Aspects Systèmes des Bases de Données : compléments, précisions et digressions

Ces notes ne sont pas un polycopié mais plutôt un ensemble de commentaires et précisions relatifs au cours.

Pr., V. Benzaken

December 5, 2005

Chapter 1

Introduction

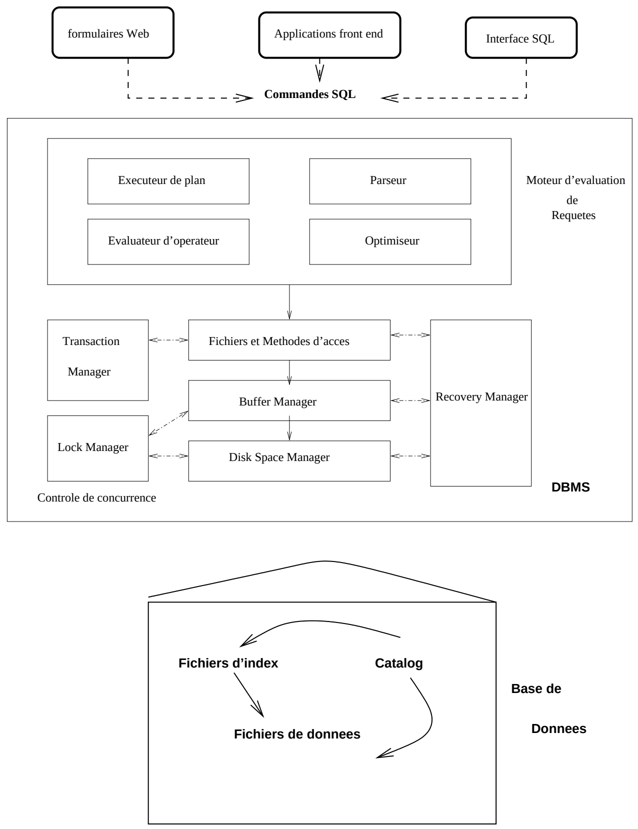


Figure 1.1: Architecture d'un SGBD

Un système de bases de données est un système complexe. Dans ces notes nous nous intéressons aux couxhes basses du DBMS : gestionnaire de stockage, organisation des fichiers, indexation, optimisation physique de requêtes, contrôle de concurrence et techniques de recouvrement sur pannes. La figure 1 illustre la complexité d'un système de gestion de bases de donn´es.

Du disque aux utilisateurs, il est indispensable de gérer les différents niveaux d'abstraction.

L'abstraction de base dans tout SGBD est la notion d'ensemble d'enregistrements ou encore de fichier. Chaque fichier consistant en une uo plusieurs pages.

Les couches *fichiers et méthodes d'accès* organisent soigneusement les données afin de garantir des accès efficaces à des portions pertinentes d'information (ensembles d'enregistrements).

Chapter 2

Stockage

2.1 Hiérarchie de mémoire

Points à retenir mémoire primaire (cache, principale) ultra rapide (nano seconde) mémoire secondaire (disques magnétiques)b plus lente enfin mémoire tertaire (bande, CD, DVD etc) très lente. Facteur de 100 en terme de temps d'accès d'un niveau à l'autre.

Or, il est **indispensable** de stocker les données en mémoire secondaire (taille, persistence).

2.1.1 Disques magnétiques

Voir slides

2.1.2 Impact sur les performances

Voir slides.

2.2 Redundant Arrays of Independent Disks : RAID

Voir slides

Choix d'un niveau de RAID Si la perte de données n'est pas un problème alors le RAID niveau 0 suffit. Il améliore les performances du système à moindre coût.

Le RAID niveau 1 est couteux (mirroir), Niveau 5 adapté à un environnement ou petites et grande demandes de données sont opérées.

2.3 Gestion de l'espace disque

2.3.1 Conserver la trace des blocs disponibles

Une base de données évolue au fur et à mesure que des entegistrments sont insérés ou détruits. Le gestionnaire de l'espace dique doit conserver la trace des blocs disques qui sont disponibles, qui sont utilisés. En outre, il doit aussi avoir quelles pages sont affectées à quels blocs (disques)

Une première méthode consiste à gérer la liste des blocs libres. Si une application (de haut niveau) opère une desallocation d'un bloc, le bloc est ajouté à la liste. Un pointeur vers le premier bloc de la liste des blocs libre est stocké à un endroit fixe (connu) du disque.

Une seconde manière de procéder consiste à maintenir un "bitma" avec un bit pour chaque bloc indiquant si le bloc est libre ou non (0 ou 1). Cette solution a le mérite de l'efficacité de la décision de la disponobilité d'un bloc mais est difficile à coupler avec une approche par chainage.

2.3.2 Utilisation du système de fichiers de l'OS ?

Les systèmes d'expolitation gèrent également l'espace disque. Un OS offre souvent l'absatraction suivante : un fichier est une séquence d'octets. L'OS permet alors des accès du style "lire l'octet i du fichier f" en les traduisant en instructions de plus bas niveau : "lire le bloc m de la piste p du cylindre c du disque d"

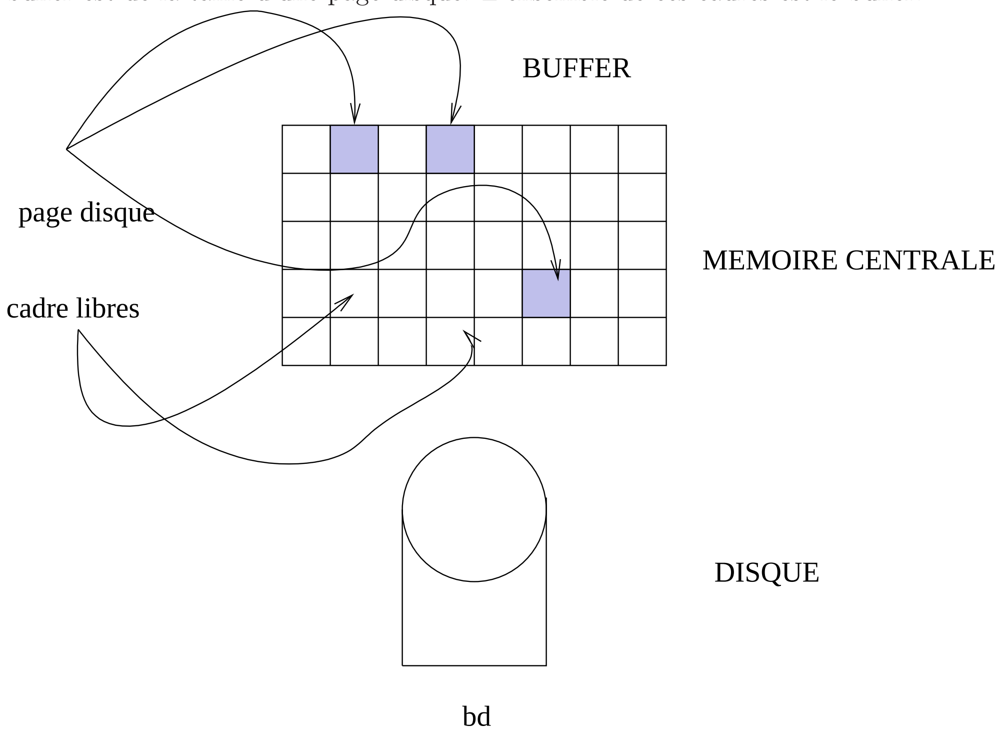
Le gestionnaire de l'espace disque d'un SGBD peut être construit en utilisant les fichiers de l'OS. Par exemple, la base peut résider sur un ou plusieurs fichiers pour lesquels un nombre de blocs seront alloués et initialisés (**par l'OS**) Le gestionnaire de l'espace disque est alors responsable de la gestion de l'espace **au sein** de ces fichiers.

Cependant, la plupart des SGBD ne se reposent pas sur le système de fichier de l'OS mais opèrent leur propre gestion de l'espace. Les raisons sont tant pratiques que techniques : en pratique les vendeurs de SGBD veulent supporter plusieurs plateformes et pour des raisons de **portabilité** préférent avoir du code "self-contained". Une seconde raison (technique celle la) est que sur un système 32-bits la plus grande taille de fichier est de 4GB alors qu'un SGBD peut vouloir avoir accès à un fichier bien plus grand que cela.

2.4 Gestion des buffers

La base entière ne peut résider en mémoire centrale. Les pages sont amenées en mémoire centrale et une table (buffer) stocke le fait que la page est résidente en MC. Lors d'un accès futur à cette page le gestionnaire de buffer saura qu'elle est présente et l'accès disque sera évité. Lorsque le buffer est plein le gestionnaire de buffer doit décider quelle page doit être renvoyée sur disque. Ce processus est régit par une *politique de replacement.*

Plus précisément le gestionnaire de buffer est le logiciel en charge d'amener et de renvoyer des pages du (sur le) disque. Il partitionne la mémoire centrale en cadres (frames) qui sont des pages mémoires centrales. donc un cadre du buffer est de la taille d'une page disque. L'ensemble de ces cadres est le buffer.



Le code des applications de plus haut niveau du SGBD n'a pas à se préoccuper de savoir si une page réside en MC ou non. La hiérarchie de mémoire est transparente. Cependant, lorsque un programme de haut niveau n'a plus besoin de la page sur laquelle il travailait, il en informe le gestionnaire de buffer afin que le cadre correspondant puisse être réutilisé. De même, le programme doit informer le gestionnaire de buffer de toute modification opérée sur la page afin que la mise à jour puisse être propagée sur la copie de la page qui réside en mémoire secondaire.

Outre la gestion du buffer, le gestionnaire de buffer maintient un certain nombre d'informations. Pour chaque cadre il gère deux variables:*pin-count* et *dirty*. Le pin-count contient le nombre d'utilisateurs courant de la page et est initialisé à 0. La variable booléenne dirty indique si la page a été modifié depuis qu'elle a été amenée du disque et est initialisé à false.

Lorsqu'une demande de page est effectuée par un appel de plus haut niveau, le gestionnaire de buffer :

1. vérifie dans le buffer si la page est contenue dans un cadre, si oui le pin-count est incrémenté pour ce cadre. Si la page n'est pas dans le buffer, le gestionnaire de buffer la charge du disque de la manière suivante :

(a) il sélectionne un cadre à renvoyer sur disque conformément à la politique de replacement, il incrémente le pin-count.

(b) si le dirty bit du cadre contenant la page à renvoyer sur disque est a vrai il copie la version de la page contenue dans le cadre sur disque

(c) il charge du disque la page demandée dans le cadre.

2. renvoie au programme l'adresse en MC du cadre contenant la page demandée.

.

2.4.1 Politiques de replacement

(LRU): : imlantée au moyen d'une liste chainée de cadres ayant leurs pin-count à 0. Dès qu'un cadre va son pin-count qi tombe à 0, il est inséré en fin de liste. La page qui sera choisie pour être réecrite sur disque sera celle contenue dans le cadre en tête de liste.

2.4.2 Gestion des buffers : OS ou SGBD?

Des similarités évidentes existent entre la notion de mémoire virtuelle des systèmes d'exploitation t la gestion de buffer des SGBD. Dans les deux cas, il s'agit d'offrir l'accès à plus de données que ne saurait en contenir la mémoire centrale. Dans les deux cas les pages sont amenées du disque quand cela est nécessaire puis renvoyées quand il le faut. Pourquoi ne pas construire le SGBD en utilisant le mécanisme de mémoire virtuelle de l'OS ?

Un premier élément de réponse est qu'un SGBD peut souvent **prédire** bien plus finement qu'un OS, l'ordre dans lequel les pages vont être accédées (ce que l'on appelle des **pages reference patterns** que l'on peut traduire par schémas de référence d'accès aux pages).et donc il est important d'exploiter ce phénomène. En outre un SGBD (pour des raisons liées à la pertinence de l'information stockée dans les données - intégrité) a besoin d'un contrôle plus poussé lors du renvoi d'une page sur diaque qu'un OS ne l'a.

Un SGBD est souvent mieux à même de prdire des schémas d'accès aux pages car les demandes de pages sont opérées par des opérations de plus haut niveau (parcours séquentiels ou implantations particulières d'opérateurs relationnels) dont on connait le mode de fonctionnement. Cette capacité de prédiction permet d'obtenir un meilleur choix des pages à replacer sur disque et permet ainsi d'avoir des politiques de replacement spécifiques au contexte bases de données.

En outre, et cela est peut-être encore plus important: pouvoir prédire des schémas d'accès aux pages permet d'effectuer du **prefetching** de pages. Cette stratégie consiste à amener les pages dans le buffer, avant même qu'elles n'aient été demandées, anticipant ainsi le besoin. Les avantages sont multiples : (i) les pages sont la quand on en a besoin, (ii) lire des blocs contigus de pages est plus efficace que faire des lectures aléatoires (rappelons nous la géométrie d'un disque) (iii) même si les pages ne sont pas dans des blocs contigus, en opérant du prefetching, il est possible d'ordonnacer les lectures de manière à minimiser les mouvement des têtes de lecture, iv une fois la demande page envoyée au disque, la CPU peut travailler à autre chose.

Enfin, un SGBD doit pouvoir avoir la possibilité de forcer explicitement l'écriture d'une page sur le disque. En particulier, il se peut qu'il faille impérativement propager sur disque la mise à jour d'une page. De même, le SGBD peut avoir besoin d'écrire des pages *avnt* d'autres (pour implanter le protocole Write Ahead Log - WAL en cas recouvrement sur panne). La gestion de la mémoire virtuelle d'un OS ne permet pas d'obtenir une telle précision du contrôle.

2.5 Fichiers et enregistrements

Nous nous intéressons à la manière d'utiliser les pages afin de stocker des enregistrements et à la manière d'organiser ces pages en fichiers logiques.Le module du SGBD en charge de cela est le **gestionnaire de l'espace disque** Un fichier est un ensemble d'enregistrements. Le gestionnaire de l'espace disque a en charge **l'organisation du fichier**.

2.5.1 L'organisationr Séquentielle (heap file organization)

Le type le plus simple d'organisation de fichier est l'organisation *séquentiel* ou **heap file organization** en anglais. Dans ce cas les enregistrements sont stockés dans le désordre sur les pages allouées au fichier. Une telle organisation offre des promitives de recherche pour tous les enregistrements du fichier, pour un enregistrement particulier identifié par son Rid (Record Id). Le gestionnaire de fichier doit conserver la trace des pages allouée au fichier.

Nous précisons maintenant l'implantation d'une organisation séquentielle du fichier.

L'unique moyen de retrouver un enregistrement dans une telle organisation est de parcourir un à un les enregistrements jusquà ce que l'on trouve celui cherché (identifié par son Rid). Evidemment chaque enregistrement du fichier possède un unique Rid ! Toutes les pages du fichier sont de la même taille. Outre les opérations de parcours (**scan**) leórganisation en heap permet de faire des *création*, *destruction* de fichiers, des opérations d *insertion*, de *destruction d'un enregistrement* , d'*accès* à un enregsitrement identifié par son Rid. Notons que nous devons être en mesure de trouver l'identificateur de la page où se trouve l'enregistrement identifié par son Rid (le Rid doit contenir cette information). .

Pour offrir ces fonctionnalités ils existe deux solutions représentées en Figures 2.5.1 et 2.5.1.

Dans les deux cas il faut conserver la trace des pages allouées pour chaque fichier ainsi que celles qui qui contiennent de l'espace disponible. Chaque page contient deux pointeurs (qui sont des identificateurs de pages) pour gérer le fichier et pas uniquement les données.

Liste chainée de pages

L'idée est de maintenir une liste chainée de page (Figure 2.5.1). Le SGBD maintient une table dont le contenu consiste en une paire (fichier, addresse première page). Cette table est stockée sur un endroit précis, connu du disque. Cette première page est appelée page de tête.

Une tache importante est de maintenir l'information relative aux slots libérés lors de la destruction d'enregistrements. Cela impose d'être capable (i) de gérer l'espace **dans** une page et (ii) de gérer les pages **possédant** de l'espace libre. Nous reviendrons sur le prmeir point en section 2.6. Pour le second point l'idée consiste à maintenir deux listes chainées différentes.

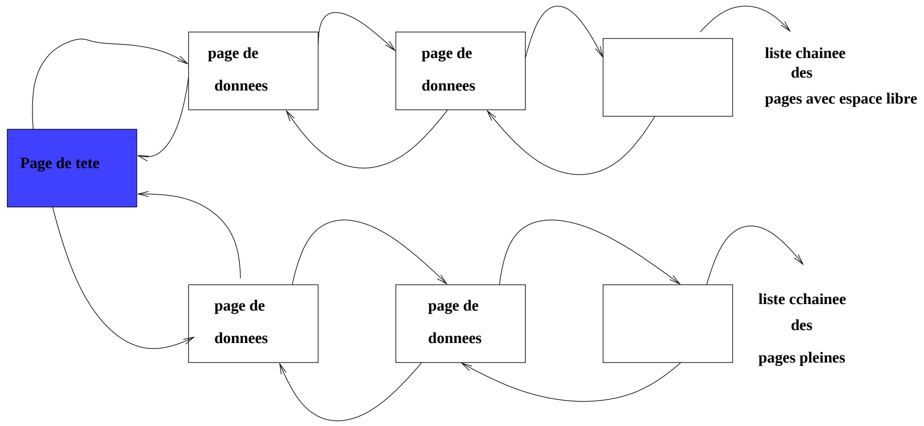


Figure 2.1: Organisation du heap avec listes chainées

Le désavantage d'une telle organisation est que si les enregistrements sont de taille variable, la plupart des pages seront sur la liste des pages ayant de l'espace libre., en cas d'insertion il faudra parcourir plusieurs pages avant de trouver une candidate.

La deuxième solution, basée sur un répertoire, permet de répondre à ce problème.

Répertoire (Directory) de pages

Dans ce cas le système maintient un répertoire de pages qui est lui même une collection de pages chainées entre elles (d'autres organisations sont possibles). Chaque entrée du répertoire identifie une page (ou une séquence de pages) du fichier séquentiel.

L'espace libre est géré en associant à chaque entrée du répertoire un bit indiquant si la page contient de l'espace libre et/ou combien.

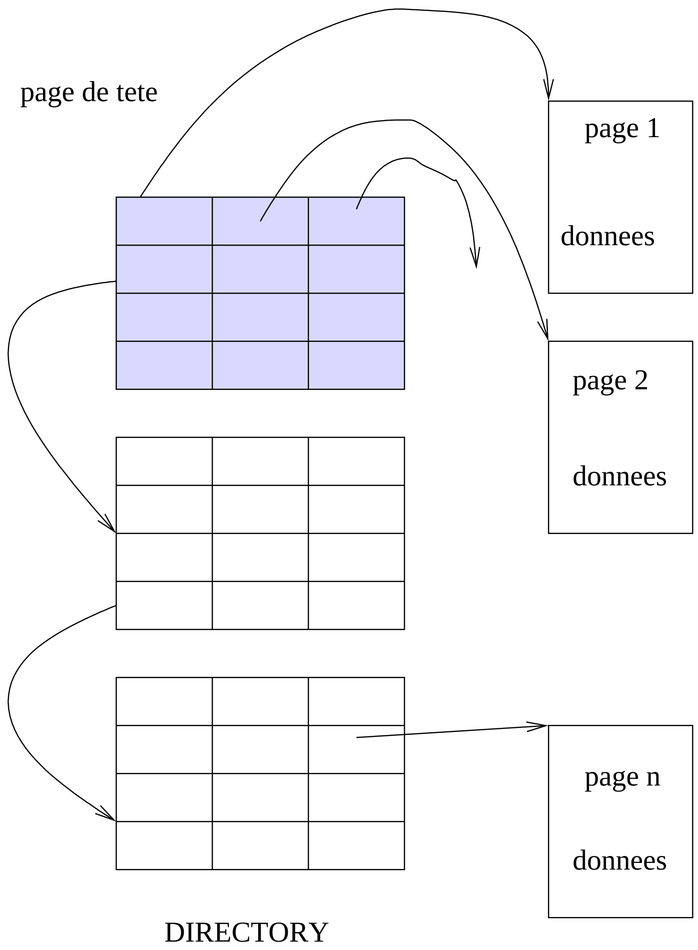


Figure 2.2: Organisation du heap avec Directory

2.6 Formats de page

La notion de page est une abstraction adaptée pour traiter des aspects liés aux I/O. Cependant les couches plus hautes du SGBD voient les données comme des enregistremants. Se pose la question de savoir comment des ensembles de tels enregistrements sont rangés dans les pages.

On peut voir une page comme une collections de slots (places), chaque slot contenant un enregistrement. Un enregistrement est identifié en utilisant la paire (id page, numéro de slot) comme rid (une autre possibilité est d'assigner à chaque enregistrement un numéro logique unique et de maintenir une table de correspondance).

Nous décrivons différentes façons de gérer les slots au sein d'une page. Les points importants sont de savoir comment ces différentes approches permettent la recherche, l'insertion ou la destruction d'enregistrements dans les pages.

2.6.1 Enregistrements de taille fixe

Si tous les enregistrements ont une taille fixe les slots sont consécutifs. A tout moment, certains sont occupés et d'autres libres. Pour insérer un nouvel enregistrement, il suffit de chercher un slot libre. Le tout est de garder trace de ces slots. La première solution consiste à stocker les enregistrements dans les N premiers slots, lorsqu'un enregistrement est détruit le dernier enregistrement est déplacé à l'endroit libéré. Cette solution permet d'accèder au ième enregistrement par un simple calcul d'offset et tous les slots libres sont en fin de page. Cependant, une telle soltion ne fonctionne pas dans le cas où des enregistrements extérieurs à la page référencent l'enregistrement (car le rid *contient* le numéro de slot).

La seconde solution consiste àgérer les destructions en maintenant un tableau de bits (un par slot) .

Les deux solutions sont représentées sur la Figure 2.6.1.

2.6.2 Enregistrements de taille variable

Voir la Figure 2.7

2.7 Formats d'enregistrements

Voir les Figures.

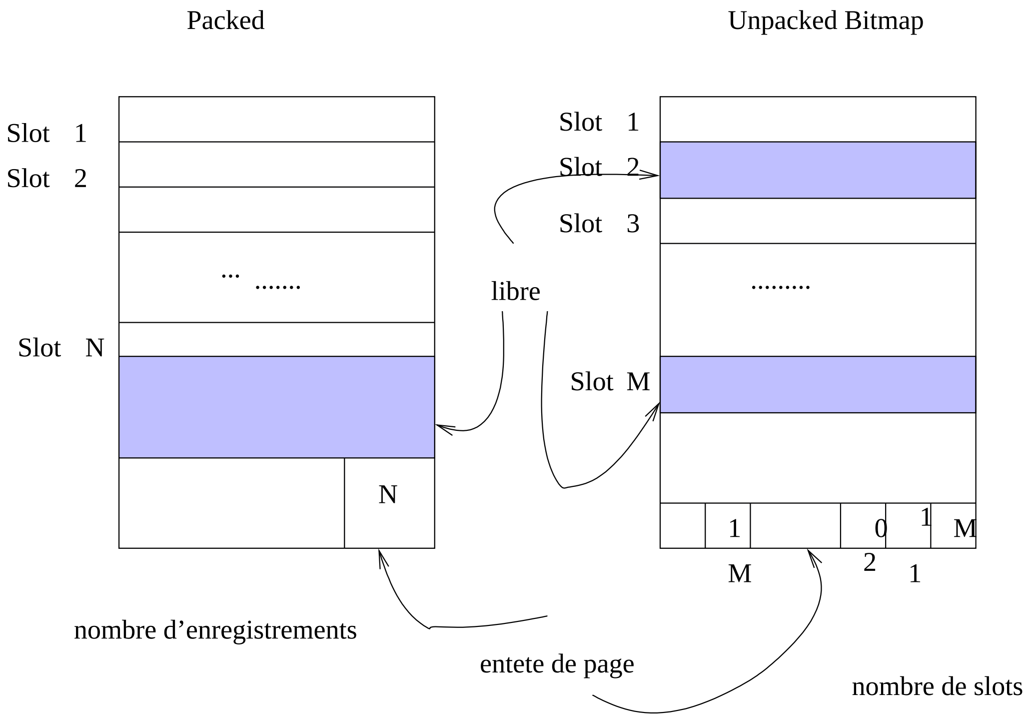


Figure 2.3: Page avec Enregistrements de taille fixe : deux possibilités

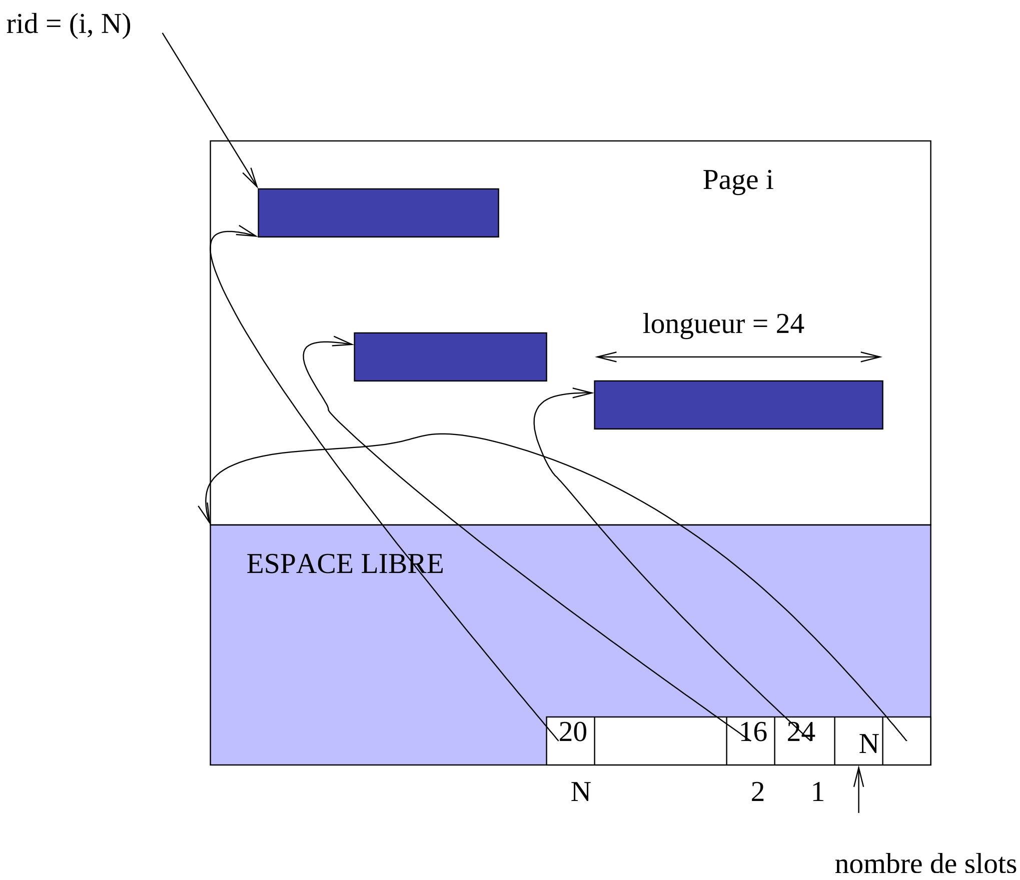


Figure 2.4: Pages avec Enregistrements de taille variable

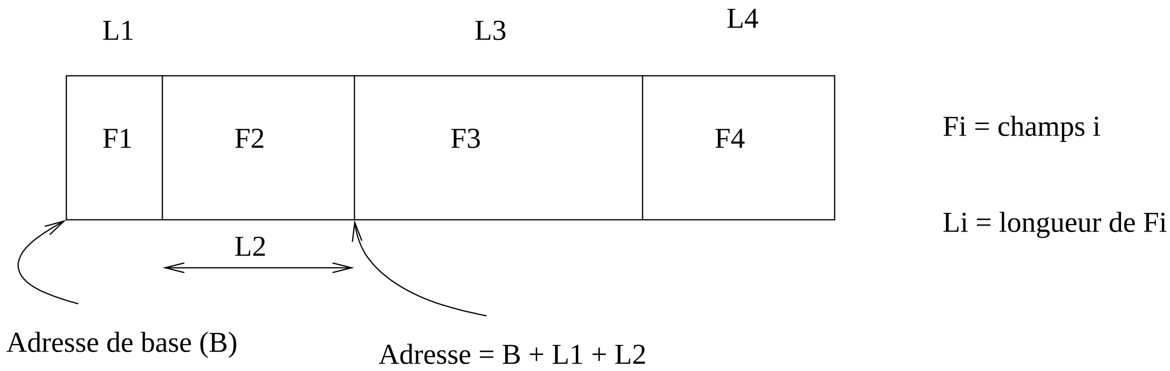


Figure 2.5: Format Enregistrements de taille fixe

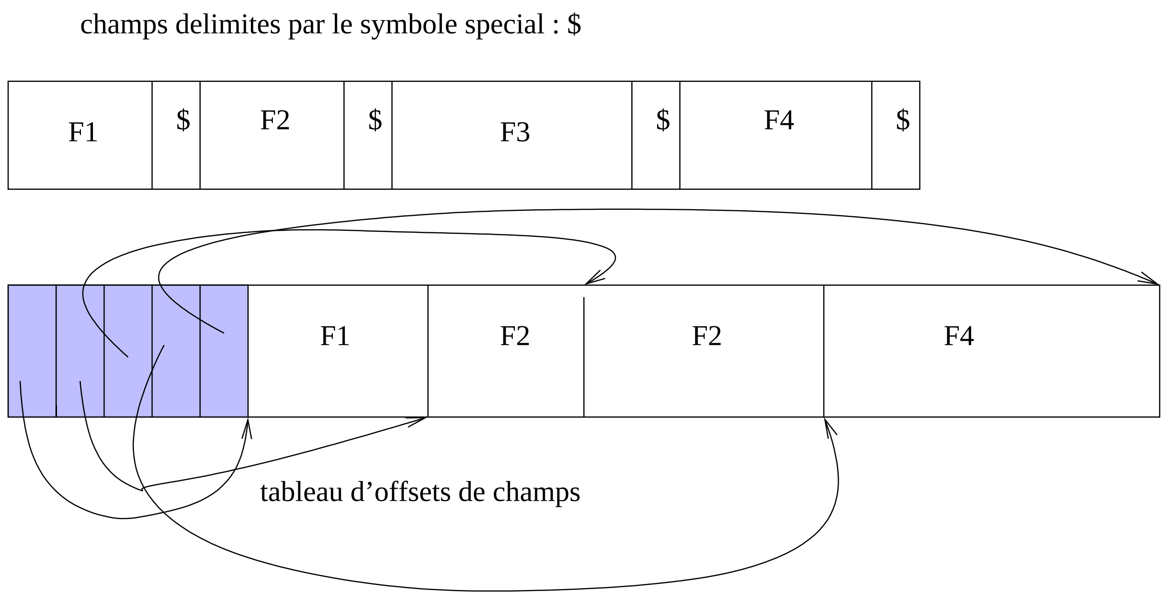


Figure 2.6: Format Enregistrements de taille variable

Chapter 3

Indexation

3.1 Introduction

Nous présentons les concepts élémentaire et la terminologie la plus admise. Il n'existe pas de terminologie standard et nous essaierons de lier les différentes appellations.

3.1.1 Préambule et Définitions

Un **index** est une structure de données qui *organise* les enregsitrements sur le disque afin d'optimiser certaines catégories d'opérations de recherche. Un index permet de retrouver efficacement tous les enregistrements satisfaisant un critère de sélection sur la **clef de recherche** de l'index. Il est tout à fait possible d'associer plusieurs index à un même ensemble d'enregistrements (sur d'autres clefs de recherche) afin d'optimiser des accès particuliers.

Nous utiliserons le terme **entrée** pour qualifier les enregistrments stockés dans i fichier d'index. Nous noterons $k*$ le contenu d'une entrée ayant pour valeur de clef de recherche : k. $k*$ contient suffisemment d'informations pour localiser (le ou les) enregistrements de données ayant comme clef de recherche k. Les enregistrements de données peuvent être distincts des entrées.

Il existe trois manières de représenter le contenu d'un entrée dans un index :

1. **cas 1** Une entrée $k*$ est le véritble enregistrement de données (de clef de recherche k).

2. **cas 2** Une entrée est une paire (k, rid) où rid est l'identificateur d'un enregistrement de données de clef k.

3. **cas 3** Une entrée est une paire (k, $liste - rid$) où $liste - rid$ est une liste d'identificateurs d'enregistrements de données de clef k.

Bien entendu, si l'index est utilisé pour stocker de véritables enregistrements de données nous sommes dans le cas 1. Nous pouvons alors considérer le cas 1 comme une organisation particulière de fichier (dans le même sens que le heap ou le fichier trié). Une telle organisation s'appelle **fichier indexé**.

Dans les cas 2 et 3, les entrées *pointent* vers les enregistrements de données. Ces deux cas sont donc *indépendants* de l'organisation du fichier de données. Le cas 3 offre une meilleure utilisation de l'espace mais les ent˚ees sont des enegistrements de taille variable.

Enfin, si l'on souhaite construire plusieurs index sur une même collection d'enregistrements au plus un index peut-être du cas 1 (sinon il faudrait stocker plusieurs fois le même enregistrement de données ce qi n'est vraiment pas souhaitable).

3.1.2 Index plaçants ou groupants - clustered indexes

Lorsqu'un fichier est organisé de sorte que l'ordre des enregistrements de données soit celui ou soit proche de celui des entrées de l'index, on dit que l'index est **plaçant** ou **groupant**. Sinon on dit que l'index n'est pas groupant.

Un index utilisant le cas 1 est groupant par définition. Un index utilisant les cas 2 ou 3 peut être groupant seulement si les enregistrements de données sont triés sur la clef de recherche.

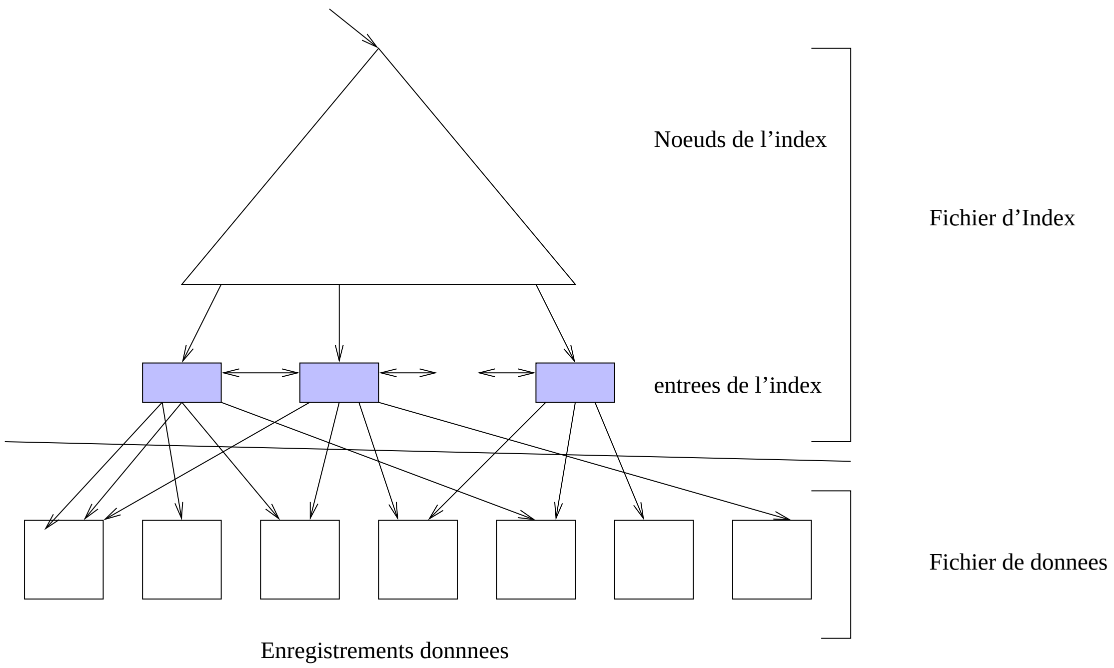


Figure 3.1: Index non groupant utilsant le cas 2

En pratique les fichiers sont rarement triés car le coût de maintenace de l'ordre est trop élevé. Ainsi, en pratique, un index groupant est un index utilisant le cas 1.

3.1.3 Index primaires et secondaires

Un index portant sur un ensemble de champs contenant une clef primaire (au sens du modèle relationnel) est dit **primaire** sinon il est **secondaire**.

Les termes primaires et secondaire sont quelquefois employés avec un autre sens. Un index conforme au cas 1 est souvent appelé primaire et secondaire pour les cas 2 ou 3.

Enfin deux entrées sont dites **duppliquées** si elles ont la même valeur pour la clef de recherche de l'index.. un index primaire ne contient pas de duppliqués. En général un index secondaire en contient. On parle aussi d'index **non dense** et **denses** selon que toutes les valeurs de clefs sont repéntées (dense) ou non (non dense).

Un point important est *l'orgsnisation* des entrées d'un fichier d'index.

3.2 Structures d'index

Il existe deux grandes classes d'organisation des entrées d'un (fichier d') index : les techniques par hachage et les technique basées sur une structure de données arborescente. Les deux type d'organisation peuvent être combinées !

3.2.1 Hachage - Hash Based

Dans une telle approche, les entregistrements sont placés dans des **paquets** (**buckets** en anglais). Un paquet consiste en une page primaire et éventuellement d'autres pages chainées entre elles. Le paquet correspondant à un enregistrement donnéest déterminé en appliquant la fonction de hash à la clef de recherche. Cette fonction donne le numéro du paquet et ainsi l'adresse de la page primaire du paquet qui peut être retrouvée en un ou deux accès disques.

Lors d'une insertion, l'enregisatrement est placé dans le paquet lui correspondant. Des pages d'overflow peuvent être allouées le cas échéant. Pour rechercher un enregistrement de clef donnée, on applique la fonction de hash à la valeur de la clef et on récupère tous les enregistrements du paquet. Si l'on souhaite faire une recherche sur un autre champs que la clef il faudra parcourir tout le fichier.

3.2.2 Arbres - Tree based

Cette approche vise à organiser les enregistrements en utilisant une structure auxilliaire arborescente. Les entrées de l'index sont placées en respectant l'ordre des valeurs de clef de recherche. Une structure de données arborescente est maintenue pour orienter les recherches vers la page correspondante contenant les entrées.

Les feuilles de l'arbre contiennent les entrées de l'index. Cette structure permet de localiser efficacement les entrées avec des valeurs de clef données. Toutes les recherches partent de la racine et les sommets intermédiaire redirigent la recherche en opérant une comparaison de clefs. Les sommets non feuilles contiennent en fait des pointeurs (ou adresses) séparés par des valeur de clef de recherche. Le pointeur situé à gauche de la valeur k pointe vers un sous-arbre ne contenant que des entrées dont la valeur est strictement inférieure à k.

Afin d'opérer des recherches par plage de valeurs (intervalles) les feuilles de l'arbre sont doublement chainées..

Le nombre d'entrées sorties pour une recherche est égal à la longueur du chemin à la page feuille (chaque sommet au pire est dans une page distincte) plus le nombre de pages feuilles contenant des entrées pertinentes.

L'arbre $\mathbf{B}^+$ est une structure qui assure que tous les chemins de la racine aux feuilles ont la même longueur : la structure est toujours équilibrée en profondeur. Avec une telle structure, trouver la feuille candidate est plus rapide qu'avec une recherche dichotomique sur une fichier trié car chaque sommet non feuille peut contenir un très grand nombre de pointeurs et la **hauteur** de l'arbre est rarement supérieure à trois ou quatre en pratique.

Le nombre moyen de fils pour un sommet est appelé le **degré** (**fan-out** en anglais) de l'arbre. Si h est la hauteur de l'arbre et d son degré moyen une bonne approximation du nombre de feuilles est d^h.

En pratique d est de l'ordre de 100. donc avec un arbre de hauteur 4, nous indexerons 100 million de feuilles. Nous pourrons avor accès à chaque page en 4 entrées/sorties. Une recherche dichotomique sur le même fichier nous donnerait $log_2(100000000)$ de l'ordre de 25 E/S.