
Cours de bases de données – Aspects système

Version 1.0

Philippe Rigaux

24 February 2016

1	Introduction	3
2	Dispositifs de stockage	5
2.1	S1 : Supports de stockage	5
2.2	S2 : Gestion des mémoires	11
3	Fichiers séquentiels	19
3.1	S1 : Enregistrements	19
3.2	S2 : fichiers	24
4	Indexation : l'arbre B	29
4.1	S1 : Indexation de fichiers	30
4.2	S2 : L'arbre-B	37
5	Opérateurs et algorithmes	43
5.1	S1 : Modèle d'exécution : les itérateurs	43
5.2	S2 : les opérateurs de base	47
5.3	S3 : Le tri externe	53
5.4	S4 : Algorithmes de jointure	57
6	Evaluation et optimisation	69
6.1	S1 : traitement de la requête	69
6.2	S2 : optimisation de la requête	73
6.3	S3 : illustration avec ORACLE	79
7	Transactions	87
7.1	S1 : Transactions	88
7.2	S2 : effets indésirables des transactions concurrentes	96
7.3	S3 : choisir un niveau d'isolation	101
8	Contrôle de concurrence	111
8.1	S1 : Les mécanismes	111
8.2	S2 : les algorithmes	114
9	Reprise sur panne	119
9.1	S1 : mécanismes fondamentaux	119
9.2	S2 : Algorithmes de reprise sur panne	122
10	Indices and tables	127

Contents : Le document que vous commencez à lire fait partie de l'ensemble des supports d'apprentissage proposés sur le site <http://www.bdpedia.fr>. Il fait partie du cours consacré aux bases de données relationnelles, divisé en deux parties :

- La version en ligne du support “Modèles et langages” est accessible à <http://sql.bdpedia.fr>, la version imprimable (PDF) est disponible à <http://sql.bdpedia.fr/files/cbd-sql.pdf>.
- La version en ligne du support “Aspects système” est accessible à <http://sys.bdpedia.fr>, la version imprimable (PDF) est disponible à <http://sys.bdpedia.fr/files/cbd-sys.pdf>.

Deux autres cours, aux contenus proches, sont également disponibles :

- Un cours sur les bases de données documentaires et distribuées à <http://b3d.bdpedia.fr>.
- Un cours sur les applications avec bases de données à <http://orm.bdpedia.fr>

Reportez-vous à <http://www.bdpedia.fr> pour plus d'explications.

Important : Ce cours de Philippe Rigaux est mis à disposition selon les termes de la licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International. Cf. <http://creativecommons.org/licenses/by-nc-sa/4.0/>.

Tout le matériel proposé ici sert de support au cours “Aspects systèmes des bases de données” proposé par le département d'informatique du Cnam. Le code du cours est NFP107 (voir le site <http://depinfocnam.fr/new/spip.php?rubrique220> pour des informations pratiques). Il est donné en

- Cours présentiel (second semestre, mercredi soir)
- Cours à distance (premier semestre, avec supports audiovisuels)

Introduction

Supports complémentaires :

- Diapositives : (à venir)
 - Vidéo de présentation (à venir)
-

Les Systèmes de Gestion de Bases de Données (SGBD) sont des logiciels complexes qui offrent un ensemble complet et cohérent d'outil de gestion de données : un langage de manipulation et d'interrogation (SQL par exemple), un gestionnaire de stockage sur disque, un gestionnaire de concurrence d'accès, des interfaces de programmation et d'administration, etc.

Les systèmes présentés ici sont les SGBD Relationnels, simplement appelés systèmes relationnels. Il s'agit de la classe la plus répandue des SGBD, avec des représentants bien connus comme Oracle, MySQL, SQL Server, etc. Tous ces systèmes s'appuient sur un modèle de données normalisé, dit relationnel, caractérisé notamment par le langage SQL.

Les documents proposés ici proposent d'aller "sous le capot" de ces systèmes pour étudier comment ils fonctionnent et réussissent le tour de force de proposer des accès sécurisés à des centaines d'utilisateurs en parallèle, tout en obtenant des temps de réponses impressionnantes même pour des bases très volumineuses. Le contenu correspond typiquement à un cours universitaire de deuxième cycle en informatique. Il couvre les connaissances indispensables à tout informaticien de niveau ingénieur.

Le cours est découpé en *chapitres*, couvrant un sujet bien déterminé, et en *sessions*. Nous essayons de structurer les sessions pour qu'elles demandent environ 2 heures de travail personnel (bien sûr, cela dépend également de vous). Pour assimiler une session vous pouvez combiner les ressources suivantes :

- La lecture du support en ligne : celui que vous avez sous les yeux, également disponible en PDF ou en ePub.
- Le suivi du cours, en vidéo ou en présentiel.
- Le test des exemples de code fournis dans chaque session.
- La réalisation des exercices proposés en fin de session.
- Dans la plupart des chapitres, des Quiz sur des questions de cours ; si vous ne savez pas répondre à une question du Quiz :, relisez le chapitre et approfondissez.

La réalisation des exercices est essentielle pour vérifier que vous maîtrisez le contenu.

Vous devez maîtriser le contenu des sessions *dans l'ordre où elles sont proposées*. Commencez par lire le support, jusqu'à ce que les principes vous paraissent clairs. Reproduisez les exemples de code : tous les exemples donnés sont testés et doivent donc fonctionner. Le cas échéant, cherchez à résoudre les problèmes par vous-mêmes : c'est le meilleur moyen de comprendre. Finissez enfin par les exercices. Les solutions sont dévoilées au fur et à mesure de l'avancement du cours, mais si vous ne savez pas faire un exercice, c'est sans doute que le cours est mal assimilé et il est plus profitable d'approfondir en relisant à nouveau que de simplement copier une solution.

Enfin, vous êtes totalement encouragé(e) à explorer par vous-mêmes de nouvelles pistes (certaines sont proposées dans les exercices).

Dispositifs de stockage

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. Un concurrent sérieux est le *Solid State Drive*, dont les performances sont très supérieures, et le coût est en baisse constante, ce qui le rend de plus en plus concurrentiel.

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.

Important : Une donnée qui n'est pas sur un support persistant est potentiellement perdue en cas de panne.

L'accès à des données stockées sur un support persistant, 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 et les algorithmes de recherche utilisés constituent donc des aspects essentiels des SGBD du point de vue des performances.

Une bonne partie du présent cours est, de fait, consacrée à des méthodes, techniques et structures de données dont le but principal est de limiter le nombre et la taille de données lues sur le support persistant.

Un bon système se doit d'utiliser au mieux les techniques disponibles afin de minimiser les temps d'accès. Dans ce chapitre nous décrivons les techniques de stockage de données et le transfert de ces dernières entre les différents niveaux de mémoire d'un ordinateur. La première partie est consacrée aux dispositifs de stockage. Nous détaillons successivement les différents types de mémoire utilisées, en insistant particulièrement sur le fonctionnement des disques magnétiques. Nous abordons en seconde partie les principales techniques de gestion de la mémoire utilisées par un SGBD.

2.1 S1 : Supports de stockage

Supports complémentaires :

- Diapositives : supports de stockage
 - Vidéo de présentation (à venir)
-

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 *persistantes* comme les disques magnétiques, les SSD, les CD ou les bandes magnétiques, préservent leur contenu même en l'absence d'alimentation électrique.

2.1.1 Mémoires

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 [Hiérarchie des mémoires](#)), 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 une mémoire intermédiaire permettant au processeur d'accéder très rapidement aux données à traiter ;
2. la *mémoire vive*, ou *mémoire principale* stocke les données et les processus constituant l'espace de travail de la machine ; toute information (donnée ou programme) doit être en mémoire principale pour pouvoir être traitée 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. les *Solid State Drive* ou SSD sont une alternative récente aux disques magnétiques ; leurs performances sont supérieures, mais leur coût élevé ;
5. enfin les CD ou les bandes magnétiques sont des supports très économiques mais leur lenteur les destine plutôt aux sauvegardes à long terme.

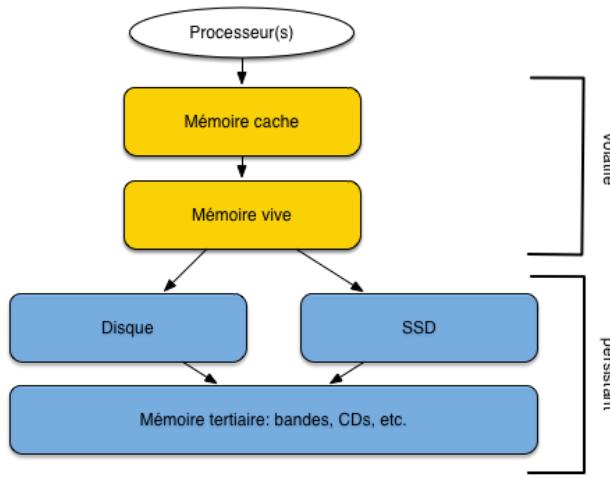


Fig. 2.1 – Hiérarchie des mémoires

La *mémoire vive* (que nous appellerons *mémoire principale*) et les *disques* (ou *mémoire secondaire*) sont les principaux niveaux à considérer pour des applications de bases de données. Une base de données doit être 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 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.

Vocabulaire : Mais qu'est-ce qu'une “donnée” ?

Le terme de *donnée* désigne le codage d'une information applicative. Dans le contexte de ce cours, “donnée” sera toujours synonyme de tuple (ligne dans une table). On parlera parfois *d'enregistrement* quand l'aspect stockage est privilégié.

Dans d'autres contextes, une donnée peut être un document, un flux d'octets, etc. Du point de vue de l'application, et de l'information qu'elle manipule, une "donnée" est l'unité de lecture et d'écriture.

La technologie évoluant rapidement, il est délicat de donner des valeurs précises pour la taille des différentes mémoires. Un ordinateur est typiquement équipé de quelques Gigaoctets de mémoire vive (typiquement 4 ou 8 Go pour un ordinateur personnel, plusieurs dizaines de Go pour un serveur de données). La taille d'un disque magnétique est de l'ordre du Téraoctet, soit un rapport de 1 à 1 000 avec les données en mémoire centrale. Les SSD ont des tailles comparables à celles des disques magnétiques (pour un coût supérieur).

2.1.2 Performances des mémoires

Comment mesurer les performances d'une mémoire ? Nous retiendrons deux critères essentiels :

- *Temps d'accès* : connaissant l'*adresse* d'une donnée, quel est le temps nécessaire pour aller à l'emplacement mémoire indiqué par cette adresse et obtenir l'information ? On parle de lecture par clé ou encore *d'accès direct* pour cette opération ;
- *Débit* : quel est le volume de données lues par unité de temps dans le meilleur des cas ?

Le premier critère est important quand on effectue des accès dits *aléatoires*. Ce terme indique que deux accès successifs s'effectuent à des adresses indépendantes l'une de l'autre, qui peuvent donc être très éloignées. Le second critère est important pour les accès dits *séquentiels* dans lesquels on lit une collection d'information, dans un ordre donné. Ces deux notions sont essentielles.

Notion : accès direct/accès séquentiel

Retenez les notions suivantes :

- *Accès direct* : étant donné une adresse dans la mémoire, on accède à la donnée stockée à cette adresse.
 - *Accès séquentiel* : on parcourt la mémoire dans un certain ordre pour lire les données au fur et à mesure.
-

Les performances des deux types d'accès sont extrêmement variables selon le type de support mémoire. Le temps d'un accès *direct* en mémoire vive est par exemple de l'ordre de 10 nanosecondes (10^{-8} sec.), de 0,1 millisecondes pour un SSD, et de l'ordre de 10 millisecondes (10^{-2} sec.) pour un disque. Cela représente un ratio approximatif de 1 000 000 (1 million !) entre les performances respectives de la mémoire centrale et du disque magnétique ! Il est clair dans ces conditions que le système doit tout faire pour limiter les accès au disque.

Le tableau suivant résumé les ordres de grandeur des temps d'accès pour les différentes mémoires.

Tableau 2.1 – Performance des divers types de mémoire

Type mémoire	Taille	Temps d'accès aléatoire	Temps d'accès séquentiel
Mémoire <i>cache</i> (Static RAM)	Quelques Mo	$\approx 10^{-8}$ (10 nanosec.)	Plusieurs dizaines de Gos par seconde
Mémoire principale (Dynamic RAM)	Quelques Go	$\approx 10^{-8} - 10^{-7}$ (10-100 nanosec.)	Quelques Go par seconde
Disque magnétique	Quelques Tos	$\approx 10^{-2}$ (10 millisec.)	Env. 100 Mo par seconde.
SSD	Quelques Tos	$\approx 10^{-4}$ (0,1 millisec.)	Jusqu'à quelques Gos par seconde.

2.1.3 Disques

Les disques sont les composants les plus lents, et pourtant ils sont indispensables pour la gestion d'une base de données. Il est donc très utile de comprendre comment ils fonctionnent. Nous parlons essentiellement des disques magnétiques, avec un bref aperçu des SSD (dont l'importance est sûrement amenée à croître rapidement).

Dispositif

Un disque magnétique 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 (1 au minimum : le secteur est l'unité de lecture sur le disque). La taille d'un secteur est le plus souvent de 512 octets.

La plus 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, les octets groupés par 64 pour former des secteurs, et une suite de secteurs 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), 1 024 octets (deux secteurs) ou 4 096 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.

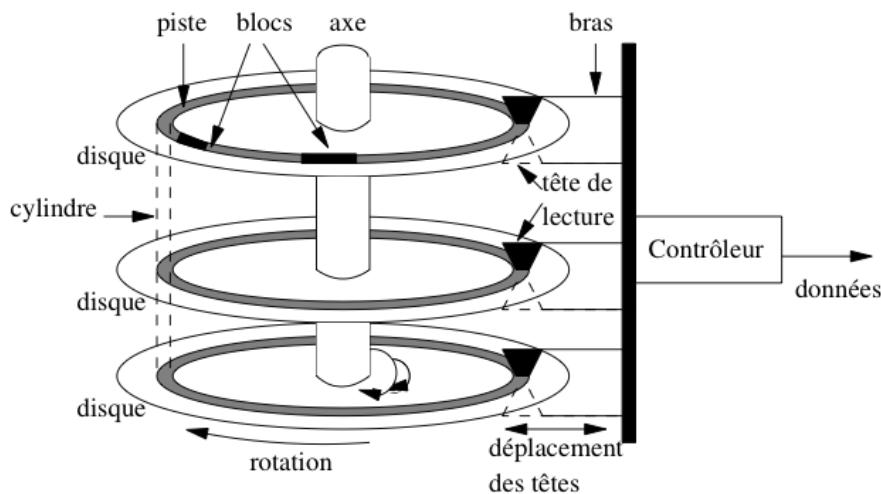


Fig. 2.2 – Fonctionnement d'un disque magnétique.

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 4 096 octets pour accéder à un entier de 4 octets, les 4 092 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. C'est un premier exemple du *principe de localité* que nous discutons plus loin.

Notion : bloc

Retenez la notion de *bloc*, zone mémoire contiguë stockée sur disque, lue ou écrite solidairement. *Le bloc est l'unité d'entrée/sortie entre la mémoire secondaire et la mémoire principale.*

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 l'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 *Fonctionnement d'un disque magnétique*). 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.

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.

Entrées/sorties sur un disque

Un disque est une mémoire à accès dit *semi-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. Mais contrairement à la mémoire principale, avant d'accéder à une adresse, il faut attendre un temps variable lié au mécanisme de rotation du disque.

L'accès est fondé sur une adresse donnée à chaque bloc au moment de l'initialisation du disque par le système d'exploitation. Cette adresse est 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 :

1. *positionnement de la tête de lecture* sur la piste contenant le bloc ;
2. *rotation du disque* pour attendre que le bloc passe sous la tête de lecture (rappelons que les têtes sont fixes, c'est le disque qui tourne) ;
3. *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.

La latence de lecture fait du disque une mémoire à accès semi-direct, comme mentionné précédemment. C'est aussi cette latence qui rend le disque lent comparé aux autres mémoires, *surtout si un déplacement des têtes de lecture est nécessaire*. Une conséquence très importante est qu'il est de très loin préférable de lire sur un disque en accès séquentiel que par une séquence d'accès aléatoires (cf. exercices).

Spécifications d'un disque

Le tableau *Spécification d'un disque* donne les spécifications d'un disque, telles qu'on peut les trouver sur le site de n'importe quel constructeur. 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 appartient au milieu de gamme.

Le disque comprend 5 335 031 400 secteurs de 512 octets chacun, la multiplication des deux chiffres donnant bien la capacité totale de 2,7 To. Les secteurs étant répartis sur 3 disques double-face, il y a donc $5\ 335\ 031\ 400 / 6 = 889\ 171\ 900$ 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 petites 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 $889\ 171\ 900 / 15300 = 58115$. On peut donc estimer qu'une piste stocke en moyenne $58115 \times 512 = 29$ Mégoctets. Ce chiffre donne le nombre d'octets qui peuvent être lus *en séquentiel*, sans délai de latence ni délai de positionnement.

Tableau 2.2 – Spécification d'un disque

Caractéristique	Performance
Capacité	2,7 To
Taux de transfert	100 Mo/s
Cache	3 Mo
Nbre de disques	3
Nbre de têtes	6
Nombre de cylindres	15 300
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

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 *Spécification d'un disque*. Le disque effectue 10 000 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.

2.1.4 Un mot sur les *Solid State Drives*

Un disque *solid-state*, ou SSD, est bâti sur la mémoire dite *flash*, celle utilisée pour les clés USB. Ce matériel est constitué de mémoires à semi-conducteurs à l'état solide. Contrairement aux disques magnétiques à rotation, les emplacements mémoire sont à accès direct, ce qui élimine le temps de latence.

Le temps d'accès direct est considérablement diminué, de l'ordre de quelques dixièmes de millisecondes, soit cent fois moins (encore une fois il s'agit d'un ordre de grandeur) que pour un disque magnétique. Le débit en lecture/écriture est également bien plus important, de l'ordre de 1 Go par sec, soit 10 fois plus efficace. La conclusion simple est que le meilleur moyen d'améliorer les performances d'une base de données est de la placer sur un disque SSD, avec des résultats spectaculaires ! Evidemment, il y a une contrepartie : le coût est plus élevé que pour un disque traditionnel, même si l'écart tend à diminuer. En 2015, il faut compter 200 à 300 Euros pour un disque SSD d'un demi Téraoctet, environ 10 fois moins pour un disque magnétique de même capacité.

Un problème technique posé par les SSD est que le nombre d'écritures possibles sur un même secteur est limité. Les constructeurs ont semble-t-il bien géré cette caractéristique, le seul inconvénient visible étant la diminution progressive de la capacité du disque à cause des secteurs devenus inutilisables.

En conclusion, les SSD ont sans doute un grand avenir pour la gestion des bases de données de taille faible à moyenne (disons de l'ordre du Téraoctet). Si on ne veut pas se casser la tête pour améliorer les performances d'un système, le passage du disque magnétique au SSD est une méthode sûre et rapide.

2.1.5 Exercices

Exercice : temps de lecture d'une base

Nous avons une base de 3 To. Quel est le temps de lecture minimal de la base complète sur un disque magnétique ? Et en mémoire centrale (en supposant la capacité de cette dernière suffisante) ?

On veut lire 100 objets de 10 octets. Combien de temps cela prend-il s'ils sont sur un disque ? Et s'ils sont en mémoire centrale ?

Exercice : lectures séquentielles et aléatoires sur un disque

On dispose d'une base de 3 Go constituée de 3 000 enregistrements (je vous laisse calculer la taille moyenne d'un enregistrement).

- Combien de temps prend la lecture complète de cette base avec un *parcours séquentiel* ?
- Combien de temps prend la lecture en effectuant un accès aléatoire pour chaque enregistrement ?

Vous pouvez prendre les valeurs du tableau *Performance des divers types de mémoire* pour les calculs.

Exercice : Spécifications d'un disque magnétique

Le tableau *Un (vieux) disque magnétique* donne les spécifications partielles d'un disque magnétique. Répondez aux questions suivantes.

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

Tableau 2.3 – Un (vieux) disque magnétique

Caractéristique	Valeur
Taille d'un secteur	512 octets
Nbre de plateaux	5
Nbre de têtes	10
Nombre de secteurs	5 335 031 400,00
Nombre de cylindres	10 000
Nombre de secteurs par piste	400
Temps de positionnement moyen	10 ms
Vitesse de rotation	7 400 rpm
Déplacement de piste à piste	0,5 ms

2.2 S2 : Gestion des mémoires

Supports complémentaires :

- Diapositives : gestion des mémoires
- Vidéo de présentation (à venir)

Un SGBD doit gérer essentiellement deux mémoires : la mémoire principale, et la mémoire secondaire (le disque). Toutes les données *doivent* être en mémoire secondaire, pour des raisons de *persistiance*. Une partie de ces données est en mémoire principale, pour des raisons de *performance*.

La figure *Le cache et le disque, ressources mémoires allouées au SGBD* illustre ces deux composants essentiels. Tout serveur de base de données s'exécute sur une machine qui lui alloue une partie de sa mémoire RAM, que nous appellerons *mémoire tampon* ou *cache* pour faire simple, ainsi qu'une partie du disque magnétique. Ces deux ressources sont gérées par un module du SGBD, le gestionnaire des accès (GA dans ce qui suit).

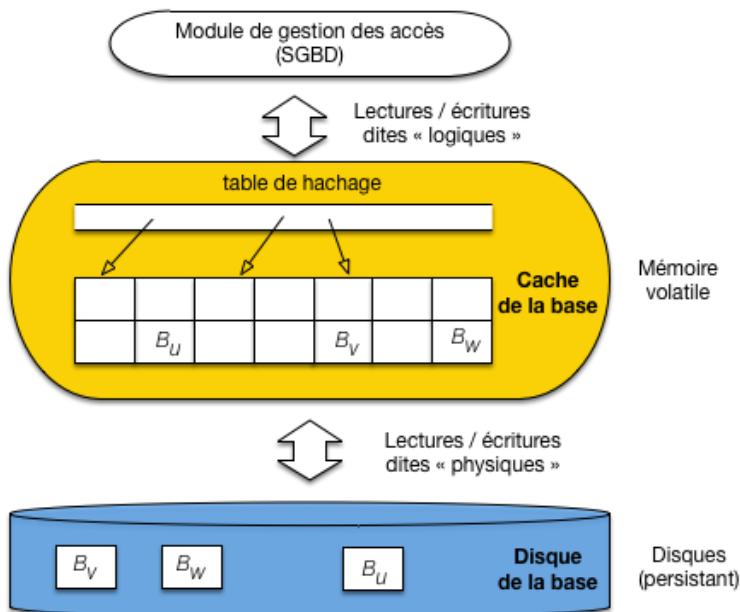


Fig. 2.3 – Le *cache* et le *disque*, ressources mémoires allouées au SGBD

Les données sont disponibles dans des *blocs* qui constituent l'unité de lecture et d'écriture sur le disque. Le GA exécute des requêtes de lecture ou d'écriture de données, et nous allons considérer comme acquis que ces requêtes comprennent *toujours* le numéro du bloc (sauf pour les insertions). Cette section explique comment ces requêtes sont exécutées.

2.2.1 Les lectures

Comme le montre la figure *Le cache et le disque, ressources mémoires allouées au SGBD*, le *cache* est constitué de blocs en mémoire principale qui sont des *copies* de blocs sur le disque. Quand une lecture est requise, deux cas sont possibles :

- la donnée fait partie d'un bloc qui est déjà dans le *cache*, le GA prend le bloc, accède à la donnée et la retourne ;
- sinon il faut d'abord lire un bloc du disque, et le placer dans le *cache*, pour se ramener au cas précédent.

La demande d'accès est appelée *lecture logique* : elle ignore si la donnée est présente en mémoire ou non. Le GA détermine si une *lecture physique* sur le disque est nécessaire. La lecture physique implique le chargement d'un bloc du disque vers le *cache*.

Note : Comment faire pour savoir si un bloc est ou non en *cache* ? Grâce à une table de hachage (illustrée sur la figure *Le cache et le disque, ressources mémoires allouées au SGBD*) qui pointe sur les blocs chargés en mémoire. L'accès à bloc, étant donnée son adresse, est extrêmement rapide avec une telle structure.

Un SGBD performant cherche à maintenir en mémoire principale une copie aussi large que possible de la base de données, *et surtout la partie la plus utilisée*. Le paramètre qui mesure l’efficacité d’une mémoire tampon est le *hit ratio*, défini comme suit :

$$\text{hit ratio} = \frac{\text{nbLecturesLogiques} - \text{nbLecturesPhysiques}}{\text{nbLecturesLogiques}}$$

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 (toutes les données demandées sont déjà en mémoire), le *hit ratio* est de 1.

Il est très important de comprendre que le *hit ratio* n’est pas simplement le rapport entre la taille de la mémoire cache et celle de la base. Ce serait vrai si tous les blocs étaient lues 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 des accès à la base qui peuvent lire n’importe quelle page avec une probabilité uniforme. Prenons un exemple pour clarifier les choses.

Un exemple pour comprendre

La base de données fait 2 GigaOctets, le cache 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 cache 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 accèderont 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 cache 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’aplatisse. En revanche sur une petite base où il est possible d’allouer un cache de taille comparable à la base, on a la possibilité d’obtenir un *hit ratio* optimal de 1.

2.2.2 Les mises à jour

Pour les lectures, les techniques sont finalement assez simple. Avec les écritures cela se complique, mais cela va nous permettre de vérifier la mise en œuvre d’un principe fondamental que nous retrouverons systématiquement par la suite :

Principe (rappel)

Il faut *toujours* éviter dans la mesure du possible d’effectuer des écritures aléatoires et préférer des écritures séquentielles*.

À ce principe correspond une technique, elle aussi fondamentale (elle se retrouve bien au-delà des SGBD), celle des fichiers journaux (*logs*).

Une approche naïve

En première approche, on peut procéder comme pour une lecture

- on trouve le bloc contenant la donnée, ou on le charge dans le *cache* s'il n'y est pas déjà ; la donnée fait partie d'un bloc qui est déjà dans le *cache*, le GA prend le bloc,
- on effectue la mise à jour sur la donnée dans le *cache* ;

On se retrouve dans la situation de la figure *Gestion naïve des écritures : les blocs doivent être écrits en ordre aléatoire* : le bloc contenant la donnée est marquée comme étant “modifié” (en pratique, chaque bloc contient un marqueur dit *dirty* qui indique si une modification a été effectuée par rapport à la version du même bloc sur le disque). On se trouve alors face à deux mauvais choix :

- soit on garde le bloc modifié en mémoire, en attendant qu'une opportunité se présente pour effectuer l'écriture sur le disque ; dans l'intervalle, toute panne du serveur entraîne la perte de la modification ;
- soit on écrit le bloc sur le disque pour remplacer la version non modifiée, et on se ramène à des écritures non ordonnées, aléatoires et donc pénalisantes.

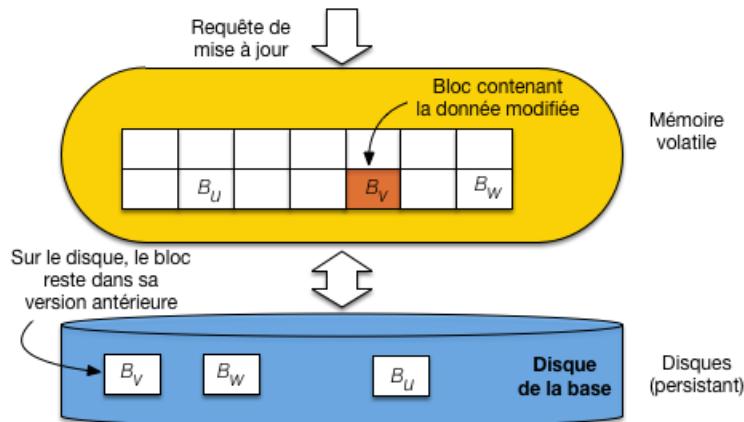


Fig. 2.4 – Gestion naïve des écritures : les blocs doivent être écrits en ordre aléatoire.

Il faut bien réaliser qu'un bloc peut contenir des *centaines* de données, et que déclencher une écriture sur disque dès que *l'une* d'entre elles est modifiée va à l'encontre d'un principe de regroupement qui vise à limiter les entrées/sorties sur la mémoire persistante.

Les fichiers journaux (*logs*)

La bonne technique est plus complexe, mais beaucoup plus performante. Elle consiste à procéder comme dans le cas naïf, *et à écrire séquentiellement* la mise à jour dans un fichier séquentiel, distinct de la base, mais utilisable pour effectuer une reprise en cas de panne.

La méthode est illustrée par la figure *Gestion des écritures avec fichier journal*. Le *cache* est divisé en deux parties, ainsi que la mémoire du disque. Idéalement, on dispose de deux disques (nous reviendrons sur ces aspects dans le chapitre *Reprise sur panne*). Le premier cache est celui de la base, comme précédemment. Le second sert de tampon, intermédiaire à un fichier particulier, le journal (ou *log*) qui enregistre toutes les opérations de mise à jour.

Au moment d'une mise à jour, le bloc de la base modifié n'est pas écrit sur disque. En revanche la commande de mise à jour est écrite *séquentiellement* dans le fichier journal.

On évite donc les écritures aléatoires, tout en s'assurant que toute commande de mise à jour est écrite sur disque et pourra donc servir à une reprise en cas de panne.

Que devient le bloc modifié dans le *cache* de la base ? Et bien, il sera écrit sur disque de manière opportuniste, quand un événement rendra cette écriture nécessaire. Par exemple :

- le *cache* est plein et il faut faire de la place ;
- le serveur est arrêté ;
- la base est inactive, et le système estime que le moment est venu de déclencher une synchronisation.

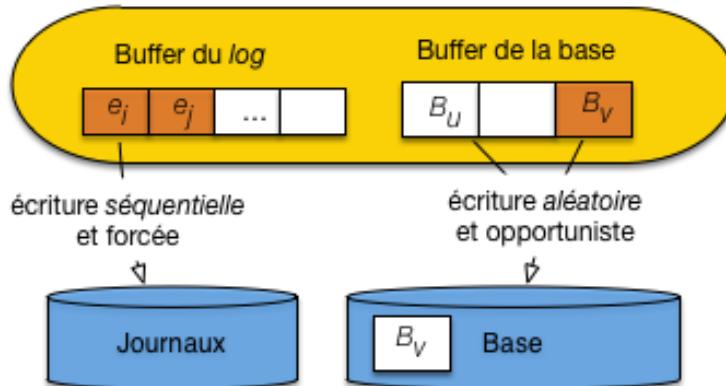


Fig. 2.5 – Gestion des écritures avec fichier journal

le point important, c'est qu'au moment de l'écriture effective, l'opération sera probablement bien plus "rentable" qu'avec la solution naïve. En premier lieu le bloc contiendra sans doute n données modifiées, et on aura donc remplacé n écritures (solution naïve) par une seule. En second lieu, il est possible que plusieurs blocs contigus sur le disque doivent être écrits, ce qui permet une écriture *séquentielle* évitant le délai de latence. Enfin, le système gagne ainsi une marge de manœuvre pour choisir le bon moment pour déclencher les écritures.

En résumé, cette technique basée sur les fichiers journaux, outre son intérêt intrinsèque, est une excellente illustration des efforts consacrés par les SGBD à privilégier les accès groupés et séquentiels au dépend des accès aléatoires et individuels.

Important : Les fichiers journaux sont à la base des techniques de reprise sur panne décrites dans le chapitre *Reprise sur panne*.

2.2.3 Le principe de localité

L'ensemble des techniques utilisées dans la gestion du stockage relève d'un principe assez général, dit de *localité*. Il résulte d'une observation pragmatique : l'ensemble des données utilisées par une application pendant une période donnée forme souvent un groupe bien identifié et présentant des caractéristiques de proximité.

- *Proximité spatiale* : Si une donnée d est utilisée, les données "proches" de d ont de fortes chances de l'être également
- *Proximité temporelle* : quand une application accède à une donnée d , il y a de fortes chances qu'elle y accède à nouveau peu de temps après.

- *Proximité de référence* : si une donnée $d1$ référence une donnée $d2$, l'accès à $d1$ entraîne souvent l'accès à $d2$.

Sur la base de ce principe, un SGBD cherche à optimiser le placement des données "proches" de celles en cours d'utilisation. Cette optimisation se résume essentiellement à déplacer dans la hiérarchie des mémoires des groupes de données proches de la donnée utilisée à un instant t . Le pari est que l'application accèdera à d'autres données de ce groupe. Voici quelques mises en application de ce principe.

Localité spatiale : 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 *Mauvaise et bonne organisation sur un disque* 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.

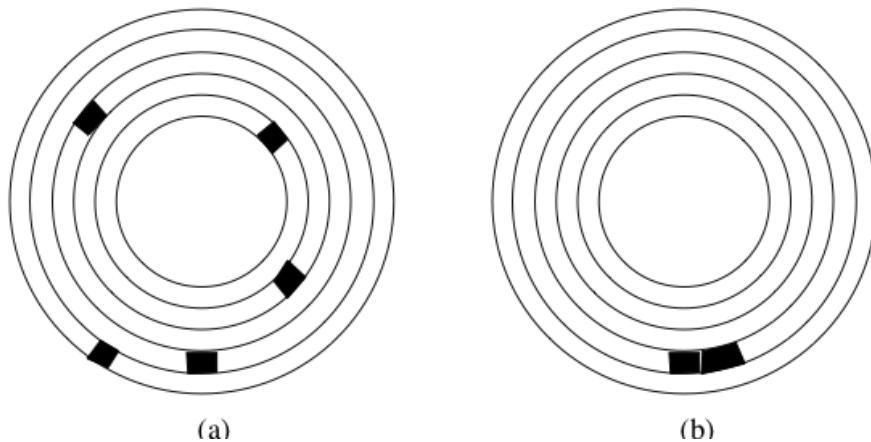


Fig. 2.6 – Mauvaise et bonne organisation sur un disque.

La lecture dans le premier cas implique 5 déplacements des têtes de lecture, et 5 délais de latence ce qui donne un temps de math : $5 \times (5.2 + 3) = 41$ 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 :

1. la proximité maximale est obtenue quand d_1 et d_2 sont dans le même bloc : elles seront alors toujours lues ensemble ;
2. le niveau de proximité suivant est obtenu quand les données sont placées dans deux blocs consécutifs ;
3. 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 ;
4. l'étape suivante est le placement des deux blocs dans un même cylindre, qui évite le déplacement des têtes de lecture ;
5. 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.

Important : Retenez qu'un critère essentiel de performance pour une base de données est le stockage le plus contigu possible des données.

Localité temporelle : ordonnancement

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 concurremment. Si un utilisateur A demande la lecture du fichier $F1$ tandis que l'utilisateur B demande la lecture du fichier $F2$, 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.

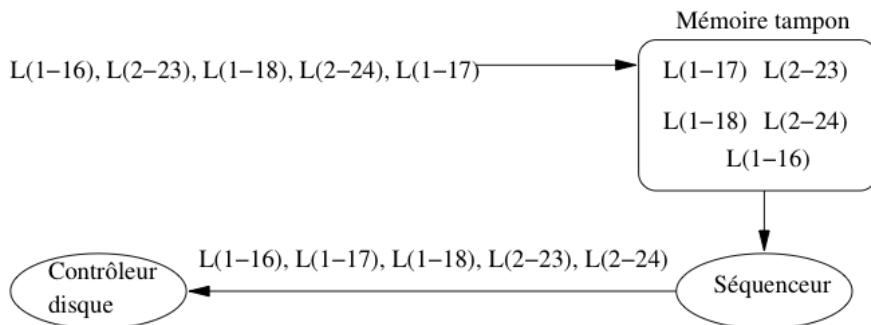


Fig. 2.7 – Séquencement des entrées/sorties

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 [Séquencement des entrées/sorties](#) 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.

Remplacement des blocs dans le cache

Pour finir sur les variantes de la localité, revenons sur ce qui se passe 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 remplacer sur le disque (opérations dites de *flush*). L'algorithme le plus courant 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. Ce bloc est alors soustrait de la mémoire centrale (il reste bien entendu sur le disque) et le nouveau bloc vient le remplacer.

Important : Si le bloc est marqué comme *dirty* (contenant des mises à jour) il faut l'écrire sur le disque.

La conséquence de cet algorithme est que le contenu du *cache* est une image fidèle de l'activité récente sur la base de données. Pour donner une illustration concrète de cet effet, supposons qu'une base soit divisée en trois parties distinctes *X*, *Y* et *Z*. L'application *A1* ne lit que dans *X*, l'application *A2* ne lit que dans *Y*, et l'application *A3* ne lit que dans *Z*.

Si, dans la période qui vient de s'écouler, la base a été utilisée à 20% par *A1*, à 30% par *A2* et à 50% par *A3*, alors on trouvera les mêmes proportions pour *X*, *Y* et *Z* dans le *cache*. Si seule *A1* a accédé à la base, alors on ne trouvera que des données de *X* (en supposant que la taille de cette dernière soit suffisante pour remplir le cache).

Quand il reste de la place dans le *cache*, 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 cache en attendant.

2.2.4 Exercices

Exercice : comprendre le *hit ratio*

On considère un fichier de 1 Go et un cache 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 cache peut-on espérer une amélioration significative du *hit ratio* ?

Prenez l'hypothèse que la stratégie de remplacement est de type LRU.

Fichiers séquentiels

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.

3.1 S1 : Enregistrements

Supports complémentaires :

- Diapositives : fichiers et enregistrements
 - Vidéo de présentation (à venir)
-

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 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.

Important : À partir de maintenant le terme vague de “données” que nous avons utilisé jusqu'à présent désigne précisément un *enregistrement*. Dit autrement, les enregistrements constituent notre unité de lecture ou d'écriture : on ne descend jamais à une granularité plus fine.

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 *Types SQL et tailles (en octets)* 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.

Tableau 3.1 – Types SQL et tailles (en octets)

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+1 si M < D
CHAR(M)	M
VARCHAR(M)	L+1, avec L ≤ M
BIT VARYING	< 2 ⁸
DATE	8
TIME	6
DATETIME	14

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.

3.1.1 Champs de tailles fixe et variable

Comme l'indique le tableau *Types SQL et tailles (en octets)*, 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 :

1. on complète les deux derniers caractères avec des blancs ;
2. 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 complétant la longueur fixée. 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 :

1. 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 un codage "3Bou}", le premier octet codant un 3 (au format binaire), les trois octets suivants des caractères 'B', 'o' et 'u', et les 7 octets suivants restant inutilisés ;
2. 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 (vous voyez pourquoi ?). 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 pas 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 autorisé 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, ..., 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, ..., 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.

3.1.2 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

Prenons l'exemple d'une table `tbl{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 *Représentation d'un enregistrement* montre cet enregistrement : notez qu'en lisant l'en-tête, on sait calculer l'adresse de l'enregistrement suivant.

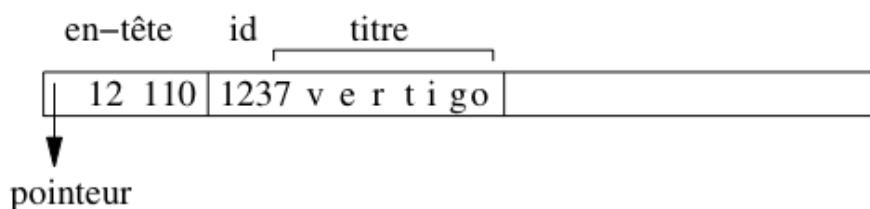


Fig. 3.1 – Représentation d'un enregistrement

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 enregistrement 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.

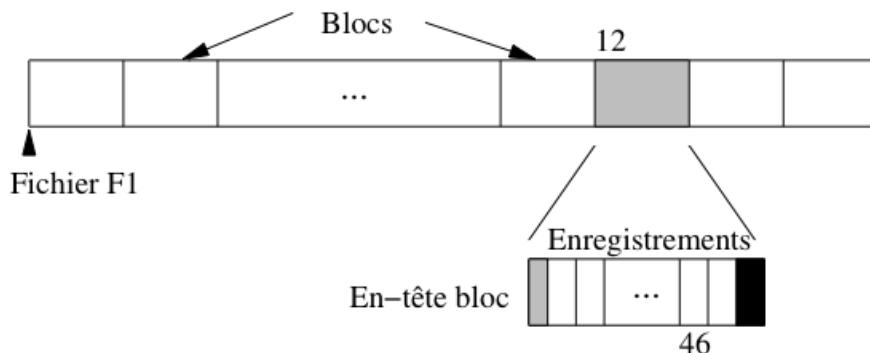


Fig. 3.2 – Stockage des enregistrements dans un bloc

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 : math : $12 \times 47 = 564$. L'enregistrement 563 est donc l'avant-dernier du bloc, avec pour numéro interne le 46 (voir figure :ref:blocenr1).

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 [Adressage avec indirection](#)). 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.

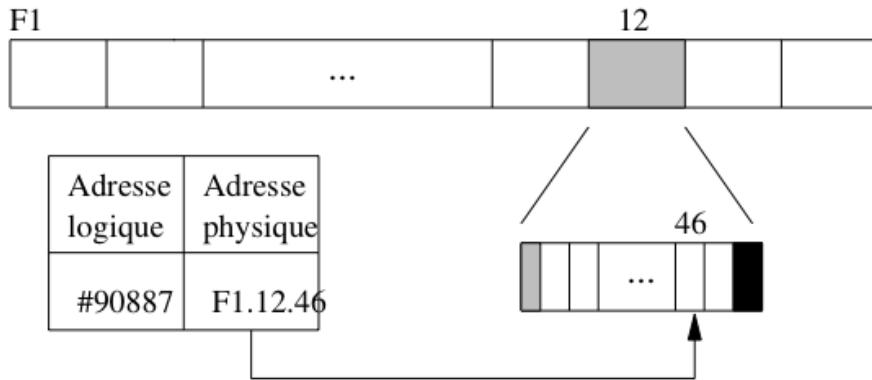


Fig. 3.3 – Adressage avec indirection

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 localisation 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 *Combinaison adresse logique/adresse physique* 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.

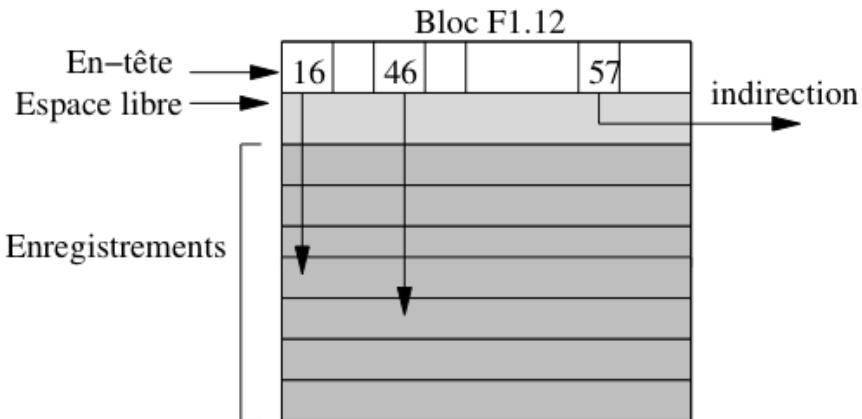


Fig. 3.4 – Combinaison adresse logique/adresse physique

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 souplement l'espace interne à un bloc.

3.1.3 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.

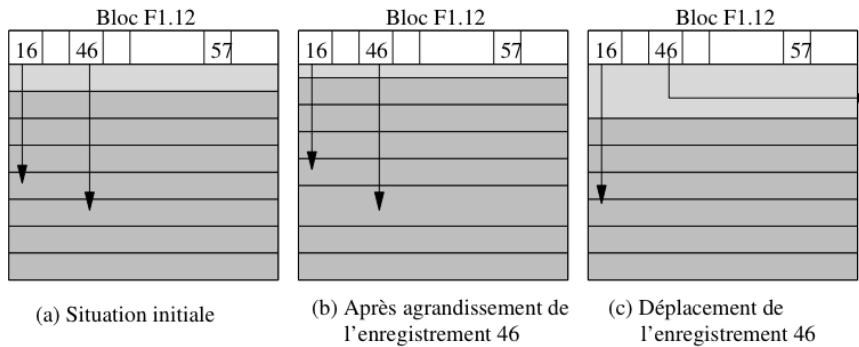


Fig. 3.5 – Mises à jour d'un enregistrement de taille variable

Considérons par exemple le scénario suivant, illustré dans la figure *Mises à jour d'un enregistrement de taille variable* :

- 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 ;
- on insère un autre film, stocké à l'emplacement 47 du bloc F1.12 ;
- 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 ;
- 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. 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.

3.2 S2 : fichiers

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.

3.2.1 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 :

- insertion d'un enregistrement ;
- recherche d'un enregistrement ;
- mise à jour d'un enregistrement ;
- 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.

3.2.2 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 à 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 :

- prendre le bloc au milieu du fichier ;
- si on y trouve *Scream* la recherche est terminée ;
- 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 ;
- 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 $\log_2(n) < k$, 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.

- 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 ;
- 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.

3.2.3 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 *Gestion des blocs libres avec liste chaînée*). 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 *Gestion des blocs libres avec liste chaînée*, en imaginant que le bloc 8 devienne plein, on chainera 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.

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.

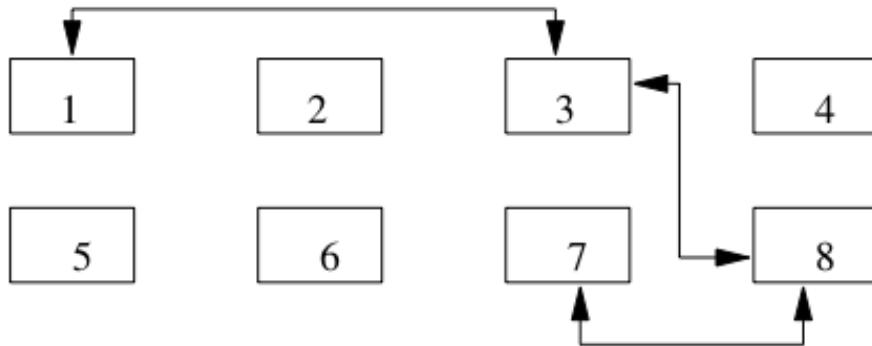


Fig. 3.6 – Gestion des blocs libres avec liste chaînée

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 *Gestion des blocs libres avec répertoire*). 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	...

Arrows point from the 'adresse' column to boxes labeled 1, 2, ..., and 7 respectively. Box 1 is connected to box 3, which is connected to box 7, forming a circular linked list.

Fig. 3.7 – 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.

Exercices

À venir.

Indexation : l'arbre B

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 conditionnant la performance d'un système.

Ce chapitre présente les structures d'index les plus classiques utilisées dans les systèmes relationnels. Après un 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 requêtes : recherche par clé, recherche par intervalle, et recherche avec un préfixe de la clé. Le "B" vient de {it 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.

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

Exemple des films

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 ci-dessous. Nous ne donnons que les deux attributs `titre` et `année` qui seront utilisés pour l'indexation.

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

Exemple d'une grosse collection

Le deuxième exemple est destiné à montrer, avec des ordres de grandeur réalistes (quoique modestes selon les normes actuelles), l'amélioration obtenue par des structures d'index, et les caractéristiques, en espace et en temps, de ces structures. Nous supposerons que la table contient un million (1 000 000) de films, la taille de chaque enregistrement étant de 1200 octets. Pour une taille de bloc de 4 096 octets, on aura donc au mieux 3 enregistrements par bloc. Il faut donc 333 334 blocs ($\lceil 1000000/3 \rceil$) occupant un peu plus de 1,3 Go (1 365 336 064 octets, le surplus étant imputable à l'espace perdu dans chaque bloc). Pour simplifier les calculs, on arrondira à 300 000 blocs. C'est sur ce fichier que nous allons construire nos index.

4.1 S1 : Indexation de fichiers

Supports complémentaires :

- Diapositives : indexation / arbre B
-

4.1.1 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 choisit 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 une ou plusieurs adresse(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 :

- l'index *plaçant* détermine la position d'un enregistrement ;
- l'index *non plaçant* est une structure indépendante du fichier de données, qui se contente de “pointer” vers les enregistrements, ces derniers étant stockés dans un ordre quelconque ;

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 prédefini. Le dictionnaire correspond à l'index plaçant, l'index d'un livre classique 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.

4.1.2 Clé et accès à un index

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, l'index peut servir à 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 300 000 blocs du fichier, pour un coût qui peut être de l'ordre de 5 mns = 300 secondes si le fichier est extrêmement 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 :

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

4.1.3 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 *Dispositifs de stockage*, 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 dépassant le Gigaoctet. La première structure que nous étudions permet d'effectuer des recherches sur un fichier trié, même si ce fichier est fragmenté.

L'index est lui-même un fichier, contenant des enregistrements particuliers de la forme [valeur, Adr] où valeur désigne une valeur de la clé de recherche, et Adr l'adresse d'un bloc.

Définition : entrée d'index

On appelle *entrée d'index* un enregistrement constitué d'une paire $[k, a]$ où k est une valeur de clé et a l'adresse d'un bloc.

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 *Un index non dense* 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*.

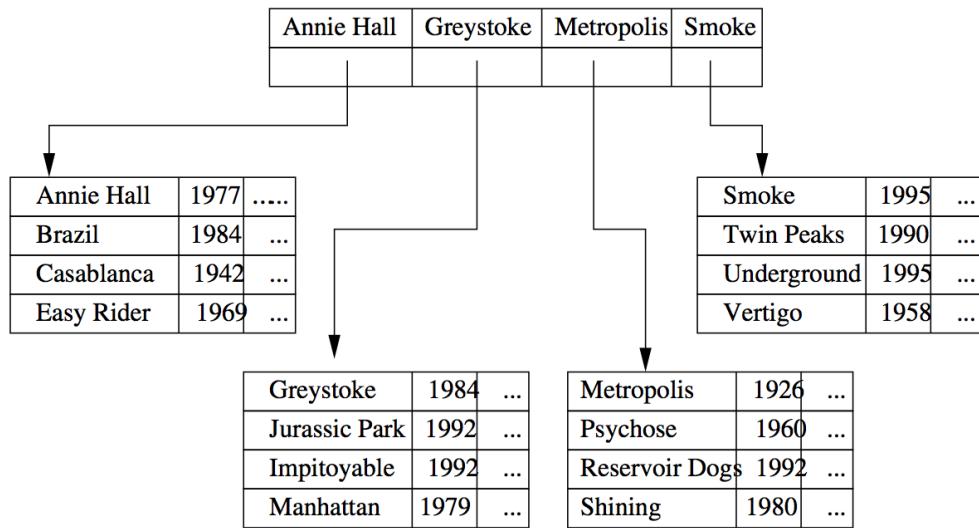


Fig. 4.1 – Un index non dense

Si on désigne par c_1, c_2, \dots, c_n la liste ordonnée des clés dans l'index, il est facile de constater qu'un enregistrement dont la valeur de clé est c est stocké dans le bloc associé à la clé 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 une adresse) y figurent. D'autre part l'index ne comprend qu'un enregistrement par bloc.

Exemple

Considérons l'exemple de notre fichier contenant un million de films. Il est constitué de 300 000 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 $300\,000 * (20 + 8) = 8,4 \text{ Mo}$ octets, à comparer aux 1,3 Go 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 une 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 dont la clé est supérieure à la borne supérieure de l'intervalle.

Exemple

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

- on recherche dans l'index la plus grande valeur strictement inférieure à ‘J’ : pour l'index de la figure *Un index non dense* c'est *Greystoke* ;
 - on accède au bloc du fichier de données, et on y trouve le premier enregistrement avec un titre commençant par ‘J’, soit *Jurassic Park* ;
 - 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 $\frac{r}{b}$ désigne le nombre d'enregistrements du résultat, et b le nombre d'enregistrements dans un bloc. Ce coût est optimal (on n'accède à aucun bloc qui ne participe pas au résultat).

Un index non 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.

4.1.4 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 plus 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 *Un index dense* 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 de fois qu'il y a de films parus cette année là.

Exemple

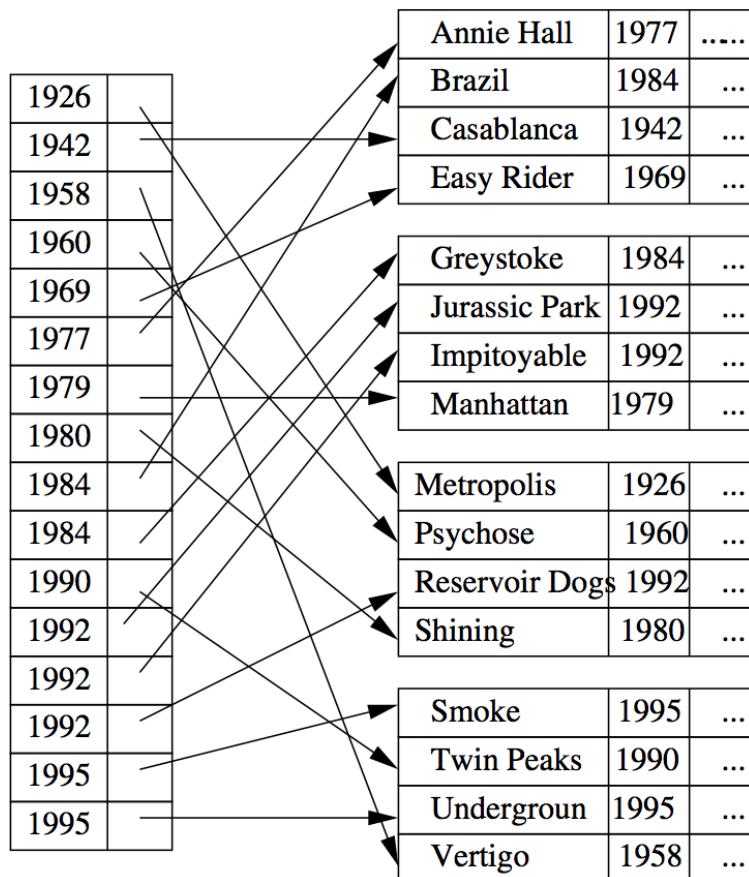


Fig. 4.2 – Un index dense

Considérons l'exemple de notre fichier contenant un million de films. 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 $1\ 000\ 000 * (4+8) = 12\ 000\ 000$ octets, soit (seulement) cent fois moins que le 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 créer un index non-dense, puis ajouter des index denses 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 (index plaçant et non plaçant est plus rigoureux).

Il est possible en fait de créer autant d'index denses que l'on veut puisqu'ils sont indépendants de l'organisation 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 *Un index dense*, on trouve d'abord la première occurrence de 1992, pointant sur *Jurasic 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

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

- on recherche dans l'index la première valeur comprise dans l'intervalle : pour l'index de la figure *Un index dense* c'est 1958 ;
 - 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 ;
 - 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, dans un ordre quelconque. C'est beaucoup moins efficace que de parcourir les 4 blocs du fichier en séquence. 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.

Cet exemple montre qu'une recherche par intervalle entraîne des accès aléatoires au bloc du fichier à cause de l'indirection inhérente à la structure. À partir d'un certain stade (rapidement en fait : le calcul peut se déduire de la petite analyse qui précède) il vaut mieux un bon parcours séquentiel qu'une mauvaise multiplication d'accès directs.

4.1.5 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'entrées [**clé**, **Adr**], trié sur la clé : *ce tri nous permet d'utiliser, dès le deuxième niveau d'indexation, un index non-dense*.

Reprendons l'exemple de l'indexation des films sur l'année de parution. Nous avons vu que la taille du fichier était 100 fois moindre que celle du fichier de données. Même s'il est possible d'effectuer une recherche par dichotomie, cette taille peut devenir pénalisante pour les opérations de recherche.

On peut alors créer un deuxième niveau d'index, comme illustré sur la figure *Index multi-niveaux*. 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.

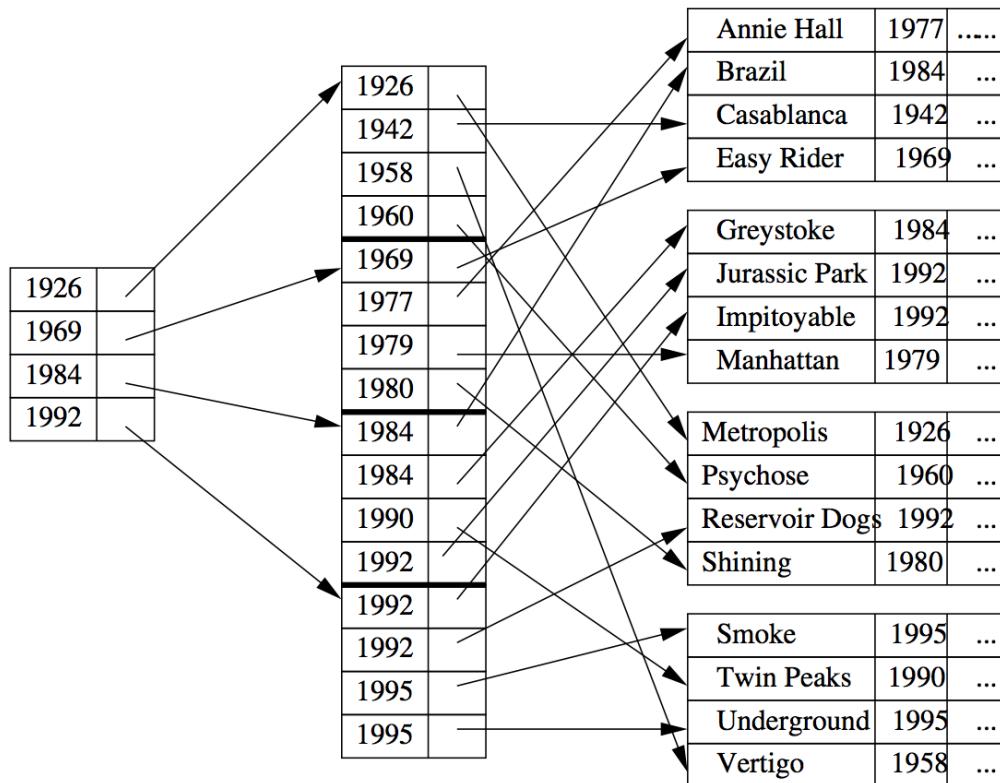


Fig. 4.3 – Index multi-niveaux

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

On recherche le ou les films parus en 1990. Partant du second niveau d'index, on sélectionne la troisième entrée correspondant à la clé 1990. 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 séquentiels en recherche, il ajoute des algorithmes de réorganisation dynamique qui résolvent la question de la maintenance d'une structure triée.

4.2 S2 : 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 ;
- chaque noeud (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 moyen de 70%) 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'entrées dans un bloc. Le nombre maximal d'entrée d'une arbre d'ordre n est donc $2n$.

À partir de maintenant, 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. C'est la situation courante. Nous allons indexer ce fichier avec un arbre B.

Note : Il existe de nombreuses variantes d'arbres B. La présentation qui suit décrit une variante non-placante et parfois désignée comme "l'arbre-B+".

4.2.1 Présentation intuitive

La figure *Le fichier des films, avec un index unique sur le titre* 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 supposerons 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.

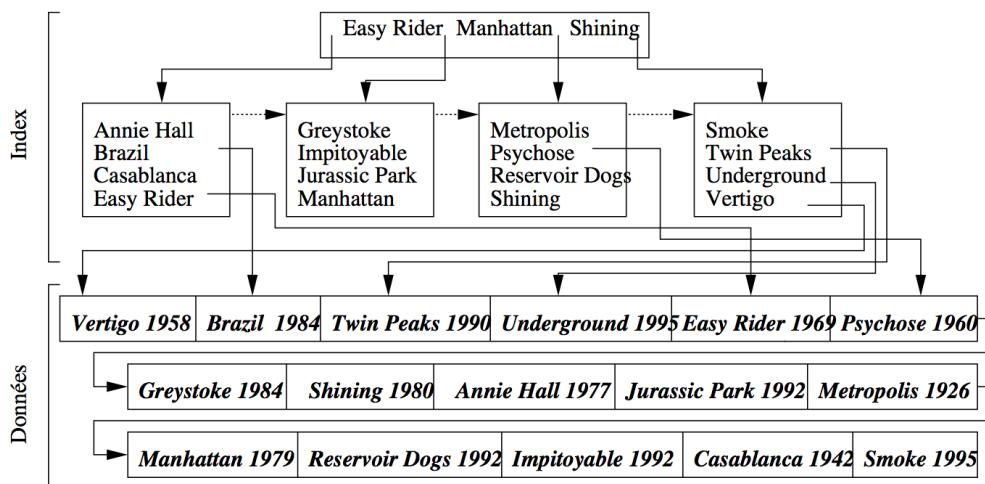


Fig. 4.4 – Le fichier des films, avec un index unique sur le titre

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.

- 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* ;
- 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* ;
- 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. Dans 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 [Le fichier des films, avec un index unique sur le titre](#), 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 [Le fichier des films, avec un index unique sur le titre](#) – permet de répondre aux recherches par intervalle.

En supposant que l'attribut `titre` est la clé unique de *Film*, il n'y a 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 associées à une même valeur.

La figure [Le fichier des films, avec un index unique](#) 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.

Un arbre-B est créé sur une table, soit implicitement par la commande `create index`, soit explicitement avec l'option `primary key`. Voici les commandes classiques : la création de la table, avec création de l'index pour assurer l'unicité de la clé primaire.

```
create table Film (titre varchar(30) not null,  
                  ...,  
                  primary key (titre)  
);
```

Et la création d'un second index, non unique, sur l'année.

```
create index filmAnnee on Film (annee)
```

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.

4.2.2 Recherches avec un arbre-B

L'arbre-B+ supporte des opérations de recherche par clé, par préfixe de la clé et par intervalle. Prenons l'exemple suivant :

```
select *  
from Film  
where titre = 'Impitoyable'
```

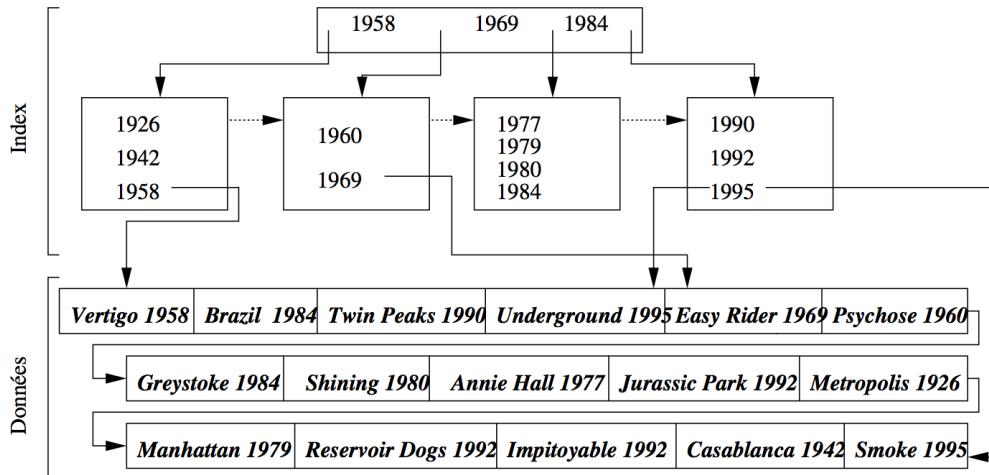


Fig. 4.5 – Le fichier des films, avec un index unique.

En l'absence d'index, la seule solution est de parcourir le fichier. Dans l'exemple de la figure *Le fichier des films, avec un index unique sur le titre*, 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

Reprendons l'exemple 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 300 000 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'arbre (pour être plus précis le calcul qui suit devrait tenir compte du fait qu'un bloc n'est pas toujours plein).

- avec un niveau d'index (la racine seulement) on peut donc référencer 146 films ;
- avec deux niveaux on indexe 146 blocs de 146 films chacun, soit $146^2 = 21\,316$ films ;
- avec trois niveaux on indexe $146^3 = 3\,112\,136$ films ;
- enfin avec quatre niveaux on indexe 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.

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 *Le fichier des films, avec un index unique.*) 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 *Le fichier des films, avec un index unique..* 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.

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 *Le fichier des films, avec un index unique sur le titre*) 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".

Opérateurs et algorithmes

Une des tâches essentielles d'un SGBD est d'exécuter les requêtes SQL soumises par une application afin de fournir le résultat avec le meilleur temps d'exécution possible. La combinaison d'un langage de haut niveau, et donc en principe facile d'utilisation, et d'un moteur d'exécution puissant apte à traiter efficacement des requêtes extrêmement complexes, est l'un des principaux atouts des SGBD (relationnels).

Ce chapitre présente les composants de base d'un moteur d'exécution : les opérateurs, et les principaux algorithmes de jointure. Ce sont les briques à partir desquels un système construit dynamiquement le programme d'exécution d'une requête, également appelé *plan d'exécution*. Dans l'ensemble du chapitre, nous donnons une spécification détaillée d'un catalogue d'opérateurs qui permettent d'évaluer toutes les requêtes SQL conjonctives (c'est-à-dire sans négation).

La manière dont le plan d'exécution est construit à la volée quand une requête est soumise fait l'objet du chapitre suivant.

5.1 S1 : Modèle d'exécution : les itérateurs

Supports complémentaires :

- Diapositives : les itérateurs
 - Vidéo de présentation (à venir)
-

L'exécution d'une requête s'effectue par combinaison d'opérateurs qui assurent chacun une tâche spécialisée. De même qu'une requête quelconque peut être représentée par une *expression* de l'algèbre relationnelle, construite à partir de cinq opérations de base, un *plan d'exécution* consiste à combiner les opérateurs appropriés, tirés d'une petite bibliothèque de composants logiciels qui constitue la "boîte à outils" du moteur d'exécution.

Ces composants ont une forme générique qui se retrouve dans tous les systèmes. D'une manière générale, ils se présentent comme des "boîtes noires" qui consomment des flux de données en entrées et produisent un autre flux de données en sortie. De plus, ces boîtes peuvent s'interconnecter, l'entrée de l'une étant la sortie de l'autre. Enfin, l'ensemble est conçu pour minimiser les ressources matérielles nécessaires, et en particulier la mémoire RAM. Nous commençons par étudier en détail ce dernier aspect.

5.1.1 Matérialisation et pipelining

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 cache s'il y a de la place, ou sur disque sinon, et d'utiliser le cache ou le fichier intermédiaire comme source de données pour o' .

Pour notre exemple, le parcours d'index (l'opération o) rechercherait toutes les adresses des enregistrements satisfaisant le critère de recherche, et les placerait dans une structure temporaire, puis l'opération o' irait lire ces adresses

pour accéder au fichier de données et fournir finalement les tuples à l’application (figure *Exécution d’une requête avec matérialisation*).

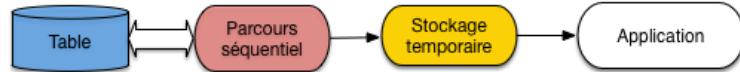


Fig. 5.1 – Exécution d’une requête avec matérialisation

Sur un serveur recevant beaucoup de requêtes, la mémoire centrale deviendra rapidement indisponible ou trop petite pour accueillir les résultats intermédiaires qui devront donc être écrits sur disque. Cette solution de *matérialisation* des résultats intermédiaires est alors 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.

Un autre inconvénient sévère est qu’il faut attendre qu’une première opération soit exécutée dans son intégralité avant d’effectuer la seconde.

L’alternative appelée *pipelinage* 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, la sortie du premier tenant lieu d’entrée au second. Dans ce scénario, o tient le rôle du producteur, o' celui du consommateur. Chaque fois que o produit une adresse d’enregistrement, o' la reçoit et va lire l’enregistrement dans le fichier (figure *Exécution d’une requête avec pipelinage*).

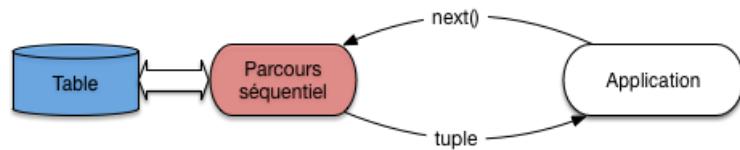


Fig. 5.2 – Exécution d’une requête avec pipelinage

On n’attend donc pas que o soit terminée, et que l’ensemble des enregistrements résultat de o ait été produit pour lancer o' . 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 :

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

Si l’application qui reçoit et traite le résultat de la requête passe un peu de temps sur chaque ligne produite, l’exécution de la requête peut même ne plus constituer un facteur pénalisant. Si, par exemple, 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 la première 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.

L’application qui exécute une requête est elle-même le consommateur ultime et décide du moment où les données doivent lui être communiquées. Elle agit en fait comme une sorte d’aspirateur branché sur des tuyaux de données qui vont prendre leurs racines dans les structures de stockage de la base de données : tables et index.

Cela se traduit, quand on accède à un SGBD avec un langage de programmation, par une structure d'accès toujours identique, consistant à :

- ouvrir un curseur par exécution d'une requête ;
- avancer le curseur sur le résultat, ligne à ligne, par des appels à une fonction de type `next()` ;
- fermer (optionnel) le curseur quand le résultat a été intégralement parcouru.

Le mécanisme est suggéré sur la figure [Exécution d'une requête avec pipelining](#). L'application demande un tuple par appel à la fonction `next()` adressé à l'opérateur d'accès direct. Ce dernier, pour s'exécuter, a besoin d'une adresse d'enregistrement : il l'obtient en adressant lui-même la fonction `next()` à l'opérateur de parcours d'index. Cet exemple simple est en fait une illustration du principe fondamental de la constitution d'un plan d'exécution : ce dernier est toujours un graphe d'opérateurs produisant *à la demande* des résultats intermédiaires non matérialisés.

5.1.2 Opérateurs bloquants

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'un opérateur dit bloquant dont le résultat doit être entièrement produit (et matérialisé en cache ou écrit sur disque) avant de démarrer l'opération suivante. Par exemple :

- le tri (`order by`) ;
- la recherche d'un maximum ou d'un minimum (`max, min`) ;
- l'élimination des doublons (`distinct`) ;
- le calcul d'une moyenne ou d'une somme (`sum, avg`) ;
- 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).

Une expérience à tenter

Si vous êtes sur une base avec une table T de taille importante, tentez l'expérience suivante. Exécutez une première requête :

```
select * from T
```

et une seconde

```
select distinct * from T
```

Vous devriez constater une attente significative avant que le résultat de la seconde commence à s'afficher. Que se passe-t-il ? Relisez ce qui précède.

Un opérateur bloquant introduit une *latence* dans l'exécution de la requête. En pratique, l'application est figée tant que la phase de préparation, exécutée à l'appel de la fonction `open()`, n'est pas terminée. À l'issue de la phase de latence, l'exécution peut reprendre et se dérouler alors très rapidement. L'inconvénient pour une application d'une exécution avec opérateur bloquant est qu'elle reste à ne rien faire pendant que le SGBD travaille, puis c'est l'inverse : le débit de données est important, mais c'est à l'application d'effectuer son travail. Le temps passé aux deux phases d'additionne, ce qui est un facteur pénalisant pour la performance globale.

Cela amène à distinguer deux critères de performance :

- *le temps de réponse* : c'est le temps mis pour obtenir un premier tuple ; il est quasi-instantané pour une exécution sans opérateur bloquant ;
- *le temps d'exécution* : c'est le temps mis pour obtenir l'ensemble du résultat.

On peut avoir une exécution avec un temps de réponse important et un temps d'exécution court, et l'inverse. En général, on choisira de privilégier le temps de réponse, pour pouvoir traiter les données dès que possible.

5.1.3 Itérateurs

Le mécanisme de pipelinage “à la demande” est implanté au moyen de la notion générique *d’itérateur*, couramment rencontré par ailleurs dans les langages de programmation offrant des interfaces de traitement de collections. Chaque itérateur peut s’implanter comme un objet doté de trois fonctions :

- `open()` commence le processus pour obtenir des articles résultats : elle alloue les ressources nécessaires, initialise des structures de données et appelle `open()` pour chacune de ses *sources* (c’est-à-dire pour chacun des itérateurs fournissant des données en entrée) ;
- `next()` effectue une étape de l’itération, retourne l’article produit par cette étape, et met à jour les structures de données nécessaires pour obtenir les résultats suivants ; la fonction peut appeler une ou plusieurs fois `next()` sur ses sources.
- `close()` termine l’itération et libère les ressources, lorsque tous les articles du résultat ont été obtenus. Elle appelle typiquement `close()` sur chacune de ses sources.

Voici une première illustration d’un itérateur, celui du balayage séquentiel d’une table que nous appellerons `FullScan`. La fonction `open()` de cet itérateur place un curseur au début du fichier à lire.

```
function openScan
{
    # Entrée: $T est la table

    # Initialisations
    $p = $T.first;      # Premier bloc de T
    $e = $p.init;        # On se place avant le premier enregistrement
}
```

La fonction `next` renvoie l’enregistrement suivant, ou `null`.

```
function nextScan
{
    # Avançons sur l'enregistrement suivant
    $e = $p.next;
    # A-t-on atteint le dernier enregistrement du bloc ?
    if ($e = null) do
        # On passe au bloc suivant
        $p = $T.next;
        # Dernier bloc dépassé?
        if ($p = null) then
            return null;
        else
            $e = $p.first;
        fi
    done

    return $e;
}
```

La fonction `close` libère les ressources.

```
function closeScan
{
    close($T);
    return;
}
```

`openScan` initialise la lecture en se placant *avant* le premier enregistrement du premier bloc de `T`. Chaque appel à `nextScan()` retourne un enregistrement. Si le bloc courant a été entièrement lu, il lit le bloc suivant et retourne le premier enregistrement de ce bloc.

Avec ce premier itérateur, nous savons déjà exécuter au moins une requête SQL ! C'est la plus simple de toutes.

```
select * from T
```

Doté de notre itérateur FullScan, cette requête s'exécute de la manière suivante :

```
# Parcours séquentiel de la table T
$curseur = new FullScan(T);
$tuple = $curseur.next();

while [$tuple != null]
do
    # Traitement du tuple
    ...
    # Récupération du tuple suivant
    $tuple = $curseur.next();
done

# Fermeture du curseur
$curseur.close();
```

Ceux qui ont déjà pratiqué l'accès à une base de données par une interface de programmation reconnaîtront sans peine la séquence classique d'ouverture d'un curseur, de parcours du résultat et de fermeture du curseur. C'est facile à comprendre pour une requête aussi simple que celle illustrée ci-dessus. La beauté du modèle d'exécution est que la même séquence s'applique pour des requêtes extrêmement complexes.

5.2 S2 : les opérateurs de base

Supports complémentaires :

- Diapositives : les opérateurs de base
 - Vidéo de présentation (à venir)
-

Cette section est consacrée aux principaux opérateurs utilisés dans l'évaluation d'une requête "simple", accédant à une seule table. En d'autres termes, il suffisent à évaluer toute requête de la forme :

```
select a1, a2, ..., an from T where condition
```

En premier lieu, il faut accéder à la table T . Il existe exactement deux méthodes possibles :

- Accès séquentiel. Tous les tuples de la table sont examinés, dans l'ordre du stockage.
- Accès par adresse. Si on connaît l'adresse du tuple, on peut aller lire directement le bloc et obtenir ainsi un accès optimal.

Chaque méthode correspond à un opérateur. Le second (l'opérateur d'accès direct) est toujours associé à une source qui lui fournit l'adresse des enregistrements. Dans un SGBD relationnel où le modèle de données connaît pas la notion d'adresse physique, cet source est *toujours* un opérateur de parcours d'index. Nous allons donc également décrire ce dernier sous forme d'itérateur. Enfin, il est nécessaire d'effectuer une sélection (pour appliquer la condition de la clause `where`) et une projection. Ces deux derniers opérateurs sont triviaux.

5.2.1 Parcours séquentiel

Le parcours 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 trouver un ou plusieurs enregistrements d'une table satisfaisant un critère de sélection.

L'opérateur est implanté par un itérateur qui a déjà été discuté dans la section précédente. Son coût est relativement élevé : il faut accéder à tous les blocs de la table, et le temps de parcours est donc proportionnel à la taille de cette dernière.

Cette mesure “brute” doit cependant être pondérée par le fait qu'une table est le plus souvent stockée de manière contiguë sur le disque, et se trouve de plus partiellement en mémoire RAM si elle a été utilisée récemment. Comme nous l'avons vu dans le chapitre *Dispositifs de stockage*, le parcours d'un segment contigu évite les déplacements des têtes de lecture et la performance est alors essentiellement limité par le débit du disque.

5.2.2 Parcours d'index

On considère dans ce qui suit que la structure de l'index est celle de l'arbre B, ce qui est presque toujours le cas en pratique.

L'algorithme de parcours d'index a été étudié dans le chapitre *Indexation : l'arbre B*. Rappelons brièvement qu'il se décompose en deux phases. La première est une *traversée* de l'index en partant de la racine jusqu'à la feuille contenant l'entrée associée à la clé de recherche. La seconde est un parcours séquentiel des feuilles pour trouver toutes les adresses correspondant à la clé (il peut y en avoir plusieurs dans le cas général) ou à l'intervalle de clés.

Ces deux phases s'implantent respectivement par les fonctions *open()* et *next()* de l'itérateur `IndexScan`. Voici tout d'abord la fonction *open()*.

```
function openIndexScan
{
    # $c est la valeur de la clé recherchée; $I est l'index

    # On parcourt les niveaux de l'index en partant de la racine
    $bloc = $I.racine();
    while [$bloc.estUneFeuille() = false]
        do
            # On recherche l'entrée correspondant à $c
            for $e in ($bloc.entrées)
                do
                    if ($e.clé > $c)
                        break;
                    done

                    $bloc = $GA.lecture ($e.adresse);
                done

            # $bloc est la feuille recherchée; on se positionne sur la
            # première occurrence de $a
            $e = $bloc.premièreOccurrence ($c)
            # Fin de la première phase
}
```

La fonction *next()* est identique à celle du parcours séquentiel d'un fichier, la seule différence est qu'elle renvoie des adresses d'enregistrement et pas des tuples. La version ci-dessous, simplifiée, ne montre pas le passage d'une feuille à une autre.

```
function nextIndexScan
{
    # Seconde phase: on est positionné sur une feuille de l'arbre, on
    # avance sur les entrées correspondant à la clé $c
    if ($e.clé = $c) then
        $adresse = $e.adresse;
        $e = $e.next();
```

```

    return $adresse;
else
    return null;
fi
}
}
```

5.2.3 Accès par adresse

Quand on connaît l'adresse d'un enregistrement, donc l'adresse du bloc où il est stocké, y accéder coûte une lecture unique de bloc. Cette adresse (rappelons que l'adresse d'un enregistrement se décompose en une adresse de bloc et une adresse d'enregistrement locale au bloc) doit nécessairement être fournie par un autre itérateur, que nous appellerons la *source* dans ce qui suit (et qui, en pratique, est toujours un parcours d'index).

L'itérateur d'accès direct (*DirectAccess*) s'implante alors très facilement. Voici la fonction *next()* .

```

function nextDirectAccess
{
    # $source est l'opérateur source; $GA est le gestionnaire d'accès

    # Récupérons l'adresse de l'enregistrement à lire
    $a = $source.next();

    # Plus d'adresse? On renvoie null
    if ($a = null) then
        return null;
    else
        # On effectue par une lecture (logique) du bloc
        $b = $GA.lecture ($a.adressBloc);
        # On récupère l'enregistrement dans le bloc
        $e = $b.get ($a.adresseLocale)
        return $e;
    fi
}
```

Cet opérateur est très efficace pour récupérer *un* enregistrement par son adresse, notamment dans le cas fréquent où le bloc est déjà dans le cache. Quand on l'exécute de manière intensive sur une table, il engendre de nombreux accès aléatoires et on peut se poser la question de préférer un parcours séquentiel. La décision relève du processus d'optimisation : nous y revenons plus loin.

5.2.4 Opérateurs de sélection et de projection

L'opérateur de sélection (le plus souvent appelé *filtre*) applique une condition aux enregistrements obtenus d'un autre itérateur. Voici la fonction *next()* de l'itérateur.

```

function nextFilter
{
    # $source est l'opérateur fournissant les tuples; $C est la condition de sélection

    # On récupère un tuple de la source
    $tuple = $source.next();
    # Et on continue tant que la sélection n'est pas satisfaite, ou la source épuisée
    while ($tuple != null and $tuple.test($C) = false)
        do
            $tuple = $source.next();
        done
}
```

```
    return $tuple;
}
```

En pratique, la source est *toujours* soit un itérateur de parcours séquentiel, soit un itérateur d'accès direct. Le filtre a pour effet de réduire la taille des données à traiter, et il est donc bénéfique de l'appliquer le plus tôt possible, immédiatement après l'accès à chaque enregistrement.

Note : Cette règle est une des heuristiques les plus courantes de la phase dite d'optimisation que nous présenterons plus tard. Elle est souvent décrite par l'expression “pousser les sélections vers le base de l'arbre d'exécution”.

Dans certains systèmes (p.e., Oracle), cet opérateur est d'ailleurs intégré aux opérateurs d'accès à une table (séquentiel ou direct) et n'apparaît donc pas dans le plan d'exécution.

L'opérateur de projection, consistant à ne conserver que certains attributs des tuples en entrée, est trivial. La fonction `next()` prend un tuple en entrée, en extrait les attributs à conserver et renvoie un tuple formé de ces derniers.

5.2.5 Exécution de requêtes mono-tables

Reprendons notre requête mono-table générique, de la forme :

```
select a1, a2, ..., an from T where condition
```

Notre petit catalogue d'opérateurs nous permet de l'exécuter. Il nous donne même plusieurs options selon que l'on utilise ou pas un index. Voici un exemple concret que nous allons examiner en détails.

```
select titre from Film where genre='SF' and annee = 2014
```

Et nous allons supposer que la table des films est indexée sur l'année. Avec notre boîte à outils, il existe (au moins) deux programmes, ou *plans d'exécution*, illustrés par la figure *Deux plans d'exécution pour une requête monutable*.

Le premier plan utilise l'index pour obtenir les adresses des films parus en 2014 grâce à un opérateur `IndexScan`. Il est associé à un opérateur `DirectAccess` qui récupère, une par une, les adresses et obtient, par une lecture directe, le tuple correspondant. Enfin, chaque tuple passe par l'opérateur de filtre qui ne conserve que ceux dont le genre est 'SF'.

On voit sur ce premier exemple qu'un plan d'exécution est un programme d'une forme très particulière. Il est constitué d'un graphe d'opérateurs connectés par des ‘tuyaux’ où circulent des tuples. L'application, qui communique avec la racine du graphe par l'interface de programmation, joue le rôle d'un “aspirateur” qui tire vers le haut ce flux de données fournissant le résultat de la requête.

Note : Pour une requête monutable, le graphe est linéaire. Quand la requête comprend des jointures, il prend la forme d'un arbre (binaire).

Examinons maintenant le second plan. Il consiste simplement à parcourir séquentiellement la table (avec un opérateur `FullScan`), suivie d'un filtre qui applique les deux critères de sélection.

Quel est le meilleur de ces deux plans ? En principe, l'utilisation de l'index donne de bien meilleurs résultats. Et en général, 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. Dans ce cas le plan d'exécution est caractérisé par l'association de l'opérateur `IndexScan` qui récupère des adresses, et de l'opérateur `DirectAccess` qui récupère les tuples en fonction des adresses.

En fait, ce choix est légèrement compliqué par les considérations suivantes :

- Quelle est la taille de la table ? Si elle tient en quelques blocs, un accès par l'index est probablement inutilement compliqué.

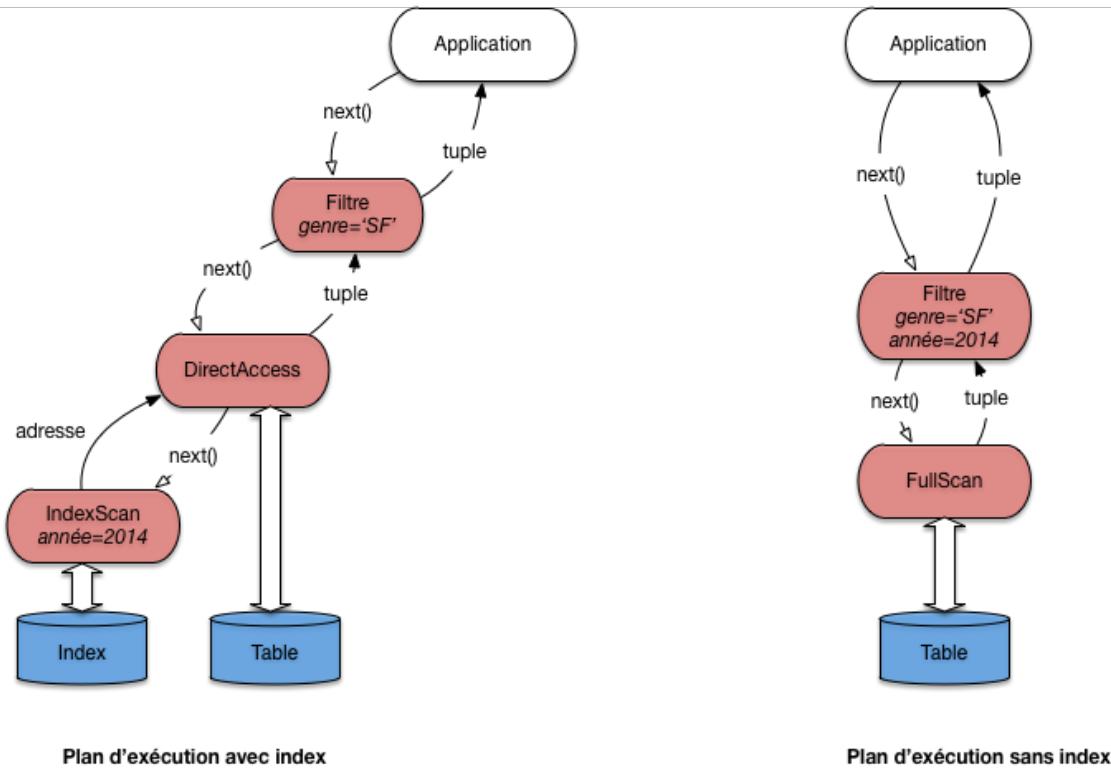


Fig. 5.3 – Deux plans d'exécution pour une requête monotable

- 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** ?
- 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 est celui d'une recherche par intervalle sur un champ indexé.

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

```
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 *Recherche par intervalle avec index*. Dans cet exemple, le fichier a quatre blocs, 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 à 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 *Indexation : l'arbre B*). À chaque valeur de clé dans l'index est associé un pointeur (une adresse) qui désigne l'enregistrement dans le fichier.

Maintenant supposons :

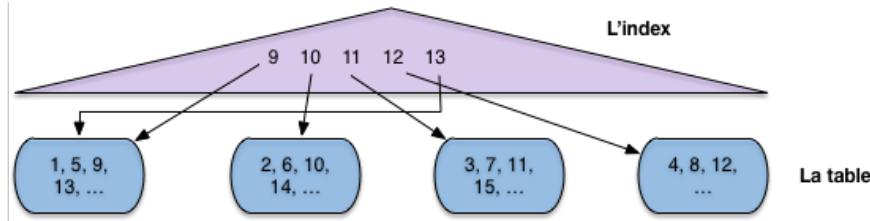


Fig. 5.4 – Recherche par intervalle avec index

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

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.

- **Étape 1** : on récupère dans l'index toutes les valeurs de clé comprises entre 9 et 13 (opérateur `IndexScan`).
- **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 (opérateur `DirectAccess`).

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

Cette petite démonstration est basée sur une situation caricaturale et 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 aléatoires et peuvent conduire à lire plusieurs fois le même bloc. Même sans cela, des recherches par adresses intenses mènent à déclencher des opérations d'accès à un bloc pour lire un unique enregistrement à chaque fois, ce qui s'avère pénalisant à terme par rapport à un simple parcours.

La leçon, c'est que le SGBD ne peut pas aveuglément appliquer une stratégie pré-déterminée pour exécuter une requête. Il doit examiner, parmi les solutions possibles (au moins celles qui semblent plausibles) celles qui vont donner le meilleur résultat. C'est un module particulier (et très sensible), *l'optimiseur*, qui se charge de cette estimation.

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*. C'est un plan similaire à celui de la figure *Deux plans d'exécution pour une requête monotable* (gauche) qui sera utilisé.

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* s'il n'y a pas d'index sur le titre. Autant donc s'épargner la recherche par index et trouver les deux films au cours du balayage. Le plan sera donc plutôt celui de la figure *Deux plans d'exécution pour une requête monotable*, à droite.

5.3 S3 : Le tri externe

Supports complémentaires :

- Diapositives : le tri-fusion
- Vidéo de présentation (à venir)

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`, `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.

Le tri d'une relation sur un ou plusieurs attributs utilise l'algorithme de tri-fusion (*sort-merge*) en mémoire externe. Il est du type “diviser pour régner”, avec deux phases. Dans la première phase on décompose le problème récursivement en sous-problèmes, et ce jusqu'à ce que chaque sous-problème puisse être résolu simplement et efficacement. La deuxième phase consiste à agréger récursivement les solutions.

Dans le cas l'algorithme de tri-fusion, les deux phases se résument ainsi :

- Partitionnement de la table en *fragments* telles que chaque fragment tienne en mémoire centrale. On trie alors chaque fragment (en mémoire), en général avec l'algorithme de *Quicksort*.
- Fusion des fragments triés.

Regardons en détail chacune des phases.

5.3.1 Phase de tri

Supposons que nous disposons pour faire le tri de M blocs en mémoire. La phase de tri commence par prendre un fragment constitué des M premiers blocs du fichier et les charge en mémoire. On trie ces blocs avec *Quicksort* et on écrit le fragment trié sur le disque dans une zone temporaire (figure [Algorithme de tri-fusion : phase de tri](#)). On recommence avec les M blocs suivants du fichier, jusqu'à ce que tout le fichier ait été lu par fragments de M blocs,

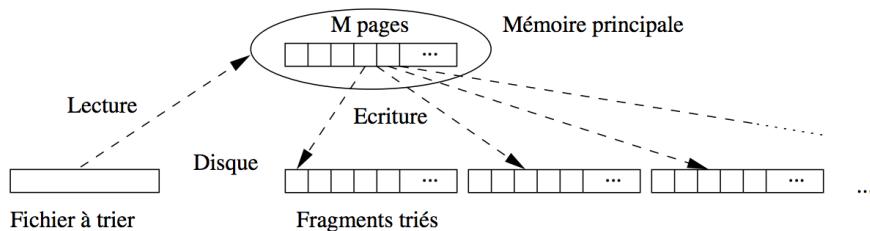


Fig. 5.5 – Algorithme de tri-fusion : phase de tri

À l'issue de cette phase on a $\lceil B/M \rceil$ fragments triés, où B est le nombre de blocs du fichier.

Exemple

Si le fichier occupe 100 000 blocs, et la mémoire disponible pour le tri est de 1 000 blocs, cette phase découpe le fichier en 100 fragments de 1 000 blocs chacun. Ces 100 fragments sont lus un par un, triés, et écrits dans la zone temporaire.

5.3.2 Phase de fusion

La phase de fusion consiste à fusionner les fragments. Si on fusionne n fragments de taille M , on obtient en effet un nouveau fragment trié de taille $n \times M$. En général, une étape de fusion suffit pour obtenir l'ensemble du fichier trié, mais si ce dernier est très gros - ou si la mémoire disponible est insuffisante - il est parfois nécessaire d'effectuer plusieurs étapes de fusion.

Commençons par regarder comment on fusionne en mémoire centrale deux listes *triées* A et B. On a besoin de trois zones en *cache*. Dans les deux premières, les deux listes à trier sont stockées. La troisième zone de cache sert pour le résultat, c'est-à-dire la liste résultante triée.

L'algorithme employé (dit *fusion* ou “*merge*”) est une technique très efficace qui consiste à parcourir en parallèle et séquentiellement les listes, en une seule fois. Le parcours unique est permis par le tri des listes sur un même critère.

La figure *Parcours linéaire pour la fusion de listes triées* montre comment on fusionne A et B. On maintient deux curseurs, positionnés au départ au début de chaque liste. L'algorithme compare les valeurs contenues dans les cellules pointées par les deux curseurs. On compare ces deux valeurs, puis :

- (choix 1) si elles sont égales, on déplace les deux valeurs dans la zone de résultat ; *on avance les deux curseurs d'un cran*
- (choix 2) sinon, on prend le curseur pointant sur la cellule dont la valeur est la plus petite, on déplace cette dernière dans la zone de résultat et on avance ce même curseur d'un cran.

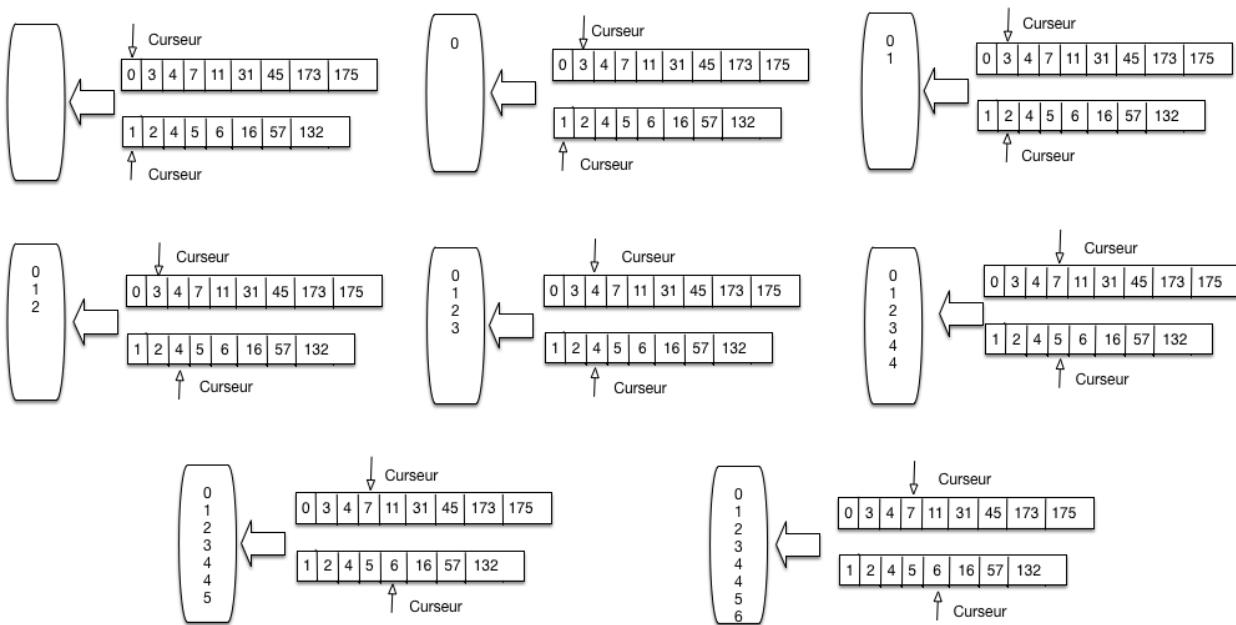


Fig. 5.6 – Parcours linéaire pour la fusion de listes triées

La figure *Parcours linéaire pour la fusion de listes triées* se lit de gauche à droite, de haut en bas. On applique tout d'abord deux fois le choix 2, avançant sur la liste A puis la liste B. On avance encore sur la liste B puis la liste A. On se trouve alors face à la situation 2, et on copie les deux valeurs 4 dans la zone de résultat.

Et ainsi de suite. Il est clair qu'*un seul parcours suffit*. Il devrait être clair également, par construction que la zone de résultat contient une liste *triée* contenant toutes les valeurs venant soit de A soit de B. .

```
// Fusion de deux listes 11 et 12
function fusion
{
    # $11 désigne la première liste
    # $12 désigne la seconde liste
    $resultat = [];
    # Début de la fusion des listes
    while ($11 != null and $12 != null) do
        if ($11.val == $12.val) then
            # On a trouvé un doublon
            $resultat += $11.val;
```

```

        $resultat += $12.val;
        # Avançons sur les deux listes
        $11 = $11.next; $12 = $12.next;
    else if ($11.val < $12.val) then
        # Avançons sur 11
        $resultat += $11.val;
        $11 = $11.next;
    else
        # Avançons sur 12
        $resultat += $12.val;
        $12 = $12.next;
    fi
done
}
    
```

Remarquons que :

- L'algorithme *fusion* se généralise facilement à plusieurs listes.
- Si on fusionne n listes de taille M , la liste résultante a une taille de $n \times M$ blocs.

La première étape de la phase de fusion de la relation consiste à fusionner les $\lceil B/M \rceil$ obtenus à l'issue de la phase de tri. On prend pour cela $M - 1$ fragments à la fois, et on leur associe à chacun un bloc en mémoire, le bloc restant étant consacré au résultat. 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 sur les listes triées, comme expliqué ci-dessus. Les enregistrements triés sont stockés dans un nouveau fragment sur disque.

On continue avec les $M - 1$ blocs suivants de chaque fragment, jusqu'à ce que les $M - 1$ fragments initiaux aient été entièrement lues et triés. On a alors sur disque une nouvelle partition de taille $M \times (M - 1)$. On répète le processus avec les $M - 1$ fragments suivants, et ainsi de suite.

À la fin de cette première étape, on obtient $\lceil \frac{B}{M \times (M-1)} \rceil$ fragments triés, chacune (sauf le dernier qui est plus petit) ayant pour taille $M \times (M - 1)$ blocs. La figure *Algorithme de tri-fusion : la phase de fusion* résume la phase de fusion sous la forme d'un arbre, chaque nœud (agrandi à droite) correspondant à une fusion de $M - 1$ partitions.

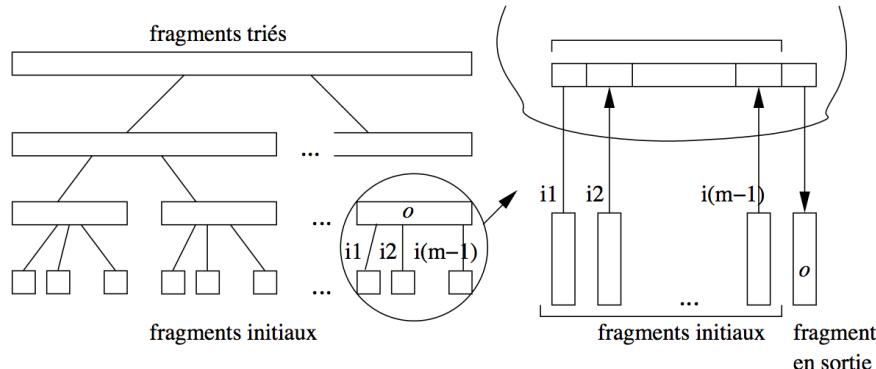


Fig. 5.7 – Algorithme de tri-fusion : la phase de fusion

Un exemple est donné dans la figure *Algorithme de tri-fusion : la phase de fusion* 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.

5.3.3 Coût du tri-fusion

La phase de tri coûte B écritures pour créer les partitions triées. À chaque étape de la phase de 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,

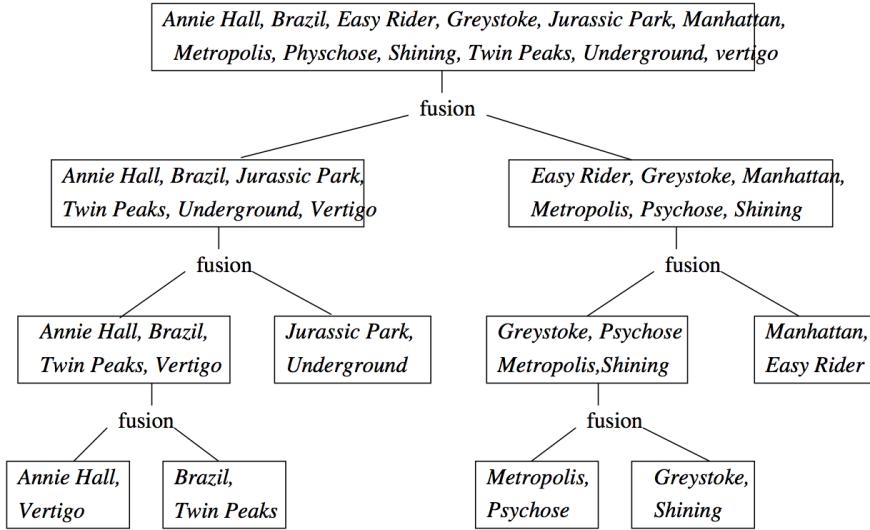


Fig. 5.8 – Algorithme de tri-fusion : un exemple

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 $O(B \times \log_{M-1} B)$. Il prédomine celui de la phase de tri.

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 bloc 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 étape de fusion en plus ou en moins, avec des différences de performance notables, comme le montre l'exemple suivant.

Exemple

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

- Avec une mémoire $M > 307\text{Mo}$, tout le fichier peut être chargé et trié en mémoire. Une seule lecture suffit.
- Avec une mémoire $M = 2\text{Mo}$, soit 500 blocs.
 - on divise le fichier en $\frac{307}{2} = 154$ 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.
 - On associe chaque fragment à une bloc en mémoire, et on effectue la fusion (noter qu'il reste 500 – 154 blocs disponibles).
On a lu encore une fois le fichier, pour un coût total de $614 + 307 = 521$ Mo.
- Avec une mémoire $M = 1\text{Mo}$, soit 250 blocs.
 - on divise le fichier en 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.
 - On associe les 249 premiers fragments à un bloc en mémoire, et on effectue la fusion (on garde le dernier bloc 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 $614 \times 2 = 1228$ Mo.
 - Finalement on prend les deux derniers fragments, F_1 et F_2 , et on les fusionne. Cela représente une lecture de plus, soit $1228 + 307 = 1535$ 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 1Mo, 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.

5.3.4 L'opérateur de tri-fusion

Comme implanter le tri-fusion sous forme d'itérateur ? Réponse : *l'ensemble du tri est effectué dans la fonction open(), le next() ne fait que lire un par un les tuples du fichier trié stocké dans la zone temporaire*. Cela s'explique par le fait qu'il est impossible de fournir un résultat tant que l'ensemble du tri n'a pas été effectué. C'est seulement alors que l'on peut savoir quel est le plus petit élément, et le fournir comme réponse au premier appel *next()*.

La conséquence essentielle est que le tri est un opérateur *bloquant*. Quand on exécute une requête contenant un tri, rien ne se passe tant que le résultat n'a pas été complètement trié. Entre la requête

```
select * from Film
```

et la requête

```
select * from Film order by titre
```

La différence est donc considérable. Quelle que soit la taille de la table, l'exécution de la première donne un premier tuple instantanément : il suffit que le plan accède au premier enregistrement du fichier. Dans le second cas, il faudra attendre que toute la table ait été lue et triée.

5.4 S4 : Algorithmes de jointure

Supports complémentaires :

- Diapositives : algorithmes de jointure
 - Vidéo de présentation (à venir)
-

Passons maintenant aux *algorithmes de jointure*. Avec les opérateurs présentés dans cette section, nous complétons notre catalogue d'opérateurs et nous saurons exécuter toutes les requêtes SQL dites conjonctives, c'es-à-dire ne comprenant ni négation (*not exists*) ni union. Cela couvre *beaucoup* de requêtes et montre que l'implantation d'un moteur d'exécution de requêtes SQL n'est finalement pas si compliquée.

```
select a1, a2, ..., an
from T1, T2, ..., Tm
where T1.x = T2.y and ...
order by ...
```

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. On peut classer les algorithmes de jointure en deux catégories, suivant l'absence ou la présence d'index sur les attributs de jointure. En l'absence d'index, les trois algorithmes les plus répandus sont les suivants :

- 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.

- 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.
- Enfin la *jointure par hachage* est une technique qui donne de très bons résultats quand une des tables au moins tient en mémoire.

Quand un index est disponible (ce qui est le cas le plus courant, notamment quand la jointure associe la clé primaire d'une table à la clé étrangère d'une autre), on utilise une variante de l'algorithme par boucles imbriquées avec traversée d'index, dite *jointure par boucles indexée*.

Note : 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 note dans ce qui suit R et S les relations à joindre et T la relation résultat. Le nombre de blocs est noté respectivement par B_R et B_S . Le nombre des enregistrements de chaque relation est respectivement N_R et N_S .

Nous commençons par l'algorithme le plus efficace et le plus courant : celui utilisant un index.

5.4.1 Jointure avec un index

La jointure entre deux tables comporte le plus souvent une condition de jointure qui associe la clé primaire d'une table à la clé étrangère de l'autre. Voici quelques exemples pour s'en convaincre.

- Les films et leur metteur en scène

```
select * from Film as f, Artiste as a
where f.id_realisateur = a.id
```

- Les artistes et leurs rôles

```
select * from Artiste as a, Role as r
where a.id = r.id_acteur
```

- Les employés et leur département

```
select * from emp e, dept d
where e.dnum = d.num
```

Cette forme de jointure est courante car elle est “naturelle” : elle consiste à reconstruire l'information dispersée entre plusieurs tables par le processus de normalisation du schéma. Le point important (pour les performances) est que la condition de jointure porte sur au moins un attribut indexé (la clé primaire) et éventuellement sur deux si la clé étrangère est, elle aussi, indexée.

Cette situation permet l'exécution d'un algorithme à la fois très simple et assez efficace (on suppose pour l'instant que seule la clé primaire est indexée) :

- on parcourt séquentiellement la table contenant la clé étrangère ;
- pour chaque tuple, on utilise la valeur de la clé étrangère pour accéder à l'index sur la clé primaire de la second table : on récupère l'adresse *adr* d'un tuple ;
- il reste à effectuer un accès direct, avec l'adresse *adr*, pour obtenir le second tuple et constituer la paire.

Prenons comme exemple la première jointure SQL donnée ci-dessus. On va parcourir la table *Film* qui contient la clé étrangère *id_realisateur*. Pour chaque tuple *film* obtenu durant ce parcours, on prend la valeur de *id_realisateur* et on recherche, avec l'index, l'adresse de l'artiste correspondant. Il reste à effectuer un accès direct à la table *Artiste*.

Nous avons un nouvel opérateur que nous appellerons *IndexedJoin*. Il consomme des données fournis par deux autres opérateurs que nous avons déjà définis : un parcours séquentiel *FullScan*, un parcours d'index *IndexScan*. Il est complété par une troisième, lui aussi déjà étudié : *DirectAccess*. La forme du programme qui effectue ce type de jointure est illustrée par la figure [Algorithme de jointure avec index](#). Elle peut paraître un peu complexe, mais elle vaut la peine d'être étudiée soigneusement. Le motif est récurrent et doit pouvoir être repéré quand on étudie un plan d'exécution.

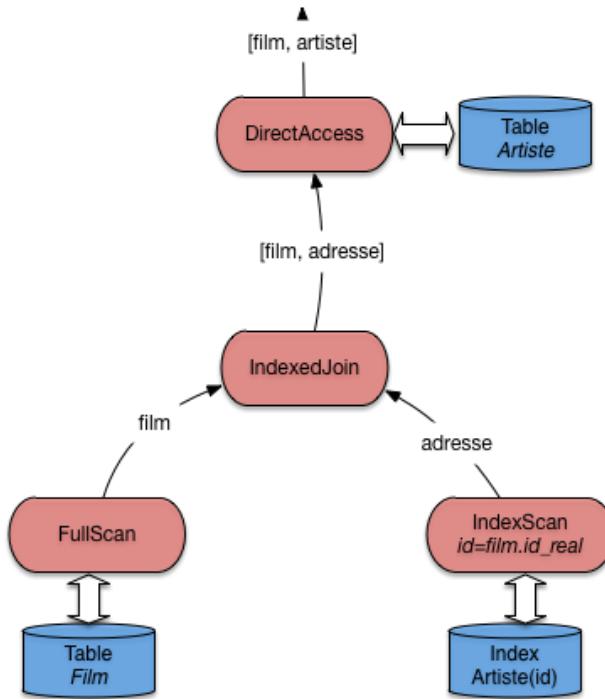


Fig. 5.9 – Algorithme de jointure avec index

L'opérateur `IndexedJoin` lui-même fait peut de choses puisqu'il s'appuie essentiellement sur d'autres composants qui font déjà une bonne partie du travail. Implanter un moteur d'exécution, tâche qui peut sembler extrêmement complexe à priori, s'avère en fait relativement simple avec cette approche très générique et décomposant les opérations nécessaires en briques élémentaires.

Voici le pseudo-code de la fonction `next()` de l'opérateur de jointure. Il faudrait, dans une implantation réelle, ajouter quelques contrôles, mais l'essentiel est là, et reste relativement simple.

```

function nextIndexJoin
{
    # $tScan est l'opérateur de parcours séquentiel de la première table
    # On récupère un tuple
    $tuple = $tScan.next();

    # On crée un opérateur de parcours d'index
    $iScan = new IndexScan ();
    # On exécute le parcours d'index avec la clé étrangère
    $iScan.open ($tuple.foreignKey);
    # On récupère l'adresse
    $addr = $iScan.next();

    # Et on renvoie la paire avec le tuple et l'adresse
    return [$tuple, $addr];
}
    
```

Cet algorithme peut être considéré comme le meilleur possible pour une jointure.

- Il s'appuie essentiellement sur un parcours d'index qui, en pratique, va s'effectuer en mémoire RAM car un arbre-B est compact, très sollicité, et résidera dans le cache la plupart du temps.
- Il permet un pipelinage complet : quelle que soit la taille des données, une application communiquant avec ce plan d'exécution recevra tout de suite la première paire-résultat, et obtiendra les suivantes avec très peu de latence à

chaque appel *next()*.

En contrepartie, l'algorithme nécessite des accès directs (aléatoires) pour obtenir les tuples de la seconde table. C'est loin d'être très efficace, pour des raisons déjà soulignées, et explique que la jointure reste une opération coûteuse.

Pour conclure sur cet algorithme, notez qu'il est présenté ici comme s'appuyant sur un parcours séquentiel, mais qu'il fonctionne tout aussi bien si la source de données (à gauche) est n'importe quel autre opérateur. Il est donc très facile à intégrer dans les plans d'exécution très complexes comprenant plusieurs jointures, sélection, projections, etc.

5.4.2 Jointure avec deux index

Peut-on faire mieux si les *deux* tables sont indexées ? Lorsque R et S ont un index sur l'attribut de jointure, on peut tirer parti du fait que les feuilles de ceux-ci sont triées sur cet attribut. En fusionnant les feuilles des index B_R et B_S de la même manière que pendant la phase de fusion de l'algorithme de jointure par tri-fusion, on obtient une liste de couples d'adresses d'enregistrements de R et S à joindre. Cette première phase est très efficace, car les deux index sont très probablement en mémoire et l'algorithme de fusion est lui-même simple et performant.

La deuxième phase consiste à lire les enregistrements par deux accès directs, l'un sur R , l'autre sur S . C'est ici que les choses se compliquent, car la multiplication des accès aléatoires devient très pénalisante. Comme déjà discuté, si une partie significative d'une table est concernée, il est préférable d'effectuer un parcours séquentiel qu'une succession d'accès directs. Pour cette raison, beaucoup de SGBD (dont Oracle), en présence d'index sur l'attribut de jointure dans les deux relations, préfèrent quand même appliquer l'algorithme `IndexedJoin`. L'amélioration permise par cette situation reste le choix de la table à parcourir séquentiellement : pour des raisons évidentes on prend la plus petite.

5.4.3 Jointure par boucles imbriquées

Nous abordons maintenant le cas des jointures où aucun index n'est disponible. Disons tout de suite que les performances sont alors nettement moins bonnes, et devraient amener à considérer la création d'un index approprié pour des requêtes fréquemment utilisées.

L'algorithme direct et naïf, que nous appellerons `NestedLoop`, s'adapte à tous les prédictats 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 prédictat de jointure. La fonction de base est la jointure de deux listes en mémoire, $L1$ et $L2$, et se décrit simplement comme suit :

```
function JoinList
{
    # $L1 est la liste dite "extérieure"
    # $L2 est la liste dite "intérieure"
    # $condition est la condition de jointure
    résultat = [];

    for tuple1 in $L1
    do
        for tuple2 in $L2
        do
            if (condition ($tuple1, $tuple2) = true) the
                $résultats[] = ($tuple1, $tuple2);
            fi
        done
    done
}
```

Le coût de cette fonction se mesure au nombre de fois où on effectue le test de la condition de jointure. Il est facile de voir que chaque tuple de $L1$ est comparé à chaque tuple de $L2$, d'où un coût de $|L1| \times |L2|$.

Maintenant, ce qui nous intéresse dans un contexte de base de données, c'est aussi (surtout) le nombre de lectures de blocs nécessaires. Dès lors que la jointure implique des accès disques, ces entrées/sorties (E/S) constituent le facteur prédominant. La méthode de base, illustrée par la figure *Boucle imbriquée sur les blocs*, consiste à charger toutes les paires de blocs en mémoire, et à appliquer la fonction *JoinList* sur chaque paire.

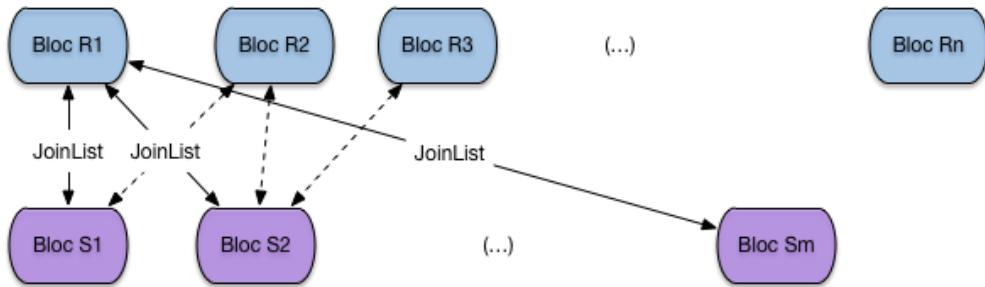


Fig. 5.10 – Boucle imbriquée sur les blocs

Le pseudo-code suivant montre la jointure par boucle imbriquée, constituant toutes les paires de blocs par un unique parcours séquentiel sur la première table, et des parcours séquentiels répétés sur la seconde.

```

function JoinList
{
    # $R est la table dite "extérieure"
    # $S est la table dite "intérieure"
    # $condition est la condition de jointure
    résultat = [];

    for blocR in $R
    do
        for blocS in $S
        do
            JoinList ($blocR, $blocS)
        done
    done
}
    
```

Le principale mérite (le seul) de cet algorithme est de demander très peu de mémoire : deux blocs suffisent. En revanche, le nombre de lectures et très important :

- il faut lire toute la table R ,
- il faut lire autant de fois la table S qu'il y a de blocs dans R .

Le nombre de lectures est donc $B_R + B_R \times B_S$. Cette petite formule montre au passage qu'il est préférable de prendre comme table extérieure la plus petite des deux.

Cela étant, on peut faire beaucoup mieux en utilisant plus de mémoire. Soit R la table la plus petite. Si le nombre de blocs M est au moins égal à $B_R + 1$, la table R tient en mémoire centrale. On peut alors lire S une seule fois, bloc par bloc, en effectuant à chaque fois la jointure entre le bloc et l'ensemble des blocs de R chargés en RAM (figure *Boucle imbriquée avec chargement complet d'une table en RAM*).

Avec cette solution (pas si rare), le coût est de $B_R + B_S$: une seule lecture des deux tables suffit. D'un coût quadratique dans les tailles des relations, lorsqu'on n'a que 3 blocs, on est passé à un coût linéaire.

S'il s'agit d'une équi-jointure, une variante encore améliorée 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 *JoinList* c'est une fonction quadratique du nombre d'enregistrements).

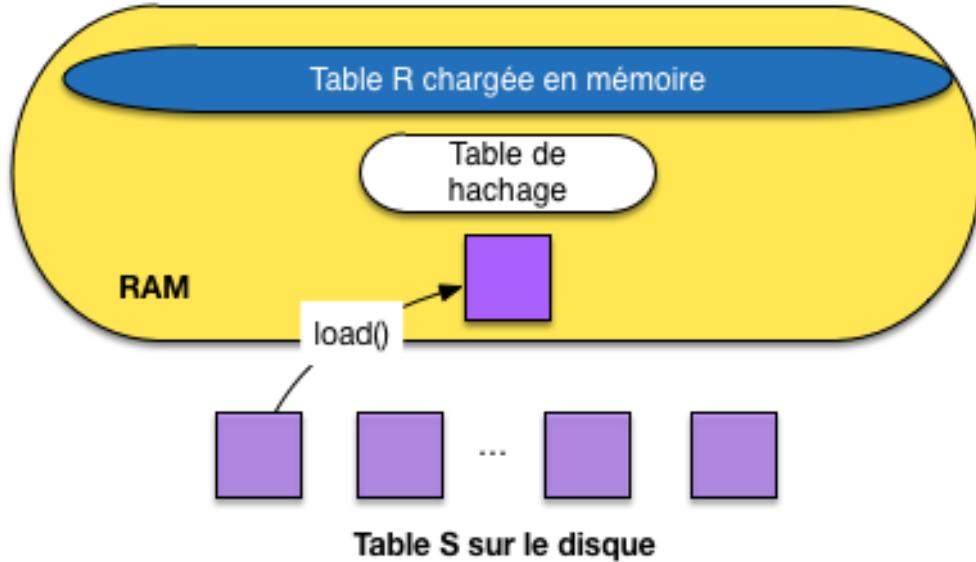


Fig. 5.11 – Boucle imbriquée avec chargement complet d'une table en RAM

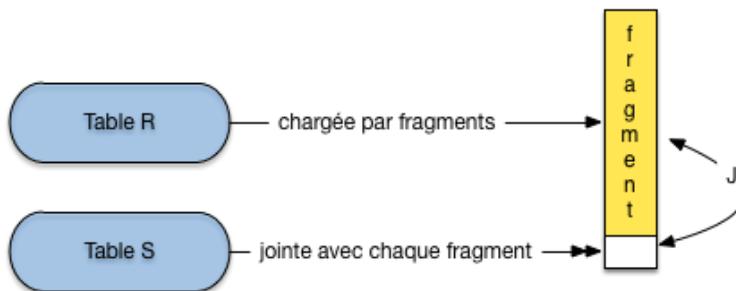


Fig. 5.12 – Boucle imbriquée avec chargement par fragments d'une table en RAM

Malheureusement il arrive souvent que R ne tient pas en mémoire car $B_R > M - 1$. Voici alors la version la plus générale de la jointure par boucles imbriquées (figure [Boucle imbriquée avec chargement par fragments d'une table en RAM](#)) : on découpe R en fragments de taille $M - 2$ blocs et on utilise la variante ci-dessus pour chaque groupe. R est lu une seule fois, groupe par groupe, S est lu $\lceil \frac{B_R}{M-1} \rceil$ fois. On obtient un coût final de :

$$B_R + \lceil \frac{B_R}{M-1} \rceil$$

Exemple

On prend l'exemple d'une jointure entre Film et Artiste en supposant, pour les besoins de la cause, qu'il n'y a pas d'index. La table Film occupe 1000 blocs, et la table Artiste 10 000 blocs. On suppose que la mémoire disponible a pour taille $M = 252$ blocs.

- En prenant la table Artiste comme table extérieure, on obtient le coût suivant :

$$10000 + \lceil \frac{10000}{250} \times 1000 \rceil = 50000$$

- Et en prenant la table Film comme table extérieure :

$$1000 + \lceil \frac{1000}{250} \times 10000 \rceil = 41000$$

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

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.

5.4.4 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 disque

Note : 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.

On utilise l'algorithme de tri externe vu précédemment pour cette première étape. La deuxième phase, dite de fusion, consiste à lire bloc par bloc chacun des deux fichiers temporaires et à parcourir séquentiellement en parallèle ces deux fichiers pour trouver les enregistrements à joindre. Comme les fichiers sont triées, sauf cas exceptionnel, chaque bloc n'est lu qu'une fois.

Prenons l'équijointure de R et S sur les attributs a et b .

```
select * from R, S where R.a = S.b
```

On va trier R et S et on parcourt ensuite les tables triées en parallèle. Regardons plus en détail la fusion. C'est une variante très proche de l'algorithme de fusion de liste. On commence avec les premiers enregistrements r_1 et s_1 de chaque table.

- Si $r_1.a = s_1.b$, on joint les deux enregistrements, on passe au enregistrements suivants, jusqu'à ce que $r_i.a \neq s_i.b$.
- Si $r_1.a < s_1.b$, on avance sur la liste de R .
- Si $r_1.a > s_1.b$, on avance sur la liste 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.

L'opérateur de jointure peut s'appuyer sur l'opérateur de tri, déjà étudié. Il suffit donc d'implanter la jointure de deux listes triées dans un opérateur Merge. Voici la fonction `next()` de cet opérateur, avec deux opérateurs de tris opérant respectivement sur la première et la seconde table (plus généralement, ces opérateurs de tri peuvent opérer sur n'importe quel sous-plan d'exécution).

```
function nextMerge
{
    # $triR est l'opérateur de tri sur la première table
    # $triS est l'opérateur de tri sur la seconde table
    # a et b désignent les attributs de jointure

    # Récupération de tuples fournis par les opérateurs
    $tupleR = $triR.next();
    $tupleS = $triS.next();

    # Tant que les deux tuples de joignent pas sur a et b, on avance
    # sur une des deux listes
    while ($tupleR.a != $tupleS.b) do
        if ($tupleR.a < $tupleS.b) then
            $tupleR = $triR.next();
        else
            $tupleS = $triS.next();
        fi
    done

    return [$tupleR, $tupleS];
}
```

Le plan d'exécution typique d'une jointure par tri-fusion avec cet opérateur est illustré par la figure *Plan d'exécution type pour la jointure par tri-fusion*.

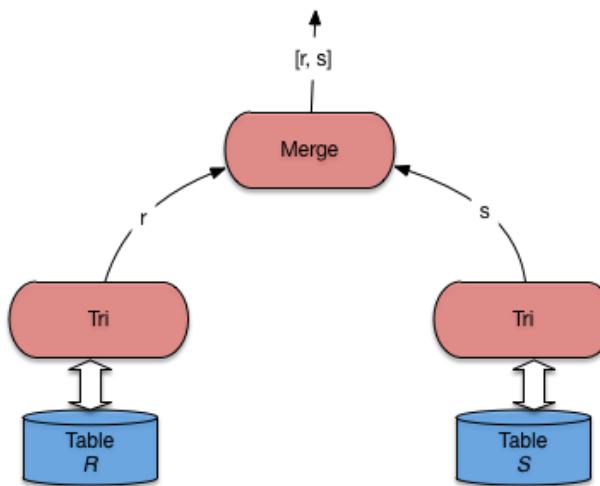


Fig. 5.13 – Plan d'exécution type pour la jointure par tri-fusion

La jointure s par tri-fusion est illustrée dans la figure [Exemple de jointure par tri-fusion](#).

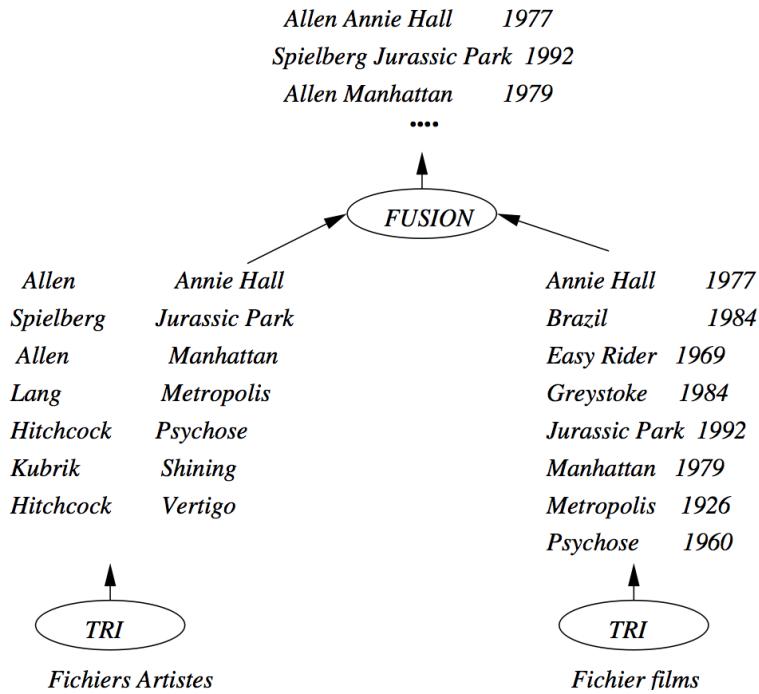


Fig. 5.14 – Exemple de jointure par tri-fusion

Le coût de la jointure par tri-fusion est important, et impose une latence due à la phase de tri initiale. Une fois la phase de fusion débutée, le débit est en revanche très rapide. La performance dépend donc essentiellement du tri, et donc de la mémoire disponible. C'est l'algorithme privilégié par les SGBD pour la jointure sans index de très grosses tables (situation qu'il vaut mieux éviter quand c'est possible).

5.4.5 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, il comprend deux phases : une phase de partitionnement des deux relations en k fragments 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 et S , on ne comparera les enregistrements de chaque fragment F_R^i de R qu'aux enregistrements du fragment F_S^i associée de S . Notez bien qu'il s'agit du même exposant i : les fragments sont associés par paire, ce qui implique que l'on a la garantie qu'aucun tuple de F_R^i ne joint avec un tuple de F_S^j , pour $i \neq j$.

Le partitionnement de R se fait par hachage. On suppose toujours que a et b sont les attributs de jointure respectifs et on note h la fonction de hachage qui s'applique à la valeur de a ou b et renvoie un entier compris entre 1 et k .

Un enregistrement r de R est donc placé dans le fragment $F_R^{h(r.a)}$; un enregistrement s de S est donc placé dans le fragment $F_S^{h(s.b)}$. On obtient exactement le même nombre de fragments pour R et S , placés sur le disque si nécessaire, comme le montre la figure [Première phase de la jointure par hachage : le partitionnement](#).

Important : Comment est choisi k , le nombre de fragments ? *Le critère que, pour la plus petite des deux tables, chaque fragment doit tenir dans la mémoire disponible.* Si, par exemple, R est la plus petite des deux tables et occupe 100 blocs, alors que 20 blocs de RAM sont disponibles, il faudra au moins $k = 5$ fragments. Pourquoi ? Lire la suite.

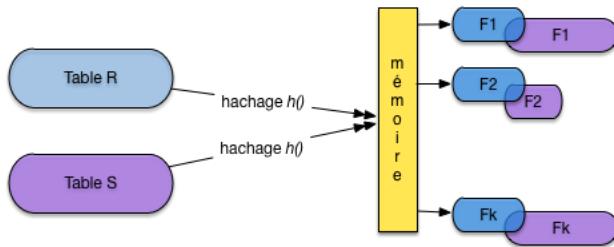


Fig. 5.15 – Première phase de la jointure par hachage : le partitionnement

On peut alors passer à la seconde phase, dite de jointure. La remarque fondamentale ici est la suivante : si deux tuples r et s doivent être joints, alors on a $h(r.a) = h(s.b) = u$ et on les trouvera, respectivement, dans F_R^u et F_S^u . En d'autres termes, il suffit d'effectuer la jointure sur les paires de fragments correspondant à la même valeur de la fonction de hachage.

Note : Le paragraphe qui précède est vraiment le cœur de l'algorithme de hachage et justifie tout son fonctionnement. Lisez-le et relisez-le jusqu'à être convaincus que vous le comprenez.

La deuxième phase consiste alors pour $i = 1, \dots, k$, à lire le fragment F_R^i de R en mémoire et à effectuer la jointure avec le fragment F_S^i de S . La technique de jointure à appliquer au fragment est exactement celle par boucle imbriquées, décrite ci-dessus, quand l'une des deux tables tient en RAM : . Le point important (et qui explique le choix du nombre de fragments) est qu'au moins l'un des deux fragments à joindre doit résider en mémoire ; l'autre, lu séquentiellement, peut avoir une taille quelconque.

La figure *Première phase de la jointure par hachage : la jointure* montre le calcul de la jointure pour deux fragments. Celui de la première table est entièrement en mémoire, celui de la seconde est lu séquentiellement et placé au fur et à mesure de la lecture dans le reste de la mémoire disponible, pour être joint avec le fragment résidant.

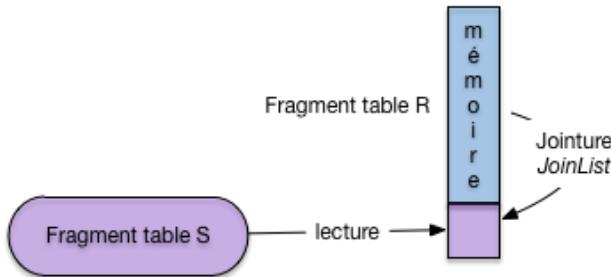


Fig. 5.16 – Première phase de la jointure par hachage : la jointure

Le coût de la première phase de partitionnement de cet algorithme est $2 \times (B_R + B_S)$. Chaque relation est lue entièrement et hachée dans les fragments qui sont écrits sur disque bloc par bloc.

Le coût de la deuxième phase est de $2 \times (B_R + B_S)$. En effet les relations partitionnées sont lues une fois chacune, fragment par fragment. Le coût total de cet algorithme est donc $3 \times (B_R + B_S)$. Noter que cet algorithme est très gourmand en mémoire. Cet algorithme est bien adapté aux jointures déséquilibrées pour lesquelles une des tables est petite en taille. Dans le meilleur des cas où un seul fragment est nécessaire (la table tient entièrement en mémoire) on retrouve tout simplement la jointure par boucles imbriquées décrite précédemment. La jointure par hachage peut être vue comme une généralisation de ce algorithme simple.

Comment implanter cet algorithme de jointure sous forme d'itérateur ? Et bien, comme pour le tri, toute la phase de hachage s'effectue dans le `open()` et cet opérateur est donc *bloquant* : la phase de hachage correspond à une latence

perçue par l'utilisateur qui attend sans que rien (en apparence) ne se passe. La phase de jointure peut, elle, être très rapide, et surtout fournit régulièrement des tuples à l'application cliente.

Concluons cette section avec deux remarques :

- Excepté les algorithmes basés sur une boucle imbriquée avec ou sans index, les algorithmes montrés ont été conçus pour le prédicat d'égalité. 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.
- 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.

Evaluation et optimisation

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.

Le module chargé de cette tâche, *l'optimiseur de requêtes*, tient donc un rôle extrêmement important puisque l'efficacité d'un SGBD est fonction, pour une grande part, du temps d'exécution des requêtes. Ce module est complexe. Il applique d'une part des techniques éprouvées, d'autre part des *heuristiques* propres à chaque système. Il est en effet reconnu qu'il est très difficile 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 circonsstances.

6.1 S1 : traitement de la requête

Supports complémentaires :

- Diapositives : traitement d'une requête
 - Vidéo de présentation (à venir)
-

Cette section est consacrée à la phase de traitement permettant de passer d'un requête SQL à une forme "opérationnelle". 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.

6.1.1 Décomposition en bloc

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 `select-from-where` sans imbrication. 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 = A
```

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, les systèmes relationnels ne sont pas toujours capables de déceler ce type d'équivalence. Le choix de la syntaxe de la requête SQL a donc une influence sur les possibilités d'optimisation laissées au SGBD.

Prenons un exemple concret pour comprendre la subtilité de certaines situations, et pourquoi le système à parfois besoin qu'on lui facilite la tâche. Notre base de données est toujours la même, rappelons le schéma de la table `Role` car il est important.

```
create table Role (id_acteur integer not null,
                   id_film integer not null,
                   nom_role varchar(30) not null,
                   primary key (id_acteur, id_film),
                   foreign key (id_acteur) references Artiste(id),
                   foreign key (id_film) references Film(id),
);
```

Le système crée un index sur la clé primaire qui est composée de deux attributs. Quelles requêtes peuvent tirer parti de cet index ? Celles sur l'identifiant de l'acteur, celles sur l'identifiant du film ? Réfléchissez-y, réponse plus loin.

Maintenant, notre requête est la 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.id_film = r.id_film
and r.id_acteur = 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.id_film = r.id_film
and f.annee = 1958
and r.id_acteur in (select id_acteur
                     from Artiste
                     where nom='Stewart'
                     and prenom='James')
```

Au lieu d'utiliser `in`, on peut également effectuer une requête *corrélée* avec `exists`.

```
select titre
from Film f, Role r
where f.id_film = r.id_film
and f.annee = 1958
```

```
and exists (select 'x'
            from Artiste a
            where nom='Stewart'
            and prenom='James'
            and r.id_acteur = a.id_acteur)
```

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

```
select titre from Film
where annee = 1958
and id_film in
    (select id_film from Role
     where id_acteur in
         (select id_acteur
          from Artiste
          where nom='Stewart'
          and prenom='James'))
```

Que l'on peut aussi formuler en :

```
select titre from Film
where annee = 1958
and exists
    (select * from Role
     where id_film = Film.id
     and exists
        (select *
         from Artiste
         where id = Role.id_acteur
         and nom='Stewart'
         and prenom='James'))
```

Dans les deux derniers cas on a trois blocs. La requête est peut-être plus facile à comprendre (vraiment ?), 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 car aucun index n'est disponible. Enfin pour chacun de ces rôles il faut utiliser l'index sur `Artiste`.

Pourquoi ne peut-on pas utiliser l'index sur `Role` ?

La clé de `Role` est une clé composite (`id_acteur, id_film`). L'index est un arbre B construit sur la concaténation des deux identifiants *dans l'ordre où ils sont spécifiés*. Souvenez-vous : un arbre B s'appuie sur l'ordre des clés, et on peut effectuer des recherches sur un *préfixe* de la clé. En revanche il est impossible d'utiliser l'arbre B sur un *suffixe*. Ici, on peut utiliser l'index pour des requêtes sur `id_acteur`, pas pour des requêtes sur `id_film`. C.Q.F.D.

Telle quelle, cette syntaxe basée sur l'imbrication 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 si 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.

6.1.2 Traduction et réécriture

Nous nous concentrerons 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.

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. À l'issue de cette phase, le système considère que la requête est bien formée.

L'étape suivante consiste à traduire la requête q en une expression algébrique $e(q)$. Nous allons prendre pour commencer une requête un peu plus simple que la précédente : trouver 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  nom_role = 'John Ferguson'
and    f.id = r.id_ilm
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 :

$$\pi_{\text{titre}}(\sigma_{\text{annee}=1958 \wedge \text{nom_role}='John Ferguson'}(Film \bowtie id=id_film Role))$$

Cette expression comprend des opérations unaires (un seul argument) et des opérations binaires. On peut la représenter sous la forme d'un arbre (figure [Expression algébrique sous forme arborescente](#)), ou *Plan d'Exécution Logique* (PEL), représentant l'expression algébrique équivalente à la requête SQL. Dans l'arbre, les feuilles correspondent les tables arguments de l'expression algébrique, et les nœuds internes aux opérateurs algébriques. Un arc entre un nœud n et son nœud père p indique que 'opération p ' s'applique au résultat de l'opération n .

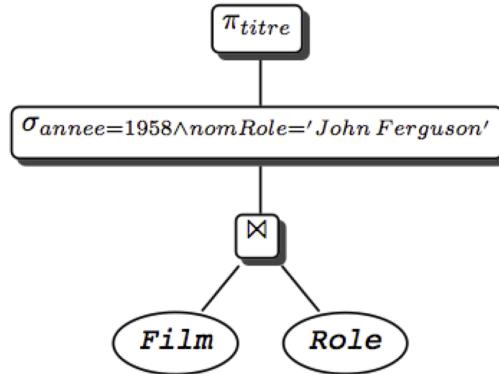


Fig. 6.1 – Expression algébrique sous forme arborescente

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.

Avec cette représentation de la requête sous une forme "opérationnelle", nous sommes prêts pour la phase d'optimisation.

6.2 S2 : optimisation de la requête

Supports complémentaires :

- Diapositives : l'optimisation d'une requête
 - Vidéo de présentation (à venir)
-

6.2.1 La réécriture

Nous disposons donc d'un plan d'exécution logique (PEL) présentant, sous une forme normalisée (par exemple, les projections, puis les sélections, puis les jointures) les opérations nécessaires à l'exécution d'une requête donnée.

On peut reformuler le PEL 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 plans d'exécution alternatifs, et tenter d'évaluer lequel est le meilleur. Voici la liste des règles de réécriture les plus importantes :

- Commutativité des jointures.

$$R \bowtie S \equiv S \bowtie R$$

- Associativité des jointures

$$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$$

- Regroupement des sélections

$$\sigma_{A='a' \wedge B='b'}(R) \equiv \sigma_{A='a'}(\sigma_{B='b'}(R))$$

- Commutativité de la sélection et de la projection

$$\pi_{A_1, A_2, \dots, A_p}(\sigma_{A_i='a}(R)) \equiv \sigma_{A_i='a}(\pi_{A_1, A_2, \dots, A_p}(R)), i \in \{1, \dots, p\}$$

- Commutativité de la sélection et de la jointure

$$\sigma_{A='a'}(R \bowtie S) \equiv \sigma_{A='a'}(R) \bowtie S$$

- Distributivité de la sélection sur l'union

$$\sigma_{A='a'}(R \cup S) \equiv \sigma_{A='a'}(R) \cup \sigma_{A='a'}(S)$$

- Commutativité de la projection et de la jointure

$$\pi_{A_1 \dots A_p}(R \bowtie_{A_i=B_j} \pi_{B_1 \dots B_q}(S)) \quad i \in \{1, \dots, p\}, j \in \{1, \dots, q\}$$

- 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 sont à la base du processus d'optimisation dont le principe est *d'enumerer* tous les plans d'exécution possible. Par exemple la règle 3 permet de gérer finement l'affectation des sélections . En effet si la relation est indexée sur l'attribut B, la règle justifie de filter 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 le plus tôt possible dans le plan (et donc le plus bas possible dans l'arbre) 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 fonction de la taille des relations en entrée. C'est particulièrement vrai pour la jointure qui est une opération coûteuse. 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 deux heuristiques le plus souvent effectives pour transformer un PEL en un meilleur PEL (équivalent).

Voici un algorithme simple résumant ces idées :

- Séparer les sélections avec plusieurs prédictats en plusieurs sélections à un prédictat (règle 3).
- Descendre les sélections le plus bas possible dans l'arbre (règles 4, 5, 6)
- Regrouper les sélections sur une même relation (règle 3).
- Descendre les projections le plus bas possible (règles 7 et 8).
- Regrouper les projections sur une même relation.

Reprenons notre requête cherchant le film paru en 1958 avec un rôle “John Ferguson”. Voici l'expression algébrique complète.

$$\pi_{titre}(\sigma_{annee=1958 \wedge nom_role='John Ferguson'}(Film \bowtie_{id=id_film} (Role)))$$

L'expression est correcte, mais probablement pas optimale. Appliquons notre algorithme. La première étape donne l'expression suivante :

$$\pi_{titre}(\sigma_{annee=1958}(\sigma_{nom_role='John Ferguson'}(Film \bowtie_{id=id_film} (Role))))$$

On a donc séparé les sélections. Maintenant on les descend dans l'arbre :

$$\pi_{titre}(\sigma_{annee=1958}(Film) \bowtie_{id=id_film} \sigma_{nom_role='John Ferguson'}(Role))$$

Finalement il reste à ajouter des projections pour limiter la taille des enregistrements. À chaque étape du plan, les projections consisteraient (nous ne les montrons pas) à supprimer les attributs inutiles. Pour conclure deux remarques sont nécessaires :

- le principe “sélection avant jointure” conduit dans la plupart des cas à un PEL plus efficace ; mais il peut arriver (très rarement) que la jointure soit plus réductrice en taille et que la stratégie “jointure d'abord, sélection ensuite”, conduise à un meilleur PEL.
- 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.

6.2.2 Plans d'exécution

Un plan d'exécution physique (PEP) est un arbre d'opérateurs (on parle *d'algèbre physique*), issus d'un “catalogue” propre à chaque SGBD. On retrouve, avec des variantes, les principaux opérateurs d'un SGBD à un autre. Nous les avons étudiés dans le chapitre *Evaluation et optimisation*, et nous les reprenons maintenant pour étudier quelques exemples de plan d'exécution.

On peut distinguer tout d'abord les opérateurs *d'accès* :

- le parcours séquentiel d'une table, `FullScan`,
- le parcours d'index, `IndexScan`,
- l'accès direct à un enregistrement par son adresse, `DirectAccess`, nécessairement combiné avec le précédent.

Puis, une second catégorie que nous appellerons opérateurs de *manipulation* :

- la sélection, `Filter` ;
- la projection, `Project` ;
- le tri, `Sort` ;
- la fusion de deux listes, `Merge` ;
- la jointure par boucles imbriquées indexées, `IndexedNestedLoop`, abrégée en `INL`.

Cela nous suffira. Nous reprenons notre requête cherchant les films parus en 1958 avec un rôle “John Ferguson”. Pour mémoire, le plan d’exécution logique auquel nous étions parvenu est le suivant.

$$\pi_{titre}(\sigma_{annee=1958}(Film) \bowtie_{id=id_film} \sigma_{nom_role='John Ferguson'}(Role))$$

Nous devons maintenant choisir des opérateurs physiques, choix qui dépend de nombreux facteurs : chemin d'accès, statistiques, nombre de blocs en mémoire centrale. En fonction de ces paramètres, l'optimiseur choisit, pour chaque nœud du PEL, un opérateur physique ou une combinaison d'opérateurs.

Une première difficulté vient du grand nombre de critères à considérer : quelle mémoire allouer, comment la partager entre opérateurs, doit-on privilégier temps de réponse ou temps d'exécution, etc. 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 d'autres nœuds (notamment concernant l'utilisation de la mémoire). Tout cela mène à une procédure d'optimisation complexe, mise au point et affinée par les concepteurs de chaque système, dont il est bien difficile (et sans doute peu utile) de connaître les détails. Ce qui suit est donc plutôt une méthodologie générale, illustrée par des exemples.

Prenons comme hypothèse directrice que l'objectif principal de l'optimiseur soit d'exécuter les jointures avec l'algorithme `IndexNestedLoop` (ce qui est raisonnable pour obtenir un bon temps de réponse et limiter la mémoire nécessaire). Pour chaque jointure, il faut donc envisager les index disponibles. Ici, la jointure s'effectue entre `Film` et `Role`, ce dernier étant indexé sur la clé primaire (`id_acteur`, `id_film`). La jointure est commutative (cf. les règles de réécriture. On peut donc effectuer, de manière équivalente,

$$Film \bowtie_{id=id_film} Role$$

ou

$$Role \bowtie_{id_film=id} Film$$

Regardons pour quelle version nous pouvons utiliser un index avec l'algorithme `IndexNestedLoop`. Dans le premier cas, nous lisons des tuples `film` (à gauche) et pour chaque film nous cherchons les rôles (à droite). Peut-on utiliser l'index sur rôle ? Non, pour les raisons déjà expliquées dans la session 1 : l'identifiant du film est un *suffixe* de la clé de l'arbre B, et ce dernier est donc inopérant.

Second cas : on lit des rôles (à gauche) et pour chaque rôle on cherche le film. Peut-on utiliser l'index sur film ? Oui, bien sûr : on est dans le cas où on lit les tuples de la table contenant la clé étrangère, et où on peut accéder par la clé primaire à la seconde table (revoir le chapitre *Opérateurs et algorithmes* pour réviser les algorithmes de jointure si nécessaire). Nos règles de réécriture algébrique nous permettent de reformuler le plan d'exécution logique, en commutant la jointure.

$$\pi_{titre}(\sigma_{nom_role='John Ferguson'}(Role) \bowtie_{id_film=id} \sigma_{annee=1958}(Film))$$

Et, plus important, nous pouvons maintenant implanter ce plan avec l'algorithme de jointures imbriquées indexées, ce qui donne l'arbre de la figure *Le plan d'exécution “standard”*.

Note : L'opérateur de projection n'est pas montré sur les figures. Il intervient de manière triviale comme racine du plan d'exécution complet.

Peut-on faire mieux ? Oui, en créant des index. La première possibilité est de créer un index pour éviter un parcours séquentiel de la table gauche. Ici, on peut créer un index sur le nom du rôle, et remplacer l'opérateur de parcours séquentiel par la combinaison habituelle (`IndexScan + DirectAccess`). Cela donne le plan de la figure *Le plan d'exécution avec deux index*.

Ce plan est certainement le meilleur. Cela ne signifie pas qu'il faut créer des index à tort et à travers : la maintenance d'index a un coût, et ne se justifie que pour optimiser des requêtes fréquentes et lentes.

Une autre possibilité pour faire mieux est de créer un index sur la *clé étrangère*, ce qui ouvre la possibilité d'effectuer les jointures dans n'importe quel ordre (pour les jointures “naturelles”, celles qui associent clé primaire et clé étrangère). Certains systèmes (MySQL) le font d'ailleurs systématiquement.

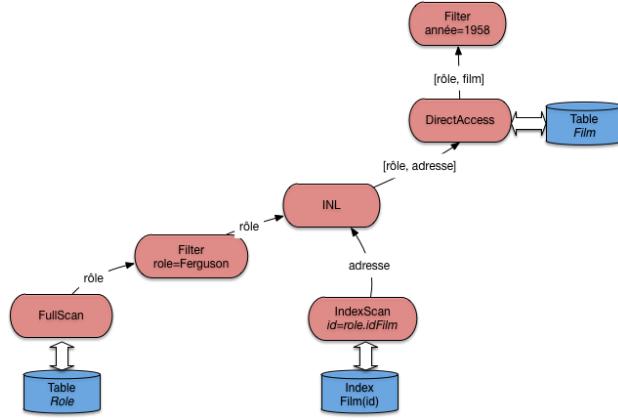


Fig. 6.2 – Le plan d'exécution “standard”

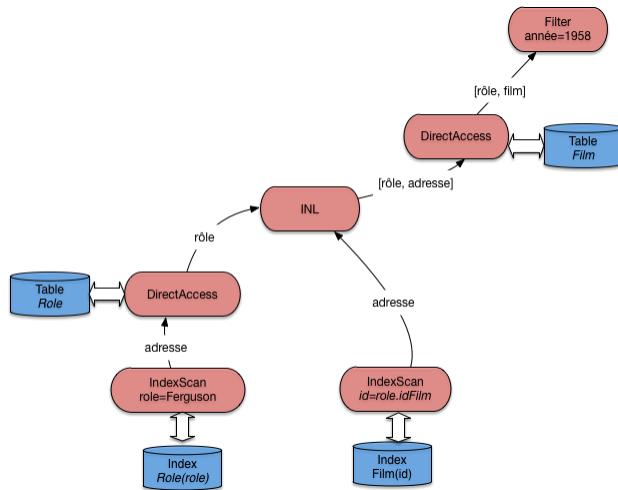


Fig. 6.3 – Le plan d'exécution avec deux index

Si, donc, la table `Role` est indexée sur la clé primaire (`id_acteur, id_film`) et sur la clé étrangère `id_film` (ce n'est pas redondant), un plan d'exécution possible est celui de la figure [Le plan d'exécution avec index sur les clés étrangères](#).

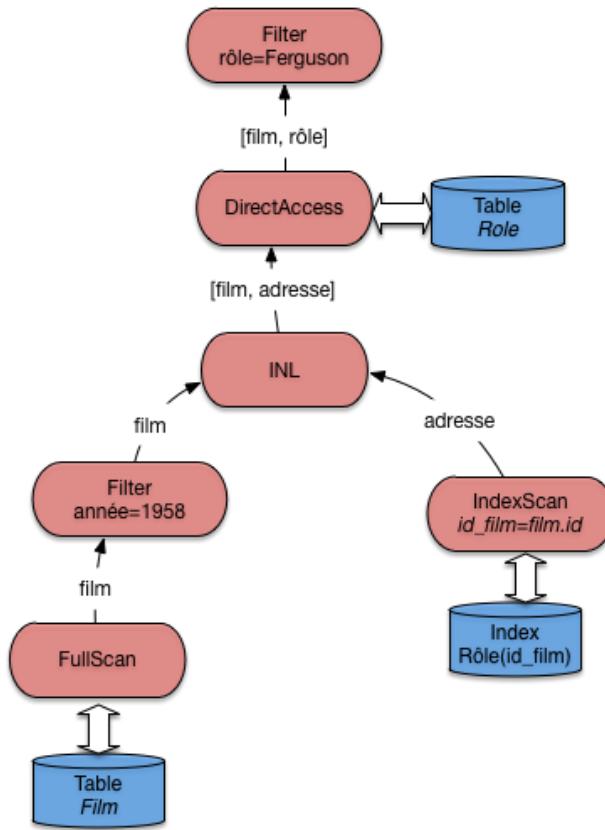


Fig. 6.4 – Le plan d'exécution avec index sur les clés étrangères

Ce plan est comparable à celui de la figure [Le plan d'exécution “standard”](#). Lequel des deux serait choisi par le système ? En principe, on choisirait comme table de gauche celle qui contient le moins de tuples, pour minimiser le nombre de demandes de lectures adressées à l'index. Mais il se peut d'un autre côté que cette table, tout en contenant moins de tuples, soit beaucoup plus volumineuse et que sa lecture séquentielle soit considérée comme trop pénalisante. Ici, statistiques et évaluation du coût entrent en jeu.

On pourrait finalement créer un index sur l'année sur `film` pour éviter tout parcours séquentiel : à vous de déterminer le plan d'exécution qui correspond à ce scénario.

Finalement, considérons le cas où aucun index n'est disponible. Pour notre exemple, cela correspondrait à une sévère anomalie puisqu'il manquerait un index sur la clé primaire. Toujours est-il que dans un tel cas le système doit déterminer l'algorithme de jointure sans index qui convient. La figure [Le plan d'exécution en l'absence d'index](#) illustre le cas où c'est l'algorithme de tri-fusion qui est choisi.

La présence de l'algorithme de tri-fusion pour une jointure doit alerter sur l'absence d'index et la probable nécessité d'en créer un.

6.2.3 Arbres en profondeur à gauche

Pour conclure cette section sur l'optimisation, on peut généraliser l'approche présentée dans ce qui précède au cas des requêtes multi-jointures, où de plus chaque jointure est “naturelle” et associe la clé primaire d'une table à la clé

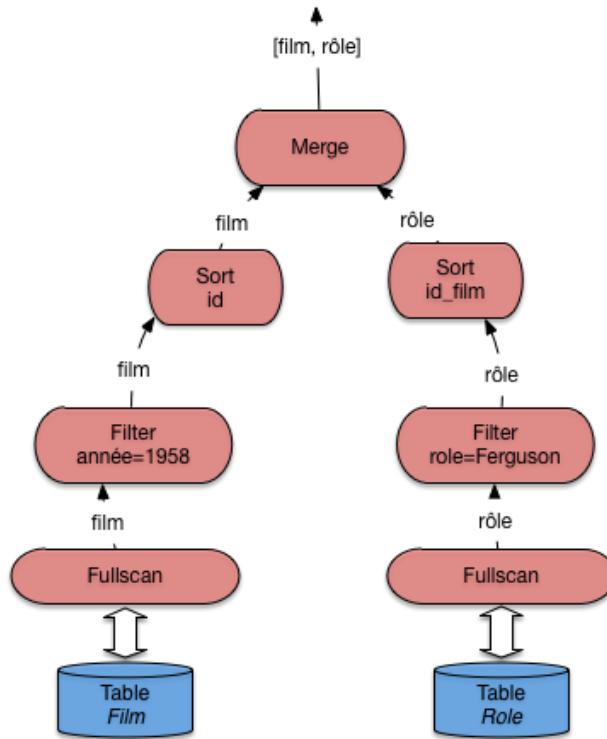


Fig. 6.5 – Le plan d'exécution en l'absence d'index

étrangère de l'autre. Voici un exemple sur notre schéma : on cherche tous les films dans lesquels figure un acteur islandais.

```

select *
from Film, Role, Artiste, Pays
where Pays.nom='Islande'
and Film.id=Role.id_film
and Role.id_acteur=Artiste.id
and Artiste.pays = Pays.code
  
```

Ces requêtes comprenant beaucoup de jointures sont courantes, et le fait qu'elles soient naturelles est également courant, pour des raisons déjà expliquées.

Quel est le plan d'exécution typique ? Une stratégie assez standard de l'optimiseur va être d'éviter les opérateurs bloquants et la consommation de mémoire. Cela mène à chercher, le plus systématiquement possible, à appliquer l'opérateur de jointure par boucles imbriquées indexées. Il se trouve que pour les requêtes considérées ici, c'est toujours possible. En fait, on peut représenter ce type de requête par une “chaîne” de jointures naturelles. Ici, on a (en ne considérant pas les sélections) :

$$Film \bowtie Role \bowtie Artiste \bowtie Pays$$

Il faut lire au moins une des tables séquentiellement pour “amorcer” la cascade des jointures par boucles imbriquées. Mais, pour toutes les autres tables, un accès par index devient possible. Sur notre exemple, le bon ordre des jointures est

$$Artiste \bowtie Pays \bowtie Role \bowtie Film$$

Le plan d'exécution consistant en une lecture séquentielle suivie de boucles imbriquées indexées est donné sur la figure *Le plan d'exécution en l'absence d'index*. Il reste bien sûr à le compléter. Mais l'aspect important est que ce plan

fonctionne entièrement en mode pipelinage, sans latence pour l'application. Il exploite au maximum la possibilité d'utiliser les index, et minimise la taille de la mémoire nécessaire.

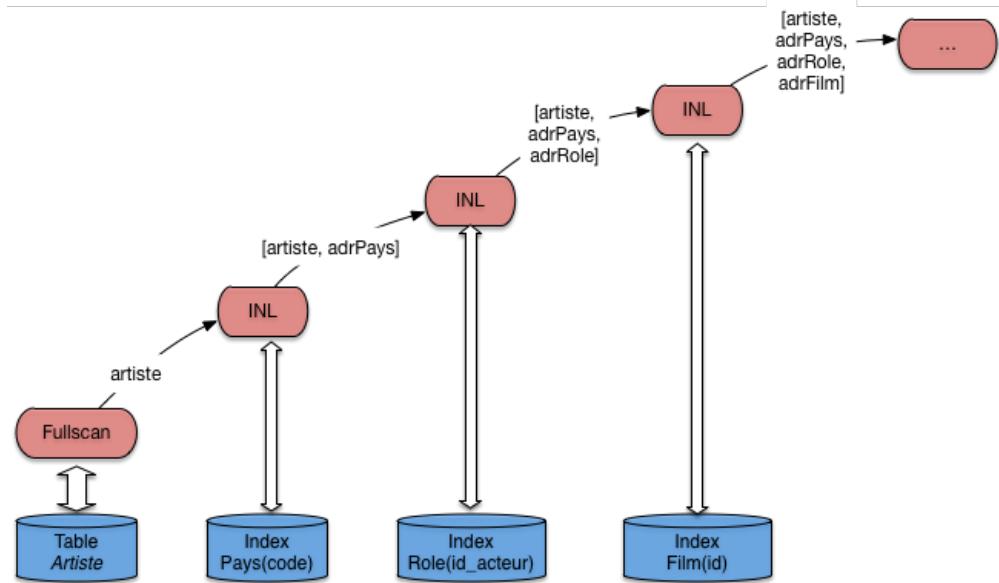


Fig. 6.6 – Le plan d'exécution en l'absence d'index

Ce plan a la forme caractéristique d'un *arbre en profondeur à gauche* ("left-deep tree"). C'est celle qui est recherchée classiquement par un optimiseur, et la forme de base que vous devriez retenir comme point de repère pour étudier un plan d'exécution. En présence d'une requête présentant les caractéristiques d'une chaîne de jointure, c'est la forme de référence, dont on ne devrait dévier que dans des cas explicables par la présence d'index complémentaires, de tables très petites, etc.

Ce plan (le sous-plan de la figure [Le plan d'exécution en l'absence d'index](#)) fournit un tuple, et autant d'*adresses* de tuples qu'il y a de jointures et donc d'accès aux index. Il faut ensuite ajouter autant d'opérateurs DirectAccess, ainsi que les opérateurs de sélection nécessaires (ici sur le nom du pays). Essayez par exemple, à titre d'exercice, de compléter le plan de la figure [Le plan d'exécution en l'absence d'index](#) pour qu'il corresponde complètement à la requête.

6.3 S3 : illustration avec ORACLE

Supports complémentaires :

- Diapositives : l'optimiseur d'Oracle
- Vidéo de présentation (à venir)

Cette section présente l'application concrète des concepts, structures et algorithmes présentés dans ce qui précède avec le SGBD ORACLE. Ce système est un bon exemple d'un optimiseur sophistiqué s'appuyant sur des structures d'index et des algorithmes d'évaluation complets. Tous les algorithmes de jointure décrits dans ce cours (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` notamment) 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).

6.3.1 Paramètres et statistiques

L'optimiseur s'appuie sur des paramètres divers et sur des statistiques. Parmi les plus paramètres les plus intéressants, citons :

- OPTIMIZER_MODE : permet 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 total ALL_ROWS.
- SORT_AREA_SIZE indique la taille de la mémoire affectée à l'opérateur de tri.
- HASH_AREA_SIZE indique la taille de la mémoire affectée à l'opérateur de hachage.
- HASH_JOIN_ENABLED indique que l'optimiseur considère les jointures par hachage.

L'administrateur de la base est responsable de la tenue à jour des statistiques. Pour analyser une table on utilise la commande `analyze table` qui produit 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 des informations statistiques dans les vues `dba_tables`, `all_tables`, `user_tables`. Par exemple :

- NUM_ROWS, le nombre de lignes.
- BLOCKS, le nombre de blocs.
- CHAIN_CNT, le nombre de blocs chaînés.
- AVG_ROW_LEN, la taille moyenne d'une ligne.

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;
```

Les informations statistiques sont placées 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 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`.

6.3.2 Plans d'exécution ORACLE

Nous en arrivons maintenant à la présentation des plans d'exécution d'ORACLE, tels qu'ils sont donnés par l'utilitaire `explain`. Ces plans ont classiquement la forme d'arbres en profondeur à gauche (voir la section précédente), chaque nœud étant un opérateur, les nœuds-feuille représentant les accès aux structures de la base, tables, index, *cluster*, etc.

Le vocabulaire de l'optimiseur pour désigner les opérateurs est un peu différent de celui utilisé jusqu'ici dans ce chapitre. La liste ci-dessous donne les principaux, en commençant par les chemins d'accès, puis les algorithmes de jointure, et enfin des opérations diverses de manipulation d'enregistrements.

- FULL TABLE SCAN, notre opérateur `FullScan`.
- ACCESS BY ROWID, notre opérateur `DirectAccess`.
- INDEX SCAN, notre opérateur `IndexScan`.
- NESTED LOOP, notre opérateur `INL` 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).
- select, opération de projection (et oui ...).

Voici un petit échantillon de requêtes sur notre base en donnant à chaque fois le plan d'exécution choisi par ORACLE. Les plans sont obtenus en préfixant la requête à analyser par `explain plan` accompagné de l'identifiant à donner au plan d'exécution. La description du plan d'exécution est alors stockée dans une table utilitaire et le plan peut être affiché de différentes manières. Nous donnons la représentation la plus courante, dans laquelle l'arborescence est codée par l'indentation des lignes.

La première requête est une sélection sur un attribut non indexé.

```
explain plan
set statement_id='SelSansInd' for
select *
from Film
where titre = 'Vertigo'
```

On obtient le plan d'exécution nommé `SelSansInd` dont l'affichage est donné ci-dessous.

```
0 SELECT STATEMENT
  1 TABLE ACCESS FULL FILM
```

ORACLE 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 [Plan ORACLE pour une requête sans index](#)) avec les conventions utilisées jusqu'à présent.

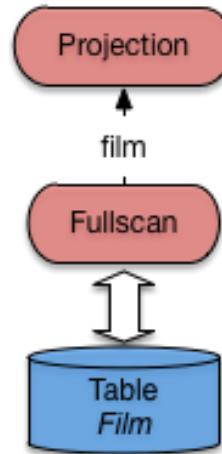


Fig. 6.7 – Plan ORACLE pour une requête sans index

Le plan est trivial. 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`.

```
explain plan
set statement_id='SelInd' for
select *
```

```
from Film
where id=21;
```

Le plan d'exécution obtenu est :

```
0 SELECT STATEMENT
1 TABLE ACCESS BY ROWID FILM
2 INDEX UNIQUE SCAN IDX-FILM-ID
```

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 *Plan ORACLE pour une requête avec index*).

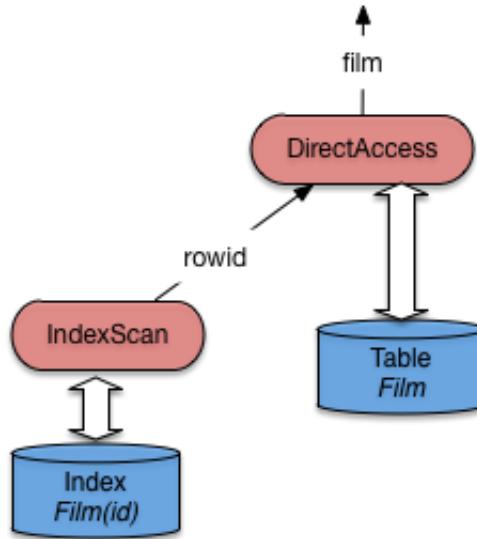


Fig. 6.8 – Plan ORACLE pour une requête avec index

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.

```
explain plan
set statement_id='JoinIndex' for
select titre, nom, prenom
from Film f, Artiste a
where f.id_realisateur = a.id;
```

Le plan d'exécution obtenu est le suivant : il s'agit d'une jointure par boucles imbriquées indexées.

```
0 SELECT STATEMENT
1 NESTED LOOPS
2 TABLE ACCESS FULL FILM
3 TABLE ACCESS BY ROWID ARTISTE
4 INDEX UNIQUE SCAN IDXARTISTE
```

Vous devriez pour décrypter ce plan est le reconnaître : c'est celui, discuté assez longuement déjà, de la jointure imbriquée indexée. Pour mémoire, il correspond à cette figure du chapitre *Opérateurs et algorithmes*.

Ré-expliquons une dernière fois. Tout d'abord la table qui n'est pas indexée sur l'attribut de jointure (ici, Film) est parcourue séquentiellement. Le nœud IndexJoin (appelé NESTED LOOPS par ORACLE) 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.

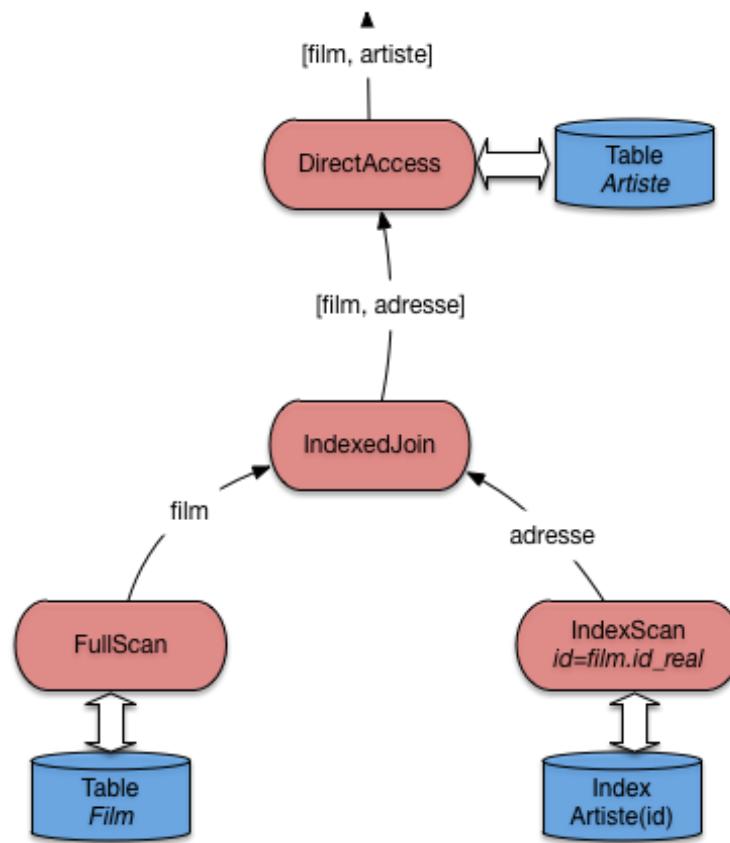


Fig. 6.9 – Plan ORACLE pour une requête avec index

On effectue alors une recherche par clé dans l'index avec la valeur `id_realisateur` provenant du film courant. La recherche renvoie un ROWID qui est utilisé pour prendre l'enregistrement complet dans la table `Artiste`.

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, et suppose qu'il existe un index sur les noms des artistes qui permet d'optimiser la recherche par nom. L'index sur la table `Rôle` est la concaténation des champs `id_acteur` et `id_film`, ce qui permet de faire une recherche par intervalle sur le préfixe constitué seulement de `id_acteur`. La requête est donnée ci-dessous.

```
explain plan
set statement_id='JoinSelIndex' for
select nom_role
from  Role r, Artiste a
where r.id_acteur = a.id
and   nom = 'Pacino';
```

Et voici le plan d'exécution.

```
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 (INDEX RANGE), et peuvent donc ramener plusieurs ROWID. Dans les deux cas on utilise en effet seulement une partie des champs définissant l'index (et cette partie constitue un préfixe, ce qui est impératif). On peut donc envisager de trouver plusieurs artistes nommés 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.

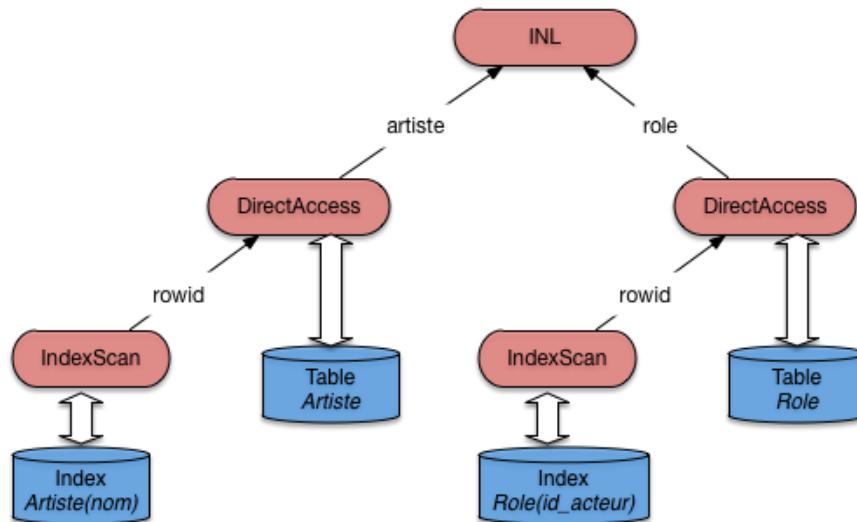


Fig. 6.10 – Plan ORACLE pour une jointure et sélection avec index

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.

```
explain plan set
statement_id='JoinSansIndex' for
select nom, prenom
from Film f, Artiste a
where f.annee = a.annee_naissance
and titre = 'Vertigo';
```

Comme il n'existe pas d'index sur ces champs, ORACLE applique un algorithme de tri-fusion, et on obtient le plan d'exécution suivant.

```
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 *Plan ORACLE pour une jointure sans index* 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.

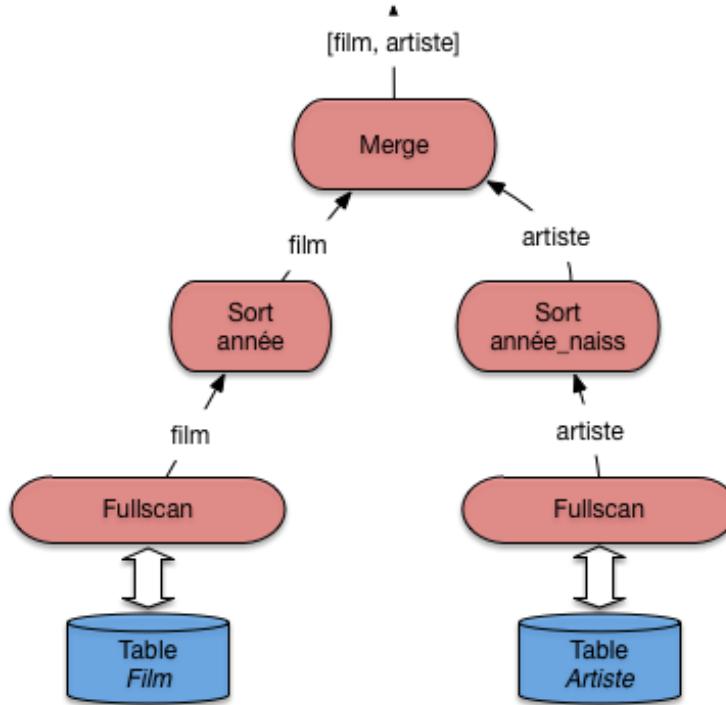


Fig. 6.11 – Plan ORACLE pour une jointure sans index

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 boucles imbriquées.

Outre l'absence d'index, 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.

Transactions

Quand on développe un programme P accédant à une base de données, on effectue en général plus ou plus explicitement deux hypothèses :

- P s'exécutera *indépendamment* de tout autre programme ou utilisateur ;
- l'exécution de P se déroulera toujours intégralement.

Il est clair que ces deux hypothèses ne se vérifient pas toujours. D'une part les bases de données constituent des ressources accessibles *simultanément* à plusieurs utilisateurs qui peuvent y rechercher, créer, modifier ou détruire des informations : les accès simultanés à une même ressource sont dits *concurrents*, et l'absence de contrôle de cette concurrence peut entraîner de graves problèmes de cohérence dus aux interactions des opérations effectuées par les différents utilisateurs. D'autre part on peut envisager beaucoup de raisons pour qu'un programme ne s'exécute pas jusqu'à son terme. Citons par exemple :

- l'arrêt du serveur de données ;
- une erreur de programmation ;
- la violation d'une contrainte amenant le système à rejeter les opérations demandées ;
- une annulation décidée par l'utilisateur.

Une interruption de l'exécution peut laisser la base dans un état transitoire *incohérent*, ce qui nécessite une opération de réparation consistant à ramener la base au dernier état cohérent connu avant l'interruption. Les SGBD relationnels assurent, par des mécanismes complexes, un partage concurrent des données et une gestion des interruptions qui permettent d'assurer à l'utilisateur que les deux hypothèses adoptées intuitivement sont satisfaites, à savoir :

- son programme se comporte, au moment où il s'exécute, *comme s'il* était seul à accéder à la base de données ;
- en cas d'interruption intempestive, les mises à jour effectuées depuis le dernier état cohérent seront annulées par le système.

On désigne respectivement par les termes de *contrôle de concurrence* et de *reprise sur panne* l'ensemble des techniques assurant ce comportement. En théorie le programmeur peut s'appuyer sur ces techniques, intégrées au système, et n'a donc pas à se soucier des interactions avec les autres utilisateurs. En pratique les choses ne sont pas si simples, et le contrôle de concurrence a pour contreparties certaines conséquences qu'il est souvent important de prendre en compte dans l'écriture des applications. En voici la liste, chacune étant développée dans le reste de ce chapitre :

- **Définition des points de sauvegardes.** La reprise sur panne garantit le retour au dernier état cohérent de la base précédent l'interruption, mais c'est au programmeur de définir ces points de cohérence (ou *points de sauvegarde*) dans le code des programmes.
- **Blocages des autres utilisateurs.** Le contrôle de concurrence s'appuie sur le verrouillage de certaines ressources (tables blocs, n-uplets), ce qui peut bloquer temporairement d'autres utilisateurs. Dans certains cas des *interblocages* peuvent même apparaître, amenant le système à rejeter l'exécution d'un des programmes en cause.
- **Choix d'un niveau d'isolation.** Une isolation totale des programmes garantit la cohérence, mais entraîne une dégradation des performances due aux verrouillages et aux contrôles appliqués par le SGBD. Or, dans beaucoup de cas, le verrouillage/contrôle est trop strict et place en attente des programmes dont l'exécution ne met pas en danger la cohérence de la base. Le programmeur peut alors choisir d'obtenir plus de concurrence (autrement dit, plus de *fluidité* dans les exécutions concurrentes), en demandant au système un niveau d'isolation moins strict, et en prenant éventuellement lui-même en charge le verrouillage des ressources critiques.

Ce chapitre est consacré à la concurrence d'accès, vue par le programmeur d'application. Il ne traite pas, ou très superficiellement, des algorithmes implantés par les SGBD. L'objectif est de prendre conscience des principales techniques nécessaires à la préservation de la cohérence dans un système multi-utilisateurs, et d'évaluer leur impact en pratique sur la réalisation d'applications bases de données. La gestion de la concurrence, du point de vue de l'utilisateur, se ramène en fait à la recherche du bon compromis entre deux solutions extrêmes :

- une cohérence maximale impliquant un risque d'interblocage relativement élevé ;
- ou une fluidité concurrentielle totale au prix de risques importants pour l'intégrité de la base.

Ce compromis dépend de l'application et de son contexte (niveau de risque acceptable vs niveau de performance souhaité) et relève donc du choix du concepteur de l'application. Mais pour que ce choix existe, et puisse être fait de manière éclairée, encore faut-il être conscient des risques et des conséquences d'une concurrence mal gérée. Ces conséquences sont insidieuses, souvent erratiques, et il est bien difficile d'imputer au défaut de concurrence des comportements que l'on a bien du mal à interpréter. Tout ce qui suit vise à vous éviter ce fort désagrément.

Le chapitre débute par une définition de la notion de *transaction*, et montre ensuite, sur différents exemples, les problèmes qui peuvent survenir. Pour finir nous présentons les niveaux d'isolation définis par la norme SQL.

7.1 S1 : Transactions

Supports complémentaires :

- Diapositives : la notion de transaction
 - Fichier de commandes pour tester les transactions sous MySQL
-

Une *transaction* est une séquence d'opérations de lecture ou de mise à jour sur une base de données, se terminant par l'une des deux instructions suivantes :

- `commit`, indiquant la validation de toutes les opérations effectuées par la transaction ;
- `rollback` indiquant l'annulation de toutes les opérations effectuées par la transaction.

Une transaction constitue donc, pour le SGBD, une unité d'exécution. Toutes les opérations de la transaction doivent être validées ou annulées solidairement.

7.1.1 Notions de base

On utilise toujours le terme de transaction, au sens défini ci-dessus, de préférence à "programme", "procédure" ou "fonction", termes à la fois inappropriés et quelque peu ambigus. "Programme" peut en effet désigner, selon le contexte, la spécification avec un langage de programmation, ou l'exécution sous la forme d'un processus client communiquant avec le (programme serveur du) SGBD. C'est toujours la seconde acception qui s'applique pour le contrôle de concurrence. De plus, l'exécution d'un programme (un *processus*) consiste en une suite d'ordres SQL adressés au SGBD, cette suite pouvant être découpée en une ou plusieurs transactions en fonction des ordres `commit` ou `rollback` qui s'y trouvent. La première transaction débute avec le premier ordre SQL exécuté ; toutes les autres débutent après le `commit` ou le `rollback` de la transaction précédente.

Note : Il est aussi possible d'indiquer explicitement le début d'une transaction avec la commande `START TRANSACTION`.

Dans tout ce chapitre nous allons prendre l'exemple de transactions consistant à réserver des places de spectacle pour un client. On suppose que la base contient les deux tables suivantes :

```
create table Client (id_client INT NOT NULL,
                     nb_places_reservees INT NOT NULL,
                     solde INT NOT NULL,
                     primary key (id_client));
create table Spectacle (id_spectacle INT NOT NULL,
                       nb_places_offertes INT NOT NULL,
```

```
nb_places_libres INT NOT NULL,
tarif DECIMAL(10,2) NOT NULL,
primary key (id_spectacle));
```

Chaque transaction s'effectue pour un client, un spectacle et un nombre de places à réserver. Elle consiste à vérifier qu'il reste suffisamment de places libres. Si c'est le cas elle augmente le nombre de places réservées par le client, et elle diminue le nombre de places libres pour le spectacle. On peut la coder en n'importe quel langage. Voici, pour être concret (et concis), la version PL/SQL.

```
/* Un programme de réservation */

create or replace procedure Reservation (v_id_client INT,
                                         v_id_spectacle INT,
                                         nb_places INT) AS
-- Déclaration des variables
v_client Client%ROWTYPE;
v_spectacle Spectacle%ROWTYPE;
v_places_libres INT;
v_places_reservees INT;
BEGIN
-- On recherche le spectacle
SELECT * INTO v_spectacle
FROM Spectacle WHERE id_spectacle=v_id_spectacle;

-- S'il reste assez de places: on effectue la réservation
IF (v_spectacle.nb_places_libres >= nb_places)
THEN
-- On recherche le client
SELECT * INTO v_client FROM Client WHERE id_client=v_id_client;

-- Calcul du transfert
v_places_libres := v_spectacle.nb_places_libres - nb_places;
v_places_reservees := v_client.nb_places_reservees + nb_places;

-- On diminue le nombre de places libres
UPDATE Spectacle SET nb_places_libres = v_places_libres
WHERE id_spectacle=v_id_spectacle;

-- On augmente le nombre de places réservées par le client
UPDATE Client SET nb_places_reservees=v_places_reservees
WHERE id_client = v_id_client;

-- Validation
commit;
ELSE
rollback;
END IF;
END;
/
```

Chaque *exécution* de ce code correspondra à une transaction. La première remarque, essentielle pour l'étude et la compréhension du contrôle de concurrence, est que l'exécution d'une procédure de ce type correspond à des échanges entre deux processus distincts : le processus *client* qui exécute la procédure, et le processus *serveur* du SGBD qui se charge de satisfaire les requêtes SQL. On prend toujours l'hypothèse que les zones mémoires des deux processus sont distinctes et étanches. Autrement dit le processus client ne peut accéder aux données que par l'intermédiaire du serveur, et le processus serveur, de son côté, est totalement ignorant de l'utilisation des données transmises au processus client (figure *Le processus client et le processus serveur pendant une transaction*).

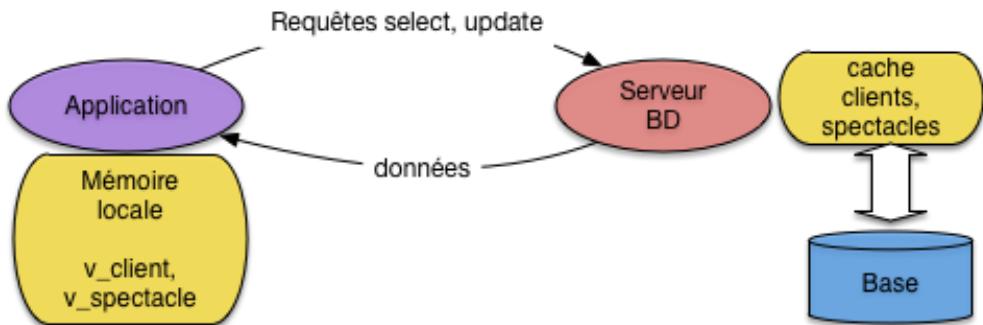


Fig. 7.1 – Le processus client et le processus serveur pendant une transaction

Il s'ensuit que non seulement le langage utilisé pour coder les transactions est totalement indifférent, mais que les variables, les interactions avec un utilisateur ou les structures de programmation (tests et boucles) du processus client sont transparentes pour le programme serveur. Ce dernier ne dispose que des instructions qui lui sont explicitement destinées, autrement dit les ordres de lecture ou d'écriture, les *commit* et les *rollback*.

D'autre part, les remarques suivantes, assez triviales, méritent cependant d'être mentionnées :

- deux exécutions de la procédure ci-dessus peuvent entraîner deux transactions différentes, dans le cas par exemple où le test effectué sur le nombre de places libres est positif pour l'une est négatif pour l'autre ;
- un processus peut exécuter répétitivement une procédure –par exemple pour effectuer plusieurs réservations– ce qui déclenche une *séquence de transactions* ;
- deux processus distincts peuvent exécuter indépendamment la même procédure, avec des paramètres qui peuvent être identiques ou non.

On fait toujours l'hypothèse que deux processus ne communiquent jamais entre eux. *En résumé, une transaction est une séquence d'instructions de lecture ou de mise à jour transmise par un processus client au serveur du SGBD.*

7.1.2 Exécutions concurrentes

Pour chaque processus il ne peut y avoir qu'une seule transaction en cours à un moment donné, mais plusieurs processus peuvent effectuer simultanément des transactions. Si elles manipulent les *mêmes* données, on peut aboutir à un entrelacement des lectures et écritures par le serveur, potentiellement générateur d'anomalies (figure *Exécution concurrentes engendrant un entrelacement*).

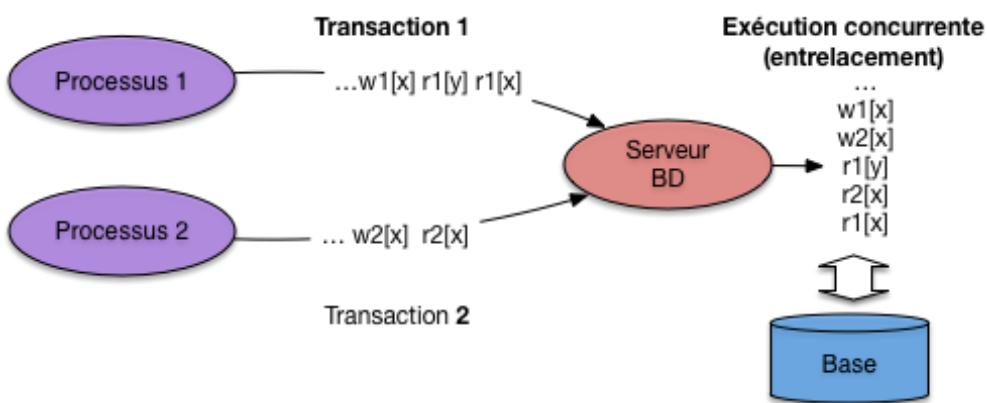


Fig. 7.2 – Exécution concurrentes engendrant un entrelacement

On suppose que chaque processus est identifié par un numéro unique qui est soumis au SGBD avec chaque ordre SQL

effectué par ce processus. Voici donc comment on représentera une transaction du processus numéro 1 exécutant la procédure de réservation.

$$read_1(s); read_1(c); write_1(s); write_1(c); C_1$$

Les symboles c et s désignent les tuples –ici un spectacle s et un client c – lus ou mis à jour par les opérations, tandis que le symbole C désigne un `commit` (on utilisera bien entendu R pour le `rollback`). Dans tout ce chapitre on supposera, sauf exception explicitement mentionnée, que l’unité d’accès à la base est le tuple (une ligne dans une table), et que tout verrouillage s’applique à ce niveau.

Voici une autre transaction, effectuée par le processus numéro 2, pour la même procédure.

$$read_2(s');$$

On a donc lu le spectacle s' , et constaté qu'il n'y a plus assez de places libres. Enfin le dernier exemple est celui d'une réservation effectuée par un troisième processus.

$$read_3(s); read_3(c'); write_3(s); write_3(c); C_3$$

Le client c réserve donc ici des places pour le spectacle s . Les trois processus peuvent s’exécuter au même moment, ce qui revient à soumettre à peu près simultanément les opérations au SGBD. Ce dernier pourrait choisir d’effectuer les transactions *en série*, en commençant par exemple par le processus 1, puis en passant au processus 2, enfin au processus 3. Cette stratégie a l’avantage de garantir de manière triviale la vérification de l’hypothèse d’isolation des exécutions, mais elle est potentiellement très pénalisante puisqu’une longue transaction pourrait mettre en attente pour un temps indéterminé de nombreuses petites transactions. C’est d’autant plus injustifié que, le plus souvent, l’entrelacement des opérations est sans danger, et qu’il est possible de contrôler les cas où il pourrait poser des problèmes. Tous les SGBD autorisent donc des *exécutions concurrentes* dans lesquelles les opérations s’effectuent alternativement pour des processus différents. Voici un exemple d’exécution concurrente pour les trois transactions précédentes, dans lequel on a abrégé *read* et *write* respectivement par r et w .

$$r_1(s); r_3(s); r_1(c); r_2(s'); r_3(c'); w_3(s); w_1(s); w_1(c); w_3(c); C_1; C_3$$

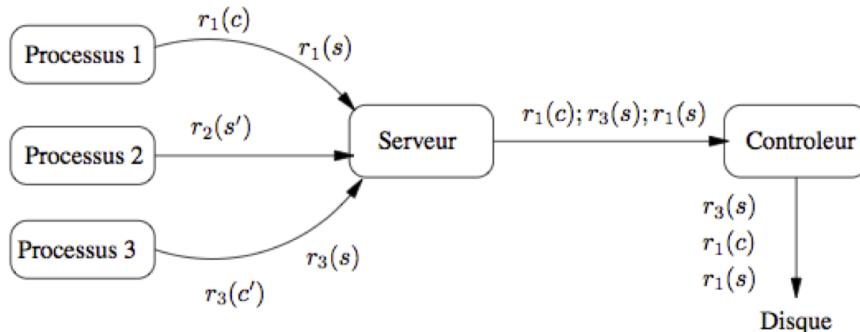


Fig. 7.3 – Processus soumettant des transactions

Dans un premier temps on peut supposer que l’ordre des opérations dans une exécution concurrente est l’ordre de transmission de ces opérations au système. Comme nous allons le voir sur plusieurs exemples, cette absence de contrôle mène à de nombreuses anomalies qu’il faut absolument éviter. Le SGBD (ou, plus précisément, le module chargé du contrôle de concurrence) effectue donc un ré-ordonnancement si cela s’impose. Cependant, l’entrelacement des opérations ou leur ré-ordonnancement ne signifie *en aucun cas* que l’ordre des opérations internes à une transaction $T : sub : i$ peut être modifié. En “effaçant” d’une exécution concurrente toutes les opérations pour les transactions

$T_j, i \neq j$ on doit retrouver exactement les opérations de T_i , dans l'ordre où elles ont été soumises au système. Ce dernier ne change *jamais* cet ordre car cela reviendrait à transformer le programme en cours d'exécution.

La figure *Processus soumettant des transactions* montre une première ébauche des composants intervenant dans le contrôle de concurrence. On y retrouve les trois processus précédents, chacun soumettant des instructions au serveur. Le processus 1 soumet par exemple $r_1(s)$, puis $r_1(c)$. Le serveur transmet les instructions, dans l'ordre d'arrivée (ici, d'abord $r_1(s)$, puis $r_3(s)$, puis $r_1(c)$), à un module spécialisé, le *contrôleur* qui, lui, peut réordonner l'exécution s'il estime que la cohérence en dépend. Sur l'exemple de la figure *Processus soumettant des transactions*, le contrôleur exécute, dans l'ordre $r_1(s)$, $r_1(c)$ puis $r_3(s)$.

7.1.3 Propriétés ACID des transactions

Les SGBD garantissent que l'exécution des transactions satisfait un ensemble de bonnes propriétés que l'on résume commodément par l'acronyme ACID (Atomicité, Cohérence, Isolation, Durabilité).

Isolation

L'isolation est la propriété qui garantit que l'exécution d'une transaction *semble* totalement indépendante des autres transactions. Le terme “semble” est bien entendu relatif au fait que, comme nous l'avons vu ci-dessus, une transaction s'exécute en fait en concurrence avec d'autres. Du point de vue de l'utilisateur, tout se passe donc comme si son programme, pendant la période de temps où il accède au SGBD, était seul à disposer des ressources du serveur de données.

Le niveau d'isolation totale, telle que défini ci-dessus, est dit *sérialisable* puisqu'il est équivalent du point de vue du résultat obtenu, à une exécution *en série* des transactions. C'est une propriété forte, dont l'inconvénient est d'impliquer un contrôle strict du SGBD qui risque de pénaliser fortement les autres utilisateurs. Les systèmes proposent en fait plusieurs niveaux d'isolation dont chacun représente un compromis entre la sérialisabilité, totalement saine mais pénalisante, et une isolation partielle entraînant moins de blocages mais plus de risques d'interactions perturbatrices.

Le choix du bon niveau d'isolation, pour une transaction donnée, est de la responsabilité du programmeur et implique une bonne compréhension des dangers courus et des options proposées par les SGBD. Le présent chapitre est essentiellement consacré à donner les informations nécessaires à ce choix.

Garantie de la commande `commit` (durabilité)

L'exécution d'un `commit` rend permanentes toutes les mises à jour de la base effectuées durant la transaction. Le système garantit que toute interruption du système survenant après le `commit` ne remettra pas en cause ces mises à jour. Cela signifie également que tout `commit` d'une transaction T rend impossible l'annulation de cette même transaction avec `rollback`. Les anciennes données sont perdues, et il n'est pas possible de revenir en arrière.

Le `commit` a également pour effet de lever tous les verrous mis en place durant la transaction pour prévenir les interactions avec d'autres transactions. Un des effets du `commit` est donc de “libérer” les éventuelles ressources bloquées par la transaction validée. Une bonne pratique, quand la nature de la transaction le permet, est donc d'effectuer les opérations potentiellement bloquantes le plus tard possible, juste avant le `commit` ce qui diminue d'autant la période pendant laquelle les données en concurrence sont verrouillées.

Garantie de la commande `rollback` (atomicité)

Le `rollback` annule toutes les modifications de la base effectuées pendant la transaction. Il relâche également tous les verrous posés sur les données pendant la transaction par le système, et libère donc les éventuels autres processus en attente de ces données.

Un `rollback` peut être déclenché explicitement par l'utilisateur, ou effectué par le système au moment d'une reprise sur panne ou de tout autre problème empêchant la poursuite normale de la transaction. Dans tout les cas l'état des

données modifiées par la transaction revient, après le `rollback`, à ce qu'il était au début de la transaction. Cette commande garantit donc l'*atomicité* des transactions, puisqu'une transaction est soit effectuée totalement (donc jusqu'au `commit` qui la conclut) soit annulée totalement (par un `rollback` du système ou de l'utilisateur).

L'annulation par `rollback` rend évidemment impossible toute validation de la transaction : les mises à jour sont perdues et doivent être resoumises.

Cohérence des transactions

Le maintien de la cohérence peut relever aussi bien du système que de l'utilisateur selon l'interprétation du concept de "cohérence".

Pour le système, la cohérence d'une base est définie par les contraintes associées au schéma. Ces contraintes sont notamment :

- les contraintes de clé primaire (clause `primary key`);
- l'intégrité référentielle (clause `foreign key`);
- les contraintes `check`;
- les contraintes implantées par *triggers*.

Toute violation de ces contraintes entraîne non seulement le rejet de la commande SQL fautive, mais également un `rollback` automatique puisqu'il est hors de question de laisser un programme s'exécuter seulement partiellement.

Mais la cohérence désigne également un état de la base considéré comme satisfaisant pour l'application, sans que cet état puisse être toujours spécifié par des contraintes SQL. Dans le cas par exemple de notre programme de réservation, la base est cohérente quand :

- le nombre de places prises pour un spectacle est le même que la somme des places réservées pour ce spectacle par les clients ;
- le solde de chaque client est supérieur à 0.

Il n'est pas facile d'exprimer cette contrainte avec les commandes DDL de SQL, mais on peut s'assurer qu'elle est respectée en écrivant soigneusement les procédures de mises à jour pour qu'elle tirent parti des propriétés ACID du système transactionnel.

Reprendons notre procédure de réservation, en supposant que la base est initialement dans un état cohérent (au sens donné ci-dessus de l'équilibre entre le nombre de places prises et le nombres de places réservées). Les propriétés d'*atomicité* (A) et de *durabilité* (D) garantissent que :

- la transaction s'effectue totalement, valide les deux opérations de mise à jour qui garantissent l'équilibre, et laisse donc après le `commit` la base dans un état cohérent ;
- la transaction est interrompue pour une raison quelconque, et la base revient alors à l'état initial, cohérent.

De plus toutes les contraintes définies dans le schéma sont respectées si la transaction arrive à terme (sinon le système l'annule).

Tout se passe bien parce que le programmeur a placé le `commit` au bon endroit. Imaginons maintenant qu'un `commit` soit introduit après le premier `UPDATE`. Si le second `UPDATE` soulève une erreur, le système effectue un `rollback` jusqu'au `commit` qui précède, et laisse donc la base dans un état incohérent –déséquilibré entre les places prises et les places réservées– du point de vue de l'application.

Dans ce cas l'erreur vient du programmeur qui a défini deux transactions au lieu d'une, et entraîné une validation à un moment où la base est dans un état intermédiaire. La leçon est simple : *tous les commit et rollback doivent être placés de manière à s'exécuter au moment où la base est dans un état cohérent*. Il faut toujours se souvenir qu'un `commit` ou un `rollback` marque la fin d'une transaction, et définit donc l'ensemble des opérations qui doivent s'exécuter solidiairement (ou "atomiquement").

Un défaut de cohérence peut enfin résulter d'un mauvais entrelacement des opérations concurrentes de deux transactions, dû à un niveau d'isolation insuffisant. Cet aspect sera illustré dans la prochaine section consacrée aux conséquences d'une absence de contrôle.

7.1.4 Un exemple de concurrence sous ORACLE

Pour conclure cette première présentation par un exemple concret, voici deux sessions effectuées en concurrence sous ORACLE. Ces sessions s'effectuent avec l'utilitaire SQLPLUS qui permet d'entrer directement des requêtes sur la base comprenant les tables *Client* et *Spectacle* décrites en début de chapitre. Les opérations effectuées consistent à réserver, pour le même spectacle, 5 places pour la session 1, et 7 places pour la session 2.

Voici les premières requêtes effectuées par la session 1.

```
Session1>SELECT * FROM Client;

ID_CLIENT NB_PLACES_RESERVEES      SOLDE
-----  -----
1           3                   2000

Session1>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARIF
-----  -----  -----
1           250                  200                   10
```

On a donc un client et un spectacle. La session 1 augmente maintenant le nombre de places réservées.

```
Session1>UPDATE Client SET nb_places_reservees = nb_places_reservees + 5
      WHERE id_client=1;

1 row updated.

Session1>SELECT * FROM Client;

ID_CLIENT NB_PLACES_RESERVEES      SOLDE
-----  -----
1           8                   2000

Session1>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARIF
-----  -----  -----
1           250                  200                   10
```

Après l'ordre UPDATE, si on regarde le contenu des tables *Client* et *Spectacle*, on voit bien l'effet des mises à jour. Notez que la base est ici dans un état instable puisqu'on n'a pas encore diminué le nombre de places libres. Voyons maintenant les requêtes de lecture pour la session 2.

```
Session2>SELECT * FROM Client;

ID_CLIENT NB_PLACES_RESERVEES      SOLDE
-----  -----
1           3                   2000

Session2>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARIF
-----  -----  -----
1           250                  200                   10
```

Pour la session 2, la base est dans l'état initial. L'isolation implique que les mises à jour effectuées par la session 1 sont invisibles puisqu'elles ne sont pas encore validées. Maintenant la session 2 tente d'effectuer la mise à jour du client.

```
Session2>UPDATE Client SET nb_places_reservees = nb_places_reservees + 7
      WHERE id_client=1;
```

La transaction est mise en attente car, appliquant des techniques de verrouillage qui seront décrites plus loin, ORACLE a réservé le tuple de la table *Client* pour la session 1. Seule la session 1 peut maintenant progresser. Voici la fin de la transaction pour cette session.

```
Session1>UPDATE Spectacle SET nb_places_libres = nb_places_libres - 5
      WHERE id_spectacle=1;

1 row updated.

Session1>commit;
```

Après le `commit`, la session 2 est libérée,

```
Session2>UPDATE Client SET nb_places_reservees = nb_places_reservees + 7
      WHERE id_client=1;

1 row updated.
Session2>
Session2>SELECT * FROM Client;

ID_CLIENT NB_PLACES_RESERVEES      SOLDE
-----  -----
1           15                  2000

Session2>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTEES  NB_PLACES_LIBRES      TARIF
-----  -----
1             250                  195                 10
```

Une sélection montre que les mises à jour de la session 1 sont maintenant visibles, puisque le `commit` les a validé définitivement. De plus on voit également la mise à jour de la session 2. Notez que pour l'instant la base est dans un état incohérent puisque 12 places sont réservées par le client, alors que le nombre de places libres a diminué de 5. La seconde session doit décider, soit d'effectuer la mise à jour de *Spectacle*, soit d'effectuer un `rollback`. En revanche il est absolument exclu de demander un `commit` à ce stade, même si on envisage de mettre à jour *Spectacle* ultérieurement. Si cette dernière mise à jour échouait, ORACLE ramènerait la base à l'état – incohérent – du dernier `commit`,

Voici ce qui se passe si on effectue le `rollback`.

```
Session2>rollback;

rollback complete.

Session2>SELECT * FROM Client;

ID_CLIENT NB_PLACES_RESERVEES      SOLDE
-----  -----
1           8                  2000

Session2>SELECT * FROM Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTEES  NB_PLACES_LIBRES      TARIF
-----  -----
1             250                  195                 10
```

Les mises à jour de la session 2 sont annulées : la base se retrouve dans l'état connu à l'issue de la session 1.

Ce court exemple montre les principales conséquences “visibles” du contrôle de concurrence effectué par un SGBD :

- chaque processus/session dispose d'une vision des données en phase avec les opérations qu'il vient d'effectuer ;
- les données modifiées mais non validées par un processus ne sont pas visibles pour les autres ;
- les accès concurrents à une même ressource peuvent amener le système à mettre en attente certains processus.

D'autre part le comportement du contrôleur de concurrence peut varier en fonction du niveau d'isolation choisi qui est, dans l'exemple ci-dessus, celui adopté par défaut dans ORACLE (`read committed`). Le choix (ou l'acceptation par défaut) d'un niveau d'isolation inapproprié peut entraîner diverses anomalies que nous allons maintenant étudier.

7.2 S2 : effets indésirables des transactions concurrentes

Supports complémentaires :

- Diapositives : anomalies transactionnelles
-

Pour illustrer les problèmes potentiels en cas d'absence de contrôle de concurrence, ou de techniques insuffisantes, on va considérer un modèle d'exécution très simplifié, dans lequel il n'existe qu'une seule version de chaque tuple (stocké par exemple dans un fichier séquentiel). Chaque instruction est effectuée par le système, dès qu'elle est reçue, de la manière suivante :

- quand il s'agit d'une instruction de lecture, on lit le tuple dans le fichier et on le transmet au processus ;
- quand il s'agit d'une instruction de mise à jour, on écrit directement le tuple dans le fichier en écrasant la version précédente.

Les problèmes consécutifs à ce mécanisme simpliste peuvent être classés en deux catégories : défauts d'isolation menant à des incohérences *–défauts de sérialisabilité–*, et difficultés dues à une mauvaise prise en compte des `commit` et `rollback`, ou *défauts de recouvrabilité*. Les exemples qui suivent ne couvrent pas l'exhaustivité des situations d'anomalies, mais illustrent les principaux types de problèmes.

7.2.1 Défauts de sérialisabilité

Considérons pour commencer que le système n'assure aucune isolation, aucun verrouillage, et ne connaît ni le `commit` ni le `rollback`. Même en supposant que toutes les exécutions concurrentes s'exécutent intégralement sans jamais rencontrer de panne, on peut trouver des situations où l'entrelacement des instructions conduit à des résultats différents de ceux obtenus par une exécution en série. De telles exécutions sont dites *non sérialisables* et aboutissent à des incohérences dans la base de données.

Les mises à jour perdues

Le problème de mise à jour perdue survient quand deux transactions lisent chacune une même donnée en vue de la modifier par la suite. Prenons à nouveau deux exécutions concurrentes du programme *Réservation*, désignées par T_1 et T_2 . Chaque exécution consiste à réserver des places pour le même spectacle, mais pour deux clients distincts c_1 et c_2 . L'ordre des opérations reçues par le serveur est le suivant :

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)w_1(s)w_1(c_1)$$

Donc on effectue d'abord les lectures pour T_1 , puis les lectures pour T_2 enfin les écritures pour T_2 et T_1 dans cet ordre. Imaginons maintenant que l'on se trouve dans la situation suivante :

- il reste 50 places libres pour le spectacle s , c_1 et c_2 n'ont pour l'instant réservé aucune place ;
- T_1 veut réserver 5 places pour s ;
- T_2 veut réserver 2 places pour s .

Voici le résultat du déroulement imbriqué des deux exécutions $T_1(s, 5, c_1)$ et $T_2(s, 2, c_2)$, en supposant que la séquence des opérations est celle donnée ci-dessus. On se concentre pour l'instant sur les évolutions du nombre de places vides.

- T_1 lit s et c_1 et constate qu'il reste 50 places libres ;
- T_2 lit s et c_2 et constate qu'il reste 50 places libres ;
- T_2 écrit s avec nb places = $50-2=48$.
- T_2 écrit le nouveau compte de c_2 .
- T_1 écrit s avec nb places = $50-5=45$.
- T_1 écrit le nouveau compte de c_1 .

À la fin de l'exécution, on constate un problème : il reste 45 places vides sur les 50 initiales alors que 7 places ont effectivement été réservées et payées. Le problème est clairement issu d'une mauvaise imbrication des opérations de T_1 et T_2 : T_2 lit et modifie une information que T_1 a déjà lue en vue de la modifier. La figure :ref :trans_anom1 montre la superposition temporelle des deux transactions. On voit que T_1 et T_2 lisent chacun, dans leurs espaces mémoires respectifs, d'une copie de S indiquant 50 places disponibles, et que cette valeur sert au calcul du nombre de places restantes sans tenir compte de la mise à jour effectuée par l'autre transaction.

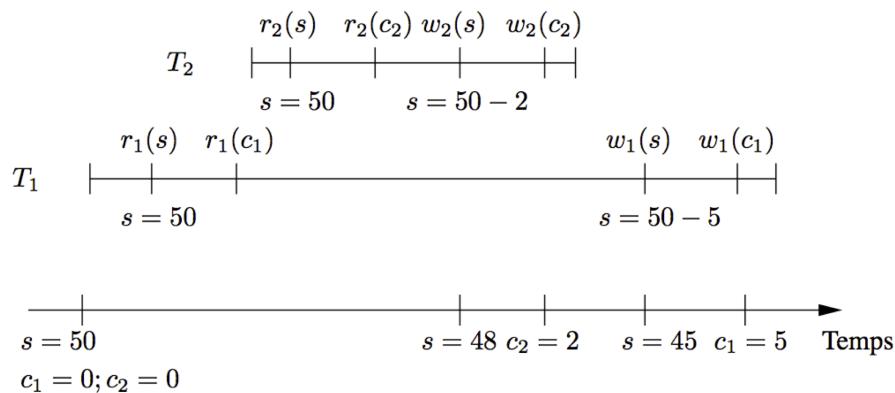


Fig. 7.4 – Exécution concurrente de T_1 et T_2

On arrive donc à une base de données incohérente alors que chaque transaction, prise isolément, est correcte, et qu'elles se sont toutes deux exécutées complètement.

Une solution radicale pour éviter le problème est d'exécuter *en série* T_1 et T_2 . Il suffit pour cela de bloquer une des deux transactions tant que l'autre n'a pas fini de s'exécuter. On obtient alors l'exécution concurrente suivante :

$$r_1(s)r_1(c)r_1(s)w_1(s)w_1(c)r_2(s)r_2(c)r_2(s)w_2(s)w_2(c)$$

On est assuré dans ce cas qu'il n'y a pas de problème car T_2 lit la donnée écrite par T_1 qui a fini de s'exécuter et ne créera donc plus d'interférence. La figure *Exécution en série de et* montre que la cohérence est obtenue ici par la lecture de s dans T_2 qui ramène la valeur 50 pour le nombre de places disponibles, ce qui revient bien à tenir compte des mises à jour de T_1 .

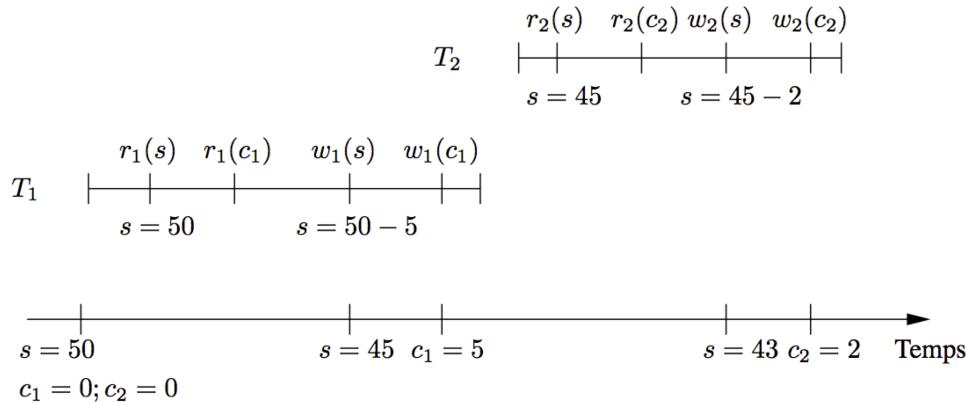
Cette solution de “concurrence zéro” est difficilement acceptable car elle revient à bloquer tous les processus sauf un. Dans un système où de très longues transactions (par exemple l'exécution d'un traitement lourd d'équilibrage de comptes) cohabitent avec de très courtes (des saisies interactives), les utilisateurs seraient extrêmement pénalisés.

Heureusement l'exécution en série est une contrainte trop forte, comme le montre l'exemple suivant.

$$r_1(s)r_1(c_1)w_1(s)r_2(s)r_2(c_2)w_2(s)w_1(c_1)w_2(c_2)$$

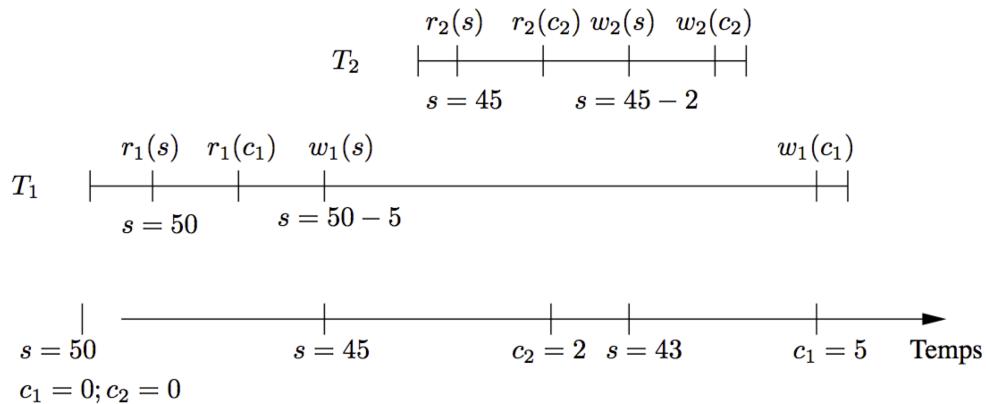
Suivons pas à pas l'exécution :

- T_1 lit s et c_1 . Nombre de places libres : 50.
- T_1 écrit s avec nb places = $50-5=45$.
- T_2 lit s . Nombre de places libres : 45.
- T_2 lit c_2 .
- T_2 écrit s avec nombre de places = $45-2=43$.


 Fig. 7.5 – Exécution en série de T_1 et T_2

- T_1 écrit le nouveau compte du client c_1 .
- T_2 écrit le nouveau compte du client c_2 .

Cette exécution est correcte : on obtient un résultat strictement semblable à celui issu d'une exécution en série. Il existe donc des exécutions imbriquées qui sont aussi correctes qu'une exécution en série et qui permettent une meilleure concurrence. Le gain, sur notre exemple, peut paraître mineur, mais il faut imaginer l'intérêt de débloquer rapidement de longues transactions qui ne rentrent en concurrence que sur une petite partie des tuples qu'elles manipulent.


 Fig. 7.6 – Exécution concurrente correcte de T_1 et T_2

On parle d'exécutions *sérialisables* pour désigner des exécutions concurrentes équivalentes à une exécution en série. Un des buts d'un système effectuant un contrôle de concurrence est d'obtenir de telles exécutions. Dans l'exemple qui précède, cela revient à mettre T_2 en attente tant que T_1 n'a pas écrit s . Nous verrons un peu plus loin par quelles techniques on peut automatiser ce genre de mécanisme.

Lectures non répétables

Voici un autre type de problème dû à l'interaction de plusieurs transactions : certaines modifications de la base peuvent devenir visibles *pendant* l'exécution d'une transaction T à cause des mises à jour effectuées *et validées* par d'autres transactions. Ces modifications peuvent rendre le résultat de l'exécution des requêtes en lecture effectuées par T *non répétables* : la première exécution d'une requête q renvoie un ensemble de tuples différent d'une seconde exécution

de q effectuée un peu plus tard, parce certain tuples ont disparu ou sont apparus dans l'intervalle (on parle de *tuples fantômes*).

Prenons le cas d'une procédure effectuant un contrôle de cohérence sur notre base de données : elle effectue tout d'abord la somme des places prises par les clients, puis elle compare cette somme au nombre de places réservées pour le spectacle. La base est cohérente si le nombre de places libres est égal au nombre de places réservées :

```

Procédure Contrôle()
Début
    Lire tous les clients et effectuer la somme des places prises
    Lire le spectacle
    SI (Somme(places prises) <> places réservées)
        Afficher ("Incohérence dans la base")
Fin
    
```

Une exécution de la procédure *Contrôle* se modélise simplement comme une séquence $r_c(c_1) \dots r_c(c_n)r_c(s)$ d'une lecture des n clients $\{c_1, \dots, c_n\}$ suivie d'une lecture de s (le spectacle). Supposons maintenant qu'on exécute cette procédure sous la forme d'une transaction T_1 , en concurrence avec une réservation $Res(c_1, s, 5)$, avec l'entrelacement suivant.

$$r_1(c_1)r_1(c_2)Res(c_2, s, 2) \dots r_1(c_n)r_1(s)$$

Le point important est que l'exécution de $Res(c_2, s, 2)$ va augmenter de 2 le nombre de places réservées par c_2 , et diminuer de 2 le nombre de places disponibles dans s . Mais la procédure T_1 a lu le spectacle s après la transaction Res , et le client c_2 avant. Seule une partie des mises à jour de Res sera donc visible, avec pour résultat la constatation d'une incohérence alors que ce n'est pas le cas.

La figure *Exécution concurrente d'un contrôle et d'une réservation* résume le déroulement de l'exécution (en supposant deux clients seulement). Au début de la session 1, la base est dans un état cohérent. On lit 5 pour le client 1, 0 pour le client 2. À ce moment-là intervient la réservation T_2 , qui met à jour le client 2 et le spectacle s . C'est cette valeur mise à jour que vient lire T_1 .

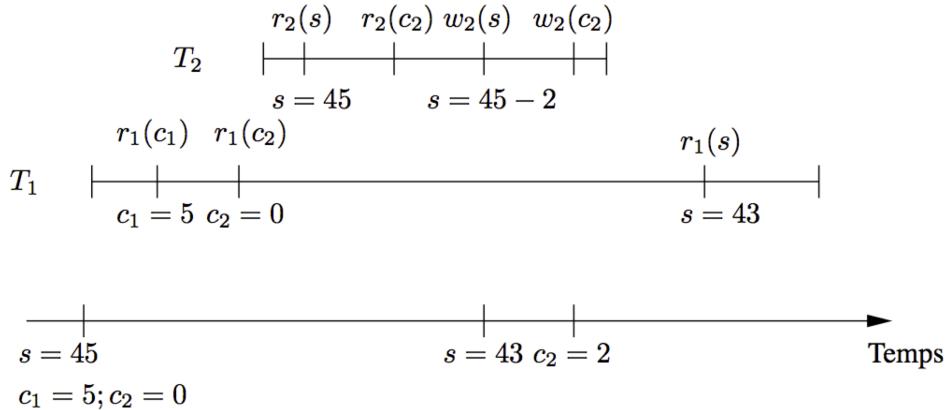


Fig. 7.7 – Exécution concurrente d'un contrôle et d'une réservation

Il faut noter que T_1 n'effectue pas de mise à jour et ne lit que des données validées. Il est clair que le problème vient du fait que la transaction T_1 accède, durant son exécution, à des versions différentes de la base et qu'un contrôle de cohérence ne peut être fiable dans ces conditions. On peut noter que cette exécution n'est pas sérialisable, puisque le résultat est clairement différent de celui obtenu par l'exécution successive de T_1 et de Res .

On désigne ce type de comportement par le terme de *lectures non répétables* (*non repeatable reads* en anglais). Si T_1 effectue deux lectures de s avant et après l'exécution de Res , elle constatera une modification.

De même toute insertion ou suppression d'un tuple dans la table Client} sera visible ou non selon que T_1 effectuera la lecture avant ou après la mise à jour concurrente. On parle alors de *tuples fantômes*.

Les lectures non répétables et tuples fantômes constituent un des effets désagréables d'une exécution concurrente. On pourrait considérer, à tort, que le risque d'avoir une telle interaction est faible. En fait l'exécution de requêtes sur une période assez longue est très fréquente dans les applications bases de données qui s'appuient sur des curseurs permettant de parcourir un résultat tuple à tuple, avec un temps de traitement de chaque tuple qui peut alonger considérablement le temps seulement consacré au parcours du résultat.

7.2.2 Défauts de recouvrabilité

Le fait de garantir une imbrication sérialisable des exécutions concurrentes serait suffisant dans l'hypothèse où tous les programmes terminent normalement en validant les mises à jour effectuées. Malheureusement ce n'est pas le cas puisqu'il arrive que l'on doive annuler les opérations d'entrées sorties effectuées par un programme. Les anomalies d'une exécution concurrente dus aux effets non contrôlés des `commit` et `rollback` constituent une seconde catégorie de problèmes qualifiés collectivement de *défauts de recouvrabilité*.

Nous allons maintenant étendre notre modèle d'exécution simplifié en introduisant les commandes `commit` et `rollback`. Attention : il ne s'agit que d'une version simplifiée de l'un des algorithmes possibles pour implanter les `commit` et `rollback` :

- le contrôleur conserve, chaque fois qu'une transaction T_i modifie un tuple t , l'image de t avant la mise à jour, dénotée t_i^{ia} ;
- quand une transaction T_i effectue un `commit`, les images avant associées à T_i sont effacées ;
- quand une transaction T_i effectue un `rollback`, toutes les images avant sont écrites dans la base pour ramener cette dernière dans l'état du début de la transaction.

Imaginons par exemple que le programme de réservation soit interrompu après avoir exécuté les instructions suivantes :

$$r_1(s)r_1(c_1)w_1(s)$$

Au moment d'effectuer $w_1(s)$, notre système a conservé l'image avant modification s_1^{ia} du spectacle s . L'interruption intervient avant le `commit`, et la situation obtenue n'est évidemment pas satisfaisante puisqu'on a diminué le nombre de places libres sans débiter le compte du client. Le `rollback` consiste ici à effectuer l'opération $w_1(s_1^{ia})$ pour réécrire l'image avant et revenir à l'état initial de la base.

L'implantation d'un tel mécanisme demande déjà un certain travail, mais cela ne suffit malheureusement toujours pas à garantir des exécutions concurrentes correctes, comme le montrent les exemples qui suivent.

Lectures sales

Revenons à nouveau à l'exécution concurrente de nos deux transactions T_1 et T_2 , en considérant maintenant l'impact des validations ou annulations par `commit` ou `rollback`. Voici un premier exemple :

$$r_1(s)r_1(c_1)w_1(s)r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2w_1(c_1)R_1$$

Le nombre de places disponibles a donc été diminué par T_1 et repris par T_2 , avant que T_1 n'annule ses réservations. On peut noter que cette exécution concurrente est sérialisable, au sens où l'ordre des opérations donne un résultat identique à une exécution en série.

Le problème vient ici du fait que T_1 est annulée *après* que la transaction T_2 a lu une information mise à jour par T_1 , manipulé cette information, effectué une écriture, et enfin validé. On parle de "lectures sales" (*dirty read* en anglais) pour désigner l'accès par une transaction à des tuples modifiés *mais non encore validés* par une autre transaction. L'exécution correcte du `rollback` est ici impossible, puisqu'on se trouve face au dilemne suivant :

- soit on écrit dans s l'image avant gérée par T_1 , s_1^{ia} , mais on écrase du coup la mise à jour de T_2 alors que ce dernier a effectué un `commit` ;
- soit on conserve le tuple s validé par T_2 , et on annule seulement la mise à jour sur c_1 , mais la base est alors incohérente.

On se trouve donc face à une exécution concurrente qui rend impossible le respect d’au moins une des deux propriétés transactionnelles requises : la durabilité (garantie du `commit`) ou l’atomicité (garantie du `rollback`). Une telle exécution est dite *non recouvrable*, et doit absolument être évitée.

La lecture sale transmet un tuple modifié et non validé par une transaction (ici T_1) à une autre transaction (ici T_2). La première transaction, celle qui a effectué la lecture sale, devient donc dépendante du choix de la seconde, qui peut valider ou annuler. Le problème est agravé irrémédiablement quand la première transaction valide avant la seconde, comme dans l’exemple précédent, ce qui rend impossible une annulation globale.

Une exécution non-recouvrable introduit un conflit insoluble entre les `commit` effectués par une transaction et les `rollback` d’une autre. On pourrait penser à interdire à une transaction T_2 ayant effectué des lectures sales d’une transaction T_1 de valider avant T_1 . On accepterait alors la situation suivante :

$$r_1(s)r_1(c_1)w_1(s)r_2(s)r_2(c_2)w_2(s)w_2(c_2)w_1(c_1)R_1$$

Ici, le `rollback` de T_1 intervient sans que T_2 n’ait validé. Il faut alors impérativement que le système effectue également un `rollback` de T_2 pour assurer la cohérence de la base : on parle d’*annulations en cascade* (noter qu’il peut y avoir plusieurs transactions à annuler).

Quoique acceptable du point de vue de la cohérence de la base, ce comportement est difficilement envisageable du point de vue de l’utilisateur qui voit ses transactions interrompues sans aucune explication liée à ses propres actions. Aucun SGBD ne pratique d’annulation en cascade. La seule solution est donc simplement d’interdire les *dirty read*. Nous verrons qu’il existe deux solutions : soit la lecture lit *l'image avant*, qui par définition est une valeur validée, soit on met en attente les lectures sur des tuples en cours de modification.

Ecriture sale

Imaginons qu’une transaction T ait modifié un tuple t , puis qu’un `rollback` intervienne. Dans ce cas, comme indiqué ci-dessus, il est nécessaire de restaurer la valeur qu’avait t *avant* le début de la transaction (“l’image avant”). Cela soulève des problèmes de concurrence illustrés par l’exemple suivant, toujours basé sur nos deux transactions T_1 et T_2 . ci-dessous.

$$r_1(s)r_1(c_1)r_2(s)w_1(s)w_1(c_1)r_2(c_2)w_2(s)R_1w_2(c_2)C_2$$

Ici il n’y a pas de lecture sale, mais une “écriture sale” (*dirty write*) car T_2 écrit s après une mise à jour T_1 sur s , et sans que T_1 ait validé. Puis T_1 annule et T_2 valide. Que se passe-t-il au moment de l’annulation de T_1 ? On doit restaurer l’image avant connue de T_1 , mais cela revient clairement à annuler la mise à jour de T_2 .

On peut sans doute envisager des techniques plus sophistiquées de gestion des `rollback`, mais le principe de remplacement par l’image avant a le mérite d’être relativement simple à mettre en place, ce qui implique l’interdiction des écritures sales.

En résumé, on peut avoir des transactions sérialisables et non recouvrables et réciproquement. Le respect des propriétés ACID des transactions impose au SGBD d’assurer :

- la sérialisabilité des transactions ;
- la recouvrabilité dite *stricte*, autrement dit sans lectures ni écritures sales.

Les SGBD s’appuient sur un ensemble de techniques assez sophistiquées dont nous allons donner un aperçu ci-dessous. Il faut noter dès maintenant que le recours à ces techniques peut être pénalisant pour les performances (ou, plus exactement, la “fluidité des exécutions). Dans certains cas –fréquents– où le programmeur sait qu’il n’existe pas de risque lié à la concurrence, on peut relâcher le niveau de contrôle effectué par le système afin d’éviter des blocages inutiles.

7.2.3 Exercices

7.3 S3 : choisir un niveau d’isolation

Supports complémentaires :

- Diapositives : niveaux d’isolation
-

Du point du programmeur d’application, l’objectif du contrôle de concurrence est de garantir la cohérence des données et d’assurer la recouvrabilité des transactions. Ces bonnes propriétés sont obtenues en choisissant un niveau d’isolation approprié qui garantit qu’aucune interaction avec un autre utilisateur ne viendra perturber le déroulement d’une transaction, empêcher son annulation ou sa validation.

Une option possible est de toujours choisir un niveau d’isolation maximal, garantissant la sérialisabilité des transactions, mais le mode `Serializable` a l’inconvénient de ralentir le débit transactionnel pour des applications qui n’ont peut-être pas besoin de contrôles aussi stricts. On peut chercher à obtenir de meilleures performances en choisissant explicitement un niveau d’isolation moins élevé, soit parce que l’on sait qu’un programme ne posera jamais de problème de concurrence, soit parce les problèmes éventuels sont considérés comme peu importants par rapport au bénéfice d’une fluidité améliorée.

On considère dans ce qui suit deux exemples. Le premier consiste en deux exécutions concurrentes du programme `Réservation`, désignées respectivement par T_1 et T_2 .

Exemple : concurrence entre mises à jour

Chaque exécution consiste à réserver des places pour le même spectacle, mais pour deux clients distincts c_1 et c_2 . L’ordre des opérations reçues par le serveur est le suivant :

Caption

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)w_1(s)w_1(c_1)$$

Au départ nous sommes dans la situation suivante :

- il reste 50 places libres pour le spectacle s , c_1 et c_2 n’ont pour l’instant réservé aucune place ;
- T_1 veut réserver 5 places pour s ;
- T_2 veut réserver 2 places pour s .

Donc on effectue d’abord les lectures pour T_1 , puis les lectures pour T_2 enfin les écritures pour T_2 et T_1 dans cet ordre. Aucun client n’a réservé de place.

Le second exemple prend le cas de la procédure effectuant un contrôle de cohérence sur notre base de données, uniquement par des lectures.

Exemple : concurrence entre lectures et mises à jour

La procédure `Contrôle` s’effectue en même temps que la procédure `Réservation` qui réserve 2 places pour le client c_2 . L’ordre des opérations reçues par le serveur est le suivant (T_1 désigne le contrôle, T_2 la réservation) :

$$r_1(c_1)r_1(c_2)r_2(s)r_2(c_2)w_2(s)w_2(c_2)r_1(s)$$

Au départ le client c_1 a réservé 5 places. Il reste donc 45 places libres pour le spectacle. La base est dans un état cohérent.

7.3.1 Les modes d’isolation SQL

La norme SQL ANSI (SQL92) définit quatre modes d’isolation correspondant à quatre compromis différents entre le degré de concurrence et le niveau d’interblocage des transactions. Ces modes d’isolation sont définis par rapport aux trois types d’anomalies que nous avons rencontrés dans les exemples qui précèdent :

- *Lectures sales* : une transaction T_1 lit un tuple mis à jour par une transaction T_2 , *avant* que cette dernière ait validé ;

- *Lectures non répétables* : une transaction T_1 accède, en lecture ou en mise à jour, à un tuple qu'elle avait déjà lu auparavant, alors que ce tuple a été modifié entre temps par une autre transaction T_2 ;
- *Tuples fantômes* : une transaction T_1 lit un tuple qui a été créé par une transaction T_2 après le début de T_1 .

Tableau 7.1 – isolation-levels

	Lectures sales	Lectures non répétables	Tuples fantômes
READ UNCOMMITTED	Possible	Possible	Possible
read committed	Impossible	Possible	Possible
repeatable read	Impossible	Impossible	Possible
serializable	Impossible	Impossible	Impossible

Il existe un mode d'isolation par défaut qui varie d'un système à l'autre, le plus courant semblant être `read committed`.

Le premier mode (`READ UNCOMMITTED`) correspond à l'absence de contrôle de concurrence. Ce mode peut convenir pour des applications non transactionnelles qui se contentent d'écrire "en vrac" dans des fichiers sans se soucier ni des interactions avec d'autres utilisateurs.

Avec le mode `read committed`, on ne peut lire que les tuples validés, mais il peut arriver que deux lectures successives donnent des résultats différents. Le résultat d'une requête est cohérent par rapport à l'état de la base *au début de la requête*. Il peut arriver que deux lectures successives donnent des résultats différents si une autre transaction a modifié les données lues, et validé ses modifications. C'est le mode par défaut dans ORACLE par exemple.

Il faut bien être conscient que ce mode ne garantit pas l'exécution sérialisable. Le SGBD garantit par défaut l'exécution correcte des `commit` et `rollback` (recouvrabilité), mais pas la sérialisabilité. L'hypothèse effectuée implicitement est que le mode `serializable` est inutile dans la plupart des cas, ce qui est sans doute justifié, et que le programmeur saura le choisir explicitement quand c'est nécessaire, ce qui en revanche est loin d'être évident.

Le mode `repeatable read` (le défaut dans MySQL/InnoDB par exemple) garantit que le résultat d'une requête est cohérent par rapport à l'état de la base *au début de la transaction*. La réexécution de la même requête donne toujours le même résultat. La sérialisabilité n'est pas assurée, et des tuples peuvent apparaître s'ils ont été insérés par d'autres transactions (les fameux "tuples fantômes").

Enfin le mode `serializable` assure les bonnes propriétés (sérialisabilité et recouvrabilité) des transactions et une isolation totale. Tout se passe alors comme si on travaillait sur une "image" de la base de données prise au début de la transaction. Bien entendu cela se fait au prix d'un risque assez élevé de blocage des autres transactions.

Le mode est choisi au début d'une transaction par la commande suivante.

```
set transaction isolation level <option>
```

Une autre option parfois disponible, même si elle ne fait pas partie de la norme SQL, est de spécifier qu'une transaction ne fera que des lectures. Dans ces conditions, on peut garantir qu'elle ne soulèvera aucun problème de concurrence et le SGBD peut s'épargner la peine de poser des verrous. La commande est :

```
SET TRANSACTION READ ONLY;
```

Il devient alors interdit d'effectuer des mises à jour jusqu'au prochain `commit` ou `rollback` : le système rejette ces instructions.

7.3.2 Verrouillage explicite

Certains systèmes permettent de poser explicitement des verrous, ce qui permet pour le programmeur averti de choisir un niveau d'isolation relativement permissif, tout en augmentant le niveau de verrouillage quand c'est nécessaire. ORACLE, PostgreSQL et MySQL proposent notamment une clause `FOR UPDATE` qui peut se placer à la fin d'une requête SQL, et dont l'effet est de réservé chaque tuple lu en vue d'une prochaine modification.

Reprendons notre programme de réservation, et réécrivons les deux premières requêtes de la manière suivante :

```
...
SELECT * INTO v_spectacle
FROM Spectacle
WHERE id_spectacle=v_id_spectacle
FOR UPDATE;
...
SELECT * INTO v_client FROM Client
WHERE id_client=v_id_client
FOR UPDATE;
...
```

On annonce donc explicitement, dans le code, que la lecture d'un tuple (le client ou le spectacle) sera suivie par la mise à jour de ce même tuple. Le système pose alors un *verrou exclusif* qui réserve l'accès au tuple, en lecture ou en mise à jour, à la transaction qui a effectué la lecture avec FOR UPDATE. Les verrous posés sont libérés au moment du commit ou du rollback.

Voici le déroulement de l'exécution pour l'exécution de l'exemple *ex-conc-trans* :

- T_1 lit s , après l'avoir verrouillé exclusivement ;
- T_1 lit c_1 , et verrouille exclusivement ;
- T_2 veut lire s , et se trouve mise en attente ;
- T_1 continue, écrit s , écrit c_1 , valide et libère les verrous ;
- T_2 est libéré et s'exécute.

On obtient l'exécution en série suivante.

$$r_1(s)r_1(c_1)w_1(s)w_1(c_1)C_1r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2$$

La déclaration, avec FOR UPDATE de l'intention de modifier un tuple revient à le réserver et donc à empêcher un entrelacement avec d'autres transactions menant soit à un rejet, soit à une annulation autoritaire du SGBD.

Les SGBDs fournissent également des commandes de verrouillage explicite. On peut réserver, en lecture ou en écriture, une table entière. Un verrouillage en lecture est *partagé* : plusieurs transactions peuvent détenir un verrou en lecture sur la même table. Un verrouillage en écriture est *exclusif* : il ne peut y avoir aucun autre verrou, partagé ou exclusif, sur la table.

Voici un exemple avec MySQL dont un des moteurs de stockage, MyISAM, ne gère pas la concurrence. Il faut donc appliquer explicitement un verrouillage si on veut obtenir des exécutions concurrentes sérialisables. En reprenant l'exemple *ex-conc-trans* avec verrouillage exclusif (WRITE), voici ce que cela donne. La session 1 verrouille (en écriture), lit le spectacle puis le client 1.

```
Session 1> LOCK TABLES Client WRITE, Spectacle WRITE;
Query OK, 0 rows affected (0,00 sec)

Session 1> SELECT * FROM Spectacle WHERE id_spectacle=1;
+-----+-----+-----+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+-----+-----+-----+
| 1 | 50 | 50 | 10.00 |
+-----+-----+-----+
1 row in set (0,00 sec)

Session 1> SELECT * FROM Client WHERE id_client=1;
+-----+-----+
| id_client | nb_places_reservees | solde |
+-----+-----+
| 1 | 0 | 100 |
+-----+-----+
```

La session 2 tente de verrouiller et est mise en attente.

```
Session 2> LOCK TABLES Client WRITE, Spectacle WRITE;
```

La session 1 peut finir ses mises à jour, et libère les tables avec la commande UNLOCK TABLES.

```
Session 1> UPDATE Spectacle SET nb_places_libres=45
    WHERE id_spectacle=1;
Query OK, 1 row affected (0,00 sec)

Session 1> UPDATE Client SET soldes=50, nb_places_reservees=5
    WHERE id_client=1;
Query OK, 1 row affected (0,00 sec)

Session 1> UNLOCK TABLES;
```

La session 2 peut alors prendre le verrou, effectuer ses lectures et mises à jour, et libérer le verrou. Les deux transactions se sont effectuées en série, et le résultat est donc correct.

La granularité du verrouillage explicite avec LOCK est la table entière, ce qui est généralement considéré comme mauvais car un verrouillage au niveau de lignes permet à plusieurs transactions d'accéder à différentes lignes de la table.

Le verrouillage des tables est une solution de “concurrence zéro” qui est rarement acceptable car elle revient à bloquer tous les processus sauf un. Dans un système où de très longues transactions (par exemple l'exécution d'un traitement lourd d'équilibrage de comptes) cohabitent avec de très courtes (des saisies interactives), les utilisateurs sont extrêmement pénalisés. Pour ne rien dire du cas où on oublie de relâcher les verrous...

De plus, dans l'exemple *ex-conc-trans*, il n'existe pas de conflit sur les clients puisque les deux transactions travaillent sur deux lignes différentes c_1 et c_2 . quand seules quelques lignes sont mises à jour, un verrouillage total n'est pas justifié.

Le verrouillage de tables peut cependant être envisagé dans le cas de longues transactions qui vont parcourir toute la table et souhaitent en obtenir une image cohérente. C'est par exemple typiquement le cas pour une sauvegarde. De même, si une longue transaction effectuant des mises à jour est en concurrence avec de nombreuses petites transactions, le risque d'interblocage, temporaire ou définitif (voir plus loin) est important, et on peut envisager de précéder la longue transaction par un verrouillage en écriture.

7.3.3 Le mode read committed

Le mode read committed, adopté par défaut dans ORACLE par exemple, amène un résultat incorrect pour nos deux exemples ! Ce mode ne pose pas de verrou en lecture, et assure simplement qu'une donnée lue n'est pas en cours de modification par une autre transaction. Voici ce qui se passe pour l'exemple *ex-conc-rw*.

- On commence par la procédure de contrôle qui lit le premier client, r_1/c_1 . Ce client a réservé 5 places. La procédure de contrôle lit c_2 qui n'a réservé aucune place. Donc le nombre total de places réservées est de 5.
- Puis c'est la réservation qui s'exécute, elle lit le spectacle, le client 2 (aucun de ces deux tuples n'est en cours de modification). Le client c_2 réserve 2 places, donc au moment où la réservation effectue un commit, il y a 43 places libres pour le spectacle, 2 places réservées pour c_2 .
- La session 1 (le contrôle) reprend son exécution et lit s . Comme s est validée on lit la valeur mise à jour juste auparavant par Res , et on trouve donc 43 places libres. La procédure de contrôle constate donc, à tort, une incohérence. Le mode read committed est particulièrement inadapté aux longues transactions pour lesquelles le risque est fort de lire des données modifiées et validées après le début de l'exécution. En contrepartie le niveau de verrouillage est faible, ce qui évite les blocages.

7.3.4 Le mode repeatable read

Dans le mode `repeatable read`, chaque lecture effectuée par une transaction lit les données telles qu'elles étaient *au début de la transaction*. Cela donne un résultat correct pour l'exemple [ex-conc-rw](#), comme le montre le déroulement suivant.

- On commence par la procédure de contrôle qui lit le premier client, r_1/c_1 . Ce client a réservé 5 places. La procédure de contrôle lit c_2 qui n'a réservé aucune place. Donc le nombre total de places réservées est de 5.
- Puis c'est la réservation qui s'exécute, elle lit le spectacle, le client 2 (aucun de ces deux tuples n'est en cours de modification). Le client c_2 réserve 2 places, donc au moment où la réservation effectue une `commit`, il y a 43 places libres pour le spectacle, 2 places réservées pour c_2 .
- La session 1 (le contrôle) reprend son exécution et lit s . Miracle ! La mise à jour de la réservation n'est pas visible car elle a été effectuée *après* le début de la procédure de contrôle. Cette dernière peut donc conclure justement que la base, *telle qu'elle était au début de la transaction*, est cohérente.

Ce niveau d'isolation est suffisant pour que les mises à jour effectuées par une transaction T' pendant l'exécution d'une transaction T ne soient pas visibles de cette dernière. Cette propriété est extrêmement utile pour les longues transactions, et elle a l'avantage d'être assurée sans aucun verrouillage.

En revanche le mode `repeatable read` ne suffit toujours pas pour résoudre le problème des mises à jour perdues. Reprenons une nouvelle fois l'exemple [ex-conc-trans](#). Voici un exemple concret d'une session sous MySQL/InnoDB, SGBD dans lequel le mode d'isolation par défaut est `repeatable read`. C'est la première session qui débute, avec des lectures.

```
Session 1> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 1> SELECT * FROM Spectacle WHERE id_spectacle=1;
+-----+-----+-----+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+-----+-----+-----+
|           1 |                 50 |                  50 |   10.00 |
+-----+-----+-----+
1 row in set (0,01 sec)

Session 1> SELECT * FROM Client WHERE id_client=1;
+-----+-----+
| id_client | nb_places_reservees | solde |
+-----+-----+
|         1 |                   0 |    100 |
+-----+-----+
```

La session 1 constate donc qu'aucune place n'est réservée. Il reste 50 places libres. La session 2 exécute à son tour les lectures.

```
Session 2> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 2> SELECT * FROM Spectacle WHERE id_spectacle=1;
+-----+-----+-----+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+-----+-----+-----+
|           1 |                 50 |                  50 |   10.00 |
+-----+-----+-----+
1 row in set (0,00 sec)

Session 2> SELECT * FROM Client WHERE id_client=2;
+-----+-----+
```

id_client nb_places_reservees solde
-----+-----+-----+
2 0 60
-----+-----+-----+

Maintenant la session 2 effectue sa réservation de 2 places. Pensant qu'il en reste 50 avant la mise à jour, elle place le nombre 48 dans la table *Spectacle*.

```
Session 2> UPDATE Spectacle SET nb_places_libres=48
      WHERE id_spectacle=1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 2> UPDATE Client SET solde=40, nb_places_reservees=2
      WHERE id_client=2;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 2> commit;
Query OK, 0 rows affected (0,00 sec)
```

Pour l'instant InnoDB ne dit rien. La session 1 continue alors. Elle aussi pense qu'il reste 50 places libres. La réservation de 5 places aboutit aux requêtes suivantes.

```
Session 1> UPDATE Spectacle SET nb_places_libres=45 WHERE id_spectacle=1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 1> UPDATE Client SET solde=50, nb_places_reservees=5 WHERE id_client=1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 1> commit;
Query OK, 0 rows affected (0,01 sec)

Session 1> SELECT * FROM Spectacle WHERE id_spectacle=1;
+-----+-----+-----+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+-----+-----+-----+
| 1 | 50 | 45 | 10.00 |
+-----+-----+-----+
1 row in set (0,00 sec)

Session 1> SELECT * FROM Client;
+-----+-----+
| id_client | nb_places_reservees | solde |
+-----+-----+
| 1 | 5 | 50 |
| 2 | 2 | 40 |
+-----+-----+
```

La base est incohérente ! les clients ont réservé (et payé) en tout 7 places, mais le nombre de places libres n'a diminué que de 5. L'utilisation de InnoDB *ne garantit pas* la correction des exécutions concurrentes, du moins avec le niveau d'isolation par défaut.

Ce point est très souvent ignoré, et source de problèmes récurrents chez les organisations qui croient s'appuyer sur un moteur transactionnel assurant une cohérence totale, et constatent de manière semble-t-il aléatoire l'apparition d'incohérences et de déséquilibres dans leurs bases.

Note : La remarque est valable pour de nombreux autres SGBD, incluant ORACLE, dont le niveau d'isolation par défaut n'est pas maximal.

On soupçonne le plus souvent les programmes, à tort puisque c'est l'exécution concurrente qui, parfois, est fautive, et pas le programme. Il est extrêmement difficile de comprendre, et donc de corriger, ce type d'erreur.

7.3.5 Le mode serializable

Si on analyse attentivement l'exécution concurrente de l'exemple *ex-conc-trans*, on constate que le problème vient du fait que les deux transactions lisent, chacune de leur côté, une information (le nombre de places libres pour le spectacle) qu'elles s'apprêtent toutes les deux à modifier. Une fois cette information transférée dans l'espace mémoire de chaque processus, il n'existe plus aucun moyen pour ces transactions de savoir que cette information a changé dans la base, et qu'elles s'appuient donc sur une valeur incorrecte.

La seule chose qui reste à faire pour obtenir une isolation maximale est de s'assurer que cette situation ne se produit pas. C'est ce que garantit le mode serializable, au prix d'un risque de blocage plus important. On obtient ce niveau avec la commande suivante :

```
SET TRANSACTION ISOLATION LEVEL serializable;
```

Reprenons une dernière fois l'exemple *ex-conc-trans*, en mode sérialisable, avec MySQL/InnoDB. La session 1 commence par ses lectures.

```
Session 1> SET TRANSACTION ISOLATION LEVEL serializable;
Query OK, 0 rows affected (0,04 sec)

Session 1> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 1> SELECT * FROM Spectacle WHERE id_spectacle=1;
+-----+-----+-----+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+-----+-----+-----+
| 1 | 50 | 50 | 10.00 |
+-----+-----+-----+
1 row in set (0,00 sec)

Session 1> SELECT * FROM Client WHERE id_client=1;
+-----+-----+
| id_client | nb_places_reservees | soldes |
+-----+-----+
| 1 | 0 | 100 |
+-----+-----+
```

Voici le tour de la session 2. Elle effectue ses lectures, et cherche à effectuer la première mise à jour.

```
Session 2> SET TRANSACTION ISOLATION LEVEL serializable;
Query OK, 0 rows affected (0,00 sec)

Session 2> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 2> SELECT * FROM Spectacle WHERE id_spectacle=1;
+-----+-----+-----+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+-----+-----+-----+
| 1 | 50 | 48 | 10.00 |
+-----+-----+
```

```
+-----+-----+-----+
1 row in set (0,00 sec)

Session 2> SELECT * FROM Client WHERE id_client=2;
+-----+-----+-----+
| id_client | nb_places_reservees | solde |
+-----+-----+-----+
| 2 | 0 | 60 |
+-----+-----+-----+
1 row in set (0,00 sec)

Session 2> UPDATE Spectacle SET nb_places_libres=48 WHERE id_spectacle=1;
```

La transaction 2 est mise en attente car, en mode sérialisable, MySQL/InnoDB pose un verrou en lecture sur les lignes sélectionnées. La transaction 1 a donc verrouillé, en mode partagé, le spectacle et le client. La transaction 2 a pu lire le spectacle, et placer à son tour un verrou partagé, mais elle ne peut pas le modifier car cela implique la pose d'un verrou exclusif.

Que se passe-t-il alors du côté de la transaction 1 ? Elle cherche à faire la mise à jour du spectacle. Voici la réaction de InnoDB.

```
Session 1> UPDATE Spectacle SET nb_places_libres=45 WHERE id_spectacle=1;
ERROR 1213 (40001): Deadlock found when trying to get lock;
try restarting transaction
```

Un interblocage (*deadlock*) a été détecté. La transaction 2 était déjà bloquée par la transaction 1. En cherchant à modifier le spectacle, la transaction 1 se trouve bloquée à son tour par les verrous partagés posés par la transaction 2.

En cas d'interblocage, les deux transactions peuvent s'attendre indéfiniment l'une l'autre. Le SGBD prend donc la décision d'annuler par `rollback` l'une des deux (ici, la transaction 1), en l'incitant à recommencer. La transaction 2 est libérée (elle garde ses verrous) et peut poursuivre son exécution.

Le mode `Serializable` garantit la correction des exécutions concurrentes, au prix d'un risque de blocage et de rejet de certaines transactions. Ce risque, et ses effets désagréables (il faut resoumettre la transaction rejetée) expliquent qu'il ne s'agit pas du mode d'isolation par défaut. Pour les applications transactionnelles, il vaut sans doute mieux voir certaines transactions rejettées que courir un risque d'anomalie.

Verrouillage d'une ligne avec `FOR UPDATE`

Une alternative au mode `Serializable` est la pause explicite de verrous sur les lignes que l'on s'apprête à modifier. La clause `FOR UPDATE` place un verrou exclusif sur les tuples sélectionnés par un ordre `SELECT`. Ces tuples sont donc réservés pour une future modification : aucune autre transaction ne peut placer de verrou en lecture ou en écriture. L'intérêt est de ne pas réserver les tuples qui sont simplement lues et non modifiées ensuite. Notez qu'en mode `Serializable` toutes les lignes lues sont réservées, car le SGBD, contrairement au programmeur de l'application, ne peut pas deviner ce qui va se passer ensuite.

Voici l'exécution de l'exemple `ex-conc-trans`, en veillant à verrouiller les lignes que l'on va modifier.

- C'est la transaction 1 qui commence. Elle lit le spectacle et le client c_1 en posant un verrou exclusif avec la clause `FOR UPDATE`.
- Ensuite c'est la seconde transaction qui transmet ses commandes au serveur. Elle aussi cherche à placer des verrous (c'est normal, il s'agit de l'exécution du même code). Bien entendu elle est mise en attente puisque la session 1 a déjàposé un verrou exclusif.
- La session 1 peut continuer de s'exécuter. Le `commit` libère les verrous, et la transaction 2 peut alors conclure.

Au final les deux transactions se sont exécutées en série. La base est dans un état cohérent. L'utilisation de `FOR UPDATE` est un compromis entre l'isolation assurée par le système, et la déclaration explicite, par le programmeur, des données lues en vue d'être modifiées. Elle assure le maximum de fluidité pour une isolation totale, et minimise

le risque d’interblocages. Le principal problème est qu’elle demande une grande discipline pendant l’écriture d’une application puisqu’il faut se poser la question, à chaque requête, des lignes que l’on va ou non modifier.

En résumé, il est de la responsabilité du programmeur, sur un SGBD n’adoptant pas le mode SERIALISABLE par défaut, de prendre lui-même les mesures nécessaires pour les transactions qui risquent d’aboutir à des incohérences en cas de concurrence sur les mêmes données. Ces mesures peuvent consister soit à passer en mode serializable pour ces transactions, soit à poser explicitement des verrous, en début de transaction, sur les données qui vont être modifiées ensuite.

7.3.6 Exercices

Contrôle de concurrence

Un SGBD doit garantir que l'exécution d'un programme effectuant des mises à jour dans un contexte multi-utilisateur s'effectue "correctement". Bien entendu la signification du "correctement" doit être définie précisément, de même que les techniques assurant cette correction : c'est l'objet du *contrôle de concurrence*.

8.1 S1 : Les mécanismes

Supports complémentaires :

- Diapositives : mécanismes transactionnels
 - Vidéo de présentation (à venir)
-

Le contrôle de concurrence est la partie de l'activité d'un SGBD qui consiste à ordonner l'exécution des transactions de manière à assurer la sérialisabilité et la recouvrabilité. Nous donnons ci-dessous un aperçu des différentes techniques utilisées, toujours dans l'optique de fournir au programmeur d'applications de bases de données une idée au moins intuitive des mécanismes sous-jacents au déroulement d'une application. Nous commençons donc par décrire les mécanismes assurant la recouvrabilité, avant de passer à la gestion de la sérialisabilité.

8.1.1 Versionnement

Il existe toujours deux choix possibles pour une transaction T en cours : effectuer un `commit` pour valider définitivement les opérations effectuées, ou un `rollback` pour les annuler. Pour que ces choix soient toujours disponibles, le SGBD doit maintenir, pendant l'exécution de T , deux versions des données mises à jour :

- une version des données *après* la mise à jour ;
- une version des données *avant* la mise à jour.

Ces deux versions sont stockées dans deux espaces de stockage séparés, que nous désignerons respectivement par les termes d'*image après* et d'*image avant* dans ce qui suit. Conceptuellement (les détails pouvant être assez compliqués), le déroulement d'une transaction T , basé sur ces deux images, s'effectue de la manière suivante. Chaque fois que T effectue la mise à jour d'un tuple, la version courante est d'abord copiée dans l'image avant, puis remplacée par la nouvelle valeur fournie par T . Dès lors le `commit` et le `rollback` sont implantés comme suit :

- le `commit` consiste à s'assurer que les données de l'image après sont vraiment écrites sur disque, afin de garantir la durabilité ; l'image avant peut alors être effacée ;
- le `rollback` prend les tuples de l'image avant et les écrit dans l'image après pour ramener la base à l'état initial de la transaction.

Un `rollback` effectué à la suite d'une panne fonctionne de la même manière, sous réserve bien entendu que la panne n'ait pas affecté l'image avant elle-même. Il est donc très important de s'assurer que les données de l'image avant sont régulièrement écrites sur disque. Pour des raisons de performance (répartition des entrées/sorties) et de minimisation des risques, on choisit d'ailleurs quand c'est possible de placer l'image avant et l'image après sur des disques différents.

Quand T effectue des lectures sur des données qu’elles vient de modifier, le système doit lire dans l’image après pour assurer une vision cohérente de la base, reflétant les opérations effectuées par une transaction. En revanche, quand c’est une autre transaction T' qui demande la lecture d’un tuple modifié par T , il faut lire dans l’image avant pour éviter les effets de lectures sales.

On peut se poser la question du nombre de versions nécessaires. Que se passe-t-il par exemple quand T' demande la mise à jour d’un tuple déjà modifié par T ? Si cette mise à jour était autorisée, il s’agirait d’une *écriture sale* et il faudrait prendre garde à créer une troisième version du tuple, l’image avant de T' étant l’image après de T . La multiplication des versions rendrait la gestion des `commit` et `rollback` extrêmement complexe. En pratique les systèmes n’autorisent pas les écritures sales, s’appuyant pour contrôler cette règle sur des mécanismes de verrouillage exclusif qui seront présentés dans ce qui suit. Il s’ensuit que la gestion de la recouvrabilité ne nécessite pas plus de deux versions pour chaque tuple, une dans l’image avant et l’autre dans l’image après.

Le versionnement n’est pas seulement utile à la gestion de la recouvrabilité. Reprenons le problème des *lectures non répétables*. Elles sont dues au fait qu’une transaction T lit un tuple t qui a été modifié par une transaction T' après le début de T . Or, quand T' modifie t , il existe avant la validation deux versions de t : l’image avant et l’image après.

Il suffirait que T lise toujours l’image avant pour que le problème des tuples fantômes soit résolu. En d’autres termes ces images avant peuvent être vues comme un “cliché” de la base pris à un moment donné, et toute transaction ne lit que dans le cliché *valide* au moment où elle a débuté. De nombreux SGBD (dont ORACLE, PostgreSQL, MySQL/InnoDB) proposent un mécanisme de *lecture cohérente* basé sur ce système de versionnement qui s’appuie sur les principes suivants :

- chaque transaction T se voit attribuer, quand elle débute, une estampille temporelle e_T ; chaque valeur d’estampille est unique et les valeurs sont croissantes : on garantit ainsi un ordre total entre les transactions.
- chaque version *validée* d’un tuple a est de même estampillée par le moment e_a de sa validation ;
- quand T doit lire un tuple a , le système regarde dans l’image après. Si t a été modifié par T ou si son estampille est inférieure à e_T , le tuple peut être lu puisqu’il a été validé avant le début de T , sinon T recherche dans l’image avant la version de t validée et immédiatement antérieure à e_T .

La seule extension nécessaire par rapport à l’algorithme de gestion de la recouvrabilité est la non-destruction des images avant, même quand la transaction qui a modifié le tuple valide par `commit`. L’image avant contient alors toutes les versions successives d’un tuple, marquées par leur estampille temporelle. Seule la plus récente de ces versions correspond à une mise à jour en cours. Les autres ne servent qu’à assurer la cohérence des lectures.

La figure *Lectures cohérentes avec image avant* illustre le mécanisme de lecture cohérente. La transaction T_{23} a débuté à l’instant 23. Elle a modifié à l’instant 25 l’enregistrement e_3 , et la version précédente (dont l’estampille est 14) de e_3 a été placée dans l’image avant. Maintenant, si T_{23} effectue une lecture, celle-ci accède à la version de e_3 placée dans l’image après puisque c’est elle qui l’a modifiée. En revanche une transaction T'_{18} dont le moment de début est 18 lira la version de e_3 dans l’image avant (il y a un pointeur de l’image avant vers l’image après).

En d’autres termes l’image avant peut être vue comme un “cliché” de la base pris à un moment donné, et toute transaction ne lit que dans le cliché “valide” au moment où elle a débuté.

L’image avant contient en revanche l’historique de toutes les versions successives d’un enregistrement, marquées par leur estampille temporelle. Seule la plus récente de ces versions correspond à une mise à jour en cours. Les autres ne servent qu’à assurer la cohérence des lectures. Certaines de ces versions n’ont plus aucune chance d’être utilisées : ce sont celles pour lesquelles il existe une version plus récente, antérieure à toutes les requêtes en cours. Cette propriété permet au SGBD d’effectuer un nettoyage (*garbage collection*) des versions devenues inutiles.

8.1.2 Verrouillage

Le mécanisme de versionnement décrit ci-dessus est nécessaire mais pas suffisant pour assurer la recouvrabilité et la sérialisabilité des exécutions concurrentes. Le contrôle de concurrence doit parfois effectuer un réordonnancement de l’ordre d’arrivée des opérations pour assurer ces deux propriétés.

La technique la plus fréquente est basée sur le *verrouillage* des tuples lus ou mis à jour. L’idée est simple : chaque transaction désirant lire ou écrire un tuple doit auparavant obtenir un verrou sur ce tuple. Une fois obtenu (sous

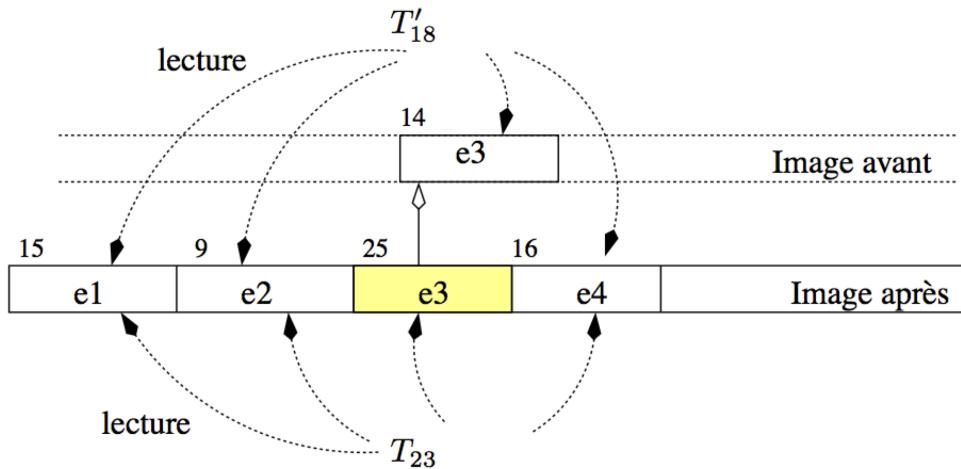


Fig. 8.1 – Lectures cohérentes avec image avant

certaines conditions explicitées ci-dessous), le verrou reste détenu par la transaction qui l'a posé, jusqu'à ce que cette transaction décide de relâcher le verrou.

Il existe essentiellement deux types de verrous :

- les *verrous partagés* autorisent la pose d'autres verrous partagés sur le même tuple.
- les *verrous exclusifs* interdisent la pose de tout autre verrou, exclusif ou partagé, et donc de toute lecture ou écriture par une autre transaction.

On ne peut poser un verrou partagé que s'il n'y a que des verrous partagés sur le tuple. On ne peut poser un verrou exclusif que s'il n'y a aucun autre verrou, qu'il soit exclusif ou partagé. Les verrous sont posés par chaque transaction, et ne sont libérés qu'au moment du *commit* ou *rollback*.

Dans ce qui suit les verrous en lecture seront notés *rl* (comme *read lock*) dans ce qui suit, et ses verrous en écritures seront notés *wl* (comme *write lock*). Donc *rl_i[x]* indique par exemple que la transaction *i* a posé un verrou en lecture sur la resource *x*. On notera de même *ru* et *wu* le relâchement des verrous (*read unlock* et *write unlock*).

Il ne peut y avoir qu'un seul verrou exclusif sur un tuple. Son obtention par une transaction *T* suppose donc qu'il n'y ait aucun verrou déjà posé par une autre transaction *T'*. En revanche il peut y avoir plusieurs verrous partagés : l'obtention d'un verrou partagé est possible sur un tuple tant que ce tuple n'est pas verrouillé exclusivement par une autre transaction. Enfin, si une transaction est la seule à détenir un verrou partagé sur un tuple, elle peut poser un verrou exclusif.

Si une transaction ne parvient pas à obtenir un verrou, elle est mise en attente, *ce qui signifie que la transaction s'arrête complètement jusqu'à ce que le verrou soit obtenu*. Rappelons qu'une transaction est une séquence d'opérations, et qu'il n'est évidemment pas question de changer l'ordre, ce qui reviendrait à modifier la sémantique du programme. Quand une opération n'obtient pas de verrou, elle est mise en attente ainsi que toutes celles qui la suivent pour la même transaction.

Les verrous sont posés de manière automatique par le SGBD en fonction des opérations effectuées par les transactions/utilisateurs. Il est également possible de demander explicitement le verrouillage de certaines ressources (tuple ou même table) (cf. chapitre d'introduction à la concurrence).

Tous les algorithmes de gestion de la concurrence (voir les deux présentés ci-dessous) utilisent le verrouillage. L'idée générale est de bloquer l'accès à une donnée dès qu'elle est lue ou écrite par une transaction ("pose de verrou") et de libérer cet accès quand la transaction termine par *commit* ou *rollback* ("libération du verrou").

En reprenant l'exemple de la réservation, et en supposant que tout accès en lecture ou en écriture pose un verrou bloquant les autres transactions, les transactions *T₁* et *T₂* s'exécuteront clairement en série et les anomalies disparaîtront.

Le bloquage systématique des transactions est cependant une contrainte trop forte. L'exécution est correcte, mais le verrouillage total bloquerait pourtant sans nécessité la transaction T_2 .

Un bon algorithme doit garantir la sérialisabilité et la recouvrabilité au prix d'un minimum de blocages. On peut réduire ce blocage en jouant sur deux critères :

- le *degré de restriction* (verrou partagé ou verrou exclusif) ;
- la *granularité* du verrouillage (i.e. le niveau de la *ressource* à laquelle il s'applique : tuple, page, table, etc).

Tous les SGBD proposent un verrouillage au niveau du tuple, et privilégient les verrous partagés tant que cela ne remet pas en cause la correction des exécutions concurrentes. Un verrouillage au niveau du tuple est considéré comme moins pénalisant pour la fluidité, puisqu'il laisse libres d'autres transactions d'accéder à tous les autres tuples non verrouillés. Il existe cependant des cas où cette méthode est inappropriée. Si par exemple un programme parcourt une table avec un curseur pour modifier chaque tuple, et valide à la fin, on va poser un verrou sur chaque alors qu'on aurait obtenu un résultat équivalent avec un seul verrou au niveau de la table.

Note : Certains SGBD pratiquent également l'*escalade des verrous* : quand plus d'une certaine fraction des tuples d'une table est verrouillée, le SGBD passe automatiquement à un verrouillage au niveau de la table. Sinon on peut envisager, en tant que programmeur, la pose explicite d'un verrou exclusif sur la table à modifier au début du programme.

8.1.3 Exercices

8.2 S2 : les algorithmes

Supports complémentaires :

- Diapositives : algorithmes de concurrence
-

Nous présentons dans cette section deux algorithmes de contrôle de concurrence, les plus utilisés en pratique. Ils s'appuient tous deux sur une condition qui permet au SGBD de *décider* si une exécution concurrente est sérialisable ou non. La notion de base est celle de *conflits* entre deux transactions sur la même ressource.

Définition.

Deux opérations $p_i[x]$ et $q_j[y]$ sont en *conflict* si $x=y$, $i \neq j$, et p ou q est une écriture.

On étend facilement cette définition aux exécutions concurrentes : deux transactions dans un exécution sont en *conflict* si elles accèdent à la même donnée et si un de ces accès au moins est une écriture.

Exemple.

Reprenons une nouvelle fois l'exemple des mises à jour perdues. L'exécution correspond à deux transactions T_1 et T_2 , accédant aux données s , c_1 et c_2 . Les conflits sont les suivants :

- $r_1(s)$ et $w_2(s)$ sont en conflit ;
- $w_2(s)$ et $w_1(s)$ sont en conflit.

Noter que $r_1(s)$ et $r_2(s)$ ne sont pas en conflit, puisque ce sont deux lectures. Il n'y a pas de conflit sur c_1 et c_2 .

De même, dans l'exécution de l'exemple~ref{ex :conc-rw} les conflits sont les suivants :

- $r_1(c_2)$ et $w_2(c_2)$ sont en conflit ;
 - $w_2(s)$ et $r_1(s)$ sont en conflit.
-

Les conflits permettent de définir un *ordre* sur les transactions d'une exécution concurrente.

Définition.

Soit H une exécution concurrente d'un ensemble de transactions $T = \{T_1, T_2, \dots, T_n\}$. Alors il existe une relation \lhd sur cet ensemble, définie par :

$$T_i \lhd T_j \Leftrightarrow \exists p \in T_i, q \in T_j, p \text{ est en conflit avec } q \text{ et } p <_H q$$

où $p <_H q$ indique que p apparaît avant q dans H .

Dans les deux exemples qui précèdent, on a donc $T_1 \lhd T_2$, ainsi que $T_2 \lhd T_1$. Une transaction T_i peut ne pas être en relation (directe) avec une transaction T_j . La condition sur la sérialisabilité s'exprime sur le graphe de la relation (T, \lhd) , dit *graphe de sérialisation* :

Théorème.

Soit H une exécution concurrente d'un ensemble de transactions T . Alors H est sérialisable si et seulement si le graphe de (T, \lhd) est acyclique.

La figure [Exemples de graphes de sérialisation](#) montre quelques exemples de graphes de sérialisation. Le premier correspond aux exemples données ci-dessus : il est clairement cyclique. Le second n'est pas cyclique et correspond donc à une exécution sérialisable. Le troisième est cyclique.

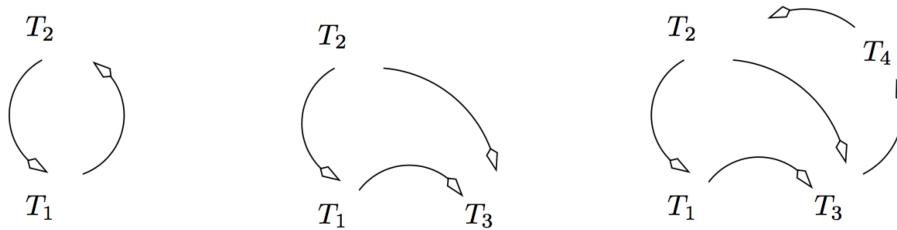


Fig. 8.2 – Exemples de graphes de sérialisation

Les algorithmes de contrôle de concurrence ont donc pour objectifs d'éviter la formation d'un cycle dans le graphe de sérialisation. Nous présentons les deux principaux algorithmes dans ce qui suit.

8.2.1 Contrôle par verrouillage à deux phases

Le verrouillage à deux phases est le protocole le plus ancien pour assurer des exécutions concurrentes correctes. Le respect du protocole est assuré par un module dit {it scheduler} qui reçoit les opérations émises par les transactions et les traite selon l'algorithme suivant :

- Le *scheduler* reçoit $p_i[x]$ et consulte le verrou déjà posé sur x , $ql_j[x]$, s'il existe.
- si $pl_i[x]$ est en conflit avec $ql_j[x]$, $p_i[x]$ est retardée et la transaction T_i est mise en attente.
- sinon, T_i obtient le verrou $pl_i[x]$ et l'opération $p_i[x]$ est exécutée.
- les verrous ne sont relâchés qu'au moment du *commit* ou du *rollback*.

Le terme “verrouillage à deux phases” s’explique par le processus détaillé ci-dessus : il y a d’abord *accumulation* de verrous pour une transaction T , puis *libération* des verrous à la fin de la transaction. Les transactions obtenues par application de cet algorithme satisfont les propriétés ACID. Il est assez facile de voir que les lectures ou écritures sales sont interdites, puisque toutes deux reviennent à tenter de lire ou d’écrire un tuple déjà écrit par une autre, et donc verrouillé exclusivement par l’algorithme.

Le protocole garantit que, en présence de deux transactions en conflit T_1 et T_2 , la dernière arrivée sera mise en attente de la première ressource conflictuelle et sera bloquée jusqu'à ce que la première commence à relâcher ses verrous (règle 1). À ce moment là il n'y a plus de conflit possible puisque T_1 ne demandera plus de verrou.

Pour illustrer l'intérêt de ces règles, on peut prendre l'exemple des deux transactions suivantes :

- $T_1 : r_1[x] w_1[y] C_1$
 - $T_2 : w_2[x] w_2[y] C_2$
- et l'exécution concurrente :

$r_1[x]w_2[x]w_2[y]C_2w_1[y]C_1$

Maintenant supposons que l'exécution avec pose et relâchement de verrous se passe de la manière suivante :

- T_1 pose un verrou partagé sur x , lit x puis relâche le verrou ;
- T_2 pose un verrou exclusif sur x , et modifie x ;
- T_2 pose un verrou exclusif sur y , et modifie y ;
- T_2 relâche les verrous sur x et y , puis valide ;
- T_1 pose un verrou exclusif sur y , modifie y , relâche le verrou et valide.

On a violé la règle 3 : T_1 a relâché le verrou sur x puis en a repris un sur y . Une “fenêtre” s'est ouverte qui a permis à T_2 de poser des verrous sur x et y . Conséquence : l'exécution n'est plus sérialisable car T_2 a écrit sur T_1 pour x , et T_1 a écrit sur T_2 pour y . Reprenons le même exemple, avec un verrouillage à deux phases :

- T_1 pose un verrou partagé sur x , lit x mais ne relâche pas le verrou ;
- T_2 tente de poser un verrou exclusif sur x : impossible puisque T_1 détient un verrou partagé, donc : T_2 est mise en attente ;
- T_1 pose un verrou exclusif sur y , modifie y , et valide ; tous les verrous détenus par T_1 sont relâchés ;
- T_2 est libérée : elle pose un verrou exclusif sur x , et le modifie ;
- T_2 pose un verrou exclusif sur y , et modifie y ;
- T_2 valide, ce qui relâche les verrous sur x et y .

On obtient donc, après réordonnancement, l'exécution suivante, qui est évidemment sérialisable :

$r_1[x]w_1[y]w_2[x]w_2[y]$

En général, le verrouillage permet une certaine imbrication des opérations tout en garantissant sérialisabilité et recouvrabilité. Notons cependant qu'il est un peu trop strict dans certains cas : voici l'exemple d'une exécution sérialisable impossible à obtenir avec un verrouillage à deux phases.

$r_1[x]w_2[x]C_2w_3[y]C_3r_1[y]w_1[z]C_1$

Un des inconvénients du verrouillage à deux phases est d'autoriser des *interblocages* : deux transactions concurrentes demandent chacune un verrou sur une ressource détenue par l'autre. Reprenons notre exemple de base : deux exécutions concurrentes de la procédure *Réservation*, désignées par T_1 et T_2 , consistant à réserver des places pour le même spectacle, mais pour deux clients distincts c_1 et c_2 . L'ordre des opérations reçues par le serveur est le suivant :

$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2w_1(s)w_1(c_1)C_1$

On effectue des lectures pour T_1 puis T_2 , ensuite les écritures pour T_2 puis T_1 . Cette exécution n'est pas sérialisable, et le verrouillage à deux phases doit empêcher qu'elle se déroule dans cet ordre. Malheureusement il ne peut le faire qu'en rejettant une des deux transactions. Suivons l'algorithme pas à pas :

- T_1 pose un verrou partagé sur s et lit s ;
- T_1 pose un verrou partagé sur c_1 et lit c_1 ;
- T_2 pose un verrou partagé sur s , ce qui est autorisé car : T_1 n'a elle-même qu'un verrou partagé et lit s ;
- T_1 pose un verrou partagé sur c_2 et lit c_2 ;
- T_2 veut poser un verrou exclusif sur s : impossible à cause du verrou partagé de T_1 : donc T_2 est mise en attente ;
- T_1 veut à son tour poser un verrou exclusif sur s : impossible à cause du verrou partagé de T_2 : donc T_1 est à son tour mise en attente.

T_1 et T_2 sont en attente l'une de l'autre : il y a *interblocage* (*deadlock* en anglais). Cette situation ne peut pas être évitée et doit donc être gérée par le SGBD : en général ce dernier maintient un *graphe d'attente des transactions* et teste l'existence de cycles dans ce graphe. Si c'est le cas, c'est qu'il y a interblocage et une des transactions doit être annulée autoritairement, ce qui est à la fois déconcertant pour un utilisateur non averti, et désagréable puisqu'il faut resoumettre la transaction annulée. Cela reste bien entendu encore préférable à un algorithme qui autoriserait un résultat incorrect.

Notons que le problème vient d'un accès aux mêmes ressources, mais dans un ordre différent : il est donc bon, au moment où l'on écrit des programmes, d'essayer de normaliser l'ordre d'accès aux données.

Dès que 2 transactions lisent la même donnée avec pour objectif d'effectuer une mise à jour ultérieurement, il y a potentiellement interblocage. D'où l'intérêt de pouvoir demander dès la lecture un verrouillage exclusif (écriture). C'est la commande `select ... for update` que l'on trouve dans certains SGBD.

8.2.2 Contrôle de concurrence multi-versions

Les systèmes qui s'appuient sur des lectures cohérentes et gèrent donc des bases multi-versions, peuvent tirer parti du fait que les lectures s'appuient toujours sur une version cohérente (le "cliché") de la base. Tout se passe comme si les lectures effectuées par une transaction $T(t_0)$ débutant à l'instant t_0 lisait la base, dès le début de la transaction, donc dans l'état t_0 .

Cette remarque réduit considérablement les cas possibles de conflits, comme le montre le raisonnement suivant. Prenons une lecture $r_1[d]$ effectuée par la transaction $T_1(t_0)$. Cette lecture accède à la version *validée* la plus récente de d qui précède t_0 , par définition de l'état de la base à t_0 . Deux cas de conflits sont envisageables :

- $r_1[d]$ est en conflit avec une écriture $w_2[d]$ qui a eu lieu *avant* t_0 ;
- $r_1[d]$ est en conflit avec une écriture $w_2[d]$ qui a eu lieu *après* t_0 .

Dans le premier cas, T_2 a forcément effectué son *commit* avant t_0 , puisque T_1 lit l'état de la base à t_0 : tous les conflits de T_1 avec T_2 sont dans le même sens, et il n'y a pas de risque de cycle (Figure [Contrôle de concurrence multi-versions : conflit avec les écritures précédentes](#)).

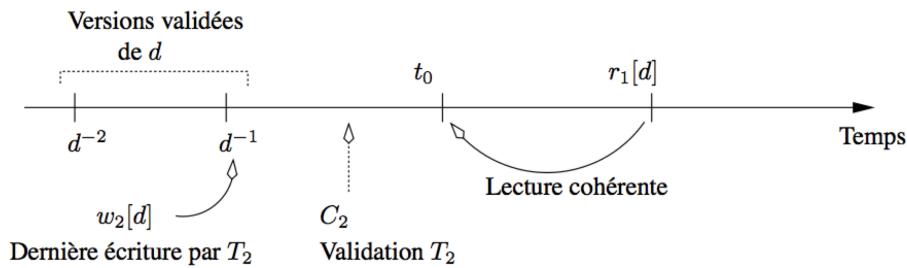


Fig. 8.3 – Contrôle de concurrence multi-versions : conflit avec les écritures précédentes

Le second cas est celui qui peut poser problème. Si T_1 cherche à écrire d après l'écriture $w_2[d]$, alors un conflit cyclique apparaît (Figure [Contrôle de concurrence multi-versions : conflit avec une écriture d'une autre transaction](#)). Notez qu'une nouvelle lecture de d par T_1 n'introduit pas de cycle puisque toute lecture s'effectue à t_0 .

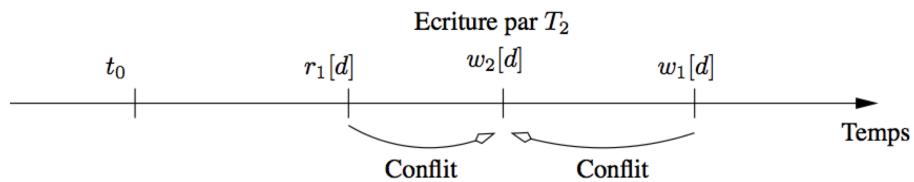


Fig. 8.4 – Contrôle de concurrence multi-versions : conflit avec une écriture d'une autre transaction.

En résumé le contrôle de concurrence peut alors se limiter à vérifier, au moment de l'écriture d'un tuple a par une transaction T , qu'aucune transaction T' n'a modifié a entre le début de T et l'instant présent. Si on autorisait la modification de a par T , des risques de conflits cycliques apparaîtraient avec T' . Autrement dit une mise à jour n'est possible que si la partie de la base à modifier n'a pas changé depuis que T a commencé à s'exécuter.

Le contrôle de concurrence multi-versions s'appuie sur les règles suivantes. Rappelons que pour chaque transaction T on connaît son estampille temporelle de début d'exécution e_T ; et pour chaque version d'un tuple a son estampille de validation e_a .

- toute lecture $r_T[a]$ lit la plus récente version de a telle que $e_a \leq e_T$; aucun verrou n'est posé ;
- **en cas d'écriture** $w_T[a]$,
 - si $e_a \leq e_T$ et aucun verrou n'est posé sur a : T pose un verrou exclusif sur a , et effectue la mise à jour ;
 - si $e_a \leq e_T$ et un verrou est posé sur a : T est mise en attente ;
- si $e_a > e_T$, T est rejetée.
- au moment du `commit` d'une transaction T , tous les enregistrements modifiés par T obtiennent une nouvelle version avec pour estampille l'instant du `commit`.

Avec cette technique, on peut ne pas poser de verrou en lecture. En revanche les verrous exclusifs sont toujours indispensables pour les écritures, afin d'éviter lectures ou écritures sales.

Voici un déroulé de cette technique, toujours sur notre exemple d'une exécution concurrente du programme de réservation avec l'ordre suivant :

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2w_1(s)w_1(c_1)C_1$$

On suppose que $e_{T_1} = 100$, $e_{T_2} = 120$. On va supposer qu'une opération est effectuée toutes les 10 unités de temps, même si seul l'ordre compte, et pas le délai entre deux opérations. Le déroulement de l'exécution est donc le suivant :

- T_1 lit s , sans verrouiller ;
- T_1 lit c_1 , sans verrouiller ;
- T_2 lit s , sans verrouiller ;
- T_2 lit c_2 , sans verrouiller ;
- T_2 veut modifier s : l'estampille de s est inférieure à $e_{T_2} = 120$, ce qui signifie que s n'a pas été modifié par une autre transaction depuis que T_2 a commencé à s'exécuter ; on pose un verrou exclusif sur s et on effectue la modification :
- T_2 modifie c_2 , avec pose d'un verrou exclusif ;
- T_2 valide et relâche les verrous ; deux nouvelles versions de s et c_2 sont créées avec l'estampille 150 ;
- T_1 veut à son tour modifier s , mais cette fois le contrôleur détecte qu'il existe une version de s avec $e_s > e_{T_1}$, donc que s a été modifié après le début de T_1 . Le contrôleur doit donc rejeter T_1 sous peine d'obtenir une exécution non sérialisable.

Comme dans le verrouillage à deux phases, on obtient le rejet de l'une des deux transactions, mais le SGBD effectue dans ce cas un contrôle *à postérieur* au lieu d'effectuer un blocage *à priori* comme le verrouillage à deux phases. On parle parfois d'approche "pessimiste" pour le verrouillage à deux phases et "optimiste" pour le contrôle multi-versions, exprimant ainsi l'idée que la première technique tente de prévenir les problèmes, alors que la seconde choisit de laisser faire et d'intervenir seulement quand ils surviennent réellement.

L'absence de verrouillage en lecture favorise la fluidité des exécutions concurrentes par rapport au verrouillage à deux phases, et limite le coût de la pose de verrous. On peut le vérifier par exemple sur l'exécution concurrente de l'exemple impliquant la procédure de contrôle. Bien entendu le coût en contrepartie est la nécessité de gérer les versions et de maintenir une vue cohérente de la base pour chaque transaction.

8.2.3 Exercices

Reprise sur panne

Supports complémentaires :

- Diapositives : reprise sur panne
-

Ce chapitre est consacré aux principes de la reprise sur panne dans les SGBD. La reprise sur panne consiste, comme son nom l'indique, à assurer que le système est capable, après une panne, de récupérer *l'état de la base* au moment où la panne est survenue. Le terme de panne désigne ici tout événement qui affecte le fonctionnement du processeur ou de la mémoire principale. Il peut s'agir par exemple d'une coupure électrique interrompant le serveur de données, ou d'une défaillance logicielle. Les pannes affectant les disques sont plus graves, mais nous verrons cependant qu'avec une stratégie appropriée de sauvegarde, il est relativement facile de garantir que l'état de la base peut être récupéré même en cas de panne de l'un des disques.

L'état de la base ?

On définit l'état de la base à un instant t comme l'état résultant de l'ensemble des transactions validées à l'instant t .

La problématique de la reprise sur panne est à rapprocher de la garantie de durabilité pour les transactions. Il s'agit d'assurer que même en cas d'interruption à $t+1$, on retrouvera la situation issue des transactions validées.

La première section discute de l'impact de l'architecture sur les techniques de reprise sur panne. Ces techniques sont ensuite développées dans les sections suivantes.

9.1 S1 : mécanismes fondamentaux

Pour bien comprendre les mécanismes utilisés, il faut avoir en tête l'architecture générale d'un serveur de données en cours de fonctionnement, et se souvenir que la performance d'un système est fortement liée au nombre de lectures/écritures qui doivent être effectuées. La reprise sur panne, comme les autres techniques mises en œuvre dans un SGBD, vise à minimiser ces entrées/sorties.

9.1.1 Pannes et mémoires

La figure *Gestion des écritures avec fichier journal* rappelle les composants d'un SGBD qui interviennent dans la reprise sur panne. On distingue la mémoire stable ou *persistante* (les disques) qui survit à une panne légère de type électrique ou logicielle, et la mémoire instable ou *volatile* qui est irrémédiablement perdue en cas, par exemple, de panne électrique.

Pour assurer la reprise sur panne, il est impératif que l'état de la base soit stocké sur support persistant, à tout instant. Une première règle simple est donc :

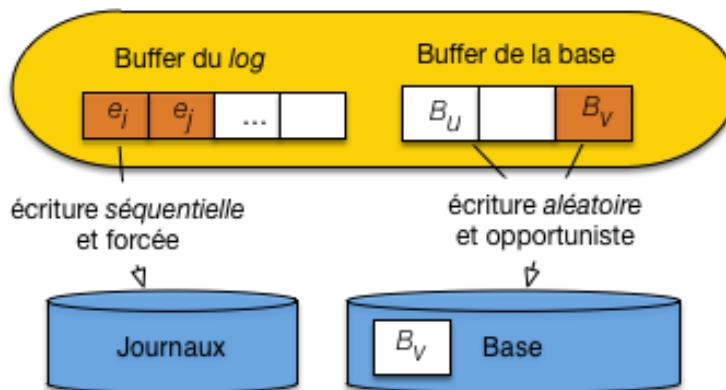


Fig. 9.1 – Gestion des écritures avec fichier journal

Règle 1

L'état de la base doit toujours être stocké sur disque

Ce qui revient à dire que toute donnée modifiée par une transaction qui valide par `commit` doit être écrite sur disque *avant* l'acquittement de l'opération de `commit`. Or, pour des raisons de performance, le serveur de données cherche à limiter les accès aux disques, et s'appuie sur une mémoire tampon (*buffer* ou *cache* en anglais) qui stocke, en mémoire principale (donc instable) les blocs de données provenant des fichiers stockés sur disque. Que ce soit en lecture ou en écriture, le serveur va chercher à s'appuyer sur la mémoire tampon.

Vocabulaire

Dans ce qui suit, on utilise le vocabulaire suivant :

- un *enregistrement* est la représentation d'une entité applicative mise à jour de manière atomique ; dans le contexte d'une base relationnelle, un enregistrement correspond à la représentation physique d'une ligne ;
- on s'autorise l'anglicisme *mémoire cache* ou simplement *cache* pour désigner la mémoire tampon ;
- enfin le *bloc* est l'unité d'échange entre la mémoire volatile et le disque ; un bloc contient en général plusieurs enregistrements.

Pour les écritures (cas qui nous intéresse ici), le serveur recherche tout d'abord si l'enregistrement est dans le *cache*. Si oui, la modification a lieu en mémoire, sinon le bloc contenant l'enregistrement est chargé du disque vers le cache, ce qui ramène au cas précédent. Dans tous les cas, la modification dans le cache n'entraîne pas une écriture sur le disque. Une écriture systématique après une modification entraînerait des performances désastreuses. Heureusement ce n'est pas nécessaire, car si une panne survient avant l'écriture sur disque, la donnée perdue est une donnée non validée par un `commit`, et elle ne fait donc pas partie de l'état de la base.

Un bloc placé dans le cache et non modifié est l'image exacte du bloc correspondant sur le disque. Quand une transaction vient modifier un enregistrement dans un bloc, son image en mémoire (l'image après) devient différente de celle sur le disque (l'image avant). La figure *Gestion des écritures avec fichier journal* illustre la situation pour deux enregistrements modifiés, e_i et e_j , situés respectivement dans les pages B_u et B_v .

Quand un enregistrement est modifié dans le cache, le bloc qui le contient est marqué. Le système tient alors compte de cette marque. En particulier, si le cache est plein et que de l'espace doit être libéré dans le cache, un bloc qui n'est pas marqué comme étant modifié est simplement supprimé du cache, alors qu'un bloc modifié doit être écrit sur le disque. Si le système applique une stratégie classique de remplacement des blocs du cache (par exemple la stratégie dite *LRU*, pour *Least Recently Used*), un bloc modifié sera tôt ou tard la cible d'un remplacement, et les modifications qu'il contient seront écrites sur le disque. Une autre possibilité (peu employée) consiste à épinglez (*pin* en anglais) une page dans le cache pour interdire son remplacement et donc son écriture. Cette stratégie paresseuse vise à n'effectuer une écriture que quand elle devient nécessaire. Avec un cache suffisamment, la partie de la base la plus utilisée n'est en fait jamais écrite sur le disque tant que le serveur n'est pas arrêté. Toutes les lectures et écritures ont lieu en mémoire.

Si on veut concilier à la fois de bonnes performances par limitation des entrées/sorties et la garantie de reprise sur panne, on réalise rapidement que le problème est plus compliqué qu'il n'y paraît. Voici par exemple un premier algorithme, simpliste, qui ne fonctionne pas. L'idée est d'utiliser le cache pour les données modifiées (donc l'image après, cf. le chapitre *Contrôle de concurrence*), et le disque pour les données validées (l'image avant).

Algorithme simpliste :

- ne jamais écrire une donnée modifiée par une transaction T avant que le `commit` n'arrive,
- au moment du `commit` de T , forcer l'écriture de tous les blocs modifiés par T .

Pourquoi cela ne marche-t-il pas ? Pour des raisons de performance et des raisons de correction (la reprise n'est pas garantie).

- Surcharge du cache. Si on interdit l'écriture des blocs modifiés, qui peut dire que le cache ne va pas, au bout d'un certain temps, contenir uniquement des blocs modifiés et donc épinglez en mémoire ? Aucune remplacement ne devient alors possible, et le système est bloqué. Entretemps il est probable que l'on aura assisté à une lente diminution des performances due à la réduction de la capacité effective du cache.
- Ecritures aléatoires. Si on décide, au moment du `commit`, d'écrire tous les blocs modifiés, on risque de déclencher des écritures, à des emplacements éloignés, de blocs donc seule une petite partie est modifiée. Or un principe essentiel de la performance d'un SGBD est de privilégier les écritures séquentielles de blocs pleins.
- Risque sur la recouvrabilité. Que faire si une panne survient après des écritures mais avant l'enregistrement du `commit` ?

Dans le dernier cas, on ne peut simplement plus assurer une reprise. Donc cette solution est inefficace et incorrecte. On en conclut que les fichiers de la base ne peuvent pas, à eux seuls, servir de support à la reprise sur panne. Nous avons besoin d'une structure auxiliaire, le journal des transactions.

9.1.2 Le journal des transactions

Un journal des transactions (*log* en anglais) est un ensemble de fichiers complémentaires à ceux de la base de données, servant à stocker sur un support non volatile les informations nécessaires à la reprise sur panne. L'idée de base est exprimée par l'équation suivante :

L'état de la base

Etat de la base = journaux de transactions + fichiers de la base

Le journal contient les types d'enregistrements suivants :

- `start(T)`
- `write(T, x, old_val, new_val)`
- `commit`
- `rollback`
- `checkpoint`

L'enregistrement dans le journal des opérations de lectures n'est pas nécessaire, sauf pour de l'audit éventuellement. Le journal est un fichier séquentiel, avec un cache dédié, qui fonctionne selon la technique classique. Quand le cache est plein, on écrit dans le fichier et on vide le cache. Les écritures sont séquentielles et maximisent la rentabilité des entrées/sorties. On doit écrire dans le journal (physiquement) à deux occasions.

Règle du point de commit.

Au moment d'un `commit` le cache du journal doit être écrit sur le disque (écriture forcée). On satisfait donc l'équation : l'état de la base est sur le disque au moment où l'enregistrement `commit` est écrit dans le fichier journal.

Règle du point de commit.

Si un bloc du fichier de données, marqué comme modifié mais non validé, est écrit sur le disque, il va écraser l'image avant. Le risque est alors de ne plus respecter l'équation, et il faut donc écrire dans le journal pour être en mesure d'effectuer un rollback éventuel.

La figure [Gestion des écritures avec fichier journal](#) explique ce choix qui peut sembler inutilement complexe. Elle montre la structure des mémoires impliquées dans la gestion du journal des transactions. Nous avons donc sur mémoire stable (c'est-à-dire non volatile, résistante aux coupures électriques) les fichiers de la base d'une part, le fichier journal de l'autre. Si possible ces fichiers sont sur des disques différents. En mémoire centrale nous avons un cache principal stockant une image partielle des fichiers de la base, et un cache pour le fichier journal. Une donnée modifiée et validée est toujours dans le fichier journal. Elle peut être dans les fichiers de la base, mais seulement une fois que le bloc modifié est écrit, ce qui finit toujours par arriver sur la durée du fonctionnement normal d'un système. Si tout allait toujours bien (pas de panne, pas de `rollback`), on n'aurait jamais besoin du journal.

9.2 S2 : Algorithmes de reprise sur panne

Si une panne légère (pas de perte de disque) survient, il faut effectuer deux types d'opérations :

- refaire (Redo) les transactions validées avant la panne qui ne seraient pas correctement écrites dans les fichiers de la base ;
- défaire (Undo) les transactions en cours au moment de la panne, qui avaient déjà effectué des mises à jour dans les fichiers de la base.

Ces deux opérations sont basées sur le journal. On doit faire un Redo pour les transactions validées (celles pour lesquelles on trouve un `commit` dans le journal) et un Undo pour les transactions actives (celles qui n'ont ni `commit`, ni `rollback` dans le journal).

9.2.1 La notion de checkpoint

En cas de panne, il faudrait en principe refaire toutes les transactions du journal, depuis l'origine de la création de la base, et défaire celles qui étaient en cours. Au moment d'un checkpoint, le SGBD écrit sur disque tous les blocs modifiés, ce qui garantit que les données validées par `commit` sont dans la base. Il devient inutile de faire un Redo pour les transactions validées avant le checkpoint.

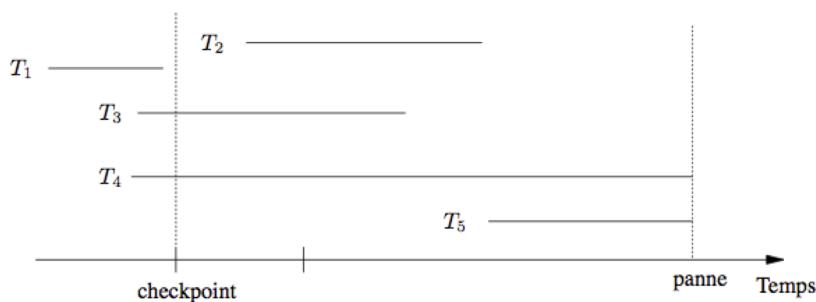


Fig. 9.2 – Reprise sur panne après un *checkpoint*

La figure [Reprise sur panne après un checkpoint](#) montre un exemple, avec un *checkpoint* survenant après la validation de *T₁*. Toutes les mises à jour de *T₁* ont été écrites dans les fichiers de la base au moment du checkpoint, et il est donc inutile d'effectuer un Redo pour cette transaction. Pour toutes les autres le Redo est indispensable. Les mises à jour de *T₂* par exemple, bien que validées, peuvent rester dans le *cache* sans être écrites dans les fichiers de la base au moment de la panne. Elles sont alors perdues et n'existent que dans le journal.

Un *checkpoint* prend du temps : il faut écrire sur le disque tous les blocs modifiés. Sa fréquence est généralement paramétrable par l'administrateur. Il est indispensable de maîtriser la taille des journaux de transactions qui, en l'absence de mesures de maintenance (comme le *checkpoint*) ne font que grossir et peuvent atteindre des volumes considérables.

9.2.2 Avec mises à jour différées

Voici maintenant un premier algorithme correct, qui s'appuie sur l'interdiction de toute écriture d'un bloc contenant des mises à jour non validées. L'intérêt est d'éviter d'avoir à effectuer des Undo à partir du journal. L'inconvénient est de devoir épingle des pages en mémoire, avec un risque de se retrouver à court d'espace disponible.

Au moment d'un `commit`, on écrit d'abord dans le journal, puis on retire l'épingle des données du cache pour qu'elles soient écrites. Il n'y a jamais besoin de faire un Undo. C'est un algorithme NO-UNDO/REDO :

- on constitue la liste des transactions validées depuis le dernier *checkpoint* ;
- on prend les entrées `write` de ces transactions dans l'ordre de leur exécution, et on s'assure que chaque donnée x a bien la valeur `new_val`.

On peut aussi refaire les opérations dans l'ordre inverse, en s'assurant qu'on ne refait que la dernière mise à jour validée.

L'opération de Redo est idempotente : on peut la réexécuter autant de fois qu'on veut sans changer le résultat de la première exécution. C'est une propriété nécessaire, car la reprise sur panne elle-même peut échouer !

9.2.3 Avec mise à jour immédiate

Dans ce second algorithme (de loin le plus répandu), on autorise l'écriture de blocs modifiés. Dans ce cas il faut défaire les mises à jours de transactions annulées. Il existe deux variantes :

- Avant un `commit`, on force les écritures dans la base : il n'y a jamais besoin de faire un Redo (Undo/No-Redo)
- Si on ne force pas les écritures dans la base, c'est un algorithme Undo/Redo, le plus souvent rencontré car il évite le flot d'écriture aléatoires à déclencher sur chaque `commit`.

L'algorithme se décrit simplement comme suit :

- on constitue la liste des transactions actives L_A et la liste des transactions validées L_V au moment de la panne ;
- on annule les écritures de L_A avec le journal : attention les annulations se font dans l'ordre inverse de l'exécution initiale ;
- on refait les écritures de L_V avec le journal.

Notez qu'avec cette technique on ne force jamais l'écriture des données modifiées (sauf aux *checkpoints*) donc on attend qu'un *flush* (mise sur disque des blocs modifiés) intervienne naturellement pour qu'elles soient placées sur le disque.

9.2.4 Journaux et sauvegardes

Le journal peut également servir à la reprise en cas de perte d'un disque. Il est cependant essentiel d'utiliser deux disques séparés. Les sauvegardes binaires (les fichiers de la base), associées aux journaux des mises à jour, vérifient en effet l'équation suivante :

Règle 3

Etat de la base = sauvegarde binaire + journaux des mises à jour

En ré-exécutant ces modifications à partir d'une sauvegarde, on récupère l'état de la base au moment de la panne d'un disque. Deux cas se présentent : panne du disque contenant le journal (appelons-le D_l) et panne du disque contenant les fichiers de la base (appelons-le D_b).

Panne du disque du log

Si D_l tombe en panne, l'état de la base peut être reconstitué en effectuant toutes les écritures des blocs modifiés du *cache*. On obtient alors des fichiers sur le disque D_b contenant toutes les mises à jour. Le bon reflexe est donc d'arrêter proprement le système, ou d'utiliser une commande *flush* si elle existe. Il n'y a plus ensuite qu'à effectuer des sauvegardes et les réparations matérielles nécessaires.

Panne du disque contenant les fichiers de la base

Le cas est un peu plus délicat. Il faut en fait effectuer une reprise sur panne à partir des journaux, en appliquant les Redo et Undo à la dernière sauvegarde disponible.

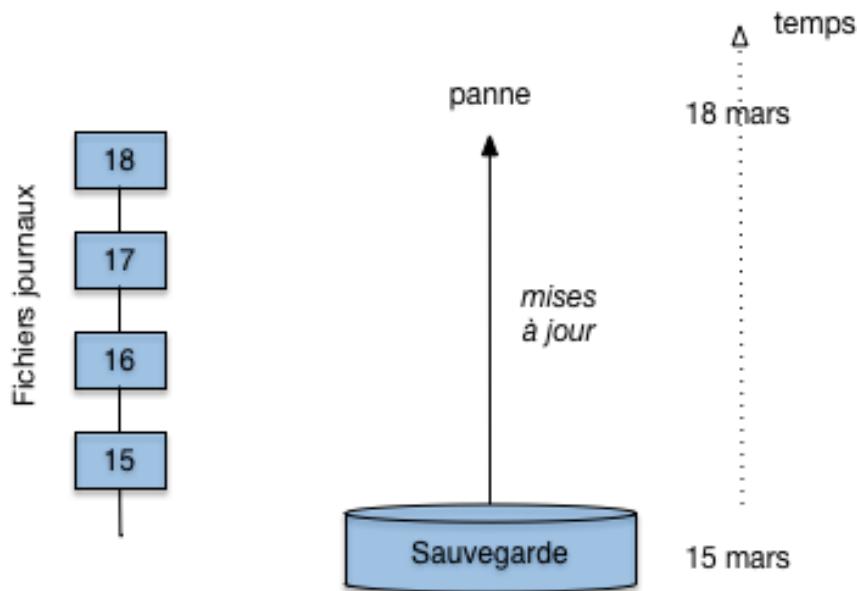


Fig. 9.3 – Reprise à froid avec une sauvegarde et des fichiers *log*

La figure *Reprise à froid avec une sauvegarde et des fichiers log* montre une situation classique, avec un sauvegarde effectuée le 15 mars, des fichiers journaux avec un *checkpoint* quotidien, chaque *checkpoint* entraînant la création d'un fichier physique supplémentaire. En théorie seul le dernier fichier journal est utile (puisque seules les opérations depuis le dernier *checkpoint* doivent être refaites). C'est vrai seulement pour des reprises à chaud, après coupure de courant. En cas de perte d'un disque tous les fichiers journaux depuis la dernière sauvegarde sont nécessaires.

Il faut donc que l'administrateur réinstalle un disque neuf et y place la sauvegarde du 15 mars. Il demande ensuite au système une reprise sur panne depuis le 15 mars, en s'assurant que les fichiers journaux sont bien disponibles depuis cette date. Sinon l'état de la base au 18 mars ne peut être récupéré, et il faut repartir de la sauvegarde.

On réalise l'importance des journaux et de leur rôle pour le maintien des données. Un soin tout particulier (sauvegardes fréquentes, disques en miroir) doit être consacré à ces fichiers sur une base sensible. Autant la reprise peut s'effectuer automatiquement après une panne légère, de type coupure d'électricité, autant elle demande des interventions de l'administrateur, parfois délicates, en cas de perte d'un disque. On parle respectivement de reprise à chaud et de reprise à froid. Bien entendu les procédures de reprise doivent être testées et validées avant qu'un vrai problème survienne, sinon on est sûr de faire face dans la panique à des difficultés imprévues.

Voici comment fonctionne alors la journalisation.

- Au moment d'un *commit*. Juste avant le *commit*, le système écrit dans le journal l'image après des enregistrements modifiés par la transaction. Pourquoi ne pas écrire les pages contenant les enregistrements modifiés ? Pour plusieurs raisons, la principale étant que s'il y a *n* enregistrements modifiés, répartis dans (au pire) *n* pages, il faut écrire ces *n* pages sur le disque, souvent sans contiguïté, ce qui est très coûteux.
En revanche ces *n* enregistrements peuvent être regroupés dans un petit nombre de pages du buffer du journal puis écrits séquentiellement dans ce dernier. En résumé cette technique est beaucoup plus performante.
- Quand le buffer principal est plein. Il faut alors replacer sur le disque certaines pages du buffer principal (*flush*). Si une page contient l'image après non validée d'un enregistrement, InnoDB risque de perdre l'image avant et donc d'être incapable d'effectuer un *rollback* en cas d'annulation ou de panne. L'image avant est donc au préalable écrite dans le journal.

Les enregistrements validés sont simplement écrits sur le disque. Leur image dans le buffer principal et sur le disque redevient donc synchronisée.

Il existe également forcément une version de cette image après dans le journal, qui n'est plus utile que pour les lectures cohérentes des transactions qui auraient commencé avant la modification de l'enregistrement.

- Au moment d'un `rollback`, InnoDB remplace les images après par les images avant, soit stockées sur le disque, soit placées dans le journal après un *flush*.

Avec cet algorithme, toutes les données validées sont toujours sur disque, soit dans les fichiers de la base, soit dans le journal. Ce dernier peut contenir aussi bien des enregistrements validés, présents dans le buffer principal, mais pas encore *flushés* dans les fichiers de la base, que des enregistrements de l'image avant qui ont été remplacés par leur image après suite à un *flush*.

Indices and tables

- genindex
- modindex
- search