait un ensemble d’entrées pertinentes pour la recherche. Autrement dit on bénéficiait du « bon » regroupement des entrées : les clés de valeurs proches – donc susceptibles d’être recherchées ensembles – sont proches dans la structure. Dès qu’on accède au fichier de données ce n’est plus vrai puisque ce fichier n’est pas organisé de manière à regrouper les enregistrements ayant des valeurs de clé proches.

Dans le pire des cas, comme nous l’avons souligné déjà pour les index simples, il peut y avoir une lecture de bloc pour chaque lecture d’un enregistrement. L’accès aux données est alors de loin la partie la plus pénalisante de la recherche par intervalle, tandis que le parcours de l’arbre-B+ peut être considéré comme négligeable.

Recherche avec un préfixe de la clé

Enfin l’arbre-B+ est utile pour une recherche avec un préfixe de la clé : il s’agit en fait d’une variante des recherches par intervalle. Prenons l’exemple suivant :

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

On veut donc tous les films dont le titre commence par 'M'. Cela revient à faire une recherche par intervalle sur toutes les valeurs comprises, selon l’ordre lexicographique, entre le 'M' (compris) et le 'N' (exclus). Avec l’index, l’opération consiste à effectuer une recherche par clé avec la lettre 'M', qui mène à

la seconde feuille (figure 2.4) dans laquelle on trouve le film *Manhattan*. En suivant le chaînage des feuilles on trouve le film *Metropolis*, puis *Psychose* qui indique que la recherche est terminée.

Le principe est généralisable à toute recherche qui peut s'appuyer sur la relation d'ordre qui est à la base de la construction d'un arbre-B+. En revanche une recherche sur un suffixe de la clé (« tous les films terminant par 'S' ») ou en appliquant une fonction ne pourra pas tirer parti de l'index et sera exécutée par un parcours séquentiel. C'est le cas par exemple de la requête suivante :

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

Ici on cherche tous les films dont le titre se finit par 'e'. Ce critère n'est pas compatible avec la relation d'ordre qui est à la base de la construction de l'arbre, et donc des recherches qu'il supporte.

Le temps d'exécution d'une requête avec index peut s'avérer sans commune mesure avec celui d'une recherche sans index, et il est donc très important d'être conscient des situations où le SGBD pourra effectuer une recherche par l'index. Quand il y a un doute, on peut demander des informations sur la manière dont la requête est exécutée (le « plan d'exécution ») avec les outils de type « EXPLAIN ».

2.4 Hachage

Les tables de hachage sont des structures très couramment utilisées en mémoire centrale pour organiser des ensembles et fournir un accès performant à ses éléments. Nous commençons par rappeler les principes du hachage avant d'étudier les spécificités apportées par le stockage en mémoire secondaire.

2.4.1 Principes de base

L'idée de base du hachage est d'organiser un ensemble d'éléments d'après une clé, et d'utiliser une fonction (dite *de hachage*) qui, pour chaque valeur de clé c , donne l'adresse $f(c)$ d'un espace de stockage où l'élément doit être placé. En mémoire principale cet espace de stockage est en général une liste chaînée, et en mémoire secondaire un ou plusieurs blocs sur le disque.

Exemple d'une table de hachage

Prenons l'exemple de notre ensemble de films, et organisons-le avec une table de hachage sur le titre. On va supposer que chaque bloc contient au plus quatre films, et que l'ensemble des 16 films occupe donc au moins 4 blocs. Pour garder une marge de manœuvre on va affecter 5 blocs à la collection de films, et on numérote ces blocs de 0 à 4.

Maintenant il faut définir la règle qui permet d'affecter un film à l'un des blocs. Cette règle prend la forme d'une fonction qui, appliquée à un titre, va donner en sortie un numéro de bloc. Cette fonction doit satisfaire les deux critères suivants :

1. le résultat de la fonction, pour n'importe quelle chaîne de caractères, doit être un adresse de bloc, soit pour notre exemple un entier compris entre 0 et 4 ;
2. la distribution des résultats de la fonction doit être uniforme sur l'intervalle $[0, 4]$; en d'autres termes les probabilités un des 5 chiffres pour une chaîne de caractère quelconque doivent être égales.

Si le premier critère est relativement facile à satisfaire, le second soulève quelques problèmes car l'ensemble des chaînes de caractères auxquelles on applique une fonction de hachage possède souvent des propriétés statistiques spécifiques. Dans notre exemple, l'ensemble des titres de film commencera souvent par « Le » ou « La » ce qui risque de perturber la bonne distribution du résultat si on ne tient pas compte de ce facteur.

Nous allons utiliser un principe simple pour notre exemple, en considérant la première lettre du titre, et en lui affectant son rang dans l'alphabet. Donc a vaudra 1, b vaudra 2, h vaudra 8, etc. Ensuite, pour se

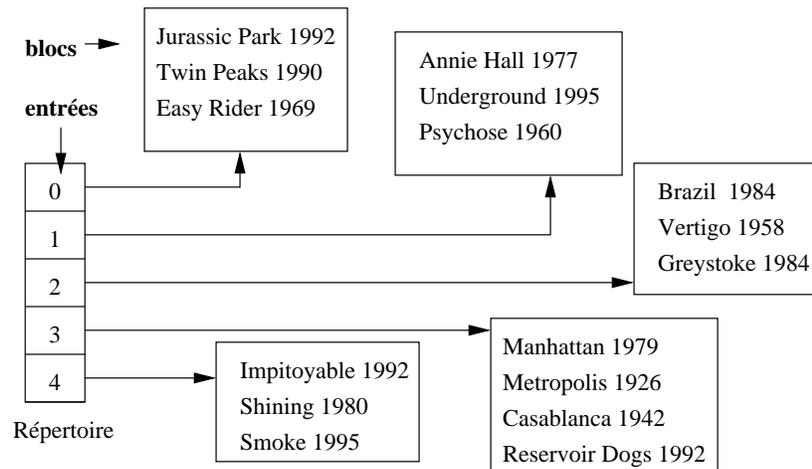


FIGURE 2.6 – Exemple d’une table de hachage

ramener à une valeur entre 0 et 4, on prendra simplement le reste de la division du rang de la lettre par 5 (« modulo 5 »). En résumé la fonction h est définie par :

$$h(\text{titre}) = \text{rang}(\text{titre}[0]) \bmod 5$$

La figure 2.6 montre la table de hachage obtenue avec cette fonction. Tous les films commençant par a , f , k , p , u et z sont affectés au bloc 1 ce qui donne, pour notre ensemble de films, *Annie Hall*, *Psychose* et *Underground*. les lettres b , g , l , q et v sont affectées au bloc 2 et ainsi de suite. Notez que la lettre e a pour rang 5 et se trouve donc affectée au bloc 0.

La figure 2.6 présente, outre les cinq blocs stockant des films, un répertoire à cinq *entrées* permettant d’associer une valeur entre 0 et 4 à l’adresse d’un bloc sur le disque. Ce répertoire fournit une indirection entre l’identification « logique » du bloc et son emplacement physique, selon un mécanisme déjà rencontré dans la partie du chapitre 1 consacrée aux techniques d’adressage de blocs (voir section 1.2.2, page 19). On peut raisonnablement supposer que sa taille est faible et qu’il peut donc résider en mémoire principale.

On est assuré avec cette fonction d’obtenir toujours un chiffre entre 0 et 4, mais en revanche la distribution risque de ne pas être uniforme : si, comme on peut s’y attendre, beaucoup de titres commencent par la lettre l , le bloc 2 risque d’être surchargé. et l’espace initialement prévu s’avèrera insuffisant. En pratique, on utilise un calcul beaucoup moins sensible à ce genre de biais : on prend par exemple les 4 ou 8 premiers caractères de la chaînes, on traite ces caractères comme des entiers dont on effectue la somme, et on définit la fonction sur le résultat de cette somme.

Recherche dans une table de hachage

La structure de hachage permet les recherches par titre, ou par le début d’un titre. Reprenons notre exemple favori :

```

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

Pour évaluer cette requête, il suffit d’appliquer la fonction de hachage à la première lettre du titre, i , qui a pour rang 9. Le reste de la division de 9 par 5 est 4, et on peut donc charger le bloc 4 et y trouver le film *Impitoyable*. En supposant que le répertoire tient en mémoire principale, on a donc pu effectuer cette recherche en lisant un seul bloc, ce qui est optimal. cet exemple résume les deux avantages principaux d’une table de hachage :

1. La structure n'occupe aucun espace disque, contrairement à l'arbre-B ;
2. elle permet d'effectuer les recherches par clé par accès direct (calculé) au bloc susceptible de contenir les enregistrements.

Le hachage supporte également toute recherche basée sur la clé de hachage, et telle que le critère de recherche fourni puisse servir d'argument à la fonction de hachage. La requête suivante par exemple pourra être évaluée par accès direct avec notre fonction basée sur la première lettre du titre.

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

En revanche, si on a utilisé une fonction plus sophistiquée basée sur tous les caractères d'une chaîne (voir ci-dessus), la recherche par préfixe n'est plus possible.

La hachage ne permet pas d'optimiser les recherche par intervalle, puisque l'organisation des enregistrement ne s'appuie pas sur l'ordre des clés. La requête suivante par exemple entraîne l'accès à tous les blocs de la structure, même si trois films seulement sont concernés.

```
SELECT *
FROM Film
WHERE titre BETWEEN 'Annie Hall' AND 'Easy Rider'
```

Cette incapacité à effectuer efficacement des recherches par intervalle doit mener à préférer l'arbre-B dans tous les cas où ce type de recherche est envisageable. Si la clé est par exemple une date, il est probable que des recherche seront effectuées sur un intervalle de temps, et l'utilisation du hachage peut s'avérer un mauvais choix. Mais dans le cas, fréquent, où on utilise une clé « abstraite » pour identifier les enregistrements pas un numéro séquentiel indépendant de leurs attributs, le hachage est tout à fait approprié car une recherche par intervalle ne présente alors pas de sens et tous les accès se feront par la clé.

Mises à jour

Si le hachage peut offrir des performances sans équivalent pour les recherches par clé, il est – du moins dans la version simple que nous présentons pour l'instant – mal adapté aux mises à jour. Prenons tout d'abord le cas des insertions : comme on a évalué au départ la taille de la table pour déterminer le nombre de blocs nécessaire, cet espace initial risque de ne plus être suffisant quand des insertions conduisent à dépasser la taille estimée initialement. La seule solution est alors de chaîner de nouveaux blocs.

Cette situation est illustrée dans la figure 2.7. On a inséré un nouveau film, *Citizen Kane*. La valeur donnée par la fonction de hachage est 3, rang de la lettre *c* dans l'alphabet, mais le bloc 3 est déjà plein.

Il est impératif pourtant de stocker le film dans l'espace associé à la valeur 3 car c'est là que les recherches iront s'effectuer. On doit alors chaîner un nouveau bloc au bloc 3 et y stocker le nouveau film. À une entrée dans la table, correspondant à l'adresse logique 3, sont associés maintenant deux blocs physiques, avec une dégradation potentielle des performances puisqu'il faudra, lors d'une recherche, suivre le chaînage et inspecter tous les enregistrements pour lesquels la fonction de hachage donne la valeur 3.

Dans le pire des cas, une fonction de hachage mal conçue affecte tous les enregistrements à la même adresse, et la structure dégénère vers un simple fichier séquentiel. Ce peut être le cas, avec notre fonction basée sur la première lettre du titre, pour tous les films dont le titre commence par *l*. Autrement dit, ce type de hachage n'est pas *dynamique* et ne permet pas, d'une part d'évoluer parallèlement à la croissance ou décroissance des données, d'autre part de s'adapter aux déviations statistiques par rapport à la normale.

- En résumé, les avantages et inconvénients du hachage statique, comparé à l'arbre-B, sont les suivantes :
- **Avantages** : (1) recherche par accès direct, en temps constant ; (2) n'occupe pas d'espace disque.
 - **Inconvénients** : (1) pas de recherche par intervalle ; (2) pas de dynamique.

Il n'est pas inutile de rappeler qu'en pratique la hauteur d'un arbre-B est de l'ordre de 2 ou 3 niveaux, ce qui relativise l'avantage du hachage. Une recherche avec le hachage demande une lecture, et 2 ou 3 avec l'arbre B, ce qui n'est pas vraiment significatif. Cette considération explique que l'arbre-B, plus généraliste et presque aussi efficace, soit employé par défaut pour l'indexation dans tous les SGBD relationnels.

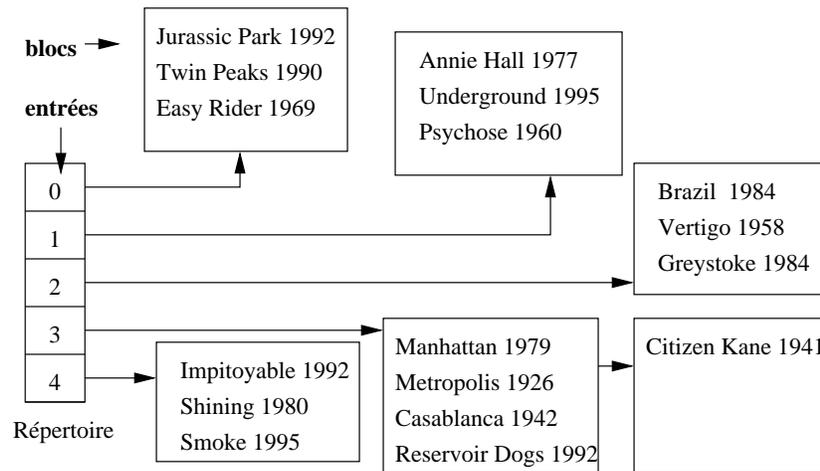


FIGURE 2.7 – Table de hachage avec page de débordement

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

TABLE 2.2 – Valeurs du hachage extensible pour les titres

Enfin signalons que le hachage est une structure *plaçante*, et qu'on ne peut donc créer qu'une seule table de hachage pour un ensemble de données, les autres index étant obligatoirement des arbres B+.

Nous présentons dans ce qui suit des techniques plus avancées de hachage dit *dynamique* qui permettent d'obtenir une structure plus évolutive. La caractéristique commune de ces méthodes est d'adapter le nombre d'entrées dans la table de hachage de manière à ce que le nombre de blocs corresponde approximativement à la taille nécessaire pour stocker l'ensemble des enregistrements. On doit se retrouver alors dans une situation où il n'y a qu'un bloc par entrée en moyenne, ce qui garantit qu'on peut toujours accéder aux enregistrements avec une seule lecture.

2.4.2 Hachage extensible

Nous présentons tout d'abord le hachage extensible sur un exemple avant d'en donner une description plus générale. Pour l'instant la structure est tout à fait identique à celle que nous avons vue précédemment, à ceci près que le nombre d'entrées dans le répertoire est variable, et toujours égal à une puissance de 2. Nous débutons avec le cas minimal où ce nombre d'entrées est égal à 2, avec pour valeurs respectives 0 et 1.

Maintenant nous supposons donnée une fonction de hachage h dont le résultat est toujours un entier sur 4 octets, soit 32 bits. La table 2.2 donne les 8 premiers bits des valeurs obtenues par application de cette fonction aux titres de nos films. Comme il n'y a que deux entrées, nous nous intéressons seulement au premier de ces 32 bits, qui peut valoir 0 ou 1. La figure 2.8 montre l'insertion des cinq premiers films de

notre liste, et leur affectation à l'un des deux blocs. Le film *Vertigo* par exemple a pour valeur de hachage 01110010 qui commence par 0, et se trouve donc affecté à la première entrée.

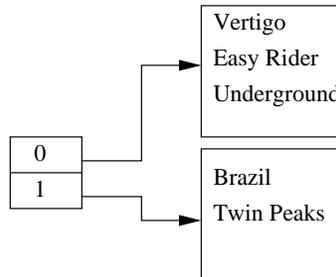


FIGURE 2.8 – Hachage extensible avec 2 entrées

Supposons, pour la clarté de l'exposé, que l'on ne puisse placer que 3 enregistrements dans un bloc. Alors l'insertion de *Psychose*, avec pour valeur de hachage 01110011, entraîne le débordement du bloc associé à l'entrée 0.

On va alors doubler la taille du répertoire pour la faire passer à quatre entrées, avec pour valeurs respectives 00, 01, 10, 11, soit les 2^2 combinaisons possibles de 0 et de 1 sur deux bits. Ce doublement de taille du répertoire entraîne la réorganisation suivante (figure 2.9) :

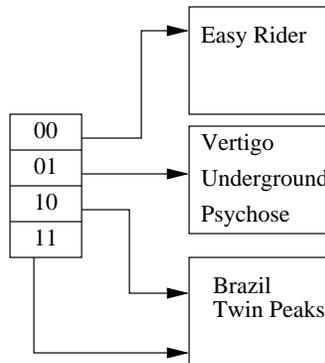


FIGURE 2.9 – Doublement du répertoire dans le hachage extensible

1. les films de l'ancienne entrée 0 sont répartis sur les entrées 00 et 01 en fonction de la valeur de leurs deux premiers bits : *Easy Rider* dont la valeur de hachage commence par 00 est placé dans le premier bloc, tandis que *Vertigo*, *Underground* et *Psychose*, dont les valeurs de hachage commencent par 01, sont placées dans le second bloc.
2. les films de l'ancienne entrée 1 n'ont pas de raison d'être répartis dans deux blocs puisqu'il n'y a pas eu de débordement pour cette valeur : *on va donc associer le même bloc aux deux entrées 10 et 11*.

Maintenant on insère *Greystoke* (valeur 10111001) et *Shining* (valeur 11010011). Tous deux commencent par 10 et doivent donc être placés dans le troisième bloc qui déborde alors. Ici il n'est cependant pas nécessaire de doubler le répertoire puisqu'on est dans une situation où plusieurs entrées de ce répertoire pointent sur le même bloc.

On va donc allouer un nouveau bloc à la structure, et l’associer à l’entrée 11, l’ancien bloc restant associé à la seule entrée 10. Les films sont répartis dans les deux blocs, *Brazil* et *Greystoke* avec l’entrée 10, *Twin Peaks* et *Shining* avec l’entrée 11 (figure 2.10).

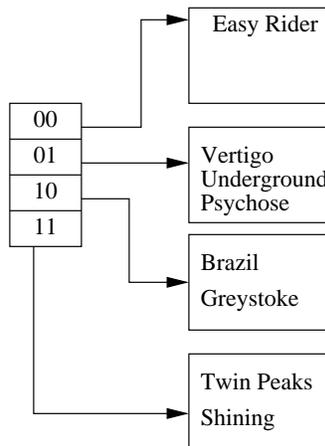


FIGURE 2.10 – Jeu de pointeurs pour éviter de doubler le répertoire

2.5 Les index *bitmap*

Un index *bitmap* repose sur un principe très différents de celui des arbres B+. Alors que dans ces derniers on trouve, pour chaque attribut indexé, les mêmes valeurs dans l’index et dans la table, un index *bitmap* considère toutes les valeurs possibles pour cet attribut, que la valeur soit présente ou non dans la table. Pour chacune de ces valeurs possibles, on stocke un tableau de bits (dit *bitmap*), avec autant de bits qu’il y a de lignes dans la table. Notons A l’attribut indexé, et v la valeur définissant le *bitmap*. Chaque bit associé à une ligne l a alors la signification suivante :

- si le bit est à 1, l’attribut A a pour valeur v dans la ligne l ;
- sinon le bit est à 0.

Quand on recherche les lignes avec la valeur v , il suffit donc de prendre le *bitmap* associé à v , de chercher tous les bits à 1, et d’accéder les enregistrements correspondant. Un index *bitmap* est très efficace si le nombre de valeurs possible pour un attribut est faible.

Exemple 13. Reprenons l’exemple de nos 16 films, et créons un index sur le genre (voir le table 2.3). L’utilisation d’un arbre-B ne donnera pas de bons résultats car l’attribut est trop peu sélectif (autrement dit une partie importante de la table peut être sélectionnée quand on effectue une recherche par valeur).

En revanche un index *bitmap* est tout à fait approprié puisque le genre fait partie d’un ensemble énuméré avec relativement peu de valeurs. Voici par exemple le *bitmap* pour les valeurs « Drame », « Science-Fiction » et « Comédie ». Chaque colonne correspond à l’un des films.

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

Pour la valeur « Drame », on place le bit à 1 pour les films de rang 4, 5, 6 et 15. Tous les autres sont à zéro. Pour « Science-Fiction » les bits à 1 sont aux rangs 2, 10 et 11. Bien entendu il ne peut y avoir qu’un seul 1 dans une colonne puisqu’un attribut ne peut prendre qu’une valeur. □

rang	titre	genre	...
1	Vertigo	Suspense	...
2	Brazil	Science-Fiction	...
3	Twin Peaks	Fantastique	...
4	Underground	Drame	...
5	Easy Rider	Drame	...
6	Psychose	Drame	...
7	Greystoke	Aventures	...
8	Shining	Fantastique	...
9	Annie Hall	Comédie	...
10	Jurassic Park	Science-Fiction	...
11	Metropolis	Science-Fiction	...
12	Manhattan	Comédie	...
13	Reservoir Dogs	Policier	...
14	Impitoyable	Western	...
15	Casablanca	Drame	...
16	Smoke	Comédie	...

TABLE 2.3 – Les films et leur genre

Un index bitmap est très petite taille comparé à un arbre-B construit sur le même attribut. Il est donc très utile dans des applications de type « Entrepôt de données » gérant de gros volumes, et classant les informations par des attributs catégoriels définis sur de petits domaines de valeur (comme, dans notre exemple, le genre d'un film). Certains requêtes peuvent alors être exécutées très efficacement, parfois sans même recourir à la table. Prenons l'exemple suivant : « Combien y a-t-il de film dont le genre est *Drame* ou *Comédie* ? ».

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

Pour répondre à cette requête, il suffit de compter le nombre de 1 dans les *bitmap* associés à ces deux valeurs !

2.6 Indexation de données géométriques

In order to efficiently process spatial queries, one needs specific access methods relying on a data structure called *index*. These access methods accelerate the access to data, i.e., they reduce the set of objects to be looked at when processing a query. Numerous examples of queries selecting or joining objects were given in Chapter 3. If the selection or join criteria are based on descriptive attributes, a classical index such as the B-tree can be used. However, we shall see that different access methods are needed for spatial queries, in which objects are selected according to their location in space.

Point and *window queries* are examples of spatial queries. In such queries, one looks for objects whose geometry contains a point or overlaps a rectangle. *Spatial join* is another important example. Given two sets of objects, one wants to keep pairs of objects satisfying some spatial relationships. Intersection, adjacency, containment, North-West are examples of spatial predicates that a pair of objects to be joined must satisfy.

Why are spatial access methods needed ?

Processing a spatial query leads to the execution of complex and costly geometric operations. For common operations like point queries, sequentially scanning and checking whether each object of a large collection contains a point involves a large number of disk accesses and the repeated expensive evaluation of

geometric predicates. Hence both the time-consuming geometric algorithms and the large volume of spatial collections stored on secondary storage motivate the design of efficient spatial access methods (SAM) which reduce the set of objects to be processed. With a SAM one expects a time which is logarithmic in the collection size or even smaller. In most of the cases, the SAM uses an explicit structure, called a *spatial index*. The collection of objects for which an index is built is said to be *indexed*.

What is indexed ?

We restrict our attention to objects whose spatial component is defined in the two-dimensional plane. We shall show some SAM for points but we focus on the indexation of lines and polygons. In this case, instead of indexing the object geometries themselves — whose shape might be complex — one usually indexes a simple approximation of the geometry. The most commonly used approximation is the minimum bounding rectangle of the objects' geometry or *minimal bounding box*, denoted *mbb* is the following. By using the *mbb* as the geometric key for constructing spatial indices, one saves the cost of evaluating expensive geometric predicates during index traversal (a geometric test against an *mbb* can be processed in constant time). A second important motivation is to use constant-size entries, a feature that simplifies the design of spatial structures.

Filter and refinement steps

A spatial index is built upon a collection of *entries*, i.e., pairs $[mbb, oid]$ where *oid* identifies the object whose minimal bounding box is *mbb*. We assume that *oid* allows one to access directly the page that contains the physical representation of the object, i.e., the values of the descriptive attributes and the value of the spatial component.

An operation involving a spatial predicate on a collection of objects indexed on their *mbb* is performed in two steps. The first step, called *filter step*, selects the objects whose *mbb* satisfies the spatial predicate. This step consists in traversing the index, applying the spatial test on the *mbb*. The output is a set of *oids*.

An *mbb* might satisfy a query predicate while the exact geometry does not. Then the objects that pass the filter step are a superset of the solution. In a second step, called *refinement step*, this superset is sequentially scanned and the spatial test is done on the actual geometries of objects whose *mbb* satisfied the filter step. This geometric operation is costly but executed only on a limited number of objects. The objects that do not pass the filter step are called *false drops*. In this chapter, we shall not look at the refinement step which is the same whatever the SAM is. We concentrate here on the design of efficient SAM for the filter step.

There is no spatial total order

In off-the-shelf database management systems, the most common access methods use B-trees and hash tables, which guarantees that the number of I/Os (Input/Output operations) to access data is respectively logarithmic and constant in the collection size, for exact search queries. A typical example of exact search is "Find County (whose name=) San Francisco". Unfortunately, these access methods cannot be used in the context of spatial data. As an example, take the B-tree. This structure indexes a collection on a *key*, i.e., an attribute value of every object of the collection. The B-tree relies on a total order on the key domain, usually the order on natural numbers, or the lexicographic order on strings of characters. Thanks to this order, interval queries are efficiently answered.

A convenient order for geometric objects in the plane is one which preserves objects proximity. Two objects *close* in the plane should be close in the index structure. Indeed take the example of objects inside a rectangle. They are close in the plane. It is desirable that their representation in the index be also close so that a window query be efficiently answered. Unfortunately, there is no such a total order. The same limitation hold for hashing. Therefore a large number of SAMs were designed which try as much as possible to preserve objects proximity. The proposals are based on some heuristics, such as rectangle inclusion.

The SAMs of the R-tree family belong to the category of data-driven access methods. Their structure adapts itself to the rectangles distribution in the plane. Similarly to the B-tree, they rely on a balanced hierarchical structure, in which each node, whether internal or leaf, is mapped onto a disk page. However, whereas B-trees are built on single value keys and rely on a total order on these keys, R-trees organize

rectangles according to a containment relationship. With each node is associated a rectangle, denoted the *directory rectangle* (dr) in the following, which is the minimal bounding box of the rectangles of its child nodes.

To access one rectangle of the indexed collection, one typically follows a path from the R-tree root down to one or several leaves, testing each directory rectangle at each level for either containment or overlapping. Since, as for the B-tree, the depth d of a R-tree is logarithmic in the number of objects indexed, and each node is mapped onto a disk page, the number of I/Os in non-degenerated situations is logarithmic in the collection size and therefore the SAM is quite time and space efficient.

We first present the R-tree structure as originally proposed. Because in some situations the search time degrades, several variants have been proposed. Two of them, called R*tree and R+tree, are described in subsequent sections.

An R-tree is a depth balanced tree in which each node corresponds to a disk page. A *leaf node* contains an array of *leaf entries*. A leaf entry is a pair (mbb, oid) with the usual meaning. mbb is of the form $(x_1, y_1, \dots, x_d, y_d)$, where d is the search space dimension. Although the R-tree structure can handle data with arbitrary dimension, we shall limit the presentation to the two-dimensional case. A *non-leaf node* contains an array of node entries.

The structure satisfies the following properties :

- For all nodes in the tree (except for the root) the number of entries is between m and M where $m \in [0, M/2]$.
- For each entry $(dr, nodeid)$ in a non-leaf node N , dr is the directory rectangle of a child node of N , whose page address is $nodeid$.
- For each leaf entry (mbb, oid) , mbb is the the minimal bounding box of the spatial component of the object stored at address oid .
- The root has at least 2 entries (unless it is a leaf).
- All leaves are at the same level.

Figure 2.11 shows an R-tree with $m = 2$ and $M = 4$. The indexed collection C contains 14 objects. The directory rectangles of leaves a , b , c and d are represented by a dot line.

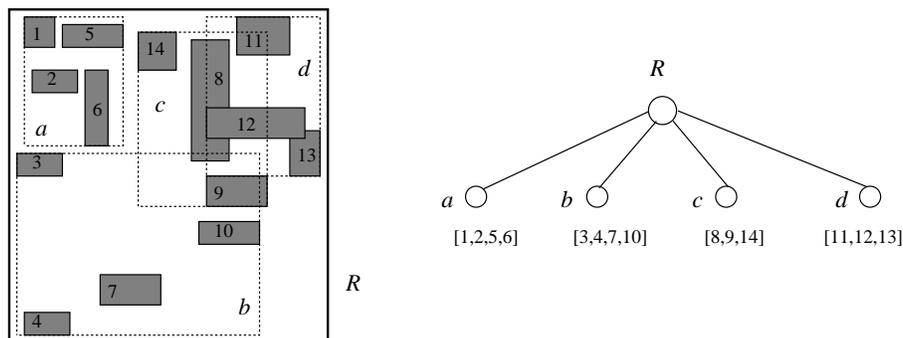


FIGURE 2.11 – An R-tree

The above properties are preserved under any dynamic insertion or deletion. Observe that, while keeping the tree balanced, the structure adapts to the skewness of a data distribution. A region of the search space populated with a large number of objects generates a large number of neighbor tree leaves. This is not the case with space-partitioning trees such as the quadtree. With the latter, some branches in the tree might be long, they correspond to regions with a high density of rectangles while others might be short, the depth of the final tree corresponding to the region with highest density.

Figure 2.12 (right-hand side) shows the directory rectangles of the leaves of an R-tree built on the hydrography dataset of the Connecticut state (left-hand side of the figure). The rectangles overlapping is obviously more important in the dense areas than in the sparse ones.

M , the maximal number of entries in a node, depends on the entry size $Size(E)$ and the disk page

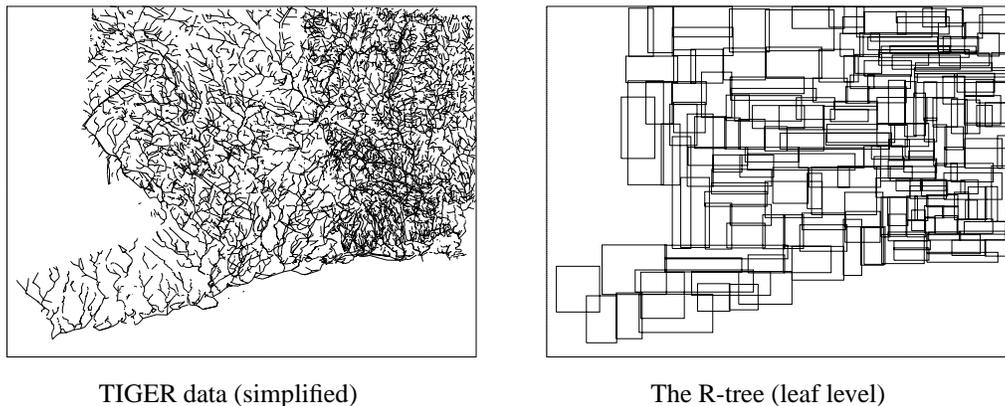


FIGURE 2.12 – Indexing the hydrography network of the Connecticut State

capacity $Size(P) : M = \lfloor Size(P)/Size(E) \rfloor$. Note that M might differ for leaf and non-leaf nodes, since the size of an entry depends on the size of $nodeid$ (non-leaf nodes) and oid (leaf nodes). oid which is an object address inside a page is generally longer than $nodeid$ which is a page address. The tuning of the minimum number of entries per node m between 0 and $M/2$ is related to the splitting strategies of nodes which will be further detailed below.

Given M and m , an R-tree of depth d indexes at least m^{d+1} objects and at most M^{d+1} objects. Conversely, the depth of an R-tree indexing a collection of N objects is at most $\lfloor \log_m(N) \rfloor - 1$ and at least $\lfloor \log_M(N) \rfloor - 1$. The exact value depends on the page utilization².

To illustrate the R-tree time efficiency, assume the page size is 4K, the entry size is 20 bytes (16 bytes for the mbb , 4 bytes for the oid) and m is set to be 40% of the page capacity. Hence $M = 204$ and $m = 81$. An R-tree of depth 1 can index at least 6 561 objects while an R-tree of depth 2 can index at least 531 441 and up to 8 489 664 objects. Then, with a collection of one million objects, it takes three page accesses to traverse the tree and a supplementary access to get the record storing the object. This example shows that even for large collections, only a few page accesses are necessary to get down the tree in order to access the objects.

Unlike the space-partitionning SAMs seen above, an object appears in one and only one of the tree leaves. However, the rectangles associated with the internal nodes do *not* constitute an exact partition of the space and hence may overlap. In Figure 2.11, rectangles b , c and d , which correspond to distinct leaves, overlap (see also Figure 2.11).

Searching with R-trees

We give a detailed algorithm for point query. The algorithm for window queries is almost identical.

The function RT-POINTQUERY is performed in two steps. First, one visits all the children of the root whose directory rectangle contains the point P . Recall that since several directory rectangles may overlap, it might happen that the point argument falls into the intersection of several rectangles. All subtrees rooted at the intersecting rectangles have then to be visited. The process is repeated at each level of the tree until the leaves are reached. For each visited node N , two situations may occur :

1. At a given node, no entry rectangle contains the point and the search terminates. This can occur even if P falls into the directory rectangle of the node. P falls into the so-called “dead space” of the node.
2. P is contained by the directory rectangle of one or several entries. One must visit each associated subtree.

2. The page utilization is defined as the ratio $(NbEntries \times Size(E))/(NbPages \times Size(P))$ between the space actually occupied by all entries (number of entries multiplied by the entry size) over the index space, i.e., the number of pages (nodes) multiplied by the page capacity.

Hence, unlike the space-partitioning SAMs (e.g., the Grid File or linear SAMs), several paths from the root to the leaves might be traversed. In a second step, the set of leaves produced in the first step are sequentially scanned : each leaf's entry whose *mbb* contains *P* is kept.

Figure 2.13 shows an example of point query with *P*, which belongs to objects 8 and 12, as an argument. The query leads to the access of 3 nodes, namely *R*, *c* and *d*. A point query on the South-East corner point of Object 8 would involve accessing 4 nodes, namely *R*, *b*, *c* and *d*.

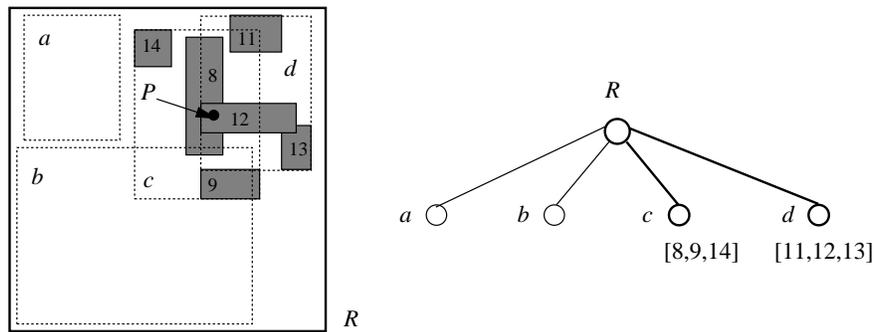


FIGURE 2.13 – Point queries with R-trees

The point query algorithm with an R-tree is given below :

Algorithm RT-POINTQUERY (*P* : Point) : set (oid)
begin
result = \emptyset
 // Step 1 : Traverse the tree from the root, and compute *SL*, the
 // set of leaves whose *dr* contains *P*
SL = RTREETRAVERSAL (*root*, *P*)
 // Step 2 : scan the leaves, keep the entries which contain *P*
for each *L* **in** *SL* **do**
 // Scan the entries in the leaf *L*
for each *e* **in** *L* **do**
if (*e.mbb* contains *P*) **then** *result* += {*e.oid*}
end for
end for
return *result*
end

The algorithm for traversing the tree is given by function RTREETRAVERSAL (see below). It implements a left and depth-first traversal of the R-tree through recursive calls. The function is initially called with the root page address as first argument.

Algorithm RTREETRAVERSAL (*nodeid* : PageID, *P* : Point) : set of leaves
begin
result = \emptyset
 // Get the node page
N = READPAGE (*nodeid*)
if (*N* is a leaf) **return** {*N*}
else
 // Scan the entries of *N*, and visit those which contain *P*

```

for each  $e$  in  $N$  do
  if ( $e.dr$  contains  $P$ ) then
     $result$  += RTREETRAVERSAL ( $e.nodeid$ ,  $P$ )
  end if
end for
end if
return  $result$ 
end

```

It should be stressed that the algorithm above is simplified from a memory management viewpoint. Keeping a set of leaves in memory would imply a large and unnecessary buffer pool. In an actual implementation, a one-page-at-a-time strategy should be used.

If the point falls into only one rectangle at each level, then the point query necessitates d page accesses where d is the depth of the tree, logarithmic in the number of indexed rectangles. Even if this seldom happens, one expects in practical situations that only a small number of paths are followed from root to leaves, and the expected number of I/Os is still logarithmic. Unfortunately there are degenerated situations where this number is not logarithmic anymore. In the worst case, it may occur that all the directory rectangles of the leaves have a common intersection into which the point argument P falls. In this case the whole tree must be scanned. However, despite its bad worst-case behavior, the R-tree is efficient in most practical situations.

The fact that the number of nodes visited is closely related to the overlapping of directory rectangles gave rise to numerous variants of the R-tree which intend to minimize this overlapping. We shall see two of them at the end of the chapter.

The window query algorithm is a straightforward generalization of point query in which the “contains P ” predicate is replaced by the “intersect W ” predicate, where W is the window argument. The larger the window, the larger the number of nodes to be visited.

Insertion and deletion in an R-tree

To insert an object, the tree is first traversed top-down, starting from the root, (function INSERT). At each level, either one finds a node whose directory rectangle contains the object’s mbb and then gets down the node subtree, or there is no such node. Then a node is chosen such that the enlargement of its dr is minimal (function CHOOSESUBTREE). We repeat the process until a leaf is reached.

If the leaf is not full, a new entry [mbb , oid] is added to the page associated with the leaf. If the leaf directory rectangle has to be enlarged, then the corresponding entry in the parent’s node must be updated with the new value of dr (function ADJUSTENTRY). Note that this might imply the parent’s directory rectangle be in turn enlarged. Thus a leaf enlargement propagates up the tree, in the worst case up to the root (function ADJUSTPATH).

If the leaf l in which the object has been inserted is full, a *split* occurs. A new leaf l' is created and the $M + 1$ entries are distributed among l and l' . How to split the leaf (function SPLIT), is the tricky part of the algorithm and will be explained later.

When the split is over, it remains to update the entry of the old leaf l in its parent node f , and to insert a new entry in f corresponding to the new leaf l' . If f itself is full because of this new insertion, the same splitting mechanism is applied to the internal node : in the worst case this splitting process propagates up the tree until the root (function SPLITANDADJUST). The tree depth is incremented by one when the root is split.

We illustrate this process by inserting two new objects in the tree of Figure 2.11. For the sake of illustration, we assume that $M = 4$.

Object 15 (Figure 2.14) has first been inserted in leaf d . The dr of d must be enlarged to enclose this new object and the d entry in the root must be adjusted.

Object 16 (Figure 2.15) is to be inserted in leaf b . Since b already contains the four entries {3, 4, 7, 10}, a node overflow occurs (recall that the capacity M is 4), b is split. A new leaf, e , is then created and the 5

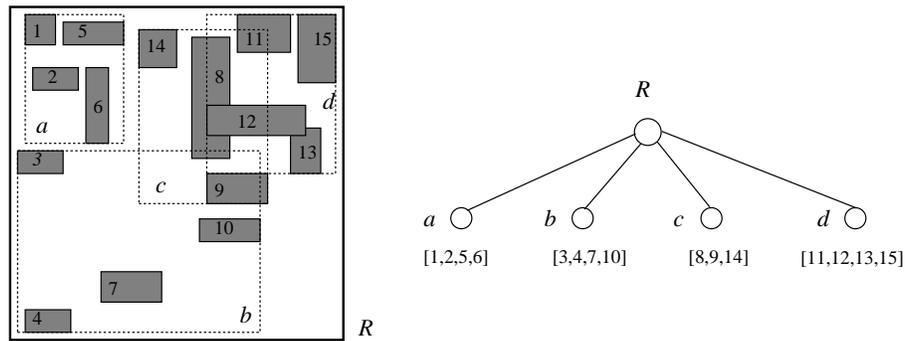


FIGURE 2.14 – Insertion of object 15

entries are distributed among b and e . Entry 10 is moved to e (where entry 16 had already been inserted). Entries 3, 4, and 7 remain in b .

Now a new entry must be inserted in the parent of b , R . Since R already contains 4 children, it must be split in turn. The process is similar to the one already described for leaf b . A new node f is created; a and b stay in R while e , c , d are attached to f . Finally, a new root R' is created, whose two children are R and f .

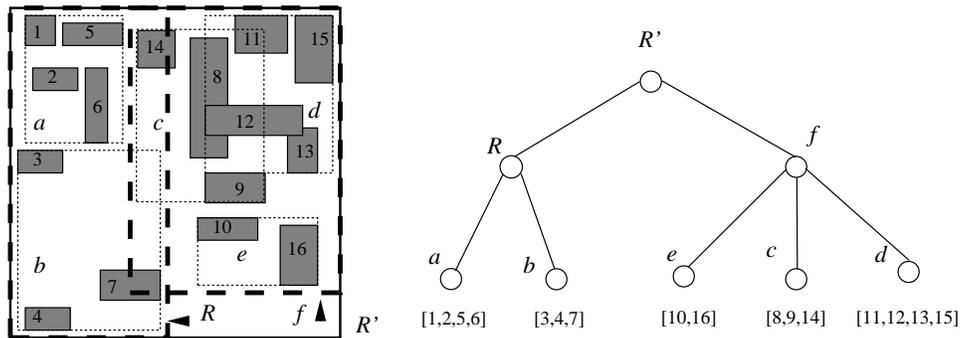


FIGURE 2.15 – Insertion of object 16

We give below the insertion algorithm.

Algorithm INSERT (e : LeafEntry)

begin

 // Initializes the search with root

$node = root$

 // Choose a path down to a leaf

while ($node$ is not a leaf) **do**

$node = \text{CHOOSESUBTREE}(node, e)$

end while

 // Insert in the leaf

 INSERTINLEAF ($node, e$)

 // Split and adjust the tree if the leaf overflows, else adjust the path

if ($node$ overflows) **then**

 SPLITANDADJUST ($node$)

```

else
  ADJUSTPATH (node)
end if
end

```

Function CHOOSESUBTREE (*node*, *e*), which is not described here, picks the entry *ne* of *node* such that *ne.dr* either contains *e.mbb* or needs the minimum enlargement to include *e.mbb*. If several entries satisfy this, choose the one with smallest area.

Function ADJUSTPATH propagates the enlargement of the node's *dr* consecutive to an insertion. The process stops when either the *dr* does not require any adjustment or the root has been reached.

Algorithm ADJUSTPATH (*node* : Node)

```

begin
  if (node is the root) return
  else
    // Find the parent of node
    parent = GETPARENT (node)
    // Adjust the entry of node in parent
    if (ADJUSTENTRY (parent, [node.mbb, node.id])) then
      // Entry has been modified : adjust the path for the parent
      ADJUSTPATH (parent)
    end if
  end if
end

```

ADJUSTENTRY (*node*, *child-entry*) is a Boolean function which looks for the entry *e* corresponding to *child-entry.id* in *node* and compares *e.mbb* with *child-entry.mbb*. If *e.mbb* needs to be updated because the *dr* of the child has been enlarged or shrunk, it is updated³, the function returns **true**, and **false** otherwise.

Function SPLITANDADJUST handles the case of nodes overflow. It performs the split and adjusts the path. It uses functions ADJUSTPATH and ADJUSTENTRY (already described) and can be recursively called in the case of propagated overflows.

Algorithm SPLITANDADJUST (*node* : Node)

```

begin
  // Create a new node and distribute the entries
  new-node = SPLIT (node)
  if (node is the root) then
    CREATENEWROOT (node, new-node)
  else
    // Get the parent of node
    parent = GETPARENT (node)
    // Adjust the entry of node in its parent
    ADJUSTENTRY (parent, [node.id, node.mbb])
    // Insert the new node in the parent.
    INSERTINNODE (parent, [new-node.mbb, new-node.id])
    if (parent overflows) then
      SPLITANDADJUST(parent)
    end if
  end if
  ADJUSTPATH (parent)
end

```

3. This possibly implies the update of *node dr* as well.

```

end if
end if
end

```

CREATE_NEW_ROOT(*node*, *new-node*) allocates a new page *p* and inserts into *p* two entries, one for *node* and one for *new-node*. Observe that the root is the only node that does not respect the minimal space utilization fixed by *m*.

Once the entries have been distributed among the two nodes, the entry of *node* must be adjusted in *parent* (call of function ADJUST_ENTRY). Then the new node is inserted into the parent node. Two cases may occur. Either the parent node overflows, then it must be split and adjusted, or it is only adjusted.

An important part of the algorithm is the *split* of a node. Recall the constraint of a minimal number of entries *m* in a node. Any solution with *m* + *i* entries in one node and *M* + 1 - *m* - *i* in the second, with $0 \leq i \leq M - 2m + 1$, is acceptable.

There are many ways of splitting a node. A well designed splitting strategy should obey the two following requirements :

- Minimize the total area of the two nodes.
- Minimize the overlapping of the two nodes.

Unfortunately these two requirements are not always compatible as illustrated in Figure 2.16. In the following we shall focus on the first criteria.

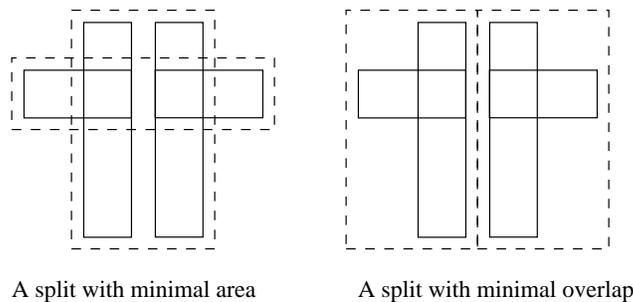


FIGURE 2.16 – Minimal area and minimal overlap

A brute force method is to exhaustively look at all possible distributions of rectangles among the two nodes and choose the one that minimizes the total area. Since *M* is of the order of 100 for common page sizes, this is clearly too expensive, the cost of this method being exponential in *M*.

Therefore a trade-off between computation time and quality of the result has to be found. We now present two algorithms along these lines.

The first one, known as quadratic split algorithm has a computation time quadratic in *M*. It relies on the following heuristic. Two groups of entries are first initialized by “picking” two “seed” entries *e* and *e'*, far away from each other, and more precisely such that the dead space is maximal. The dead space is defined as the area of the *mbb* of *e* and *e'* minus the sum of the areas of *e* and *e'*.

Then the remaining *M* - 2 entries are inserted in one of the two groups as follows. We call *expansion* of a group for an entry the dead space that would result from inserting the entry in this group. Then at each step, we look for the entry *e* for which the difference in group expansions is maximal. We choose to add *e* to the group “closest” to the entry (for which the expansion is the smallest). In the case of equality, the following secondary criterias can be used : (i) choose the group with the smallest area, (ii) choose the group with the fewest elements.

Algorithm QUADRATIC_SPLIT (*E* : set of entries)
begin

```

MBB  $J$ ; integer  $worst = 0$ ,  $d1$ ,  $d2$ ,  $expansion=0$ 
// Choose the first element of each group,
//  $E'$  is the set of remaining entries
for each  $e$  in  $E$  do
  for each  $e'$  in  $E$  do
     $J = mbb(e.mbb, e'.mbb)$ 
    if  $((area(J) - area(e.mbb) - area(e'.mbb)) > worst)$  then
       $worst = (area(J) - area(e.mbb) - area(e'.mbb))$ 
       $G_1 = \{e\}$ 
       $G_2 = \{e'\}$ 
    end if
  end for
end for
// Each group has been initialized. Now insert the remaining entries
 $E' = E - (G_1 \cup G_2)$ 
while  $(E' \neq \emptyset)$  do
  // Compute the cost of inserting each entry in  $E'$ 
  for each  $e$  in  $E'$  do
     $d1 = area(mbb(e.mbb, G_1.mbb) - e.mbb)$ 
     $d2 = area(mbb(e.mbb, G_2.mbb) - e.mbb)$ 
    if  $(d2 - d1 > expansion)$  then
       $best\text{-}group = G_1$ ;  $best\text{-}entry = e$ ,  $expansion=d2-d1$ 
    end if
    if  $(d1 - d2 > expansion)$  then
       $best\text{-}group = G_2$ ;  $best\text{-}entry = e$ ;  $expansion=d1-d2$ 
    end if
  end for
  // Insert the best candidate entry in the best group
   $best\text{-}group = best\text{-}group \cup best\text{-}entry$ 
   $E' = E' - \{best\text{-}entry\}$ 
  // Each group must contain at least  $m$  entries.
  // If the group size added to the number of remaining
  // entries is equal to  $m$ , then just add them to the group.
  if  $(|G_1| = m - |E'|)$  then  $G_1 = G_1 \cup E'$ ;  $E' = \emptyset$ 
  if  $(|G_2| = m - |E'|)$  then  $G_2 = G_2 \cup E'$ ;  $E' = \emptyset$ 
end while
end

```

The two parts of the algorithm (group initialization and insertion of entries) are both quadratic in M . In the last part of the algorithm, if it turns out that one of the group (say G_1) has been preferred to the other one (say G_2), the set of remaining entries E' must be assigned to G_2 independently from their location. This might lead to a bad distribution in some cases. Obviously, this depends on the parameter m which defines the minimal cardinality of a group. It seems that $m=40\%$ is a suitable value for this algorithm.

2.7 Indexation dans Oracle

Oracle propose plusieurs techniques d'indexation : arbres B, arbres B+, tables de hachage. Par défaut la structure d'index est un arbre B+, stocké dans un *segment d'index* séparément de la table à indexer. Il est possible de demander explicitement qu'une table soit physiquement organisée avec un arbre-B (*Index-organized table*) ou par une table de hachage (*hash cluster*).

2.7.1 Arbres B+

La principale structure d'index utilisée par Oracle est l'arbre B+, ce qui s'explique par les bonnes performances de cet index dans la plupart des situations (recherche, recherches par préfixe, mises à jour). Par défaut un arbre B+ est créé la clé primaire de chaque table, ce qui offre un double avantage :

1. l'index permet d'optimiser les jointures, comme nous le verrons plus tard ;
2. au moment d'une insertion, l'index permet de vérifier très rapidement que la nouvelle clé n'existe pas déjà.

Oracle maintient automatiquement l'arbre B+ au cours des insertions, suppressions et mises à jour affectant la table, et ce de manière transparente pour l'utilisateur. Ce dernier, s'il effectue des recherches par des attributs qui ne font pas partie de la clé, peut optimiser ses recherches en créant un nouvel index avec la commande `CREATE INDEX`. Par exemple on peut créer un index sur les nom et prénom des artistes :

```
CREATE UNIQUE INDEX idxNomArtiste ON Artiste (nom, prenom)
```

Cette commande ne fait pas partie de la norme SQL ANSI mais on la retrouve à peu de choses près dans tous les SGBD relationnels. Notons que Oracle crée le même index si on spécifie une clause `UNIQUE(nom, prenom)` dans le `CREATE TABLE`.

La clause `UNIQUE` est optionnelle. On peut créer un index non-unique sur des attributs susceptibles de contenir la même valeur pour deux tables différentes. Voici un exemple permettant d'optimiser les recherches portant sur le genre d'un film.

```
CREATE INDEX idxGenre ON Film (genre)
```

Attention cependant à la sélectivité des recherches avec un index non-unique. Si, avec une valeur, on en arrive à sélectionner une partie importante de la table, l'utilisation d'un index n'apportera pas grand chose et risque même de dégrader les performances. Il est tout à fait déconseillé par exemple de créer un index sur une valeur booléenne car une recherche séquentielle sur le fichier sera beaucoup plus efficace.

L'arbre B+ est placé dans le segment d'index associé à la table. On peut indiquer dans la clause `CREATE INDEX` le *tablespace* où ce segment doit être stocké, et paramétrer alors ce *tablespace* pour optimiser le stockage des blocs d'index.

Au moment de la création d'un index, Oracle commence par trier la table dans un segment temporaire, puis construit l'arbre B+ de bas en haut afin d'obtenir un remplissage des blocs conforme au paramètre `PCTFREE` du *tablespace*. Au niveau des feuilles de l'arbre B+, on trouve, pour chaque valeur, le (cas de l'index unique) ou les (cas de l'index non-unique) ROWID des enregistrements associés à cette valeur.

2.7.2 Arbres B

Rappelons qu'un arbre-B consiste à créer une structure d'index et à stocker les enregistrements dans les nœuds de l'index. Cette structure est plaçante, et il ne peut donc y avoir qu'un seul arbre-B pour une table.

Oracle nomme *index-organized tables* la structure d'arbre-B. Elle est toujours construite sur la clé primaire d'une table, des index secondaires (en arbre-B+ cette fois) pouvant toujours être ajoutés sur d'autres colonnes.

À la différence de ce qui se passe quand la table est stockée dans une structure séquentielle, un enregistrement n'est pas identifié par son ROWID mais par sa clé primaire. En effet les insertions ou destructions entraînent des réorganisations de l'index qui amènent à déplacer les enregistrements. Les pointeurs dans les index secondaires, au niveau des feuilles, sont donc les valeurs de clé primaire des enregistrements. Étant donné une clé primaire obtenue par traversée d'un index secondaire, l'accès se fait par recherche dans l'arbre-B principal, et non plus par accès direct avec le ROWID.

Une table organisée en arbre-B fournit un accès plus rapide pour les recherches basées sur la valeur de clé, ou sur un intervalle de valeur. La traversée d'index fournit en effet directement les enregistrements, alors qu'elle ne produit qu'une liste de ROWID dans le cas d'un arbre-B+. Un autre avantage, moins souvent utile, est que les enregistrements sont obtenus triés sur la clé primaire, ce qui peut faciliter certaines jointures, ou les requêtes comportant la clause `ORDER BY`.

En contrepartie, Oracle met en garde contre l'utilisation de cette structure quand les enregistrements sont de taille importante car on ne peut alors mettre que peu d'enregistrements dans un nœud de l'arbre, ce qui risque de faire perdre à l'index une partie de son efficacité.

Pour créer une table en arbre-B, il faut indiquer la clause `ORGANIZATION INDEXED` au moment du `CREATE TABLE`. Voici l'exemple pour la table *Internaute*, la clé primaire étant l'email.

```
CREATE TABLE Internaute
    (email VARCHAR (...),
    ...
    PRIMARY KEY (email),
    ORGANIZATION INDEX)
```

2.7.3 Indexation de documents

Oracle ne fournit pas explicitement de structure d'index inversé pour l'indexation de documents, mais propose d'utiliser une table en arbre-B. Rappelons qu'un index inversé est construit sur une table contenant typiquement trois attributs :

1. le mot-clé apparaissant dans le ou les document(s) ;
2. l'identifiant du ou des documents où figure le mot ;
3. des informations complémentaires, comme le nombre d'occurrences.

Il est possible de créer une table stockée séquentiellement, et de l'indexer sur le mot-clé. L'inconvénient est que c'est alors une arbre-B+ qui est créé, ce qui implique un accès supplémentaire par ROWID pour chaque recherche, et la duplication des clés dans l'index et dans la table. La structuration de cette table par un arbre-B est alors recommandée car elle résout ces deux problèmes.

2.7.4 Tables de hachage

Les tables de hachage sont nommées *hash cluster* dans Oracle. Elles sont créées indépendamment de toute table, puis une ou plusieurs tables peuvent être affectées au *cluster*. Pour simplifier nous prendrons le cas où une seule table est affectée à un *cluster*, ce qui correspond à une organisation par hachage semblable semblable à celle que nous avons présentée précédemment. Il est important de noter que le hachage dans Oracle est *statique*.

Remarque : Il existe dans Oracle un autre type de regroupement dit *indexed cluster*, qui n'est pas présenté ici. Elle consiste à grouper dans des blocs les lignes de plusieurs tables en fonction de leurs clés.

La création d'une table de hachage s'effectue en deux étapes. On définit tout d'abord les paramètres de l'espace de hachage avec la commande `CREATE CLUSTER`, puis on indique ce *cluster* au moment de la création de la table. Voici un exemple pour la table *Film*.

```
CREATE CLUSTER HachFilms (id INTEGER)
SIZE 500
HASHKEYS 500;

CREATE TABLE Film (idFilm INTEGER,
    ... )
CLUSTER HachFilms (idFilm)
```

La commande `CREATE CLUSTER`, combinée avec la clause `HASHKEYS`, crée une table de hachage définie par paramètres suivants :

1. la *clé de hachage* est spécifiée dans l'exemple comme étant un `id` de type `INTEGER` ;
2. le *nombre d'entrées* dans la table est spécifié par `HASHKEYS` ;

3. la taille estimée pour chaque entrée est donnée par `SIZE`.

Oracle utilise une fonction de hachage appropriée pour chaque type d'attribut (ou combinaison de type). Il est cependant possible pour l'administrateur de donner sa propre fonction, à ses risques et périls.

Dans l'exemple qui précède, le paramètre `SIZE` est égal à 500. L'administrateur estime donc que la somme des tailles des enregistrements qui seront associés à une même entrée est d'environ 500 octets. Pour une taille de bloc de 4 096 octets, Oracle affectera alors $\lfloor \frac{4096}{500} \rfloor = 4$ entrées de la table de hachage à un même bloc. Cela étant, si une entrée occupe à elle seule tout le bloc, les autres seront insérées dans un bloc de débordement.

Pour structurer une table en hachage, on l'affecte simplement à un *cluster* existant :

```
CREATE TABLE Film (idFilm INTEGER,  
                  ... )  
CLUSTER HachFilms (idFilm)
```

Contrairement à l'arbre-B+ qui se crée automatiquement et ne demande aucun paramétrage, l'utilisation des tables de hachage demande de bonnes compétences des administrateurs, et l'estimation – délicate – des bons paramètres. De plus, le hachage dans Oracle n'étant pas extensible, cette technique est réservée à des situations particulières.

2.7.5 Index bitmap

Oracle propose une indexation par index *bitmap* et suggère de l'utiliser dès que la *cardinalité* d'un attribut, défini comme le nombre moyen de répétition pour les valeurs de cet attribut, est égal ou dépasse cent. Par exemple une table avec un million de lignes et seulement 10 000 valeurs différentes dans une colonne, cette colonne peut avantageusement être indexée par un index *bitmap*. Autrement dit ce n'est pas le nombre de valeurs différentes qui est important, mais le ratio entre le nombre de lignes et ce nombre de valeurs.

L'optimiseur d'Oracle prend en compte l'indexation par index *bitmap* au même titre que toutes les autres structures.

2.8 Exercices

Chapitre 3

Organisation Physique

Exercice 8.

On prend ici comme exemple la relation **Directeur** (nom_directeur, nom_film).

1. Organisation Séquentielle : Expliquer l'organisation séquentielle et représenter un exemple sur la relation **Directeur**. Montrer le fichier après une insertion et après quelques suppressions d'articles.
2. Organisation Indexée : Montrer des exemples d'index non dense (primaire) et dense (secondaire) sur la relation **Directeur**.

Exercice 9. Soit l'arbre B+ de la figure 3.1, avec 4 entrées par bloc au maximum. Expliquez les opérations suivantes, et donnez le nombre d'entrées-sorties de blocs nécessaires.

1. Recherche de l'enregistrement 41.
2. Recherche des enregistrements 17 à 30.
3. Insertion des enregistrements 1 et 4.
4. Destruction de l'enregistrement 23.

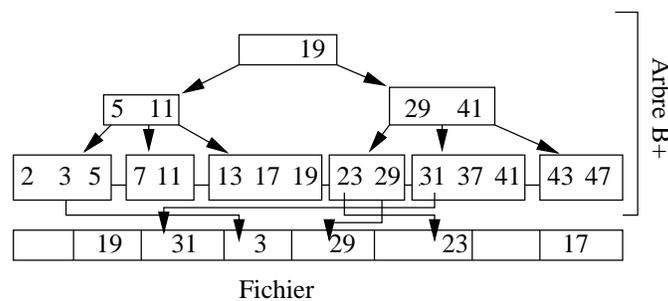


FIGURE 3.1 – Un arbre B+

Exercice 10. Soit la liste des départements suivants.

- 3 Allier
- 36 Indre
- 18 Cher
- 75 Paris
- 39 Jura
- 9 Ariège

81 Tarn
 11 Aude
 12 Aveyron
 25 Doubs
 73 Savoie
 55 Meuse
 15 Cantal
 51 Marne
 42 Loire
 40 Landes
 14 Calvados
 30 Gard
 84 Vaucluse
 7 Ardèche

1. Construire, en prenant comme clé le numéro de département, un index dense à deux niveaux sur le fichier contenant les enregistrements dans l'ordre donné ci-dessus, en supposant 2 enregistrements par bloc pour les données, et 8 par bloc pour l'index.
2. Construire un index non-dense sur le fichier trié par numéro, avec les mêmes hypothèses.
3. Construire un arbre-B sur les noms de département, en supposant qu'il y a au plus 4 enregistrements par bloc, et en prenant les enregistrements dans l'ordre donné ci-dessus.
4. On suppose que les départements sont stockés dans un fichier séquentiel, dans l'ordre donné, et que le fichier de données contient deux enregistrements par bloc. Construisez un arbre-B+ sur le nom de département avec au plus 4 entrées par bloc.
5. Quel est le nombre de lectures nécessaires, pour l'arbre-B puis pour l'arbre-B+, pour les opérations suivantes, :
 - (a) rechercher le Cher ;
 - (b) rechercher les départements entre le Cantal et les Landes (donner le nombre de lectures dans le pire des cas).

Exercice 11 (Index dense et non-dense). Soit un fichier de données tel que chaque bloc peut contenir 10 enregistrements. On indexe ce fichier avec un niveau d'index, et on suppose qu'un bloc d'index contient 100 entrées [valeur, adresse].

Si n est le nombre d'enregistrements, donnez la fonction de n permettant d'obtenir le nombre minimum de blocs pour les structures suivantes :

1. Un fichier avec un index dense.
2. Un fichier trié sur la clé avec un index non-dense.

Exercice 12 (Arbre-B+). On reprend les hypothèses précédentes, et on indexe maintenant le fichier avec un arbre-B+. Les feuilles de l'arbre contiennent donc des pointeurs vers le fichier, et les nœuds internes des pointeurs vers d'autres nœuds. On suppose qu'un bloc d'arbre B+ est plein à 70% (69 clés, 70 pointeurs).

1. Le fichier est indexé par un arbre-B+ dense. Donnez (1) le nombre de niveaux de l'arbre pour un fichier de 1 000 000 articles, (2) le nombre de blocs utilisés (y compris l'index), et (3) le nombre de lectures pour rechercher un article par sa clé.
2. On effectue maintenant une recherche par intervalle ramenant 1 000 articles. Décrivez la recherche et donnez le nombre de lectures dans le pire des cas.
3. Mêmes questions que (1), mais le fichier est trié sur la clé, et l'arbre-B+ est non dense.

Exercice 13. Soit un fichier de 1 000 000 enregistrements répartis en pages de 4 096 octets. Chaque enregistrement fait 45 octets et il n'y a pas de chevauchement de blocs. Répondez aux questions suivantes en justifiant vos réponses (on suppose que les blocs sont pleins).

1. Combien faut-il de blocs ? Quelle est la taille du fichier ?
2. Quelle est la taille d'un index de type arbre-B+ si la clé fait 32 octets et un pointeur 8 octets ? Même question si la clé fait 4 octets.
3. Si on suppose qu'un E/S coûte 10 ms, quel est le coût moyen d'une recherche d'un enregistrement par clé unique, avec index et sans index ?

Exercice 14. Un arbre B+ indexe un fichier de 300 enregistrements. On suppose que chaque nœud stocke au plus 10 clés et 10 pointeurs. Quelle est la hauteur minimale de l'arbre et sa hauteur maximale ? (Un arbre constitué uniquement de la racine a pour hauteur 0).

Inversement, on constate que l'arbre B+ a pour hauteur 1. Quel est le nombre minimal de pointeurs par bloc ? Le nombre maximal ?

Exercice 15. On indexe une table par un arbre B+ sur un identifiant dont les valeurs sont fournies par une séquence. À chaque insertion un compteur est incrémenté et fournit la valeur de clé de l'enregistrement inséré.

On suppose qu'il n'y a que des insertions dans la table. Montrez que tous les nœuds de l'index qui ont un frère droit sont exactement à moitié pleins.

Exercice 16. Soit un fichier non trié contenant N enregistrements de 81 octets chacun. Il est indexé par un arbre-B+, comprenant 3 niveaux, chaque entrée dans l'index occupant 20 octets. On utilise des blocs de 4 096 octets, sans entête, et on suppose qu'ils sont utilisés à 100 % pour le fichier et à 70 % pour l'index.

On veut effectuer une recherche par intervalle dont on estime qu'elle va ramener m enregistrements. On suppose que tous les blocs sont lus sur le disque pour un coût uniforme.

1. Donnez la fonction exprimant le nombre de lectures à effectuer pour cette recherche avec un parcours séquentiel.
2. Donnez la fonction exprimant le nombre de lectures à effectuer en utilisant l'index.
3. Pour quelle valeur de m la recherche séquentielle est-elle préférable à l'utilisation de l'index, en supposant un temps d'accès uniforme pour chaque bloc ?

En déduire le pourcentage d'enregistrements concernés par la recherche à partir duquel le parcours séquentiel est préférable.

On pourra simplifier les équations en éliminant les facteurs qui deviennent négligeables pour des grandes valeurs de N et de m .

Exercice 17. Soit les deux tables suivantes :

```
CREATE TABLE R (idR VARCHAR(20) NOT NULL,
                PRIMARY KEY (idR));
```

```
CREATE TABLE S (idS INT NOT NULL,
                idR VARCHAR(20) NOT NULL,
                PRIMARY KEY (idS),
                FOREIGN KEY idR REFERENCES R);
```

Indiquez, pour les ordres SQL suivants, quels index peuvent améliorer les performances ou optimiser la vérification des contraintes PRIMARY KEY et FOREIGN KEY.

1. `SELECT * FROM R WHERE idR = 'Bou'`
2. `SELECT * FROM R WHERE idR LIKE 'B%'`
3. `SELECT * FROM R WHERE LENGTH(idR) = 3`
4. `SELECT * FROM R WHERE idR LIKE '_ou'`
5. `INSERT INTO S VALUES (1, 'Bou')`

6. `SELECT * FROM S WHERE idS BETWEEN 10 AND 20`

7. `DELETE FROM R WHERE idR LIKE 'Z%'`

Exercice 18. Écrire l'algorithme de recherche par intervalle dans un arbre-B.

Exercice 19. Soit la liste des départements données dans l'exercice 10, la clé étant le numéro de département. On suppose qu'un bloc contient 5 enregistrements.

1. Proposez une fonction de hachage et le nombre d'entrées de la table, puis construisez la structure en prenant les enregistrements dans l'ordre indiqué.
2. Insérez un ou plusieurs autres départements de manière à provoquer un chaînage pour l'une des entrées.

Exercice 20. Même exercice, mais avec une structure basée sur le hachage extensible.

On prendra une fonction de hachage sur le nom, définie de la manière suivante : $f(\text{nom}) = i_1 i_2 \dots i_4$ avec $i_j = 1$ si la lettre $\text{nom}[i_j]$ est en position impaire dans l'alphabet, et 0 sinon. Donc $f(\text{Aude}) = 1101$. Voici la liste des valeurs de hachage, en ne prenant que les 4 premiers bits.

Allier	1001	Indre	1000	Cher	1010	Paris	0101	
Jura	0101	Ariège		1011	Tarn	0100	Aude	1101
Aveyron	1011	Doubs	0110	Savoie	1101	Meuse	1111	
Cantal	1100	Marne	1100	Loire	0110	Landes	0100	
Calvados	1100	Gard	1100	Vaucluse	0111	Ardèche	1001	

Le hachage extensible consiste à considérer les n derniers bits de la valeur de hachage (en d'autres termes, on prend $h(v) \bmod 2^n$, le reste de la division de la valeur de hachage par 2^n). Au départ on prend $n = 1$, puis $n = 2$ quand une extension est nécessaire, etc. Montrez l'évolution de la structure de hachage extensible en insérant les départements dans l'ordre indiqué (de gauche à droite, puis de haut en bas).

Exercice 21. On indexe l'ensemble des pistes de ski du domaine skiable alpin français. Chaque piste a une couleur qui peut prendre les valeurs suivantes : Verte, Rouge, Bleue, Noire. On suppose qu'il y a le même nombre de pistes de chaque couleur, et que la taille d'un bloc est de 4 096 octets.

1. Donnez la taille d'un index bitmap en fonction du nombre de pistes indexées.
2. Combien de blocs faut-il lire, dans le pire des cas, pour rechercher les pistes vertes ?
3. Combien de blocs pour compter le nombre de pistes vertes ?
4. Que se passerait-il si on utilisait un arbre B+ ?

Exercice 22. Soit un arbre-R dont chaque nœud a une capacité maximale de 4 entrées. Quelle est à votre avis la meilleure distribution possible au moment de l'éclatement du nœud de la figure 3.2 ?

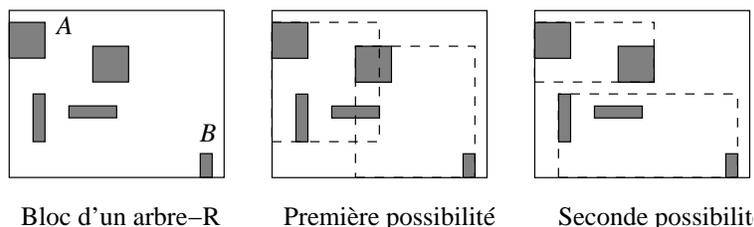
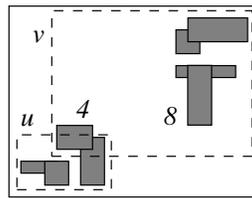


FIGURE 3.2 – Éclatement d'un nœud d'arbre-R

Exercice 23. Même hypothèse que précédemment : on insère le rectangle 8 dans le nœud de la figure 3.3.

1. Quel est le résultat après réorganisation ?



Insertion du rectangle 8

FIGURE 3.3 – Éclatement avec réinsertion

2. Comment pourrait-on faire mieux, en s'autorisant la réinsertion dans l'arbre de un ou plusieurs rectangles d'un nœud avant éclatement.

Exercice 24. Soit un rectangle R de dimension $[h, l]$ dans un espace de dimension $[d, d]$.

1. Quelle est la probabilité qu'un pointé $P(x, y)$ ramène le rectangle, en supposant une distribution uniforme ?
2. Quelle est la probabilité qu'un fenêtrage $W(w_h, w_l)$ ramène le rectangle ?

Exercice 25. Écrire l'algorithme de pointé avec un arbre- R .

Chapitre 4

Évaluation de requêtes

Sommaire

4.1	Introduction à l'optimisation des performances	72
4.1.1	Opérations exprimées par SQL	72
4.1.2	Traitement d'une requête	72
4.1.3	Mesure de l'efficacité des opérations	73
4.1.4	Organisation du chapitre	74
4.1.5	Modèle d'exécution	75
4.2	Algorithmes de base	77
4.2.1	Recherche dans un fichier (sélection)	77
4.2.2	Quand doit-on utiliser un index ?	79
4.2.3	Le tri externe	82
4.3	Algorithmes de jointure	85
4.3.1	Jointure par boucles imbriquées	86
4.3.2	Jointure par tri-fusion	88
4.3.3	Jointure par hachage	90
4.3.4	Jointure avec un index	92
4.3.5	Jointure avec deux index	93
4.4	Compilation d'une requête et optimisation	94
4.4.1	Décomposition en bloc	94
4.4.2	Traduction et réécriture	96
4.4.3	Plans d'exécution	98
4.4.4	Modèles de coût	102
4.5	Oracle, optimisation et évaluation des requêtes	103
4.5.1	L'optimiseur d'ORACLE	104
4.5.2	Plans d'exécution ORACLE	105
4.6	Exercices	110
4.6.1	Opérateurs d'accès aux données	110
4.6.2	Plans d'exécution ORACLE	113
4.6.3	Utilisation de EXPLAIN et de TKPROF	117
4.6.4	Exercices d'application	119

L'objectif de ce chapitre est de montrer comment un SGBD analyse, optimise et exécute une requête. SQL étant un langage *déclaratif* dans lequel on n'indique ni les algorithmes à appliquer, ni les chemins d'accès aux données, le système a toute latitude pour déterminer ces derniers et les combiner de manière à obtenir les meilleures performances. Les modules chargés de ces tâches (et notamment *l'optimiseur de requêtes*) ont donc un rôle extrêmement important puisque l'efficacité d'un SGBD est fonction, pour une grande part, du temps d'exécution des requêtes. Ces modules sont également extrêmement complexes et

combinent des techniques éprouvées et des heuristiques propres à chaque système. Il est en effet reconnu qu'il est impossible de trouver en un temps raisonnable l'algorithme *optimal* pour exécuter une requête donnée. Afin d'éviter de consacrer des ressources considérables à l'optimisation, ce qui se ferait au détriment des autres tâches du système, les SGBD s'emploient donc à trouver, en un temps limité, un algorithme raisonnablement bon.

La compréhension des mécanismes d'exécution et d'optimisation fournit une aide très précieuse quand vient le moment d'analyser le comportement d'une application et d'essayer de distinguer les goulets d'étranglements. Comme nous l'avons vu dans les chapitres consacrés au stockage des données et aux index, des modifications très simples de l'organisation physique peuvent aboutir à des améliorations (ou des dégradations) extrêmement spectaculaires des performances. Ces constatations se transposent évidemment au niveau des algorithmes d'évaluation : le choix d'utiliser ou non un index conditionne fortement les temps de réponse, sans que ce choix soit d'ailleurs évident dans toutes les circonstances.

4.1 Introduction à l'optimisation des performances

Nous commençons par une description d'assez haut niveau des diverses de traitement d'une requête, suivie de quelques rappels sur les critères d'évaluation de performance utilisés.

4.1.1 Opérations exprimées par SQL

La lecture de ce chapitre suppose le lecteur familier avec SQL et avec l'interprétation d'une requête SQL sous forme d'opérations ensemblistes appliquées à des tables (l'algèbre relationnelle). Voici un bref rappel des notations utilisées pour ces opérations :

1. La sélection, notée $\sigma_F(R)$, sélectionne dans la table R tous les enregistrements satisfaisant le critère F ;
2. La projection, notée $\pi_{A_1, \dots, A_n}(R)$, parcourt tous les enregistrements de R et ne garde que les attributs A_1, \dots, A_n ;
3. La jointure, notée $R \bowtie_F S$, calcule les paires d'enregistrements venant de R et S , satisfaisant le critère F .
4. L'union, $R \cup S$, effectue l'union des enregistrements des deux tables R et S ;
5. La différence, $R - S$, calcule les enregistrements de R qui ne sont pas dans S .

Le noyau du langage SQL (en mettant à part des extensions comme les fonctions, et les opérations de regroupement) permet d'exprimer les mêmes requêtes qu'un petit langage de programmation contenant les seules opérations ci-dessus. Pour décrire l'optimisation, on traduit donc une requête SQL en une *expression* de l'algèbre, ce qui permet alors de raisonner concrètement en termes d'opérations à effectuer sur les tables.

4.1.2 Traitement d'une requête

Le traitement d'une requête s'effectue en trois étapes (figure 4.1) :

1. **Analyse de la requête** : à partir d'un texte SQL, le système vérifie la correction syntaxique puis contrôle la validité de la requête : existence des tables (ou vues) et des attributs (analyse sémantique). Enfin des normalisations sont effectuées (par exemple en essayant de transformer une requête imbriquée en requête plate) ainsi que des tests de cohérence (une clause `WHERE` avec le test `1 = 2` s'évalue toujours à `false`).

Le résultat est un *plan d'exécution logique* qui représente la requête sous la forme d'une arbre composé des opérations de l'algèbre relationnelle¹.

1. Il s'agit d'une vue conceptuelle : les systèmes ne sont pas requis d'utiliser l'algèbre, mais elle fournit pour nous un moyen simple et pratique d'interpréter une requête en terme d'opérations sur des tables.

2. **Optimisation** : le plan logique est transformé en un plan d'exécution physique, comprenant les opérations d'accès aux données et les algorithmes d'exécution, placées dans un ordre supposé optimal (ou quasi-optimal).

Le choix du meilleur plan dépend des opérations physiques implantant les divers opérateurs de l'algèbre, disponibles dans le processeur de requêtes. Il dépend également des *chemins d'accès* aux fichiers, c'est-à-dire de l'existence d'index ou de tables de hachage. Enfin il peut s'appuyer sur des données statistiques enregistrées ou estimées pour chaque relation (nombre d'enregistrements et de pages dans une relation, sélectivité d'un attribut, etc.).

3. **Exécution de la requête** : le plan d'exécution est finalement compilé, ce qui fournit un programme d'un type assez particulier. L'exécution de ce programme calcule le résultat de la requête.

Ce qui fait la particularité d'un plan d'exécution physique, c'est en premier lieu qu'il s'agit d'un programme sous forme d'arbre, dans lequel chaque nœud correspond à une opération appliquée aux données issues des nœuds inférieurs. Un mécanisme de transfert des données entre les nœuds (*pipelining*) évite de stocker les résultats intermédiaires. En second lieu, les opérations disponibles sont en nombre limitées et constituent la « bibliothèque » du processeur de requête pour créer des plans d'exécution. Les principales opérations sont les *algorithmes de jointure* dont le choix conditionne souvent de manière décisive l'efficacité d'une requête.

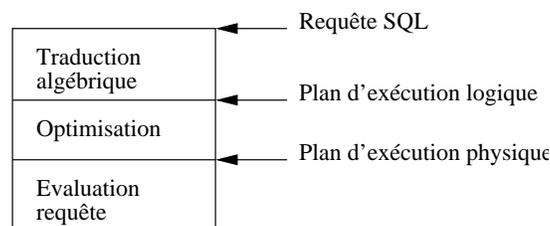


FIGURE 4.1 – Les étapes de traitement d'une requête

Les informations telles que l'existence et la spécification d'index sur une table et les données statistiques enregistrées sont stockées dans le *catalogue* de la base. Celui-ci est également représenté sous forme de relations, gérées par le système, auxquelles l'administrateur accède simplement par des requêtes SQL. Le contenu du catalogue, également appelé *schéma physique*, est parfois désigné par le terme de *méta-données* (« des données sur des données »). Le catalogue est en général conservé en mémoire ce qui explique que le processus d'analyse/optimisation, s'il paraît complexe, occupe en fait un temps négligeable comparé à celui de l'exécution qui est le seul à engendrer des entrées/sorties.

4.1.3 Mesure de l'efficacité des opérations

Par évaluation efficace, on entend généralement évaluation qui prend un temps le plus court possible. L'optimisation consiste à trouver le plan qui minimisera le temps pris pour exécuter la séquence d'opérations. Il faut alors prendre en compte les principaux facteurs qui influent sur ce temps d'exécution, à savoir :

- la mémoire centrale disponible ;
- le nombre d'accès en mémoire secondaire nécessaire ;
- le nombre d'utilisateurs accédant simultanément au système.

Les trois facteurs sont en partie liés. Par exemple la place disponible en mémoire centrale peut dépendre du nombre d'utilisateurs. Le nombre de lectures en mémoire secondaire dépend du nombre de pages déjà chargées en mémoire principale, et donc de la taille de cette dernière. Il faut également souligner que pour des applications transactionnelles, on peut chercher à favoriser le débit en nombre de transactions au dépens du temps d'évaluation d'une requête utilisateur, ce qui change les critères d'optimisation.

La complexité de prise en compte de tous ces facteurs implique que l'on n'a aucune garantie sur l'optimalité de la solution choisie. L'optimisation consiste en un certain nombre d'heuristiques qui garantissent l'exécution d'une requête dans un délai raisonnable.

Parmi les nombreux autres facteurs, peut-être moins important, entrant en jeu, citons les cas où l'efficacité requise concerne l'arrivée du premier résultat (le « temps de réponse »), plutôt que le temps pour obtenir le résultat total (« temps d'évaluation »). Quand la taille du résultat est importante, les stratégies d'optimisation peuvent différer suivant qu'on cherche à optimiser le temps de réponse ou le temps d'évaluation. En particulier le pipelining que nous verrons en fin de la section sur l'optimisation permet de favoriser le temps de réponse.

Mesure du coût d'une opération

Le nombre de lectures/écritures de page sur disque sera notre critère essentiel d'estimation du coût d'une opération. Ce type d'estimation est justifié par le fait qu'amener en mémoire centrale des enregistrements coûte beaucoup plus cher que de les traiter en mémoire.

Nous ne comptons jamais le coût d'écriture du résultat d'une opération sur disque. En effet, d'une part le nombre de pages à écrire sur disque, si nécessaire, peut être facilement ajouté : il dépend uniquement de la taille du résultat et pas de l'algorithme. D'autre part, nous verrons dans la section sur le pipelining que les résultats intermédiaires ne sont pas nécessairement réécrits sur disque : chaque enregistrement est utilisé par l'opération suivante dès sa production.

Un certain nombre de paramètres sont nécessaires pour l'estimation du coût d'une opération. Le premier est la mémoire centrale disponible, notée M et mesurée en nombre de pages. M est donc le nombre de pages en mémoire centrale destinés à contenir les entrées d'une opération et les résultats intermédiaires. Le nombre de pages qu'occupe sur disque une relation en entrée d'une opération est le deuxième paramètre, noté B . Enfin le nombre d'enregistrements d'une table est noté N .

4.1.4 Organisation du chapitre

Ce chapitre se concentre sur les étapes d'optimisation et d'exécution d'une requête. La phase d'analyse relève de techniques classiques d'analyse syntaxique et de compilation, et ne présente pas vraiment d'intérêt du point de vue des performances. Nous commençons (section 4.2, page 77) par décrire les principaux algorithmes d'accès et de manipulation des données : tri, sélection, et surtout jointures. Nous expliquons ensuite (section 4.4, page 94) comment ces algorithmes sont combinés durant la phase d'optimisation pour former des plans d'exécution. Enfin le chapitre se conclut par une présentation détaillée du mécanisme d'optimisation et d'évaluation des requêtes dans le SGBD Oracle (section 4.5, page 103).

Les exemples sont basés sur le schéma de base de données suivant, les champs en gras indiquant comme d'habitude la clé primaire :

- Film (**idFilm**, titre, année, genre, résumé, *idMES*, *codePays*)
- Artiste (**idArtiste**, nom, prénom, annéeNaissance)
- Role (*idActeur*, *idFilm*, nomRôle)
- Internaute (**email**, nom, prénom, région)
- Notation (*email*, *idFilm*, note)
- Pays (**code**, nom, langue)

On supposera de plus, occasionnellement, que les artistes sont identifiés de manière unique par leur nom et prénom.

Une partie des exercices proposés pour accompagner ce chapitre consiste à expérimenter l'optimiseur d'Oracle, et à tester les différences de performances obtenues en modifiant l'organisation physique d'une (grosse) base de données. Le site <http://cortes.cnam.fr:8080/CoursBD> propose un petit outil de génération qui permet d'engendrer quelques centaines de milliers ou millions de lignes, à la demande, dans la base *Films* (le script de création se trouve également sur le site). À partir de cette base, on peut tester les outils décrits dans la section 4.5 et obtenir une idée concrète de l'importance d'une administration avisée pour obtenir des applications performantes.

4.1.5 Modèle d'exécution

Même si la phase d'optimisation a choisi le meilleur plan d'exécution, c'est-à-dire la meilleure séquence d'opérations, les performances de l'exécution dépendront fortement de l'espace disponible en mémoire centrale. Cet espace est important pour le choix et les performances d'une opération.

Si la meilleure allocation d'espace entre opérateurs est un problème difficile à résoudre dans toute généralité, il est néanmoins facile de minimiser l'espace nécessaire pour l'exécution de deux opérateurs en séquence. Imaginons qu'il faille effectuer deux opérations o et o' (par exemple un parcours d'index suivi d'un accès au fichier) pour évaluer une requête. Une manière naïve de procéder est d'exécuter d'abord l'opération o , de stocker le résultat intermédiaire en mémoire s'il y a de la place ou sur disque sinon, et d'utiliser le buffer en mémoire ou le fichier intermédiaire comme source de données pour o' . Par exemple le parcours d'index (l'opération o) rechercherait toutes les adresses d'enregistrement et les placerait dans une structure temporaire, puis l'opération o' irait lire ces adresses pour accéder au fichier de données.

Sur un serveur recevant beaucoup de requêtes, la mémoire centrale deviendrait rapidement disponible ou trop petite pour accueillir les résultats intermédiaires qui devront donc être écrits sur disque. Cette solution de *matérialisation* des résultats intermédiaires est donc très pénalisante car les écritures/lectures sur disque répétées entre chaque opération coûtent d'autant plus cher en temps que la séquence d'opérations est longue.

L'alternative appelée *pipelining* consiste à ne pas écrire les enregistrements produits par o sur disque mais à les utiliser immédiatement comme entrée de o' . Les deux opérateurs, o et o' , sont donc connectés, o tenant le rôle du producteur et o' celui du consommateur. Chaque fois que o produit une adresse d'enregistrement, o' la reçoit et va lire l'enregistrement dans le fichier.

On n'attend donc pas que o soit terminée, et que l'ensemble des enregistrements résultat de o ait été produit pour lancer o' . *Il n'est donc pas nécessaire de stocker un enregistrement intermédiaire.* On peut ainsi combiner l'exécution de plusieurs opérations, cette combinaison constituant justement le plan d'exécution final. Cette méthode a deux avantages très importants :

1. *il n'est donc pas nécessaire de stocker un résultat intermédiaire*, puisque ce résultat est consommé au fur et à mesure de sa production ;
2. l'utilisateur reçoit les premières lignes du résultat alors même que ce résultat n'est pas calculé complètement.

On peut ainsi combiner l'exécution de plusieurs opérations, constituant le plan d'exécution final, produisant au fur et à mesure le résultat. Si l'utilisateur met un peu de temps à traiter chaque ligne produite, l'exécution de la requête peut même ne plus constituer du tout un facteur pénalisant. Si, par exemple, l'utilisateur ou l'application qui demande l'exécution met 0,1 seconde pour traiter chaque ligne alors que le plan d'exécution peut fournir 20 lignes par secondes, c'est le premier qui constitue le point de contention, et le coût de l'exécution disparaît dans celui du traitement. Il en irait tout autrement s'il fallait d'abord calculer *tout* le résultat avant de lui appliquer le traitement : dans ce cas le temps passé dans chaque phase s'additionnerait.

Cette remarque explique une dernière particularité : un plan d'exécution se déroule en fonction de la demande et pas de l'offre. C'est toujours le consommateur, o' dans notre exemple, qui « tire » un enregistrement de son producteur o , quand il en a besoin, et pas o qui « pousse » un enregistrement vers o' dès qu'il est produit. La justification est simplement que le consommateur, o' , pourrait se retrouver débordé par l'afflux d'information sans avoir assez de temps pour les traiter, ni assez de mémoire pour les stocker.

Opérations bloquantes

Parfois il n'est pas possible d'éviter le calcul complet de l'une des opérations avant de continuer. On est alors en présence d'une opération dite *bloquante* dont le résultat doit être entièrement produit et écrit sur disque avant de démarrer l'opération suivante. Par exemple :

1. le tri (ORDER BY) ;
2. la recherche d'un maximum ou d'un minimum (MAX, MIN) ;
3. l'élimination des doublons (DISTINCT) ;

4. le calcul d'une moyenne ou d'une somme (SUM, AVG) ;
5. un partitionnement (GROUP BY)

sont autant d'opérations qui doivent lire complètement les données en entrée avant de produire un résultat (il est facile d'imaginer qu'on ne peut pas produire le résultat d'un tri tant qu'on n'a pas lu le dernier élément en entrée).

Donc lorsque le pipelining de deux opérations o et o' en séquence est possible, on évite l'écriture des résultats intermédiaires sur disque, et on minimise la place nécessaire en mémoire centrale. Aussitôt qu'un article a été produit par o il est utilisé comme entrée de o' : on a juste besoin de la place en mémoire nécessaire pour écrire un article. En fait c'est o' qui *demande* à o un enregistrement. Si o' est un opération binaire il demande à chacun de ses opérations filles dans l'arbre, respectivement o et o'' , un enregistrement.

Ce mécanisme de pipelining « à la demande » est implanté au moyen d'un arbre d'*itérateurs* (correspondant au PEP) dont les fonctions s'appellent à des moments appropriés. À chaque opérateur physique (chaque nœud dans l'arbre) correspond un itérateur. Les données demandées par l'itérateur implémentant o' sont générées par l'itérateur fils implémentant o .

Pour faciliter la construction de plans d'exécution par assemblage d'opérations, chaque itérateur peut s'implanter comme un objet doté de trois fonctions :

1. OPEN. Cette fonction commence le processus pour obtenir des articles résultats. Elle alloue les ressources nécessaires, initialise des structures de données et appelle OPEN pour chacun de ses arguments (c'est-à-dire pour chacun des itérateurs fils).
2. NEXT. Cette fonction remplit une étape de l'itération, retourne l'article résultat suivant en séquence et met à jour les structures de données nécessaires pour obtenir les résultats suivants. Pour obtenir un article résultat, elle peut appeler une ou plusieurs fois NEXT sur ses arguments.
3. CLOSE. cette fonction termine l'itération et libère les ressources, lorsque tous les articles du résultat ont été obtenus. Typiquement elle appelle CLOSE sur chacun de ses arguments.

Nous illustrons la notion d'itérateur en construisant deux itérateurs : celui du balayage séquentiel d'une table et celui de la boucle imbriquée.

```

OpenScan (R)
{
  p:= première page de R;
  n:= premier enregistrement dans la page p;
  FIN = false;
}

NextScan (R)
{
  IF (n a dépassé le dernier enregistrement de la page p) {
    incrémenter p à la page suivante;
    IF (pas de page suivante) {
      FIN := true;
      RETURN;
    }
    ELSE
      lire page p;
      n:= premier enregistrement de la page p;
  }
  vieuxn:= n;
  incrémenter n au enregistrement suivant dans la page p;
  RETURN vieuxn;
}

CloseScan (R)
{
  liberer ressources;
}

```

```

    RETURN ;
}

```

OpenScan (R) initialise la lecture au premier enregistrement de la première page de R . La variable `FIN` est initialisée à `false`. *NextScan* (R) retourne un enregistrement. Si la page courante a été entièrement lue, il lit la page suivante et retourne le premier enregistrement de cette page.

L'itérateur I qui appelle l'itérateur de balayage séquentiel de R commence par appeler *OpenScan* (R). Tant que la variable globale 'FIN' n'est pas à `true`, un enregistrement est retourné (vieuxn) par un appel de *NextScan*(R). Quand la table est entièrement lue (`FIN=true`), I lance l'itérateur *CloseScan* (R).

4.2 Algorithmes de base

L'évaluation d'une requête consiste à exécuter une séquence d'opérations appliquées à des enregistrements lus dans des fichiers. Pour chaque opération, il existe plusieurs algorithmes possibles, dont l'efficacité dépend de la présence ou non d'index, de la taille des relations, de la sélectivité des attributs, etc.

Cette section est consacrée aux principaux opérateurs utilisés dans l'évaluation d'une requête. Nous commençons par la *recherche dans un fichier* (opération de sélection), qui peut se faire par balayage ou par accès direct. *Le tri* est une autre opération fondamentale pour l'évaluation des requêtes. On a besoin du tri par exemple lorsqu'on fait une projection ou une union et qu'on désire éliminer les enregistrements en double (clauses `DISTINCT` ou `ORDER BY` de SQL). On verra également qu'un algorithme de jointure courant consiste à trier au préalable sur l'attribut de jointure les relations à joindre.

Enfin cette section consacre une part importante aux *algorithmes de jointures*. La jointure est une des opérations les plus courantes et les plus coûteuses, et savoir l'évaluer de manière efficace est une condition indispensable pour obtenir un système performant. Les algorithmes couramment utilisés dans les SGBD relationnels sont décrits : jointure par boucles imbriquées, jointure par tri-fusion et jointure par hachage.

4.2.1 Recherche dans un fichier (sélection)

Il existe deux algorithmes pour effectuer une recherche dans un fichier (ou une sélection dans une table, pour employer un vocabulaire de plus haut niveau) en fonction d'un critère de recherche :

1. *Accès séquentiel ou balayage*. Tous les enregistrements du fichier sont examinés lors de l'opération, en général suivant l'ordre dans lequel ils sont stockés.
2. *Accès par adresse*. Si on connaît l'adresse du ou des enregistrements concernés, on peut aller lire directement les blocs et obtenir ainsi un accès optimal.

Voici une description de ces deux méthodes, avant de discuter des situations où on peut les utiliser.

Balayage

Le balayage séquentiel d'une table est utile dans un grand nombre de cas. Tout d'abord on a souvent besoin de parcourir tous les enregistrements d'une relation, par exemple pour faire une projection. Certains algorithmes de jointure utilisent le balayage d'au moins une des deux tables. On peut enfin vouloir accéder à un ou plusieurs enregistrements d'une table satisfaisant un critère de sélection.

Dans le cas de la sélection, le balayage est utilisé soit parce qu'il n'y a pas d'index sur le critère de sélection, soit parce que la table est stockée sur un petit nombre de pages. L'algorithme est très simple. Il consiste à lire en séquence les pages de la relation, une à une, en mémoire centrale. Il suffit d'un tampon en mémoire pour stocker la page courante lue du disque. Les enregistrements de cette page sont parcourus séquentiellement et traités au fur et à mesure. Ce balayage séquentiel permet de faire en même temps que la sélection une projection. Chaque enregistrement du tampon est testé en fonction du critère de sélection. S'il le satisfait, il est projeté et rangé dans un tampon de sortie. L'algorithme ci-dessous, dénommé *SelBal* par la suite, résume ces opérations. Comme la plupart des algorithmes de cette section, il correspond à une boucle sur les lectures/écritures de pages et une seconde boucle sur les enregistrements en mémoire centrale.

```

p:= première page de R;
n:= premier enregistrement de p;
s:= premier octet de S;
Pour chaque page p de R {
    lire p dans le tampon d'entrée E;
    pour chaque enregistrement n dans E {
        IF (n satisfait le critère de sélection) {
            projeter n;
            IF (S plein) {
                vider S sur disque;
                s:= premier octet
            }
            ranger le résultat dans le tampon de sortie S;
            incrémenter s avec la taille de la projection;
        }
    }
}

```

Le coût de cette opération est donc de $B E/S$. On essaie de limiter ce nombre non seulement en compactant les tables, mais également en réduisant le temps de lecture d'une page. Lors d'un balayage, les pages du fichier sont lues en séquence. Si ces pages sont contigues sur disque et si on lit plusieurs pages contigues du disque à la fois, le temps de lecture sera moindre que si on lit ces pages une à une (voir chapitre 1). On cherche donc :

1. à regrouper les pages d'une table dans des espaces contigus (appelés *extensions* dans ORACLE) ;
2. à effectuer des *lectures à l'avance* : quand on lit une page, on lit également les n pages suivantes dans l'extension. Une valeur typique de n est 7 ou 15.

Par conséquent l'unité d'entrée/sortie d'un SGBD (un bloc ou page) est souvent un multiple de celle du gestionnaire de fichier, et la taille d'un tampon en mémoire est en général un multiple de la taille d'un bloc (ou page) physique.

Accès direct

Quand on connaît l'adresse d'un enregistrement, donc l'adresse de la page où il est stocké, y accéder coûte une lecture unique de page. Le problème est un petit peu plus compliqué quand il s'agit d'accéder à plusieurs enregistrements dont on connaît les adresses.

Il peut se faire que les enregistrements soient tous dans des pages différentes, mais il peut arriver également que plusieurs enregistrements soient dans la même page. Si on a un seul tampon en mémoire ($M=1$) alors il est fort possible qu'on relise plusieurs fois la même page. Par exemple, l'accès à 6 enregistrements dont les pages ont pour numéros respectivement 4, 1, 3, 1, 5, 4, demandera 6 lectures de page. Les pages 1 et 4 seront lues deux fois chaque.

Une manière d'économiser le nombre de lectures est de trier la liste des enregistrements à lire sur l'adresse de leur page. De cette manière, on économisera le nombre de lectures, en ne lisant qu'une fois toutes les pages et en traitant en même temps les enregistrements de la même page. Dans l'exemple précédent, si les adresses des enregistrements sont triées avant accès à la table, on ne lira que 4 pages : après tri, on obtient la liste de pages $\langle 1, 1, 3, 4, 4, 5 \rangle$. Lorsque la page 1 (la page 4) est lue on accède aux deux enregistrements de cette page.

Exemple 14. Comparons les performances de recherche avec et sans index, en prenant les hypothèses suivantes : le fichier fait 500 Mo, et une lecture de bloc prend 0,01 s (10 millisecondes).

1. Un parcours séquentiel lira tout le fichier (ou la moitié pour une recherche par clé). Donc ça prendra 5 secondes.
2. Une recherche par index implique 2 ou 3 accès pour parcourir l'index, et un seul accès pour lire l'enregistrement : soit $4 \times 0.01 = 0.04$ s, (4 millisecondes).

En gros, c'est mille fois plus cher. Ces chiffres sont bien sûr à pondérer par le fait qu'une bonne partie des pages du fichier peuvent être déjà en mémoire. □

4.2.2 Quand doit-on utiliser un index ?

L'accès direct suppose (sauf cas exceptionnel où on connaît l'adresse d'un enregistrement) qu'il existe un index ou une table de hachage sur la table permettant d'obtenir les adresses en fonction du critère de recherche. Pour être complet il faut signaler qu'il existe une troisième méthode, la recherche par dichotomie, mais elle suppose que le fichier est trié sur les attributs sur lesquels s'effectue la recherche, ce qui est difficile à satisfaire en pratique.

Le choix d'utiliser le parcours séquentiel ou l'accès par index peut sembler trivial : on regarde si un index est disponible, et si oui on l'utilise comme chemin d'accès. En fait ce choix est légèrement compliqué par les considérations suivantes :

1. Le critère de recherche porte-t-il sur un ou sur plusieurs attributs ? S'il y a plusieurs attributs, les critères sont-ils combinés par des **and** ou des **or** ?
2. Quelle est la sélectivité de la recherche ? On constate que quand une partie significative de la table est sélectionnée, il devient inutile, voire contre-performant, d'utiliser un index.

Le cas réellement trivial est celui – fréquent – d'une recherche avec un critère d'égalité sur la clé primaire (ou plus généralement sur un attribut indexé par un index unique). Dans ce cas l'utilisation de l'index ne se discute pas. Exemple :

```
SELECT *
FROM   Film
WHERE  idFilm = 100
```

Dans beaucoup d'autres situations les choses sont un peu plus subtiles. Le cas le plus délicat – car le plus fréquemment rencontré – est celui d'une recherche par intervalle sur un champ indexé.

Cas des recherches par intervalle

Voici un exemple simple de requête dont l'optimisation n'est pas évidente a priori. Il s'agit d'une recherche par intervalle (comme toute sélection avec $<$, $>$, ou une recherche par préfixe exprimée avec LIKE).

```
SELECT *
FROM   Film
WHERE  idFilm BETWEEN 100 AND 1000
```

L'utilisation d'un index n'est pas toujours appropriée dans ce cas, comme le montre le petit exemple de la figure 4.2. Dans cet exemple, le fichier a quatre pages, et les enregistrements sont identifiés (clé unique) par un numéro. On peut noter que le fichier n'est pas ordonné sur la clé (il n'y a aucune raison a priori pour que ce soit le cas).

L'index en revanche s'appuie sur l'ordre des clés (il s'agit ici typiquement d'un arbre B+, voir chapitre 2). À chaque valeur de clé dans l'index est associé un pointeur (une adresse) qui désigne l'enregistrement dans le fichier.

Maintenant supposons :

1. que l'on effectue une recherche par intervalle pour ramener tous les enregistrements entre 9 et 13 ;
2. que la mémoire centrale disponible soit de trois pages.

Si on choisit d'utiliser l'index, comme semble y inviter le fait que le critère de recherche porte sur la clé primaire, on va procéder en deux étapes.

1. **Étape 1** : on récupère dans l'index toutes les valeurs de clé comprises entre 9 et 13.

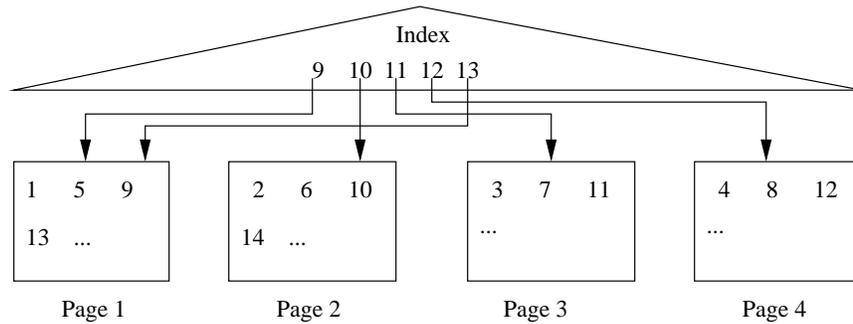


FIGURE 4.2 – Recherche par intervalle avec un index

2. **Étape 2** : pour chaque valeur obtenue dans l'étape 1, on prend le pointeur associé, on lit le bloc dans le fichier et on en extrait l'enregistrement.

Donc on va lire la page 1 pour l'enregistrement 9, puis la page 2 pour l'enregistrement 10, puis la page 3 pour l'enregistrement 11. À ce moment-là la mémoire (trois pages est pleine). Quand on lit la page 4 pour y prendre l'enregistrement 12, on va replacer sur disque la page la plus anciennement utilisée, à savoir la page 1. Pour finir, on doit relire, *sur le disque*, la page 1 pour l'enregistrement 13. Au total on a effectué 5 lectures, alors qu'un simple balayage du fichier se serait contenté de 4.

Cette petite démonstration s'appuie sur des ordres de grandeurs qui ne sont clairement pas représentatifs d'une vraie base de données. Elle montre simplement que les accès au fichier, à partir d'un index de type arbre B+, sont anarchiques et peuvent conduire à lire plusieurs fois la même page. Même sans cela, des recherches par adresses intenses mènent à déclencher des opérations d'accès à une page pour lire un enregistrement à chaque fois, ce qui s'avère très pénalisant.

Sélectivité

Le seul moyen pour l'optimiseur de déterminer la bonne technique est de disposer de *statistiques* sur la sélectivité des attributs, afin de pouvoir estimer, pour une sélection donnée, la partie d'un fichier concernée par cette sélection.

La *sélectivité* d'un attribut A d'une table R (notée $S(R, A)$) est le rapport entre le nombre d'enregistrements pour lesquels A a une valeur donnée, et le nombre total d'enregistrements. Cette définition prend l'hypothèse que la répartition de valeurs de A est uniforme, ce qui est loin d'être toujours le cas. Admettons pour l'instant cette hypothèse. La sélectivité de A est calculée de la manière suivante :

1. Soit $nbVals$ le nombre de valeurs distinctes dans R , alors le nombre d'enregistrements ayant une valeur donnée est

$$nbRecs = \frac{|R|}{nbVals}$$

2. la sélectivité de A est donc

$$\frac{nbRecs}{|R|}$$

ce qui revient aussi à $1/nbVals$.

Si est une clé unique, on a $nbVals = |R|$, $nbRecs = 1$, et la sélectivité est égale à $1/R$. Il s'agit du cas où l'attribut est le plus sélectif. Si au contraire on a un attribut dont les seules valeurs possibles sont « Oui » ou « Non », on aura $nbVals = 2$, $nbrecs = |R|/2$, et la sélectivité est égale à $\frac{1}{2}$. L'attribut est très peu sélectif.

Si un optimiseur ne dispose pas de la sélectivité d'un attribut, il utilisera l'index, ce qui donnera, dans l'exemple précédent, des résultats catastrophiques. Il est donc essentiel, d'abord de faire attention quand on

créer des index à utiliser des attributs ayant une sélectivité raisonnable, ensuite de récolter des statistiques sur la base pour fournir à l'optimiseur des informations permettant de le guider dans le choix d'utiliser ou non un index.

Revenons maintenant sur l'hypothèse d'uniformité des valeurs sur laquelle s'appuient les calculs précédents. Si cette hypothèse n'est pas vérifiée en pratique, cela peut mener l'optimiseur à commettre des mauvais choix. Le fait par exemple d'avoir 1 % de « Oui » et 99 % de « Non » devrait l'inciter à utiliser l'index pour une recherche sur les « Oui », et un balayage pour une recherche sur les « Non ».

Une technique plus sophistiquée consiste à gérer des *histogrammes* décrivant la répartition des valeurs d'un attribut et permettant d'estimer, pour un intervalle donné, la partie de la table qui sera sélectionnée. Le principe d'un histogramme est de découper les enregistrements en n groupes, chaque groupe contenant des valeurs proches. Il existe deux types d'histogrammes :

- Les histogrammes *en hauteur* : les valeurs prises par un attribut sont triées, puis divisées en n intervalles *égaux* ; à chaque intervalle I on associe alors le nombre d'enregistrements pour lesquels l'attribut analysé a une valeur appartenant à I .
- Les histogrammes *en largeur* : on trie les enregistrements sur l'attribut, et on définit alors les n intervalles de telle sorte que chacun contienne le même nombre d'enregistrements.

La figure 4.3 montre deux exemples d'histogrammes, l'un en hauteur, l'autre en largeur. Le premier nous indique par exemple qu'il y a 8 % des données entre les valeurs 70 et 80. Connaissant le nombre total de lignes dans la table, on en déduit la sélectivité de la requête portant sur cette partie des valeurs.

Les histogrammes peuvent être tenus à jour automatiquement par le système, ou créés explicitement par l'administrateur. Nous verrons dans la section 4.5 comment effectuer ces opérations sous ORACLE.

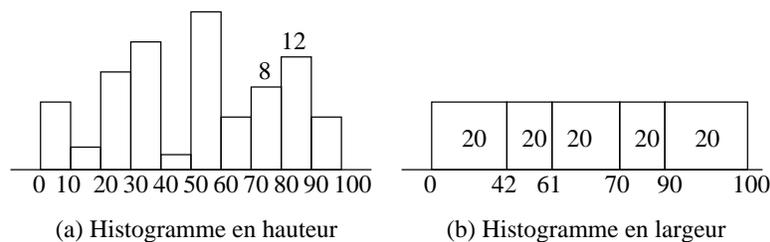


FIGURE 4.3 – Exemples d'histogrammes

Index ou pas index ? Quelques autres exemples

Voici quelques autres exemples, plus faciles à traiter.

```
SELECT *
FROM   Film
WHERE  idFilm = 20
AND    titre = 'Vertigo'
```

Ici on utilise évidemment l'index pour accéder à l'unique film (s'il existe) ayant l'identifiant 20. Puis, une fois l'enregistrement en mémoire, on vérifie que son titre est bien *Vertigo*. Voici le cas complémentaire :

```
SELECT *
FROM   Film
WHERE  idFilm = 20
OR     titre = 'Vertigo'
```

On peut utiliser l'index pour trouver le film 20, mais il faudra de toutes manières faire un parcours séquentiel pour rechercher *Vertigo*. Autant donc s'épargner la recherche par index et trouver les deux films au cours du balayage.

4.2.3 Le tri externe

Le tri d'une relation sur un ou plusieurs attributs utilise l'algorithme de tri-fusion. Celui-ci est du type «diviser pour régner»²

Voici les deux étapes de l'algorithme :

1. Découpage de la table en partitions telles que chaque partition tienne en mémoire centrale et tri de chaque partition en mémoire. On utilise en général l'algorithme de *Quicksort*.
2. Fusion des partitions triées.

Regardons en détail chacune des phases.

Phase de tri

Supposons que nous disposons pour faire le tri de M pages en mémoire. On prend un fragment constitué des M premières pages du fichier et on les charge en mémoire. On les trie alors avec *Quicksort* et on écrit le fragment trié sur le disque (figure 4.4). On recommence avec les M pages suivantes du fichier, jusqu'à ce que tout le fichier ait été lu par fragments de M . À l'issue de cette phase on a $\lceil B/M \rceil$ partitions triées, où B est le nombre de blocs du fichier.

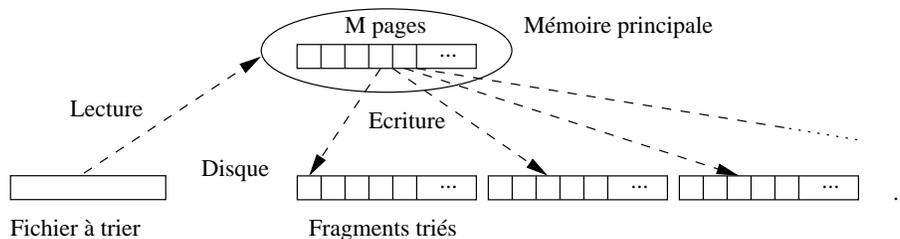


FIGURE 4.4 – Algorithme de tri-fusion : phase de tri

Phase de fusion

La phase de fusion consiste à fusionner récursivement les partitions. À chaque étape on obtient des partitions triées plus grosses, et le processus termine quand la dernière fusion délivre la relation tout entière.

Commençons par regarder comment on fusionne en mémoire centrale deux listes triées A et B . On a besoin de trois tampons. Dans les deux premiers, les deux listes à trier sont stockées. Le troisième tampon sert pour le résultat c'est-à-dire la liste résultante triée. L'algorithme est donné ci-dessous. Par convention, lorsque tous les éléments de A (B) ont été lus, l'élément suivant est égal à *eof*.

Remarquons que :

- L'algorithme *fus* se généralise facilement au cas de $M > 3$ tampons, où on fusionne en même temps $M - 1$ listes
- Si on fusionne $M - 1$ listes de taille p pages chacune, la liste résultante a une taille de $(M - 1) * p$ pages.

La première étape de la phase de fusion de la relation consiste à fusionner les $\lceil B/M \rceil$ fragments triés obtenues après la phase de tri. On prend $M - 1$ fragments à la fois, et on leur associe à chacun un tampon en mémoire, le tampon restant étant consacré à la sortie. On commence par lire le premier bloc des $M - 1$ premiers fragments dans les $M - 1$ premiers blocs, et on applique l'algorithme de fusion. Les enregistrements triés sont stockés dans un nouveau fragment sur disque.

2. Ce type d'algorithme a deux phases. Dans la première phase on décompose le problème récursivement en sous problèmes jusqu'à ce que chaque sous-problème puisse être résolu de façon simple. La deuxième phase consiste à fusionner récursivement les solutions.

```

a= premier élément de A;
b= premier élément de B;
tant qu'il reste un élément dans A ou B
{
  si a avant b
  {
    si tampon sortie T plein
    {
      vider T
    }
    écrire a dans T
    a= élément suivant dans A
  }
  sinon
  {
    si tampon sortie T plein
    {
      vider
    }
    écrire b dans T
    b= élément suivant dans B
  }
}

```

FIGURE 4.5 – Algorithme de fusion de deux listes

On continue avec les $M - 1$ blocs suivants de chaque partition, jusqu'à ce que les $M - 1$ partitions initiales aient été entièrement lues et triées. On a alors sur disque une nouvelle partition de taille $M \times (M - 1)$. On répète le processus avec les $M - 1$ partitions suivantes, etc.

À la fin de cette première étape, on obtient $\lceil \frac{B}{M*(M-1)} \rceil$ partitions triées, chacune (sauf la dernière qui est plus petite) ayant pour taille $M \times (M - 1)$ blocs.

La première phase de la fusion est résumée par l'algorithme ci-dessous en supposant que $M = 3$ et que la table R est représentée par un ensemble P de fichiers, un par fragment trié. Le résultat est un ensemble P' de fragment.

```

Tant qu'il reste deux partitions à lire dans P
{
  p= partition suivante dans P
  q= partition suivante dans P
  pour chaque bloc b dans p, bloc c dans q
  {
    lire b dans le premier tampon
    lire c dans le deuxième tampon
    fusion (b,c).
  }
  mettre la partition résultat dans P'
}
S'il reste une partition p dans P
{
  mettre p dans P'
}

```

La deuxième étape consiste à recommencer le même processus mais avec $M - 1$ fois moins de partitions chacune étant $M - 1$ fois plus grande.

La figure 4.6 résume la phase de fusion. La phase de fusion peut être représentée par un arbre, chaque noeud (agrandi à droite) correspondant à une fusion de $M - 1$ partitions.

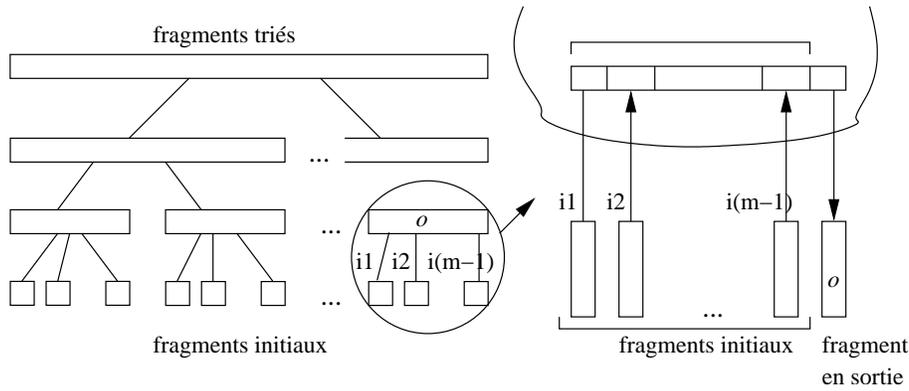


FIGURE 4.6 – Algorithme de tri-fusion : la phase de fusion

Un exemple est donné dans la figure 4.7 sur un ensemble de films qu'on trie sur le nom du film. Il y a trois phases de fusion, à partir de 6 fragments initiaux que l'on regroupe 2 à 2.

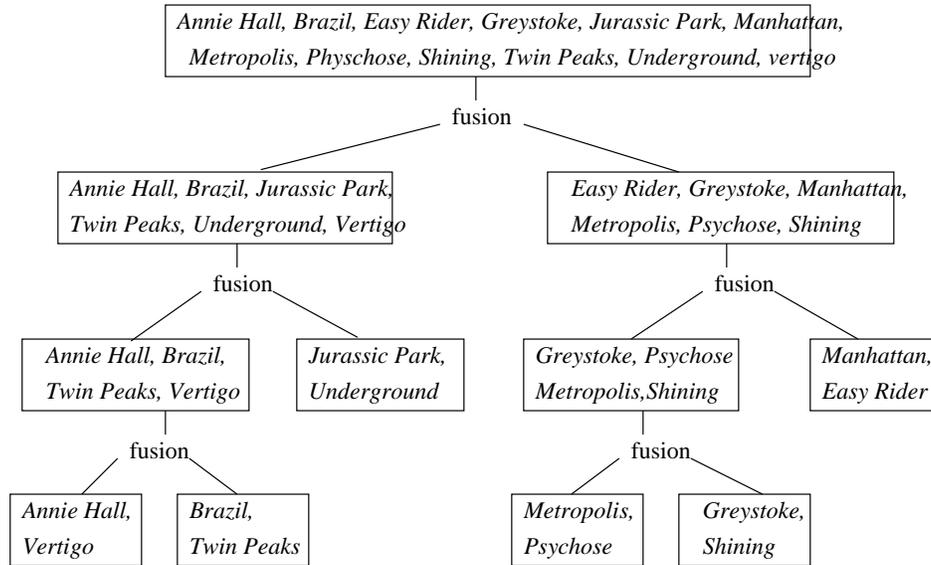


FIGURE 4.7 – tri-fusion : un exemple

Coût du tri-fusion

La phase de tri coûte B lectures et B écritures pour créer les partitions triées. À chaque étape de la phase fusion, chaque fragment est lu une fois et les nouveaux fragments créés sont $M - 1$ fois plus grands mais $M - 1$ fois moins nombreux. Par conséquent à chaque étape (pour chaque niveau de l'arbre de fusion), il y a $2 \times B$ entrées/sorties. Le nombre d'étapes c'est-à-dire le nombre de niveaux dans l'arbre est $O(\log_{M-1} B)$.

Le coût de la phase de fusion est $O(\text{Blog}_M B)$. Il prédomine celui de la phase de tri. En résumé, le coût de l'algorithme de tri-fusion est $O(\text{Blog}_M B)$.

En pratique, un niveau de fusion est en général suffisant. Idéalement, le fichier à trier tient complètement en mémoire et la phase de tri suffit pour obtenir un seul fragment trié, sans avoir à effectuer de fusion par la suite. Si possible, on peut donc chercher à allouer un nombre de page suffisant à l'espace de tri pour que tout le fichier puisse être traité en une seule passe.

Il faut bien réaliser que les performances ne s'améliorent pas de manière continue avec l'allocation de mémoire supplémentaire. En fait il existe des « seuils » qui vont entraîner un étage de fusion en plus ou en moins, avec des différences de performance notables, comme le montre l'exemple suivant.

Exemple 15. Prenons l'exemple du tri d'un fichier de 75 000 pages de 4 096 octets, soit 307 Mo. Voici quelques calculs, pour des tailles mémoires différentes.

1. **Avec une mémoire** $M > 307\text{Mo}$, tout le fichier peut être chargé et trié en mémoire. Une seule lecture suffit.
2. **Avec une mémoire** $M = 2\text{Mo}$, soit 500 pages.
 - (a) on divise le fichier en $\lceil \frac{307}{2} \rceil = 154$ fragments. Chaque fragment est trié en mémoire et stocké sur le disque.
On a lu et écrit une fois le fichier en entier, soit 614 Mo.
 - (b) On associe chaque fragment à une page en mémoire, et on effectue la fusion (noter qu'il reste $500 - 154$ pages disponibles).
On a lu encore une fois le fichier, pour un coût total de $614 + 307 = 921$ Mo.
3. **Avec une mémoire** $M = 1\text{Mo}$, soit 250 pages.
 - (a) on divise le fichier en $\lceil \frac{307}{1} \rceil = 307$ fragments. Chaque fragment est trié en mémoire et stocké sur le disque.
On a lu et écrit une fois le fichier en entier, soit 714 Mo.
 - (b) On associe les 249 premiers fragments à une page en mémoire, et on effectue la fusion (on garde la dernière page pour la sortie). On obtient un nouveau fragment F_1 de taille 249 Mo.
On prend les $307 - 249 = 58$ fragments qui restent et on les fusionne : on obtient F_2 , de taille 58 Mo.
On a lu et écrit encore une fois le fichier, pour un coût total de $614 \times 2 = 1\,228$ Mo.
 - (c) Finalement on prend les deux derniers fragments, F_1 et F_2 , et on les fusionne. Cela représente une lecture de plus, soit $1\,228 + 307 = 1\,535$ Mo.

□

Il est remarquable qu'avec seulement 2 Mo, on arrive à trier en une seule étape de fusion un fichier qui est 150 fois plus gros. Il faut faire un effort considérable d'allocation de mémoire (passer de 2 Mo à 307) pour arriver à éliminer cette étape de fusion. Noter qu'avec 300 Mo, on garde le même nombre de niveaux de fusion qu'avec 2 Mo (quelques techniques subtiles, non présentées ici, permettent quand même d'obtenir de meilleures performances dans ce cas).

En revanche, avec une mémoire de 1 Mo, on doit effectuer une étape de fusion en plus, ce qui représente plus de 700 E/S supplémentaires.

En conclusion : on doit pouvoir effectuer un tri avec une seule phase de fusion, à condition de connaître la taille des tables qui peuvent être à trier, et d'allouer une mémoire suffisante au tri.

4.3 Algorithmes de jointure

On peut classer les algorithmes de jointure en deux catégories, suivant l'absence ou la présence d'index sur les attributs de jointure. Nous allons présenter successivement trois algorithmes de la première catégorie et deux algorithmes de la seconde catégorie.

Jointure sans index

Les trois algorithmes les plus répandus sont les suivants :

1. L'algorithme le plus simple est la *jointure par boucles imbriquées*. Il est malheureusement très coûteux dès que les tables à joindre sont un tant soit peu volumineuses.
2. L'algorithme de *jointure par tri-fusion* est basé, comme son nom l'indique, sur un tri préalable des deux tables. C'est le plus ancien et le plus répandu des concurrents de l'algorithme par boucles imbriquées, auquel il se compare avantageusement dès que la taille des tables dépasse celle de la mémoire disponible.
3. Enfin la *jointure par hachage* est une technique plus récente qui donne de très bons résultats quand une des tables au moins tient en mémoire.

Jointure avec index

1. Quand une des tables est indexée sur l'attribut de jointure, on utilise une variante de l'algorithme par boucles imbriquées avec traversée d'index, dite *jointure par boucles indexée*.
2. Enfin si les deux tables sont indexées, on utilise parfois une variante du tri-fusion sur les index, mais cette technique pose quelques problèmes et nous ne l'évoquerons que brièvement.

On notera R et S les relations à joindre et T la relation résultat. Le nombre de pages est noté respectivement par B_R et B_S . Le nombre des enregistrements de chaque relation est respectivement N_R et N_S . On précisera pour chaque algorithme s'il peut être utilisé quel que soit le prédicat de jointure.

4.3.1 Jointure par boucles imbriquées

Cet algorithme s'adapte à tous les prédicats de jointure. Il consiste à énumérer tous les enregistrements dans le produit cartésien de R et S (en d'autres termes, toutes les paires possibles) et garde ceux qui satisfont le prédicat de jointure. La technique employée est un exemple de ce que nous avons appelé *technique d'itération*. On balaye l'une des deux relations, disons R , appelée *relation extérieure*. Pour chaque enregistrement r de R , on balaye entièrement l'autre relation S , appelée *relation intérieure*. Pour chaque enregistrement s de S , on compare l'attribut de jointure de s avec celui de r . Si le prédicat de jointure est satisfait, on concatène r et s dans un tampon T qu'on vide sur disque lorsqu'il est plein. La technique est résumée par la procédure *BIM* (R, S) ci-dessous.

```

Pour chaque r dans R
{
  Pour chaque s dans S
  {
    si r et s sont joignables pour donner t, alors
      si T est plein vider T sur disque,
      sinon ajouter t à T
  }
}

```

Illustrons cet algorithme sur un exemple. Soit les tables *Films* et *Artistes* de schéma :

```

Films(nomFilm, année)
Artistes(Nom, nomFilm)

```

La jointure naturelle $Films \bowtie Artistes$ par boucles imbriquées est illustrée dans la figure 4.8.

En fait l'algorithme précédent ne fonctionne que si R et S sont entièrement en mémoire centrale. Comme en général ce n'est pas le cas, on applique une variante qui effectue une première boucle sur les blocs, et une seconde, une fois les blocs en mémoire, sur les enregistrements qu'ils contiennent.

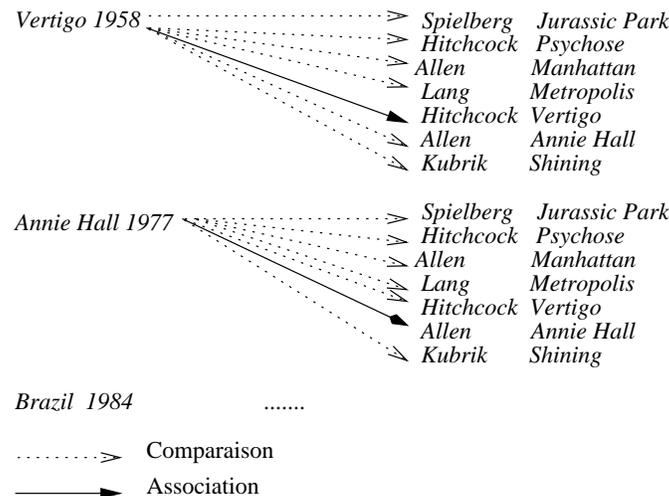


FIGURE 4.8 – Jointure par boucles imbriquées

Voici la variante la plus simple pour commencer. On lit en séquence les pages de R . Pour chaque page p de R , on lit en séquence les pages de S . Pour chaque page q de S on fait la jointure avec la page p . On applique l'algorithme BIM ci-dessus pour les enregistrements de la page p et les enregistrements de la page q qui sont dans deux pages. Lorsque la jointure entre p et q est terminée, on passe à la page suivante de S . Lorsque toutes les pages de S ont été visitées, on recommence tout le processus avec la page suivante de R .

Cet algorithme résumé ci-dessous ne nécessite donc que 3 pages en mémoire, deux pour lire une page de chaque relation et un pour le résultat de la jointure. Le coût de la jointure avec cet algorithme est de $B_R + B_R \times B_S$ lectures.

```
Pour chaque p dans pages(R) {
  lire p
  Pour chaque q dans pages(S) {
    lire q
    BIM(p,q)
  }
}
```

Maintenant on peut faire beaucoup mieux en utilisant plus de mémoire. Soit R la relation la plus petite. Si le nombre de pages M est au moins égal à $B_R + 2$, la relation R tient en mémoire centrale. On la lit dans B_R pages, et pour chaque page q de S on fait la jointure (procédure $BIM(R, q)$). La table S n'est lue qu'une seule fois. le coût est de $B_R + B_S$ lectures : d'un coût quadratique dans les tailles des relations, lorsqu'on n'a que 3 pages, on est passé à un coût linéaire.

S'il s'agit d'une équi-jointure, une variante de cet algorithme consiste à hacher R en mémoire à l'aide d'une fonction de hachage h . Alors pour chaque enregistrement de S on cherche par $h(s)$ les enregistrements de R joignables. Le coût en E/S est inchangé, mais le coût CPU est linéaire dans le nombre d'enregistrement des tables $N_R + N_S$ (alors qu'avec la procédure BIM c'est une fonction quadratique du nombre d'enregistrements).

Malheureusement il arrive souvent que R ne tient pas en mémoire : $B_R > M - 2$. Voici alors la version la plus générale de la jointure par boucles imbriquées (voir figure 4.9) : on découpe R en groupes de taille $M - 2$ pages et on utilise la variante ci-dessus pour chaque groupe. R est lue une seule fois, groupe par

groupe, S est lue $\lceil \frac{B_R}{M-2} \rceil$ fois. On obtient un coût final de :

$$B_R + \lceil \frac{B_R}{M-2} \rceil \times B_S$$

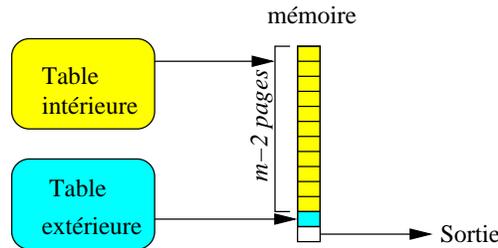


FIGURE 4.9 – Allocation mémoire dans la jointure par boucles imbriquées

Exemple 16. On prend l'exemple d'une jointure $Film \bowtie_{idMES=idArtiste} Artiste$ en supposant, pour les besoins de la cause, qu'il n'y a pas d'index. La table *Film* occupe 1 000 blocs, et la table *Artiste* 10 000 blocs. On suppose que la mémoire disponible a pour taille $M = 252$ blocs.

1. En prenant la table *Artiste* comme table extérieure, on obtient le coût suivant :

$$10\,000 + \lceil \frac{10\,000}{250} \rceil \times 1\,000 = 50\,000$$

2. Et en prenant la table *Film* comme table extérieure :

$$1\,000 + \lceil \frac{1\,000}{250} \rceil \times 10\,000 = 41\,000$$

Conclusion : il faut prendre la table la plus petite comme table extérieure. Cela suppose bien entendu que l'optimiseur dispose des statistiques suffisantes. \square

En résumé, cette technique est simple, et relativement efficace quand une des deux relations peut être découpée en un nombre limité de groupes (autrement dit, quand sa taille par rapport à la mémoire disponible reste limitée). Elle tend vite cependant à être très coûteuse en E/S, et on lui préfère donc en général la jointure par tri-fusion, ou la jointure par hachage, présentées dans ce qui suit.

4.3.2 Jointure par tri-fusion

L'algorithme de jointure par tri-fusion que nous présentons ici s'applique à l'équijointure (jointure avec égalité). C'est un exemple de technique à deux phases : la première consiste à trier les deux tables sur l'attribut de jointure (si elles ne le sont pas déjà). Ce tri facilite l'identification des paires d'enregistrement partageant la même valeur pour l'attribut de jointure.

À l'issue du tri on dispose de deux fichiers temporaires stockés sur disque³. On utilise l'algorithme de tri externe vu précédemment pour cette première étape. La deuxième phase, dite de fusion, consiste à lire page par page chacun des deux fichiers temporaires et à parcourir séquentiellement en parallèle ces deux fichiers pour trouver les enregistrements à joindre. Comme les fichiers sont triés, sauf cas exceptionnel, chaque page n'est lue qu'une fois. Regardons plus en détail la fusion.

Soit l'équijointure de R et S sur les attributs $R.A$ et $S.B$. On commence avec les premiers enregistrements r_1 et s_1 de chaque relation.

3. En fait on évite d'écrire le résultat de la dernière étape de fusion du tri, en prenant « à la volée » les enregistrements produits par l'opérateur de tri. Il s'agit d'un exemple de petites astuces qui peuvent avoir des conséquences importantes, mais dont nous omettons en général la description pour des raisons évidentes de clarté.

1. Si $r_1.A = s_1.B$, on joint les deux enregistrements, on passe au deuxième enregistrement de S et on fait le test $r_1.A = s_2.B$, etc., jusqu'à ce que l'enregistrement s_p de S soit tel que $s_p.B > r.A$. On avance alors au deuxième enregistrement de R et on fait le test $r_2.A = s_p.P$.
2. Si $r_1.A < s_1.B$ on lit le deuxième enregistrement de R et on recommence.
3. Si enfin $r_1.A > s_1.B$, on passe à l'enregistrement suivant de S .

Donc on balaie une table tant que l'attribut de jointure a une valeur inférieure à la valeur courante de l'attribut de jointure dans l'autre table. Quand il y a égalité, on fait la jointure. Ceci peut impliquer la jointure entre plusieurs enregistrements de R en séquence et plusieurs enregistrements de S en séquence. Ensuite on recommence.

Voici un pseudo-code pour cet algorithme, en supposant comme d'habitude qu'on a trois pages en mémoire, une pour lire une page de chaque table triée, et une pour le résultat.

```

tant que r différent de eof et s différent de eof
{
  p = le page de R;
  lire p;
  r = ler enregistrement de p;
  q = le page de S;
  lire q;
  s = ler enregistrement de q;
  v = ler enregistrement de q;
  tant que r.A < s.B {
    si p lu entièrement {
      p = page suivante de R;
      lire p;
      r = ler enregistrement de p;
    }
    sinon r = enregistrement suivant dans p;
  }
  tant que s.B < r.A {
    si q lu entièrement {
      q = page suivante de S;
      lire q;
      s = ler enregistrement de q;
    }
    sinon s = enregistrement suivant dans q;
  }
  v = s;
  tant que r.A = s.B {
    s = v;
    tant que s.B = r.A {
      joindre r et s, resultat: t;
      si tampon de sortie plein, vider;
      sinon mettre t dans tampon de sortie;
      s = enregistrement suivant dans q;
    }
    r = enregistrement suivant dans p;
  }
}

```

La jointure *Films* ⋈ *Artistes* par tri-fusion est illustrée dans la figure 4.10

En général, on doit parcourir un groupe d'enregistrements en séquence de S ayant même valeur pour B , autant de fois qu'il y a d'enregistrements dans R ayant la même valeur pour A (boucle *tant que* $r.A = s.B$). Si le groupe d'enregistrements de S est à cheval sur plusieurs pages, il va falloir rappeler plusieurs fois les

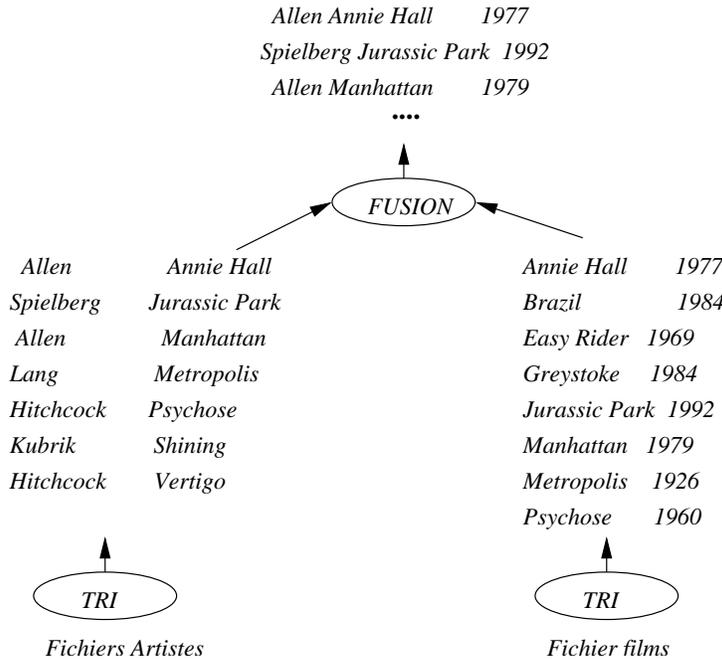


FIGURE 4.10 – Jointure par tri-fusion

mêmes pages de S , ce qui peut augmenter sensiblement le nombre de lectures disque⁴. L'algorithme ci-dessus suppose que ce cas n'arrive jamais, et que lorsqu'un groupe d'enregistrements en séquence de R est joint à un groupe d'enregistrements de S , les deux groupes sont entièrement dans les pages p et q courantes. L'hypothèse d'un parcours unique de chacune des pages des deux relations est justifiée au moins dans le cas où il n'y a pas de doublés pour l'attribut de jointure dans l'une des relations. Ce cas est extrêmement courant : par exemple lorsque l'attribut de jointure est une clé ou une clé étrangère.

Avec l'hypothèse ci-dessus, le coût de la fusion est alors de $B_R + B_S$ lectures disque (linéaire). En général, les relations n'étant pas triées, le tri domine le coût de l'algorithme de tri-fusion qui est alors :

$$O(B_R \log_M(B_R) + B_S \log_M(B_S) + B_R + B_S)$$

4.3.3 Jointure par hachage

Comme tous les algorithmes à base de hachage, cet algorithme ne peut s'appliquer qu'à une équi-jointure. Comme l'algorithme de tri-fusion (section 4.3.2, page 88), il a deux phases : une phase de partitionnement des deux relations en k partitions chacune, avec la même fonction de hachage, et une phase de jointure proprement dite.

La première phase a pour but de réduire le coût de la jointure proprement dite de la deuxième phase. Au lieu de comparer tous les enregistrements de R à tous les enregistrements de S , on ne comparera les enregistrements de chaque partition de R qu'aux enregistrements de la partition associée de S .

Le partitionnement de R se fait par hachage. Soit A et B les attributs respectifs de l'équijointure de R et S . Soit h la fonction de hachage. Un enregistrement r de R (s de S) va dans la partition $h(r.A)$ ($h(s.B)$). Les enregistrements de la partition p de R ne peuvent être joints qu'avec les enregistrements s de S tels que $h(s.B) = p$. Ces enregistrements forment la partition de S associée à la partition p de R (voir figure 4.11).

La deuxième phase consiste alors pour $i = 1, \dots, k$, à lire la partition i de R en mémoire (la partition doit tenir *entièrement* en mémoire), à lire ensuite tous les enregistrements de la partition de S associée à i ,

4. Dans le pire des cas, hautement improbable, où A dans R et B n'ont qu'une seule valeur identique dans chaque relation, il faut lire $B_R \times B_S$ pages comme dans le cas de l'algorithme de boucles imbriquées

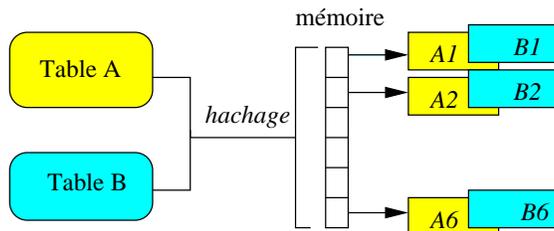


FIGURE 4.11 – Première phase de la jointure par hachage

et à comparer chacun aux enregistrements de la partition i . Ceci se fait par accès séquentiel de la partition. Noter que, alors que les enregistrements d'une partition de R doivent être tous en mémoire centrale en même temps, l'algorithme ne nécessite pas qu'une partition de S réside en mémoire. Sa taille peut donc être arbitrairement grande.

L'algorithme de jointure par hachage présenté ci-dessous suppose qu'on a $k = M - 2$ pages pour une partition de R , une page pour une partition de S et une page pour le résultat. La première phase consiste à lire séquentiellement le fichier en utilisant un balayage et à hacher chaque enregistrement dans l'un des $k = M - 2$ tampons, en vidant chacun des tampons lorsqu'il est plein dans un fichier correspondant à la partition.

La deuxième phase consiste à itérer k fois (i) la lecture d'une partition de R en mémoire centrale dans les $M - 2$ tampons dédiés, (ii) à lire page par page la partition associée de S et à faire la jointure en mémoire centrale (procédure *BIM*) de cette page avec la partition de R entière, en utilisant pour le résultat le dernier des M tampons (figure 4.12).

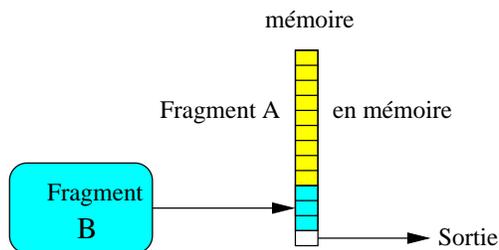


FIGURE 4.12 – Seconde phase de la jointure par hachage

```

pour chaque page p de R
{
  lire p dans premier tampon,
  pour chaque enregistrement r de p {
    si tampon h(r.A) est plein vider dans partition (R, h(r.A));
    écrire r dans tampon h(r.A)
  }
}

pour chaque page p de S
{
  lire p dans premier tampon,
  pour chaque enregistrement s de p {
    si tampon h(s.B) est plein vider dans partition (S, h(s.B));
  }
}

```

```

        écrire s dans tampon h(s.B)
    }
}

pour i=1,...,k
{
    lire partition (R,i);
    pour chaque page p de partition (S,i) {
        lire p;
        BIM(partition(R,i), p)
    }
}

```

Le coût de la première phase de partitionnement de cet algorithme est $2(B_R + B_S)$. Chaque relation est lue entièrement et hachée dans les partitions qui sont recopiées sur disque page par page. À la fin les tampons non pleins sont également recopiés sur disque dans la partition correspondante. Bien que le nombre de pages de R (S) après partitionnement (hachage) puisse être légèrement supérieur, on supposera qu'il est égal à B_R (B_S).

Le coût de la deuxième phase est de $B_R + B_S$. En effet les relations partitionnées sont lues une fois, partition par partition. Par conséquent le coût total de cet algorithme est $3(B_R + B_S)$. Noter que cet algorithme est très gourmand en mémoire. Il suppose que toute partition de la plus petite relation R tienne entièrement en mémoire centrale, c'est-à-dire ait moins de $M - 2$ pages. Cet algorithme est donc bien adapté aux jointures pour lesquelles une des relations est petite en taille. En supposant que les partitions de R sont égales en taille, celle-ci est égale à $B_R/(M - 2)$. On doit donc avoir $M \geq B_R/(M - 2) + 2$. Donc on doit avoir approximativement $M \geq \sqrt{B_R}$.

Remarques :

- il existe de nombreuses variantes d'algorithmes de jointure par hachage. Celle que nous avons présentée s'appelle dans la littérature anglo-saxonne *Grace hash join*.
- il faut prévoir le cas où la taille d'une partition dépasse $M - 2$ tampons (voir la méthode employée pour la projection par hachage).
- comme pour l'algorithme par boucles imbriquées, on peut diminuer le temps CPU de la jointure en mémoire centrale (voir le paragraphe sur les boucles imbriquées par bloc).
- si la relation R est si petite qu'elle tient en mémoire centrale ($B_R \leq M - 2$), il existe une variante de l'algorithme qui consiste (i) à partitionner R comme ci-dessus dans l'un des $M - 2$ tampons⁵ et (ii) à accéder séquentiellement S sans partitionnement préalable et à faire la jointure en mémoire centrale entre R et les enregistrements de S lus dans un tampon. Soit s un tel enregistrement. On le comparera avec les enregistrements du tampon $h(s.B)$. Pour évaluer le coût de cette variante, observons que chaque relation n'est lue qu'une fois. Le coût est alors de $B_R + B_S$.

4.3.4 Jointure avec un index

S'il existe un index I sur le ou les attributs de jointure B de l'une des relations S , on applique une variante de l'algorithme des boucles imbriquées, très efficace, et utilisée dans tous les SGBD. Elle prend comme relation extérieure la relation non-indexée (disons que c'est R) sur laquelle on fait un accès séquentiel. Pour chaque enregistrement r de R , $r.A$ sert de clé d'accès à l'index I . La traversée de I donne les adresses des enregistrements s de S qui peuvent être joints avec r , c'est-à-dire ceux pour lesquels $s.B = r.A$. Il suffit ensuite d'accéder par adresse aux enregistrements s et de les joindre avec r . On a évité le balayage de toute la relation S qui est fait pour chaque enregistrement r de R dans l'algorithme initial. On donne l'algorithme ci-dessous :

```

Pour chaque p dans pages(R)
{

```

5. La différence avec l'algorithme ci-dessus est qu'on n'a qu'une partition pour R .

```

lire p dans tampon E
Pour chaque enregistrement r dans p {
  adresses = TRAV (I,r.A)
  pour chaque a dans adresses {
    acces par adresse au enregistrement s
    t= jointure de r et s
    si tampon resultat T plein {
      vider T
    }
    écrire t dans T
  }
}
}

```

Le coût de cet algorithme est le produit (i) du nombre de pages B_R de R par (ii) le coût de la traversée d'index suivi du coût d'accès au(x) enregistrement(s) de S . Celui-ci dépend du type d'index et du nombre d'enregistrements de S , voir discussion à ce sujet dans la section 4.2.2. En résumé le coût de l'algorithme de jointure est $B_R \times O(\log(B_S))$.

Une variante de cet algorithme existe dans le cas où S est partitionnée par hachage sur la valeur de l'attribut B . Alors que la variante ci-dessus s'applique à tous les prédicats de jointure, la variante avec hachage ne marche que pour l'équijointure.

4.3.5 Jointure avec deux index

Cet algorithme pour un prédicat avec égalité est un autre exemple d'algorithme à deux phases, l'une de partitionnement, et l'autre de jointure proprement dite. Lorsque R et S ont un index sur l'attribut de jointure, comme les feuilles de ceux-ci sont triées sur cet attribut, en fusionnant les feuilles des index F_A et F_B de la même manière que pendant la phase de fusion de l'algorithme de jointure par tri-fusion (section 4.3.2), on obtient une liste de couples d'adresses d'enregistrements de R et S à joindre. Dans le pire des cas, où les deux index sont non uniques, pour chaque couple de valeurs des attributs de jointure $[a, b]$, on obtient un couple d'ensembles d'adresses des enregistrements $[A_r, B_s]$.

La deuxième phase consiste à lire les enregistrements, faire la jointure et mettre le résultat dans le tampon de sortie. Cette phase est détaillée ci-dessous dans le cas où les deux index sont denses.

```

For each a in Ar {
  lire r d'adresse a
  for each b in Bs {
    lire s d'adresse b
    si le tampon T est plein {
      vider T
    }
    joindre r et s, résultat dans T
  }
}
}

```

Algorithme FI de jointure par fusion d'index:
jointure d'un couple (index denses)

La première phase permet de déceler efficacement les jointures possibles. Le coût de cette phase de partitionnement est en général (voir discussion sur la jointure par tri-fusion) la somme du nombre de feuilles F_A et du nombre de feuilles F_B . Cet algorithme est très intéressant dans le cas où la deuxième phase d'accès aux enregistrements de l'une ou des deux relations n'est pas nécessaire. Le coût de la deuxième phase dépend de l'index et de la taille du résultat c'est-à-dire des produits cartésiens. En particulier si les deux index sont denses et non uniques, non seulement on accède à de nombreuses pages, mais de plus on risque de lire plusieurs fois les mêmes pages.

Pour éviter cette lecture multiple des mêmes pages, plusieurs techniques existent. Par exemple si le nombre de tampons disponibles est grand, il se peut qu'une page déjà lue soit encore en mémoire centrale (voir discussion de la section 4.2.2). On peut également essayer de trier les pages à lire. Ceci demande de faire le produit cartésien des couples d'adresses et ensuite de trier les couples d'adresses des enregistrements obtenus sur les pages de la première relation pour regrouper les enregistrements à lire dans la deuxième relation par page de la première relation. Cette dernière amélioration n'empêche pas des lectures multiples d'une même page de la deuxième relation. Pour toutes ces raisons, beaucoup de SGBD (dont ORACLE), en présence d'index sur l'attribut de jointure dans les deux relations, préfèrent l'algorithme par boucles imbriquées et traversée d'index à cet algorithme.

Concluons cette section avec deux remarques :

1. Excepté les algorithmes basés sur une boucle imbriquée avec ou sans index, les algorithmes montrés ont été conçus pour le prédicat d'égalité. Observons qu'une thétajointure pour d'autres prédicats peut être traduite par une équijointure ou un produit suivi d'une sélection. Naturellement, indépendamment de l'algorithme, le nombre des enregistrements du résultat est vraisemblablement beaucoup plus important pour de telles jointures que dans le cas d'égalité.
2. Cette section a montré que l'éventail des algorithmes de jointure est très large et que le choix d'une méthode efficace n'est pas simple. Il dépend notamment de la taille des relations, des méthodes d'accès disponibles et de la taille disponible en mémoire centrale. Ce choix est cependant fondamental parce qu'il a un impact considérable sur les performances. La différence entre deux algorithmes peut dans certains cas atteindre plusieurs ordres de grandeur.

4.4 Compilation d'une requête et optimisation

Cette section est consacrée à la tâche d'optimisation proprement dite : comment, à partir d'une requête SQL, déterminer le meilleur programme pour évaluer cette requête ? Nous présentons successivement la traduction de la requête SQL en langage algébrique représentant les opérations nécessaires, puis les réécritures symboliques qui organisent ces opérations de la manière la plus efficace.

La notion de *plans d'exécution* est ensuite détaillée : ces plans sont des programmes en forme d'arbre constitués d'opérateurs physique (les nœuds) échangeant des données (les arêtes). Nous décrivons le fonctionnement d'un plan d'exécution, discutons de leurs propriétés, et montrons quelques exemples de plans possibles pour une même requête. La dernière partie de cette section propose quelques algorithmes et fonctions de coût pour choisir un plan parmi les candidats.

4.4.1 Décomposition en bloc

Nous supposons qu'une requête SQL est décomposée en une collection de *blocs*. L'optimiseur se concentre sur l'optimisation d'un bloc à la fois. Un bloc est une requête sans imbrication avec une seule clause SELECT, une seule clause FROM et au plus une clause WHERE, une clause GROUP BY et une clause HAVING⁶. La décomposition en blocs est nécessaire à cause des requêtes imbriquées. Toute requête SQL ayant des imbrications peut être décomposée en une collection de blocs. Considérons par exemple la requête suivante qui calcule le film le mieux ancien :

```
SELECT titre
FROM   Film
WHERE  annee = (SELECT MIN (annee)
                FROM Film)
```

On peut décomposer cette requête en deux blocs : le premier calcule l'année minimale A . Le deuxième bloc calcule le(s) film(s) paru en A grâce à une référence au premier bloc.

```
SELECT titre
FROM   Film
WHERE  annee = (référence au bloc imbriqué)
```

6. Pour l'instant nous laissons de côté ces deux dernières clauses.

Bien entendu cette méthode peut s'avérer très inefficace et il est préférable de transformer la requête avec imbrication en une requête équivalente sans imbrication (un seul bloc) quand cette équivalence existe. Malheureusement de nombreux systèmes relationnels sont incapables de déceler ce type d'équivalence.

Prenons l'exemple de la requête suivante : « Dans quel film paru en 1958 joue James Stewart » (vous avez sans doute deviné qu'il s'agit de *Vertigo*) ? Voici comment on peut exprimer la requête SQL.

```
SELECT titre
FROM   Film f, Role r, Artiste a
WHERE  a.nom = 'Stewart' AND a.prenom='James'
AND    f.idFilm = r.idFilm
AND    r.idActeur = a.idArtiste
AND    f.annee = 1958
```

Cette requête est en un seul « bloc », mais il est tout à fait possible – question de style ? – de l'écrire de la manière suivante :

```
SELECT titre
FROM   Film f, Role r
WHERE  f.idFilm = r.idFilm
AND    f.annee = 1958
AND    r.idActeur IN (SELECT idArtiste
                      FROM Artiste
                      WHERE nom='Stewart'
                      AND prenom='James')
```

Au lieu d'utiliser IN, on peut effectuer une requête *corrélée* avec EXISTS.

```
SELECT titre
FROM   Film f, Role r
WHERE  f.idFilm = r.idFilm
AND    f.annee = 1958
AND    EXISTS (SELECT 'x'
               FROM Artiste a
               WHERE nom='Stewart'
               AND prenom='James'
               AND r.idActeur = a.idArtiste)
```

Encore mieux (ou pire), on peut utiliser deux imbrications :

```
SELECT titre FROM Film
WHERE annee = 1958
AND idFilm IN
  (SELECT idFilm FROM Role
   WHERE idActeur IN (SELECT idArtiste
                     FROM Artiste
                     WHERE nom='Stewart'
                     AND prenom='James'))
```

Dans ce dernier cas on a trois blocs. La requête est peut-être facile à comprendre, mais le système a très peu de choix sur l'exécution : on doit parcourir tous les films parus en 1958, pour chacun on prend tous les rôles, et pour chacun de ces rôles on va voir s'il s'agit bien de James Stewart.

S'il n'y a pas d'index sur le champ *annee* de *Film*, il faudra balayer *toute la table*, puis pour chaque film, c'est la catastrophe : il faut parcourir tous les rôles pour garder ceux du film courant (notez qu'il n'existe pas d'index sur les films pour accéder à *Role*). Enfin pour chacun de ces rôles il faut utiliser

l'index sur *Artiste*. Telle quelle, cette dernière syntaxe a toutes les chances d'être extrêmement coûteuse à évaluer.

Or il existe un plan bien meilleur (lequel ?) mais le système ne peut le trouver que s'il a des degrés de liberté suffisants, autrement dit quand la requête est « à plat », en un seul bloc. Il est donc recommandé de limiter l'emploi des requêtes imbriquées à de petites tables dont on est sûr qu'elles résident en mémoire.

4.4.2 Traduction et réécriture

Nous nous concentrons maintenant sur le traitement d'un bloc, étant entendu que ce traitement doit être effectué autant de fois qu'il y a de blocs dans une requête. Il comprend plusieurs phases. Tout d'abord une analyse syntaxique est effectuée, puis une traduction algébrique permettant d'exprimer la requête sous la forme d'un ensemble d'opérations sur les tables. Enfin l'optimisation consiste à trouver les meilleurs chemins d'accès aux données et à choisir les meilleurs algorithmes possibles pour effectuer ces opérations.

Analyse syntaxique

L'analyse syntaxique vérifie la validité (syntaxique) de la requête. On vérifie notamment l'existence des relations (arguments de la clause FROM) et des attributs (clauses SELECT et WHERE). On vérifie également la correction grammaticale (notamment de la clause WHERE). D'autres transformations sémantiques simples sont faites au delà de l'analyse syntaxique. Par exemple, on peut détecter des contradictions comme *année = 1998 and année = 2003*. Enfin un certain nombre de simplifications sont effectuées.

Traduction algébrique

L'étape suivante du traitement d'une requête consiste à la traduire en une expression algébrique E . Voici l'expression algébrique correspondant à la requête précédente. Nous allons prendre pour commencer une requête un peu plus simple que la précédente : donner le titre du film paru en 1958, où l'un des acteurs joue le rôle de John Ferguson (rassurez-vous c'est toujours *Vertigo*). Voici la requête SQL :

```
SELECT titre
FROM   Film f, Role r
WHERE  nomRole = 'John Ferguson'
AND    f.idFilm = r.idFilm
AND    f.annee = 1958
```

Cette requête correspond aux opérations suivantes : une *jointure* entre les rôles et les films, une *sélection* sur les films (année=1958), une *sélection* sur les rôles ('John Ferguson), enfin une *projection* pour éliminer les colonnes non désirées. La combinaison de ces opérations donne l'expression algébrique suivante :

$$\pi_{\text{titre}}(\sigma_{\text{annee}=1958}(\text{Film}) \bowtie_{\text{idFilm}=\text{idFilm}} (\sigma_{\text{nomRole}='John Ferguson'}(\text{Role}))$$

Cette expression comprend des opérations unaires (un seul argument) et des opérations binaires. On peut la représenter sous la forme d'un arbre (figure 4.13), ou *Plan d'Exécution Logique* (PEL). C'est en fait un arbre représentant l'expression algébrique équivalente à la requête SQL⁷. Dans l'arbre, les feuilles représentent les tables arguments de l'expression algébrique. Les nœuds internes correspondent aux opérateurs algébriques. Un arc entre un nœud n et son nœud père p correspond à la relation résultat de l'opération n et argument d'entrée de l'opération p .

L'interprétation de l'arbre est la suivante. On commence par exécuter les opérations sur les feuilles (ici les sélections) ; sur le résultat, on effectue les opérations correspondant aux nœuds de plus haut niveau (ici une jointure), et ainsi de suite, jusqu'à ce qu'on obtienne le résultat (ici après la projection). Cette interprétation est bien sûr rendue possible par le fait que tout opérateur prend une table en entrée et produit une table en sortie.

7. Toute requête SQL a une expression algébrique équivalente.

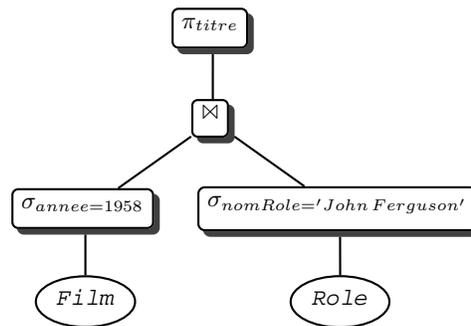


FIGURE 4.13 – Expression algébrique sous forme arborescente

Lois algébriques

On peut améliorer le PEL obtenu par traduction de la requête SQL grâce à l'existence de propriétés sur les expressions de l'algèbre relationnelle. Ces propriétés appelées *lois algébriques* ou encore *règles de réécriture* permettent de transformer l'expression algébrique en une expression équivalente et donc de réagencer l'arbre. Le PEL obtenu est équivalent, c'est-à-dire qu'il conduit au même résultat. En transformant les PEL grâce à ces règles, on peut ainsi obtenir des PEL qui s'exécutent plus rapidement. Voici la liste des règles de réécriture les plus importantes :

1. **Commutativité des jointures :**

$$R \bowtie S \equiv S \bowtie R$$

2. **Associativité des jointures :**

$$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$$

3. **Regroupement des sélections :**

$$\sigma_{A='a' \wedge B='b'}(R) \equiv \sigma_{A='a'}(\sigma_{B='b'}(R))$$

4. **Commutativité de la sélection et de la projection**

$$\pi_{A_1, A_2, \dots, A_p}(\sigma_{A_i='a'}(R)) \equiv \sigma_{A_i='a'}(\pi_{A_1, A_2, \dots, A_p}(R)), i \in \{1, \dots, p\}$$

5. **Commutativité de la sélection et de la jointure.**

$$\sigma_{A='a'}(R(\dots A \dots) \bowtie S) \equiv \sigma_{A='a'}(R) \bowtie S$$

6. **Distributivité de la sélection sur l'union.**

$$\sigma_{A='a'}(R \cup S) \equiv \sigma_{A='a'}(R) \cup \sigma_{A='a'}(S)$$

NB : valable aussi pour la différence.

7. **Commutativité de la projection et de la jointure**

$$\begin{aligned} \pi_{A_1 \dots A_p B_1 \dots B_q}(R \bowtie_{A_i=B_j} S) &\equiv \\ \pi_{A_1 \dots A_p}(R) \bowtie_{A_i=B_j} \pi_{B_1 \dots B_q}(S) & \quad i \in \{1, \dots, p\}, j \in \{1, \dots, q\} \end{aligned}$$

8. **Distributivité de la projection sur l'union** $\pi_{A_1 A_2 \dots A_p}(R \cup S) \equiv \pi_{A_1 A_2 \dots A_p}(R) \cup \pi_{A_1 A_2 \dots A_p}(S)$

Ces règles simples sont très utiles. Par exemple la règle 3 permet des sélections très efficaces. En effet si la relation est indexée sur l'attribut B , la règle justifie de filtrer sur A seulement les enregistrements satisfaisant le critère $B = b$ obtenus par traversée d'index. La commutativité de la projection avec la sélection et la jointure (règles 4 et 7) d'une part et de la sélection et de la jointure d'autre part (règle 5) permettent de faire les sélections et les projections d'abord pour *réduire les tailles des relations manipulées*, ce qui est l'idée de base pour le choix d'un « meilleur » PEL. En effet nous avons vu que l'efficacité des algorithmes implantant les opérations algébriques est très sensible à la taille des relations en entrée. Ceci est plus particulièrement vrai pour la jointure qui est une opération chère. Donc quand une séquence comporte une jointure et une sélection, il est préférable de faire la sélection d'abord : on réduit ainsi la taille d'une ou des deux relations à joindre, ce qui peut avoir un impact considérable sur le temps de traitement de la jointure.

« Pousser » les sélections le plus bas possible dans l'arbre, c'est-à-dire essayer de les appliquer le plus rapidement possible et éliminer par projection les attributs non nécessaires pour obtenir le résultat de la requête sont donc les deux idées pour transformer un PEL en un PEL équivalent meilleur.

Voici un algorithme simple résumant ces idées :

1. Séparer les sélections avec plusieurs prédicats en plusieurs sélections à un prédicat (règle 3).
2. Descendre les sélections le plus bas possible dans l'arbre (règles 4, 5, 6)
3. Regrouper les sélections sur une même relation (règle 3).
4. Descendre les projections le plus bas possible (règles 7 et 8).
5. Regrouper les projections sur une même relation.

Un exemple

Reprenons notre requête cherchant le film paru en 1958 dans lequel figure James Stewart. Voici l'expression algébrique complète. Notez qu'on a groupé toutes les sélections en une, placée *après* les jointures.

$$\pi_{titre}(\sigma_{annee=1958 \text{ and } nom='Stewart' \text{ and } prenom='James'}(Film \bowtie_{idFilm=idFilm} (Role \bowtie Artiste)))$$

L'expression est correcte, mais évidemment inutilement coûteuse à évaluer. Appliquons notre algorithme. La première étape donne l'expression suivante :

$$\pi_{titre}(\sigma_{annee=1958}(\sigma_{nom='Stewart'}(\sigma_{prenom='James'}(Film \bowtie_{idFilm=idFilm} (Role \bowtie Artiste))))))$$

On a donc séparé les sélections. Maintenant on les descend dans l'arbre :

$$\pi_{titre}(\sigma_{annee=1958}(Film) \bowtie_{idFilm=idFilm} (Role \bowtie \sigma_{nom='Stewart'}(\sigma_{prenom='James'}(Artiste))))$$

Finalement il reste à ajouter des projections pour limiter la taille des enregistrements. Pour conclure deux remarques sont nécessaires :

1. le principe « sélection avant jointure » conduit dans la plupart des cas à un PEL plus efficace. Mais il peut arriver (rarement) que la jointure soit plus réductrice en taille et que la stratégie « jointure d'abord, sélection ensuite », conduise à un meilleur PEL.
2. cette optimisation du PEL, si elle est nécessaire, est loin d'être suffisante. Il faut ensuite choisir le « meilleur » algorithme pour chaque opération du PEL. Ce choix va dépendre des chemins d'accès et des statistiques sur les tables de la base et bien entendu des algorithmes d'évaluation implantés dans le noyau. Le PEL est alors transformé en un plan d'exécution physique.

Cette transformation constitue la dernière étape de l'optimisation. Elle fait l'objet de la section suivante.

4.4.3 Plans d'exécution

Un plan d'exécution physique (PEP) est une séquence d'opérations (on parle d'*algèbre physique*) propres au SGBD⁸. Dans la section 4.2 nous avons vu les algorithmes implantant les principales opérations. Nous redonnons ici une liste d'opérateurs physiques qu'on retrouve dans tous les SGBD.

Choix d'un plan d'exécution

Le choix du PEP dépend de nombreux facteurs : chemin d'accès, statistiques, nombre de pages en mémoire centrale. Typiquement en fonction de ces paramètres, l'optimiseur choisit, pour chaque nœud du PEL, une opération physique ou une séquence d'opérations. La première difficulté vient du grand nombre de paramètres. Une autre difficulté vient du fait que le choix d'un algorithme pour un nœud du PEL peut avoir un impact sur le choix d'un algorithme pour le nœud suivant dans le PEL. Nous ne regarderons pas ce problème. Disons pour simplifier que la connaissance ou l'estimation de la taille du résultat d'une opération

8. Il n'y a pas d'interface standard, même si tous les éditeurs ont à peu près les mêmes stratégies d'évaluation de requêtes et implantent des algorithmes similaires pour les opérations algébriques.

est utile pour le choix de l'opération suivante. Par ailleurs un autre problème difficile est la répartition de la mémoire disponible entre les différentes opérations (voir section 4.1.5).

La connaissance des chemins d'accès aux tables restreint le choix d'opérations physiques pour une opération donnée. Pour faire un choix définitif, l'optimiseur compare une estimation des coûts des opérations restantes. L'estimation du coût d'une opération physique utilise un *modèle de coût* qui dépend de l'algorithme et qui a pour paramètres des grandeurs dont la valeur est soit connue et stockée ou bien *estimée*. De telles grandeurs sont la taille des relations ou la sélectivité d'un attribut.

Par exemple pour faire une sélection avec un critère $A \geq a$, sachant qu'il existe un index dense sur A , on comparera une estimation du coût de la sélection par balayage séquentiel à une estimation du coût par traversée de l'index. Ces estimations utilisent la taille exacte de la relation. Nous illustrerons la méthode dans ce cas simple et étudierons ensuite successivement une stratégie simplifiée de choix d'un opérateur physique pour la sélection et pour la jointure.

Plan d'exécution physique (PEP)

Un plan d'exécution physique est le résultat de la phase d'optimisation d'une requête. Il contient les opérateurs physiques choisis pour évaluer la requête ainsi que l'ordre dans lequel exécuter ces opérateurs (sauf cas exceptionnel, le plan comprend plusieurs opérateurs). Le plan contient aussi des détails comme le chemin d'accès aux tables et index, et si une relation doit être triée. Ce plan est représenté par un arbre dont les feuilles sont les chemins d'accès aux tables et index, et les nœuds internes sont les opérateurs physiques. Les feuilles représentent les opérations effectuées en premier. L'arc entre un nœud et son nœud parent représente le flot de données entre deux opérations en séquence : o et o' . Ce flot est le résultat de o qui sert de source (entrée) pour l'opération o' . Le résultat final est le flot de données qui sort de la racine de l'arbre.

On peut noter qu'une opération algébrique peut donner naissance à plusieurs opérations physiques. La jointure (par boucles imbriquées, par tri-fusion) est un exemple typique. Inversement, une expression algébrique de plusieurs opérations peut être implantée par une seule opération physique : par exemple le parcours séquentiel d'une table permet l'exécution d'une sélection et d'une projection.

Quelques exemples

La figure 4.14 donne une notation pour quelques opérateurs physiques courants. Cette notation sera utilisée dans les exemples de plans d'exécution physique plus bas.

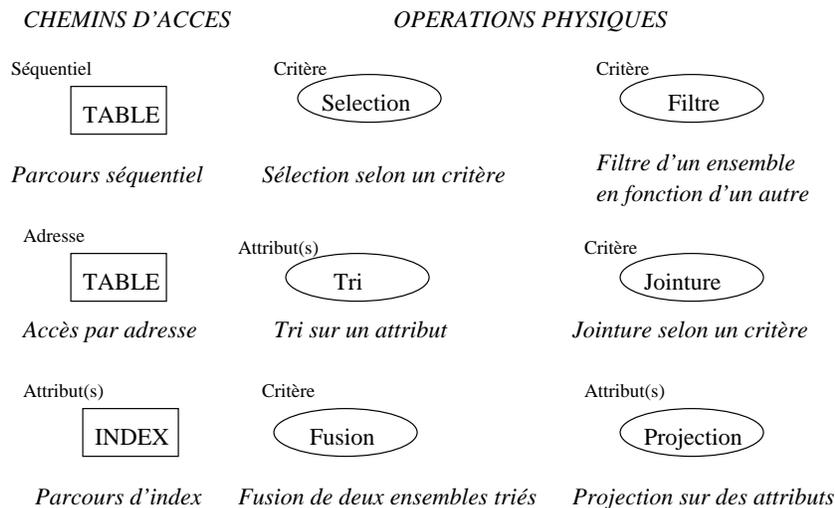


FIGURE 4.14 – Algèbre physique

On distingue les opérations d'accès à des tables ou index stockés dans des fichiers, des opérations de manipulation de données. Les opérations d'accès à des fichiers sont représentées dans les arbres par des rectangles (feuilles du PEP), le nom du fichier étant indiqué à l'intérieur du rectangle. La première est le balayage séquentiel. La deuxième est l'accès par adresse, c'est-à-dire l'accès à un ou plusieurs enregistrements connaissant son (leur) adresse. Le troisième enfin est la traversée d'index. On indique le ou les attribut(s) clé(s).

Les opérations de manipulation sont représentées par des ellipses. L'étiquette à l'intérieur de l'ellipse est le nom de l'opération. Les données en entrée sont obtenues par les branches qui descendent du nœud dans l'arbre. Ce sont soit le résultat de l'opération d'accès, soit le résultat d'une opération fille dans l'arbre. Un paramètre éventuel est indiqué en haut à gauche de l'ellipse (par exemple le(s) attribut(s) sur le(s)quel(s) est fait le tri). On reconnaît les algorithmes décrits dans la section 4.2. Le filtre est une opération utile qui ne sera pas détaillée ici. La jointure correspond à l'algorithme de boucles imbriquées. Cette liste est juste indicative et loin d'être complète (il y manque notamment la jointure par hachage).

Voici maintenant quelques exemples de plans d'exécution, en fonction d'hypothèses sur l'organisation de la base. Le premier plan d'exécution (figure 4.15) suppose qu'il n'y a pas d'index sur les attributs (nom, prénom) des artistes. On en est donc réduit à effectuer un balayage séquentiel de la table *Artiste*. En revanche il est possible, connaissant un artiste, d'utiliser l'index sur *Rôle* pour accéder à tous les rôles de cet artiste. On utilise donc l'algorithme de boucles imbriquées indexées. Même remarque pour la seconde jointure : on connaît l'identifiant du film (il est dans l'enregistrement décrivant un rôle) donc on peut utiliser l'index. Notez que les sélections se font systématiquement « en bas », au moment où on accède à la table, soit en balayage, soit en accès direct.

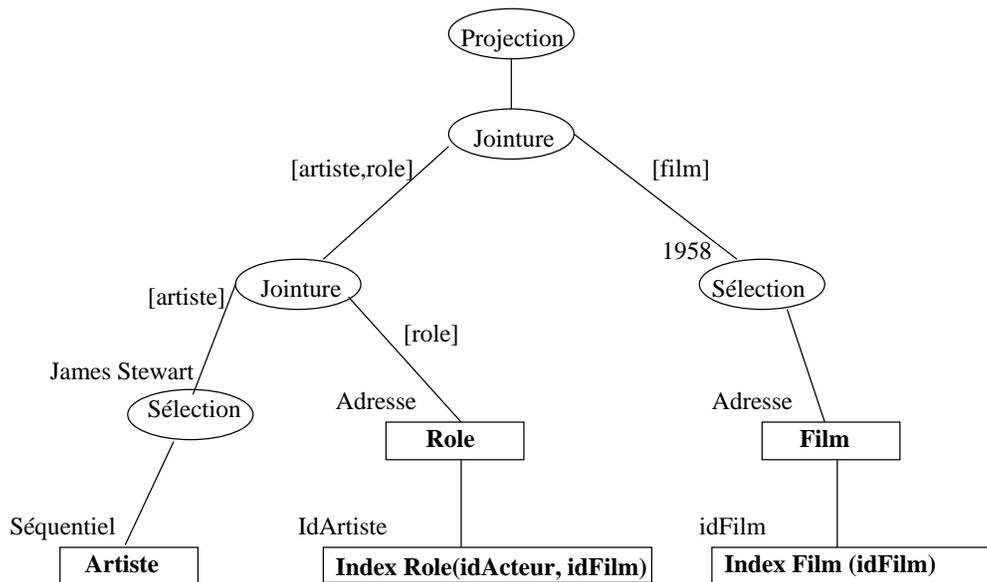


FIGURE 4.15 – Plan d'exécution physique, sans index sur nom et prénom

La figure 4.16 montre le plan d'exécution dans le cas où un index existe sur les nom et prénom des artistes. Ce plan est certainement optimal puisque on peut utiliser un index sur chaque table.

Enfin, quand il n'y a pas d'index, les sélections sont faites par balayage séquentiel et les jointures par tri-fusion. La table *Film* est balayée séquentiellement. La sélection garde seulement les films parus en 1958 qui sont ensuite triés sur l'attribut *idFilm*. La table *Rôle* est également balayée séquentiellement et triée sur l'attribut *idFilm*. Les fichiers résultant du tri sont fusionnés et triés sur l'attribut *idActeur*. Le fichier résultant est fusionné avec les artistes nommés James Stewart, eux-mêmes triés sur l'attribut *idArtiste* et obtenus par balayage séquentiel et sélection de la table *Artiste*.

Notons que :

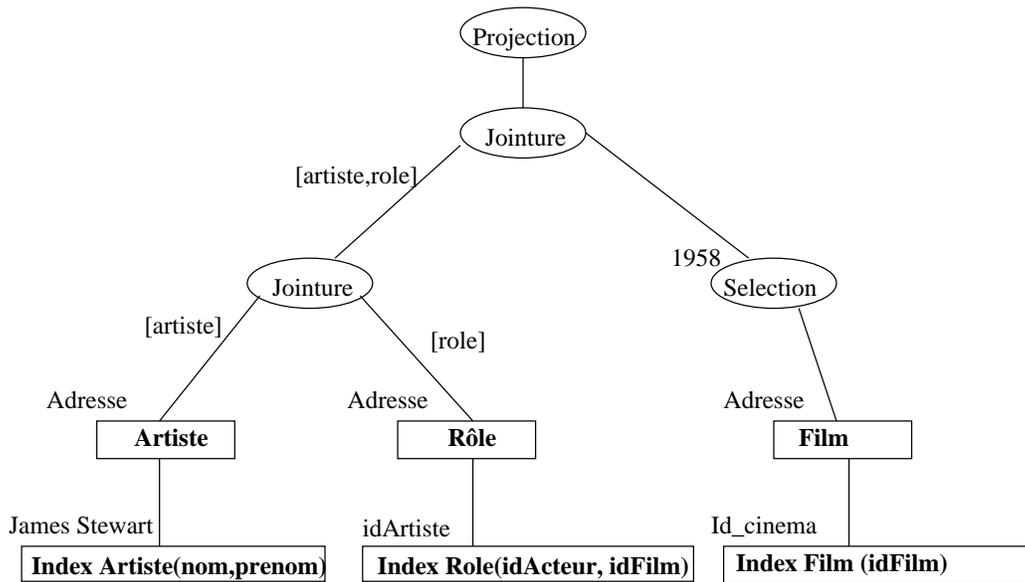


FIGURE 4.16 – Plan d’exécution physique, avec index sur nom et prénom

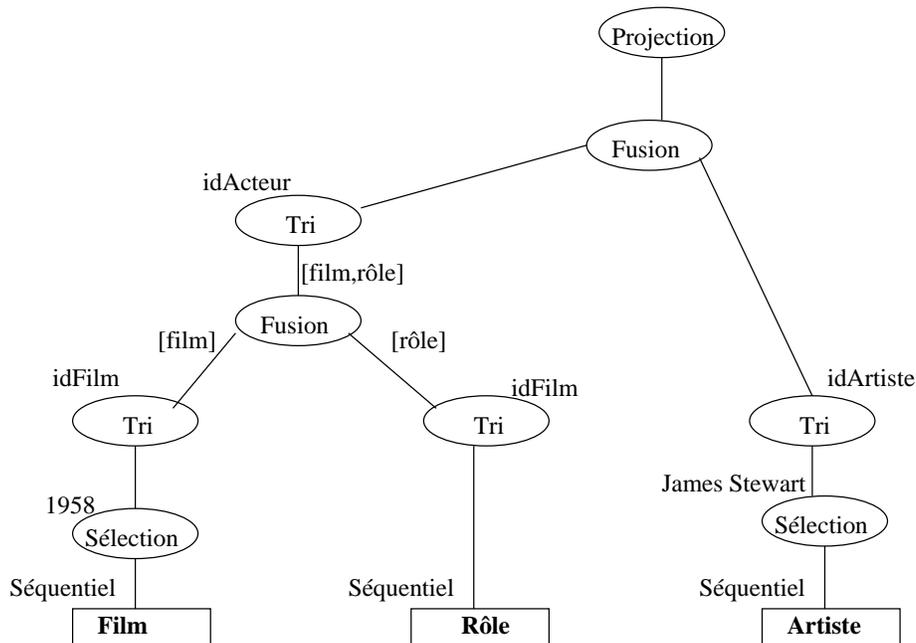


FIGURE 4.17 – plan d’exécution physique, sans aucun index

1. l’opération algébrique de sélection est chaque fois traduite par un balayage séquentiel (algorithme *Selbal* de la section 4.2. Même si les deux opérations d’accès et de sélection sont faites en même temps, on garde par souci de généralité le découpage en deux opérations distinctes, l’une d’accès séquentiel (feuille de l’arbre) et l’autre de sélection proprement dite (nœud interne). La même remarque joue pour le tri de la table *Artiste*.

2. chaque arc de l'arbre correspond à un flot des enregistrements (résultat de l'opération de plus bas niveau dans l'arbre). Soit celui-ci est stocké dans un fichier temporaire soit il est au fur et a mesure utilisé par l'opération de plus haut niveau (pipelining, voir section 4.1.5).

4.4.4 Modèles de coût

Soit la sélection $\sigma_{A \geq a}(R)$. La table a $N = 3000$ enregistrements stockés sur $B = 90$ blocs et un index dense unique sur A . Il existe deux algorithmes pour évaluer cette sélection :

1. le balayage séquentiel de R : la table n'étant pas en général triée sur A , dans le cas du balayage séquentiel, il faut parcourir toute la table et tester pour chaque enregistrement l'inégalité.
2. la traversée d'index : on accède à la feuille contenant le couple $[a, \text{adressed'enregistrement}]$ et on parcourt ensuite toutes les feuilles de l'index chaînées en séquence. Pour chacun des couples $[a', \text{adresse}]$, on accède au enregistrement d'adresse adresse (une lecture de page), voir section 4.2 et chapitre 2.

Pour choisir entre ces deux algorithmes, l'optimiseur compare les estimations du coût C de chacun :

1. balayage séquentiel : $C = B = 100$
2. index : $C = I + B/10 + N/10 = 312$

L'estimateur du coût de l'algorithme utilisant l'index est la somme de trois estimations :

- estimation du nombre de niveaux I de l'index : la première étape consiste à traverser l'index. Une estimation raisonnable de I est : $I = 3$
- estimation du nombre de feuilles chaînées à lire en séquence : les feuilles de l'index sont triées sur A . L'estimateur simpliste consiste à dire qu'un dixième des feuilles satisfait au critère $B/10 = 9$ ⁹
- estimation du nombre de pages de R lues pour accéder aux enregistrements : $N/10$. Cet estimateur simpliste est justifié par les hypothèses suivantes : (i) environ $N/10$ enregistrements satisfont le critère ; (ii) pour chaque enregistrement tel que $A = a$ il faut lire une page différente. Ceci est une hypothèse pessimiste parce que la page à lire est peut-être encore en mémoire, ce qui suppose qu'un certain nombre de tampons soient disponibles en mémoire centrale.

Par conséquent l'optimiseur choisira le balayage séquentiel. Notons que

1. s'il y a moins de 10 nulets par page, l'optimiseur choisira toujours le balayage séquentiel,
2. ce choix peut être erroné à cause de l'estimation simpliste du coût de l'algorithme utilisant l'index. En utilisant un histogramme des valeurs de A (stocké ou évalué sur un échantillon), on peut arriver à une estimation plus fine du nombre des enregistrements satisfaisant le critère conduisant éventuellement au choix de la stratégie avec index.

Choix d'un opérateur pour la sélection

Considérons le cas général d'une condition de sélection qui est un *AND* de n conditions de la forme $A = a$ ou $B \leq b$ (voir note de bas de page de la section 4.4.2). Supposons pour simplifier l'exposé que les n conditions portent sur n attributs différents. Nous supposons également que la relation a un index sur p parmi les attributs apparaissant dans un critère. Alors l'optimiseur a le choix entre le balayage séquentiel (la condition complète est testée sur chaque enregistrement) et l'accès par index suivi d'un accès par adresse au enregistrement (et le test sur ce enregistrement des $n - 1$ conditions restantes). Comme il y a p critères portant sur un attribut avec index, l'optimiseur a $p + 1$ cas à considérer. L'estimation du coût C de chacun des cas est résumé ci-dessous en fonction du nombre des enregistrements N , du nombre de blocs B de la relation et de la sélectivité $S(R, A)$ de l'attribut A .

1. balayage séquentiel : $C = B$

⁹. On pourrait penser qu'en moyenne $N/2$ enregistrements satisfont le critère. L'intuition du rapport 1 sur 10 vient de ce qu'une requête avec inégalité donne seulement une petite fraction des enregistrements.

2. stratégie avec index sur un critère avec égalité (l'index est supposé être dense et non unique, voir section 4.2) : $C = I + F + P$ où I est le coût de traversée d'index, F est le nombre de feuilles en séquence à lire et P le nombre de pages à lire pour accéder aux enregistrements (et tester les $n - 1$ conditions restantes) : $F = (N \times S)/(N/B) = B \times S$. En effet le terme numérateur $N \times S$ indique le nombre moyen des enregistrements par valeur d'index et le dénominateur N/B indique le nombre des enregistrements par page. $P = N \times S$. En effet P est le nombre des enregistrements qui satisfont le critère. Pour chacun d'eux il faut accéder une page (ce qui est une hypothèse pessimiste, voir discussion sur l'exemple de la section précédente). En résumé l'estimation du coût de cette stratégie est $C = I + (B + N) \times S$.
3. stratégie avec index sur un critère avec inégalité : $I + (B + N)/10$: le premier terme concerne la traversée d'index ; le terme $B/10$ est l'estimation du nombre de feuilles parcourues et le terme $N/10$ est l'estimation du nombre des enregistrements accédés et testés sur les $n - 1$ autres critères (voir discussion de la section précédente)

Choix d'un opérateur pour la jointure

Lors de l'étude des algorithmes de jointure dans la section 4.2 nous avons évalué le coût de chacun des algorithmes. Cette évaluation peut servir de base pour un estimateur permettant à l'optimiseur de choisir entre plusieurs stratégies possibles. Cependant un certain nombre de critères simples aident l'optimiseur dans son choix. Ceux-ci sont résumés ci-dessous :

- Le prédicat de jointure est-il une inégalité ? Le produit cartésien suivi d'une sélection est une stratégie possible si la taille de ce produit est raisonnable. Cependant en général on choisit un algorithme simple par boucles imbriquées (procédure BIS). Dans la suite on suppose que le prédicat de jointure est l'égalité.
- Si l'une au moins des deux relations a un index on utilise l'algorithme par boucles imbriquées et traversée d'index (procédure BIT). Cet algorithme est d'autant plus intéressant que la relation extérieure (qu'on balaie) est plus petite (elle est par exemple le résultat d'une sélection qui produit un ou quelques enregistrements)¹⁰.
- La taille de la mémoire disponible est grande par rapport à celle des tables : on lit les tables en mémoire et on fait la jointure en mémoire centrale (Procédure BIM). Une stratégie d'exception si les tables sont légèrement trop grandes consiste à les découper en deux morceaux et faire la jointure en deux étapes.
- si l'une des deux relations est déjà triée sur l'attribut de jointure, utiliser l'algorithme de tri-fusion. S'il n'y a pas d'index sur l'attribut de jointure, certains systèmes utilisent systématiquement cette stratégie¹¹.
- l'algorithme de tri-fusion doit être mis en compétition avec l'algorithme par hachage lequel est intéressant si les partitions issues de sa première phase tiennent en mémoire centrale.

4.5 Oracle, optimisation et évaluation des requêtes

Cette section présente l'application concrète des concepts, structures et algorithmes présentés dans ce qui précède dans le SGBD ORACLE. Ce système est un très bon exemple d'un optimiseur sophistiqué s'appuyant sur des structures d'index et des algorithmes d'évaluation extrêmement complets. Tous les algorithmes de jointure décrits dans ce chapitre (boucles imbriquées, tri-fusion, hachage, boucles imbriquées indexées) sont en effet implantés dans ORACLE. De plus le système propose des outils simples et pratiques (EXPLAIN et TKPROF) pour analyser le plan d'exécution choisi par l'optimiseur, et obtenir des statistiques sur les performances (coût en E/S et coût CPU, entre autres).

10. Si les deux relations ont un index sur l'algorithme de jointure on peut utiliser la variante consistant à fusionner les feuilles de l'index s'il n'est pas nécessaire d'accéder aux enregistrements des deux relations (voir discussion sur le coût de cet accès dans la section 4.2).

11. Ceci est justifié si les tailles des relations sont grandes. Un autre cas justifiant l'utilisation de cet algorithme est que le tri d'une table peut servir ultérieurement dans le même plan (par exemple pour une autre jointure sur le même attribut).

Nous commençons par décrire l'environnement de l'optimiseur et ses deux principaux modes, *optimisation basée sur les règles (rule-based)* et *optimisation basée sur les coûts (cost-based)*. Nous donnons ensuite, pour un ensemble représentatif de requêtes sur notre base exemple, les plans d'exécution choisis par l'optimiseur.

4.5.1 L'optimiseur d'ORACLE

À l'origine, l'optimiseur d'ORACLE déterminait un plan d'exécution en fonction d'un ensemble de *règles* indiquant quels étaient les chemins d'accès prioritaires parmi ceux disponibles pour exécuter une requête. Depuis la version 7.3, c'est l'optimisation par les coûts qui est privilégiée.

Optimisation par les règles

Prenons l'exemple de la requête suivante :

```
SELECT *
FROM   Film
WHERE  ROWID = '00000DD5.000.001'
```

L'optimiseur a le choix entre deux accès : par l'adresse donnée par le ROWID, ou par parcours séquentiel. Les règles (voir tableau 4.1) indiquant une priorité 1 pour le premier choix, et 15 pour le second, c'est évidemment la première solution qui est choisie.

Priorité	Description
1	Accès à un enregistrement par ROWID
2	Accès à un enregistrement dans un <i>cluster</i>
3	Accès à un enregistrement dans une table de hachage (<i>hash cluster</i>)
4	Accès à un enregistrement par clé unique (avec un index)
...	...
15	Balayage complet de la table

TABLE 4.1 – Règles pour l'optimiseur

L'optimisation basée sur les règles trouve ses limites dans l'incapacité à prédire le coût réel d'un chemin d'accès en fonction du degré de sélectivité de la requête. Nous avons par exemple expliqué dans la section 4.2.2 qu'il n'était pas toujours pertinent d'utiliser un index quand la sélectivité des attributs sur lesquels portent les critères de recherche est faible. Un nouveau mode introduit à partir de la version 7.3, *l'optimisation basée sur les coûts*, est venu palier ces insuffisances.

Optimisation par les coûts

Ce mode s'appuie sur des statistiques, et il est le seul à prendre en compte les évolutions récentes du SGBD. Les index bitmap par exemple sont uniquement pris en compte dans ce mode. Il est également plus contraignant puisqu'il implique la récolte régulière de statistiques sur les tables et attributs « sensibles », à savoir ceux qui influent sur le coût des requêtes. L'administrateur de la base est responsable de la tenue à jour de ces statistiques.

Il existe de très nombreux paramètres qui permettent de régler le fonctionnement de l'optimiseur basé sur les coûts qui, contrairement à celui basé sur les règles, est très flexible. Parmi les plus intéressants, citons les suivants :

1. OPTIMIZER_MODE indique le mode de fonctionnement de l'optimiseur. Si la valeur est RULE, c'est l'optimisation par les règles qui est choisie. Si la valeur est CHOOSE, l'optimiseur choisit l'optimisation par les coûts quand les statistiques sont disponibles

Ce paramètre permet également d'indiquer si le coût considéré est le temps de réponse (temps pour obtenir la première ligne du résultat), FIRST_ROW ou le temps d'exécution ALL_ROWS.

2. `SORT_AREA_SIZE` indique la taille de la zone de tri.
3. `HASH_AREA_SIZE` indique la taille de la zone de hachage.
4. `HASH_JOIN_ENABLED` indique que l'optimiseur considère les jointures par hachage.

Nous renvoyons à la documentation pour les (nombreux) autres paramètres. Il faut retenir que l'optimisation par les coûts n'est possible que si les statistiques (taille des tables et distribution des valeurs) existent pour au moins une table de la requête. Ces statistiques sont récoltées avec l'outil `ANALYZE`.

Pour analyser une table on utilise la commande `ANALYZE TABLE` qui stocke la taille de la table (nombre de lignes) et le nombre de blocs utilisés. Cette information est utile par exemple au moment d'une jointure pour utiliser comme table externe la plus petite des deux. Voici un exemple de la commande.

```
ANALYZE TABLE Film COMPUTE STATISTICS FOR TABLE;
```

On trouve alors les informations du tableau 4.2 dans les vues `DBA_TABLES`, `ALL_TABLES` et `USER_TABLES`.

Champ	Description
<code>NUM_ROWS</code>	Nombre de lignes
<code>BLOCKS</code>	Nombre de blocs
<code>EMPTY_BLOCKS</code>	Nombre de blocs non utilisés (?)
<code>AVG_SPACE</code>	Nombre moyen d'octets libres dans un bloc
<code>CHAIN_CNT</code>	Nombre de blocs chaînés
<code>AVG_ROW_LEN</code>	Taille moyenne d'une ligne
<code>NUM_FREELIST_BLOCKS</code>	Nombre de blocs dans la <i>freelist</i> (?)
<code>AVG_SPACE_FREELIST_BLOCKS</code>	(?)
<code>SAMPLE_SIZE</code>	Taille de l'échantillon utilisé
<code>LAST_ANALYZED</code>	Date de la dernière analyse

TABLE 4.2 – Les champs de la vue `USER_TABLES` après analyse

On peut également analyser les index d'une table, ou un index en particulier. Voici les deux commandes correspondantes.

```
ANALYZE TABLE Film COMPUTE STATISTICS FOR ALL INDEXES;
ANALYZE INDEX PKFilm COMPUTE STATISTICS;
```

On trouve alors les informations du tableau 4.3 dans les vues `DBA_INDEX`, `ALL_INDEX` et `USER_INDEXES`.

Pour finir on peut calculer des statistiques sur des colonnes. ORACLE utilise des histogrammes en hauteur (voir section 4.2.2) pour représenter la distribution des valeurs d'un champ. Il est évidemment inutile d'analyser toutes les colonnes. Il faut se contenter des colonnes qui ne sont pas des clés uniques, et qui sont indexées. Voici un exemple de la commande d'analyse pour créer des histogrammes avec vingt groupes sur les colonnes `titre` et `genre`.

```
ANALYZE TABLE Film COMPUTE STATISTICS FOR COLUMNS titre, genre SIZE 20;
```

On peut remplacer `COMPUTE` par `ESTIMATE` pour limiter le coût de l'analyse. ORACLE prend alors un échantillon de la table, en principe représentatif (on sait ce que valent les sondages !). Les informations sont stockées dans les vues `DBA_TAB_COL_STATISTICS` et `DBA_PART_COL_STATISTICS`. Le tableau 4.4 donne les champs de ces vues.

4.5.2 Plans d'exécution ORACLE

Nous en arrivons maintenant à la présentation des plans d'exécution d'ORACLE, tels qu'ils sont donnés par `EXPLAIN`. Ces plans ont classiquement la forme d'arbres en profondeur à gauche, chaque nœud étant un opérateur, les nœuds-feuille représentant les accès aux structures de la base, tables, index, *cluster*, etc.

Champ	Description
BLEVEL	Nombre de niveaux de l'arbre B
LEAF_BLOCKS	Nombre de blocs-feuille
DISTINCT_KEYS	Nombre de clés distinctes
AVG_LEAF_BLOCKS_PER_KEY	Nombre moyen de blocs dans lesquels on trouve une valeur de clé.
AVG_DATA_BLOCKS_PER_KEY	Nombre moyen de blocs de données à lire pour une valeur de clé.
CLUSTERING_FACTOR	Probabilité que deux enregistrements ayant des clés voisines soient également proches dans la table. Plus ce facteur est élevé, et plus il est avantageux d'effectuer des recherches par intervalle avec l'index.
NUM_ROWS	Nombre de lignes dans la table
SAMPLE_SIZE	Taille de l'échantillon utilisé
LAST_ANALYSED	Date de la dernière analyse

TABLE 4.3 – Les champs de la vue *USER_INDEXES* après analyse

Champ	Description
NUM_DISTINCT	Nombre de valeurs distinctes
LOW_VALUE	Plus petite valeur
HIGH_VALUE	Plus grande valeur
DENSITY	Distribution des valeurs
NUM_NULLS	Nombre de champs à NULL.
NUM_BUCKETS	Nombre d'intervalles de l'histogramme.
SAMPLE_SIZE	Taille de l'échantillon utilisé
LAST_ANALYSED	Date de la dernière analyse

TABLE 4.4 – Les champs de la vue *DBA_TAB_COL_STATISTICS* après analyse

Opérateur	Description
FULL TABLE SCAN	désigne un parcours séquentiel d'une table
ACCESS BY ROWID	désigne un accès par adresse
CLUSTER SCAN	parcours de <i>cluster</i> . On récupère alors dans une même lecture les enregistrements des 2 tables du <i>cluster</i> .
HASH SCAN	parcours de <i>hash cluster</i> .
INDEX SCAN	traversée d'index.
NESTED LOOP	Algorithme de boucles imbriquées indexées, utilisé quand il y a au moins un index.
SORT/MERGE	Algorithme de tri-fusion.
HASH JOIN	Jointure par hachage.
INTERSECTION	intersection de deux ensembles d'enregistrements.
CONCATENATION	union de deux ensembles.
FILTER	élimination d'enregistrements (utilisé dans un négation).
PROJECTION	opération de l'algèbre relationnelle.

TABLE 4.5 – Les opérations d'accès aux données et de jointure dans ORACLE

Le vocabulaire utilisé par l'optimiseur est un peu différent de celui utilisé précédemment dans ce chapitre. Le tableau 4.5 donne la liste des principaux opérateurs, en commençant par les chemins d'accès, puis les algorithmes de jointure, et enfin des opérations diverses de manipulation d'enregistrements.

Voici un petit échantillon de requêtes sur notre base *Film*, en donnant à chaque fois le plan d'exécution choisi par ORACLE. Les plans sont obtenus par application de l'optimisation par règles : vous êtes invités à appliquer l'optimisation par coût en utilisant les outils proposés dans les exercices.

La première requête est une sélection sur un attribut non indexé (partie gauche ci-dessous). On obtient le plan d'exécution nommé *SelSansInd* que l'on peut afficher comme montré dans la partie droite ¹² :

Requête	Plan d'exécution
<pre>EXPLAIN PLAN SET STATEMENT_ID='SelSansInd' FOR SELECT * FROM Film WHERE titre = 'Vertigo'</pre>	<pre>0 SELECT STATEMENT 1 TABLE ACCESS FULL FILM</pre>

On effectue donc un balayage complet de la table *Film*. L'affichage représente l'arborescence du plan d'exécution par une indentation. Pour plus de clarté, nous donnons également l'arbre complet (figure 4.18). On y distingue les structures de la base (tables et index principalement), représentées par des ovales, les opérateurs d'accès à ces structures, en foncé, et les opérateurs de manipulation de données, avec un fond clair.

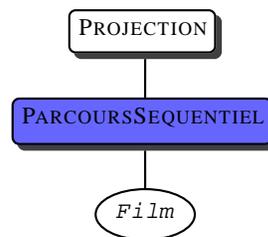


FIGURE 4.18 – Plan d'exécution pour une sélection sans index

La figure 4.18 montre un plan très simple : l'opérateur de parcours séquentiel extrait un à un les enregistrements de la table *Film*. Un filtre (jamais montré dans les plans donnés par EXPLAIN, car intégré aux opérateurs d'accès aux données) élimine tous ceux dont le titre n'est pas *Vertigo*. Pour ceux qui passent le filtre, un opérateur de projection (malencontreusement nommé *SELECT* dans ORACLE ...) ne conserve que les champs non désirés.

Voici maintenant une sélection avec index sur la table *Film*. Comme précédemment, la partie gauche montre la requête avec l'instruction EXPLAIN, et la partie droite le plan d'exécution obtenu.

Requête	Plan d'exécution
<pre>EXPLAIN PLAN SET STATEMENT_ID='SelInd' FOR SELECT * FROM Film WHERE idFilm=21;</pre>	<pre>0 SELECT STATEMENT 1 TABLE ACCESS BY ROWID FILM 2 INDEX UNIQUE SCAN IDX-FILM-ID</pre>

L'optimiseur a détecté la présence d'un index unique sur la table *Film*. La traversée de cet index donne un ROWID qui est ensuite utilisé pour un accès direct à la table (figure 4.19).

Passons maintenant aux jointures. La requête donne les titres des films avec les nom et prénom de leur metteur en scène, ce qui implique une jointure en *Film* et *Artiste*. Le plan d'exécution est donné à droite :

12. La section consacrée aux exercices donne des détails techniques sur la mise en œuvre de EXPLAIN et sur l'affichage du résultat

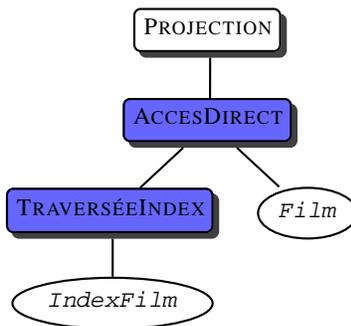


FIGURE 4.19 – Plan d'exécution pour une sélection avec index

il s'agit d'une jointure par boucles imbriquées indexées.

Requête	Plan d'exécution
EXPLAIN PLAN	0 SELECT STATEMENT
SET STATEMENT_ID='JoinIndex' FOR	1 NESTED LOOPS
SELECT titre, nom, prenom	2 TABLE ACCESS FULL FILM
FROM Film f, Artiste a	3 TABLE ACCESS BY ROWID ARTISTE
WHERE idMES = idArtiste;	4 INDEX UNIQUE SCAN IDXARTISTE

La représentation graphique est sans doute plus claire pour comprendre le mécanisme (fig 4.20). Tout d'abord la table qui n'est pas indexée sur l'attribut de jointure (ici, *Film*) est parcourue séquentiellement. Le nœud BOUCLESIMBRIQUÉES récupère les enregistrements *Film* un par un du côté gauche. Pour chaque film on va alors récupérer l'artiste correspondant avec le sous-arbre du côté droit.

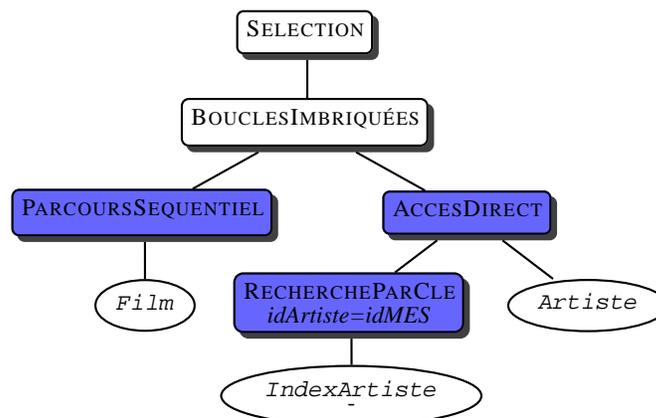


FIGURE 4.20 – Plan d'exécution pour une jointure avec index

On effectue d'abord une recherche par clé dans l'index avec la valeur *idMES* provenant du film courant. La recherche renvoie un ROWID qui est alors utilisé pour prendre l'enregistrement complet dans la table *Artiste*. Le nœud de jointure appelle alors le film suivant, et ainsi de suite.

Dans certains cas on peut éviter le parcours séquentiel à gauche de la jointure par boucles imbriquées, si une sélection supplémentaire sur un attribut indexé est exprimée. L'exemple ci-dessous sélectionne tous les rôles joués par Al Pacino. Il existe un index sur les noms des artistes qui permet d'optimiser la recherche par nom, et l'index sur la table *Rôle* est la concaténation des champs *idActeur* et *idFilm*, ce qui permet de faire une recherche par intervalle sur le préfixe constitué seulement de *idActeur*. La

requête et le plan d'exécution sont donnés ci-dessous, (voir figure 4.21 pour l'arbre correspondant).

Requête	Plan d'exécution
EXPLAIN PLAN SET STATEMENT_ID='JoinSelIndex' FOR SELECT nomRole FROM Role r, Artiste a WHERE r.idActeur = a.idArtiste AND nom = 'Pacino';	0 SELECT STATEMENT 1 NESTED LOOPS 2 TABLE ACCESS BY ROWID ARTISTE 3 INDEX RANGE SCAN IDX-NOM 4 TABLE ACCESS BY ROWID ROLE 5 INDEX RANGE SCAN IDX-ROLE

Notez bien que les deux recherches dans les index s'effectuent par intervalle, et peuvent donc ramener plusieurs ROWID. Dans les deux cas on utilise en effet seulement une partie des champs définissant l'index (et un partie constituant un préfixe, ce qui est impératif). On peut donc envisager de trouver plusieurs artistes nommé Pacino (avec des prénoms différents), et pour un artiste, on peut trouver plusieurs rôles (mais pas pour le même film). Tout cela résulte de la conception de la base.

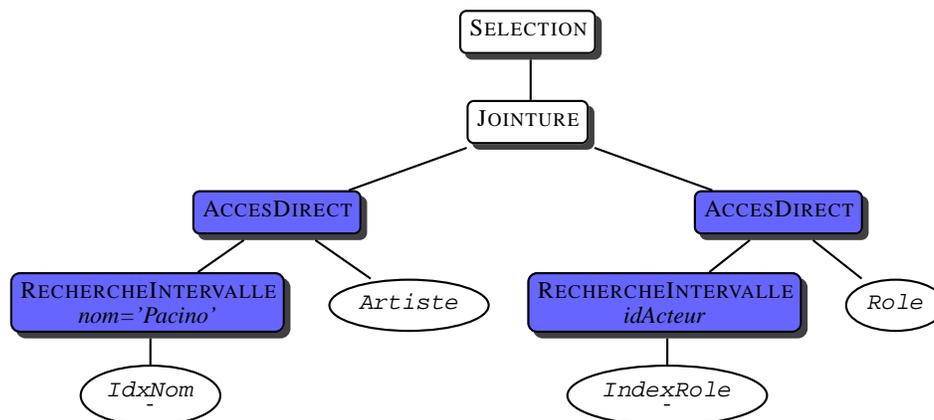


FIGURE 4.21 – Plan d'exécution pour jointure et sélection avec index

Pour finir voici une requête sans index. On veut trouver tous les artistes nés l'année de parution de *Vertigo* (et pourquoi pas ?). La requête est donnée ci-dessous : elle effectue une jointure sur les années de parution des films et l'année de naissance des artistes. Comme il n'existe pas d'index sur ces champs, ORACLE applique un algorithme de tri-fusion.

Requête	Plan d'exécution
EXPLAIN PLAN SET STATEMENT_ID='JoinSansIndex' FOR SELECT nom, prenom FROM Film f, Artiste a WHERE f.annee = a.anneeNaiss AND titre = 'Vertigo';	0 SELECT STATEMENT 1 MERGE JOIN 2 SORT JOIN 3 TABLE ACCESS FULL ARTISTE 4 SORT JOIN 5 TABLE ACCESS FULL FILM

L'arbre de la figure 4.22 montre bien les deux tris, suivis de la fusion. Au moment du parcours séquentiel, on va filtrer tous les films dont le titre n'est pas *Vertigo*, ce qui va certainement beaucoup simplifier le calcul de ce côté-là. En revanche le tri des artistes risque d'être beaucoup plus coûteux.

Dans un cas comme celui-là, on peut envisager de créer un index sur les années de parution ou sur les années de naissance. Un seul index suffira, puisqu'il devient alors possible d'effectuer une jointure par boucle imbriquées.

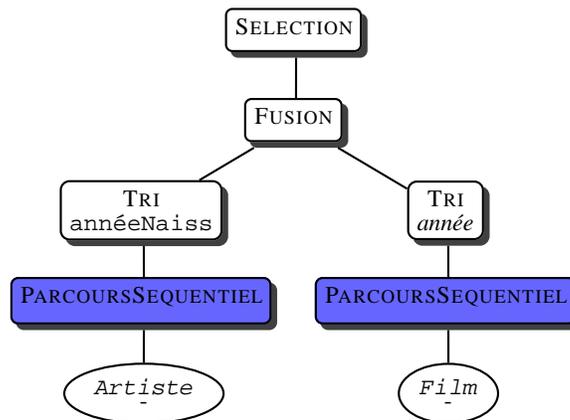


FIGURE 4.22 – Plan d'exécution pour une jointure sans index (tri-fusion)

Outre l'absence, il existe de nombreuses raisons pour qu'ORACLE ne puisse pas utiliser un index : par exemple quand on applique une fonction au moment de la comparaison. Il faut être attentif à ce genre de détail, et utiliser EXPLAIN pour vérifier le plan d'exécution quand une requête s'exécute sur un temps anormalement long.

4.6 Exercices

Les premiers exercices sont à faire en TD, et adoptent le modèle d'exécution par itération/pipelining. Les exercices suivants consistent à expérimenter les concepts proposés avec le SGBD ORACLE, en créant une base de données, en l'alimentant avec un nombre choisi d'enregistrements, et en évaluant l'effet de manipulations diverses sur les performances du système.

4.6.1 Opérateurs d'accès aux données

Exercice 26. Soit la liste des départements de l'exercice 10, page 65. On suppose qu'on peut stocker deux enregistrements par bloc. Décrire l'algorithme de tri-fusion sur le numéro de département appliqué à ce fichier dans les cas suivants :

1. $M = 4$ blocs ;
2. $M = 3$ blocs.

Exercice 27. Soit un fichier de 10 000 blocs et un buffer en mémoire centrale de 3 blocs. On trie ce fichier avec l'algorithme de tri-fusion.

- Combien de fragments sont produits pendant la première passe ?
- Combien de passes faut-il pour trier complètement le fichier ?
- Quel est le coût total en entrées/sorties ?
- Combien faut-il de blocs en mémoire faut-il pour trier le fichier en une fusion seulement.

Répondre aux mêmes questions en prenant un fichier de 20 000 blocs et 5 blocs de buffer, puis un fichier de 2 000 000 de blocs et 17 blocs de buffer.

Exercice 28. Donnez une spécification en pseudo-langage orienté-objet des méthodes constructeur, open, next et close de l'itérateur PARCOURSSEQUENTIEL qui lit séquentiellement un fichier.

Exercice 29. Donner des plans d'exécution pour les requêtes suivantes :

- SELECT titre FROM Film ORDER BY annee
 - SELECT MIN(annee) FROM Film
- Il faut définir un itérateur MIN.

- SELECT DISTINCT genre FROM Film
Il faut définir un itérateur *DISTINCT*, en supposant que ce dernier s'appuie sur une source de données (un autre itérateur) triée.
- SELECT idMES, COUNT(*) FROM Film GROUP BY idMES
Il faut définir un itérateur *GROUP BY*, en supposant encore qu'il s'appuie sur une source de données (un autre itérateur) triée.

Exercice 30. Soit deux relations R et S , de tailles respectives $|R|$ et $|S|$ (en nombre de blocs). On dispose d'une mémoire mem de taille M , dont les blocs sont dénotés $mem[1], mem[2], \dots, mem[M]$.

1. Donnez les formules exprimant le coût d'une jointure $R \bowtie S$, en nombre d'entrées/sorties, pour l'algorithme suivant :

```

posR = 1
Tant que posR <= |R| Faire
  Lire R[posR] dans mem[1]
  posS = 1
  Tant que posS <= |S| faire
    Lire S[posS] dans mem[2]
    Pour chaque enregistrement r de mem[1]
      Pour chaque enregistrement s de mem[2]
        Si r.a = s.b alors retourner [r, s]
      Fin pour
    Fin pour
  posS = posS + 1
  Fait
  posR = posR + 1
Fait

```

2. Même question avec l'algorithme suivant :

```

posR = 1
Tant que posR <= |R| Faire
  Lire R[posR..(posR+M-1)] dans mem[1..M-1]
  posS = 1
  Tant que posS <= |S| faire
    Lire S[posS] dans mem[M]
    Pour chaque enregistrement r de mem[1..M-1]
      Pour chaque enregistrement s de mem[M]
        Si r.a = s.b alors retourner [r, s]
      Fin pour
    Fin pour
  posS = posS + 1
  Fait
  posR = posR + M
Fait

```

3. Quelle table faut-il prendre pour la boucle extérieure ? La plus petite ou la plus grande ?

Exercice 31 (Jointures). On suppose que $|R| = 10\,000$, $|S| = 1\,000$ et $M = 51$. On a 10 enregistrements par bloc, b est la clé primaire de S et on suppose que pour chaque valeur de a on trouve en moyenne 5 enregistrements dans R . On veut calculer $\pi_{R.c}(R \bowtie_{a=b} S)$,

- Donnez le nombre d'entrée-sorties dans le pire des cas pour les algorithmes de l'exercice 30.
- Même question en supposant (a) qu'on a un index sur R , (b) qu'on a un index sur S , (c) qu'on a deux index, sachant que dans tous les cas l'index a 3 niveaux.
- Même question pour une jointure par hachage.
- Même question avec un algorithme de tri-fusion.

Exercice 32. Donner le meilleur plan d'exécution pour les requêtes suivantes, en supposant qu'il existe un index sur `idFilm`.

- `SELECT * FROM Film WHERE idFilm = 20 AND titre = 'Vertigo'`
- `SELECT * FROM Film WHERE idFilm = 20 OR titre = 'Vertigo'`
- `SELECT COUNT(*) FROM Film`
- `SELECT MAX(idFilm) FROM Film`

Lesquelles de ces requêtes peuvent s'évaluer uniquement avec l'index ?

Exercice 33. Soit les tables relationnelles suivantes (les attributs qui forment une clé sont en gras) :

- Produit (**code**, nom, marque, prix)
- Client (**id**, nom, prénom)
- Achat (**codeProduit**, **idClient**, date, quantité)

On donne ci-dessous une requête SQL et le plan d'exécution fourni par Oracle :

```
select quantité
from Produit p, Client c, Achat a
where p.code = a.codeProduit
and   c.id = a.idClient
and   prix > 50
```

Plan d'exécution :

```
0 SELECT STATEMENT
1 MERGE JOIN
2 SORT JOIN
3 NESTED LOOPS
4 TABLE ACCESS FULL ACHAT
5 TABLE ACCESS BY INDEX ROWID PRODUIT
6 INDEX UNIQUE SCAN A34561
7 SORT JOIN
8 TABLE ACCESS FULL Client
```

1. Que peut-on déduire de ce plan : peut-on savoir s'il existe un index sur la table Client ? Sur la table Produit ? Sur la table Achat ? Si vous pensez qu'un index existe, donnez les attributs indexés. Justifiez vos réponses.
2. Algorithme de jointure : Expliquer en détail le plan d'exécution appliqué par ORACLE (accès aux tables, sélections, jointure, projections)
3. Quel(s) index peut-on ajouter pour obtenir les meilleures performances possibles ? Donnez, sous la forme que vous souhaitez, le plan d'exécution après ajout d'index.
4. On fait une jointure entre les tables Produit et Achat. Produit occupe 500 blocs, Achat 10 000 blocs. Vaut-il mieux appliquer un algorithme de boucles imbriquées ou de hachage si la mémoire M disponible est de 600 blocs ? Et si elle est de 101 blocs ? Indiquez dans chaque cas le nombre d'entrées/sorties (sans prendre en compte l'écriture du résultat final).

Exercice 34. Soit le schéma relationnel :

```
Journaliste (jid, nom, prénom)
Journal (titre, rédaction, id_rédacteur)
```

La table `Journaliste` stocke les informations (nom, prénom) sur les journalistes (`jid` est le numéro d'identification du journaliste). La table `Journal` stocke pour chaque rédaction d'un journal le titre du journal (titre), le nom de la rédaction (rédaction) et l'id de son rédacteur (`rédacteur_id`). Le titre du journal est une clé. On a un index dense sur la table `Journaliste` sur l'attribut `jid`, nommé `Idx-Journaliste-jid`.

On considère la requête suivante :

```

SELECT nom
FROM Journal, Journaliste
WHERE titre='Le Monde'
AND jid=id_redacteur
AND prenom='Jean'

```

1. Voici deux expressions algébriques :

- (a) $\pi_{nom}(\sigma_{titre='Le Monde' \wedge prenom='Jean'}(Journaliste \bowtie_{jid=redacteur_id} Journal))$
 (b) $\pi_{nom}(\sigma_{prenom='Jean'}(Journaliste) \bowtie_{jid=redacteur_id} \sigma_{titre='Le Monde'}(Journal))$

Les deux expressions retournent-elles le même résultat (sont-elles équivalentes) ? Une expression est-elle meilleure que l'autre si on les considère comme des plans d'exécution ?

2. Donner le meilleur plan d'exécution physique sous forme arborescente ou sous forme d'une expression EXPLAIN, et expliquez en détail ce plan.

Exercice 35. Soit la base d'une société d'informatique décrivant les clients, les logiciels vendus, et les licences indiquant qu'un client a acquis un logiciel.

```

Société (id, intitulé)
Logiciel (id, nom)
Licence (idLogiciel, idSociété, durée)

```

Bien entendu un index unique est créé sur les clés primaires. Pour chacune des requêtes suivantes, donner le plan d'exécution qui vous semble le meilleur.

```

- SELECT intitulé
  FROM Société, Licence
  WHERE durée = 15
  AND id = idSociete

- SELECT intitule
  FROM Société, Licence, Logiciel
  WHERE nom='EKIP'
  AND Société.id = idSociete
  AND Logiciel.id = idLogiciel

- SELECT intitule
  FROM Société, Licence
  WHERE Société.id = idSociete
  AND idLogiciel IN (SELECT id FROM Logiciel WHERE nom='EKIP')

- SELECT intitule
  FROM Société s, Licence c
  WHERE s.id = c.idSociete
  AND EXISTS (SELECT * FROM Logiciel l
              WHERE nom='EKIP' AND c.idLogiciel=l.idLogiciel)

```

4.6.2 Plans d'exécution ORACLE

Exercice 36. On prend les tables suivantes, abondamment utilisées par ORACLE dans sa documentation :

- Emp (empno, ename, sal, mgr, deptno)
- Dept (deptno, dname, loc)

La table Emp stocke des employés, la table Dept stocke les départements d'une entreprise. La requête suivante affiche le nom des employés dont le salaire est égal à 10000, et celui de leur département.

```

SELECT e.ename, d.dname
FROM emp e, dept d
WHERE e.deptno = d.deptno
AND e.sal = 10000

```

Voici des plans d'exécution donnés par ORACLE, qui varient en fonction de l'existence ou non de certains index. Dans chaque cas expliquez ce plan.

1. Index sur Dept (deptno) et sur Emp (Sal).

Plan d'exécution

```
-----
0 SELECT STATEMENT
  1 NESTED LOOPS
    2 TABLE ACCESS BY ROWID EMP
      3 INDEX RANGE SCAN EMP_SAL
    4 TABLE ACCESS BY ROWID DEPT
      5 INDEX UNIQUE SCAN DEPT_DNO
```

2. Index sur Emp (sal) seulement.

Plan d'exécution

```
-----
0 SELECT STATEMENT
  1 NESTED LOOPS
    2 TABLE ACCESS FULL DEPT
    3 TABLE ACCESS BY ROWID EMP
      4 INDEX RANGE SCAN EMP_SAL
```

3. Index sur Emp (deptno) et sur Emp (sal).

4. Voici une requête légèrement différente.

```
SELECT e.ename
FROM emp e, dept d
WHERE e.deptno = d.deptno
AND    d.loc = 'Paris'
```

On suppose qu'il n'y a pas d'index. Voici le plan donné par ORACLE.

Plan d'exécution

```
-----
0 SELECT STATEMENT
  1 MERGE JOIN
    2 SORT JOIN
      3 TABLE ACCESS FULL DEPT
    4 SORT JOIN
      5 TABLE ACCESS FULL EMP
```

Indiquer quel(s) index on peut créer pour obtenir de meilleures performances (donner le plan d'exécution correspondant).

5. Que pensez-vous de la requête suivante par rapport à la précédente ?

```
SELECT e.ename
FROM emp e
WHERE e.deptno IN (SELECT d.deptno
                   FROM Dept d
                   WHERE d.loc = 'Paris')
```

Voici le plan d'exécution donné par ORACLE :

```

0 SELECT STATEMENT
  1 MERGE JOIN
    2 SORT JOIN
      3 TABLE ACCESS FULL EMP
    4 SORT JOIN
      5 VIEW
        6 SORT UNIQUE
          7 TABLE ACCESS FULL DEPT

```

Qu'en dites vous ?

Sur le même schéma, voici maintenant la requête suivante.

```

SELECT *
FROM Emp e1 WHERE sal IN (SELECT sall
                          FROM Emp e2
                          WHERE e2.empno=e1.mgr)

```

Cette requête cherche les employés dont le salaire est égal à celui de leur patron. On donne le plan d'exécution avec Oracle (outil EXPLAIN) pour cette requête dans deux cas : (i) pas d'index, (ii) un index sur le salaire et un index sur le numéro d'employé.

Expliquez dans les deux cas ce plan d'exécution (éventuellement en vous aidant d'une représentation arborescente de ce plan d'exécution).

1. Pas d'index.

Plan d'exécution

```

-----
0 FILTER
  1 TABLE ACCESS FULL EMP
  2 TABLE ACCESS FULL EMP

```

2. Index sur empno et index sur sal.

Plan d'exécution

```

-----
0 FILTER
  1 TABLE ACCESS FULL EMP
  2 AND-EQUAL
    3 INDEX RANGE SCAN I-EMPNO
    4 INDEX RANGE SCAN I-SAL

```

3. Dans le cas où il y a les deux index (salaire et numéro d'employé), on a le plan d'exécution suivant :

Plan d'exécution

```

-----
0 FILTER
  1 TABLE ACCESS FULL EMP
  2 TABLE ACCESS ROWID EMP
    3 INDEX RANGE SCAN I-EMPNO

```

Expliquez-le.

Exercice 37. Soit le schéma suivant :

```

CREATE TABLE Artiste (id_artiste INT NOT NULL,
                      nom VARCHAR (60),
                      prénom VARCHAR (60),
                      année_naissance INT,

```

```

        PRIMARY KEY (id_artiste));

CREATE TABLE film (id_film INT NOT NULL,
                   titre   VARCHAR(60),
                   année   INT,
                   id_réalisateur INT,
                   PRIMARY KEY (id_film),
                   FOREIGN KEY (id_réalisateur) REFERENCES Artiste);

CREATE TABLE seance (nom_cinema VARCHAR (60) NOT NULL,
                     no_salle   NUMBER(2) NOT NULL,
                     no_séance  NUMBER(2) NOT NULL,
                     heure_debut INT,
                     heure_fin  INT,
                     id_film    INT NOT NULL,
                     PRIMARY KEY (nom_cinema, no_salle, no_séance),
                     FOREIGN KEY (id_film) REFERENCES Film
                               ON DELETE CASCADE);

```

Questions :

1. Donner l'ordre SQL pour la requête : Quels sont les films d'Hitchcock visibles après 20h00 ?
2. Donner l'expression algébrique correspondante et proposez un arbre de requête qui vous paraît optimal.
3. Sous ORACLE, l'outil EXPLAIN donne le plan d'exécution suivant :

```

0 SELECT STATEMENT
  1 MERGE JOIN
    2 SORT JOIN
      3 NESTED LOOPS
        4 TABLE ACCESS FULL ARTISTE
        5 TABLE ACCESS BY ROWID FILM
          6 INDEX RANGE SCAN IDX-ARTISTE-ID
    7 SORT JOIN
      8 TABLE ACCESS FULL SEANCE

```

Commentez le plan donné par EXPLAIN. Pourrait-on améliorer les performances de cette requête ?

Exercice 38. Soit le schéma, la requête et le plan d'exécution ORACLE suivants :

```

CREATE TABLE TGV (
  NumTGV integer,
  NomTGV varchar(32),
  GareTerm varchar(32));

CREATE TABLE Arret (
  NumTGV integer,
  NumArr integer,
  GareArr varchar(32),
  HeureArr varchar(32));

EXPLAIN PLAN
  SET statement_id = 'eds0'
  FOR select NomTGV
  from TGV, Arret
  where TGV.NumTGV = Arret.NumTGV
  and GareTerm = 'Aix';

```

```
@exbdb;

/* sans index */
0 SELECT STATEMENT
  1 MERGE JOIN
    2 SORT JOIN
      3 TABLE ACCESS FULL ARRET
    4 SORT JOIN
      5 TABLE ACCESS FULL TGV
```

- Que calcule la requête ?
- Que pouvez-vous dire sur l’existence d’index pour les tables “TGV” et “Arret” ? Décrivez en détail le plan d’exécution : quel algorithme de jointure a été choisi, quelles opérations sont effectuées et dans quel ordre ?
- On fait la création d’index suivante :

```
CREATE INDEX index arret_numtgv ON Arret(numtgv);
```

 L’index créé est-il dense ? unique ? Quel est le plan d’exécution choisi par Oracle ? Vous pouvez donner le plan avec la syntaxe ou sous forme arborescente. Expliquez en détail le plan choisi.
- On rajoute encore un index :

```
CREATE INDEX tgv_gareterm on tgv(gareterm);
```

 Quel est le plan d’exécution choisi par Oracle ? Expliquez le en détail.

C’est presque le même plan, sauf que au lieu de balayer tous les TGV (toute la table TGV), on accède directement à ceux dont le terminus est ‘Aix’ en traversant l’index sur les gares terminus

4.6.3 Utilisation de EXPLAIN et de TKPROF

Le site *Films* propose les fichiers suivants pour créer et alimenter une base (sur des films bien sûr...).

1. Le fichier *Schema.sql* contient les commandes de création du schéma sous ORACLE.
2. Le fichier *CrFilms.pc* est un programme PRO*C pour alimenter cette base. L’entête du fichier contient les instructions pour le compiler et l’exécuter.

Avec ces deux fichiers vous pouvez créer une base de taille quelconque, et tester l’évaluation des requêtes avec EXPLAIN et TKPROF.

Utilisation de EXPLAIN

Voici le mode d’utilisation de EXPLAIN.

- Les plans d’exécution sont stockés dans une table *PLAN_TABLE* dont le script de création se trouve, en principe, dans *\$ORACLE_HOME/rdbms/admin/utlxplan.sql*. Vous pouvez également la récupérer sur le site
- Ensuite on stocke le plan d’exécution d’une requête dans cette table avec la commande EXPLAIN PLAN. Voici un exemple :

```
EXPLAIN PLAN
      SET statement_id = 'plan0'
      FOR SELECT a.nom
            FROM   film f, artiste a
            WHERE  f.idMES = s.idArtiste
            AND    f.titre = 'Vertigo';
```

La clause `statement_id = 'plan0'` attribue un identifiant au plan d’exécution de cette requête dans la table *PLAN_TABLE*. Bien entendu vous devez donner à chaque requête stockée un identifiant spécifique.

- Pour connaître le plan d'exécution, on interroge la table `PLAN_TABLE`. L'information est un peu difficile à interpréter : le plus simple est de faire tourner le fichier *Explain.sql* (à récupérer sur le site) dont voici le code :

```
undef query;
```

```
SELECT LPAD(' ',2*(LEVEL-1))||operation||' '||options
      ||' '||object_name
      "Plan d'exécution"
FROM plan_table
START WITH id = 0 AND statement_id = '&&query'
CONNECT BY PRIOR id = parent_id AND statement_id = '&&query';
```

Quand on exécute ce fichier, il demande (2 fois) le nom du plan d'exécution à afficher. Dans le cas de l'exemple ci-dessus, on répondrait deux fois 'plan0'. On obtient l'affichage suivant qui présente de manière relativement claire le plan d'exécution.

```
Plan d'exécution
```

```
-----
0 SELECT STATEMENT
  1 NESTED LOOPS
    2 TABLE ACCESS FULL Film
    3 TABLE ACCESS BY ROWID Artiste
      4 INDEX UNIQUE SCAN SYS_C004709
```

Ici, le plan d'exécution est le suivant : on parcourt en séquence la table *Film* (ligne 2) ; pour chaque séance, on accède à la table *Artiste* par l'index¹³ (ligne 4), puis pour chaque ROWID provenant de l'index, on accède à la table elle-même (ligne 3). Le tout est effectué dans une boucle imbriquée (ligne 1).

Utilisation de TKPROF

TKPROF est complémentaire de EXPLAIN : il donne les différents coûts constatés pour l'exécution d'une requête. Le principe d'utilisation est le suivant : on passe en mode TRACE avec la commande suivante (sous SQLPLUS)

```
ALTER SESSION SET SQL_TRACE = TRUE;
```

À partir de ce moment-là, toutes les requêtes exécutées sous SQLPLUS entraîneront l'insertion dans un fichier des informations sur le coût d'exécution. Quand on a exécuté toutes les requêtes à analyser, on stoppe le mode TRACE avec :

```
ALTER SESSION SET SQL_TRACE = FALSE;
```

Il reste à récupérer le fichier de trace et à l'analyser avec l'utilitaire TKPROF. Pour localiser le fichier, on peut utiliser la requête suivante :

```
SELECT value
FROM v$parameter
WHERE name = 'user_dump_dest';
```

On obtient le répertoire où se trouve le fichier, dont le nom suit le format *oraSID_ora_noProcessus*. Il reste à lui appliquer le programme TKPROF :

```
tkprof nomFichier
```

Vous devez bien entendu avoir les droits de lecture sur le fichier, ce qui peut poser problème si vous n'êtes pas l'administrateur. Il reste une troisième solution, peut-être la plus simple, avec l'option AUTOTRACE de SQLPLUS.

13. Cet index a été automatiquement créé en association avec la commande PRIMARY KEY.

Cumuler EXPLAIN et TKPROF avec AUTOTRACE

En passant en mode AUTOTRACE sous SQLPLUS, on obtient les principales informations fournies par EXPLAIN et TKPROF sans avoir besoin d'effectuer les manipulations détaillées précédemment. Avant de pouvoir utiliser cette option, il faut avoir effectué la petite installation suivante :

1. créer le rôle PLUSTRACE avec le script `$ORACLE_HOME/sqlplus/admin/plustrce.sql` (il faut exécuter ce script sous le compte SYS) ;
2. donner ce rôle avec la commande GRANT à tout utilisateur souhaitant faire appel à AUTOTRACE.

Si, maintenant, on est un de ces utilisateurs, on se place en mode AUTOTRACE avec la commande suivante :

```
ALTER SESSION SET AUTOTRACE {OFF | ON | TRACEONLY} [EXPLAIN] [STATISTICS]
```

Le mode TRACEONLY permet de ne pas afficher le résultat de la requête, ce qui est le but recherché en général.

4.6.4 Exercices d'application

Exercice 39. *Effectuer des requêtes en mode TRACE, afin d'étudier les plans d'exécution. Pour l'instant, se mettre en mode « règles ».*

1. Des sélections sur des champs indexés ou non indexés (faire une sélection sur le genre par exemple)
2. Des jointures (penser à inverser l'ordre des tables dans le FROM)
3. Des requêtes avec NOT IN ou NOT EXISTS

Exercice 40. *Maintenant, analyser les tables, les index et les distributions avec ANALYSE, et regarder les modifications des plans d'exécution en optimisation par les coûts. En particulier*

- Chercher des films par le genre en mode règles
- Chercher des films par le genre en mode coûts

Le genre étant peu sélectif, l'optimiseur ne devrait pas utiliser l'index dans le second cas.

Exercice 41. *Reprendre les requêtes du polycopié cherchant le film paru en 1958 avec James Stewart. Analyser les différentes versions (sans imbrication, avec imbrication mais sans corrélation (IN), avec imbrication et corrélation (EXISTS), etc). Comparer les plans d'exécution obtenus dans chaque cas.*

Exercice 42. *Supprimer quelques index, et regarder le changement dans les plans d'exécutions des requêtes précédentes. NB : vous pouvez obtenir la liste des index existant sur vos tables avec la commande :*

```
SELECT table_name, index_name FROM user_indexes;
```

Exercice 43. *Maintenant appliquez des fonctions de groupe, des GROUP BY et des HAVING, regardez les plans d'exécution et interprétez-les.*

Exercice 44. *Essayer d'appliquer des fonctions dans les conditions de jointure. Que constate-t-on au niveau du plan d'exécution ?*

Exercice 45. *Essayer d'appliquer l'opérateur LIKE dans des sélections sur des attributs indexés. Que constate-t-on ?*