Cours de bases de données – Aspects système

Version Février 2023

Philippe Rigaux

févr. 07, 2023

Table des matières

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 constitue, sous le titre de « Aspects systèmes », la seconde partie d'un cours complet consacré aux bases de données relationnelles.

— La version en ligne du présent support est accessible à http://sys.bdpedia.fr,
— la version imprimable (PDF) est disponible à http://sys.bdpedia.fr/files/cbd-sys.pdf,
— la version pour liseuse / tablette est disponible à http://sys.bdpedia.fr/files/cbd-sys.epub (format EPUB).

Ce support a pour auteur Philippe Rigaux, Professeur au Cnam. Je suis également l'auteur de trois autres cours, aux contenus proches :

— Un cours sur le modèle relationnel et SQL à http://sql.bdpedia.fr.
— 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.

CHAPITRE 1

Introduction

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 systèmes non-relationnels, vaguement rassemblés sous le terme générique « NoSQL » reprennent une partie des techniques utilisées par les systèmes relationnels, mais en diffèrent par deux aspects essentiels : l'absence d'un langage structuré et normalisé d'interrogation (et donc des techniques d'optimisation qui l'accompagnent), avec en contrepartie un grande facilité de passage à l'échelle par distribution. Reportez-vous à http://b3d.bdpedia.fr.

Le présent support de cours propose d'aller "sous le capot" des systèmes relationnels 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 impressionnants 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 amené à mettre en place des applications professionnelles s'appuyant sur une base de données (soit une classe d'application extrêmement courante).

Il semble difficile de comprendre le contenu du cours sans avoir au préalable étudié les concepts principaux du modèle relationnel, et notamment le langage SQL. Reportez-vous au support http://sql.bdpedia.fr si vous avez un doute.

1.1 Contenu et plan du cours

Le cours est constitué d'un ensemble de chapitres consacrés aux techniques implantées dans les systèmes relationnels, et plus précisément

- les *méthodes de stockage* qui exploitent les ressources physiques de la machine pour assurer la disponibilité et la sécurité des bases de données ;
- les *structures de données*, parfois sophistiquées, utilisées par obtenir de très bonnes performances même en présence de très gros volumes ;
- les *algorithmes* et *protocoles* que l'on trouve à différents niveaux pour garantir un comportement robuste et efficace du système : optimisation des requêtes, contrôle de concurrence, gestion des pannes.

Le cours comprend trois parties consacrées successivement au stockage et aux structures de données, au méthodes et algorithmes d'optimisation, et enfin aux transactions et à la reprise sur panne.

1.2 Apprendre avec ce cours

Le cours est découpé en *chapitres*, couvrant un sujet bien déterminé, et en *sessions*. J'essaie de structurer les sessions pour que les concepts principaux puissent être présentés dans une vidéo d'à peu près 20 minutes. J'estime que chaque session demande 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 EPUB.
- Le suivi du cours consacré à la session, soit en vidéo, soit en présentiel.
- La réponse au Quiz proposant des QCM sur les principales notions présentées dans la session. Le quiz permet de savoir si vous avez compris : si vous ne savez pas répondre à une question du Quiz, il faut relire le texte, écouter à nouveau la vidéo, approfondir.
- La pratique avec les travaux pratiques en ligne proposés dans plusieurs chapitres.
- Et enfin, la réalisation des exercices proposés en fin de chapitre.

Note : Au Cnam, ce cours est proposé dans un environnement de travail Moodle avec forum, corrections en lignes, interactions avec l'enseignant.

Tout cela constitue autant de manière d'aborder les concepts et techniques présentées. Lisez, écoutez, pratiquez, recommencez autant de fois que nécessaire jusqu'à ce que vous ayez la conviction de maîtriser l'esentiel du sujet abordé. Vous pouvez alors passer à la session suivante. La réalisation des exercices est essentielle pour vérifier que vous maîtrisez le contenu.

Les définitions

Pour vous aider à identifier l'essentiel, la partie rédigée du cours contient des définitions. Une définition n'est pas nécessairement difficile, ou compliquée, mais elle est toujours importante. Elle identifie un concept à connaître, et vise à lever toute ambiguité sur l'interprétation de ce concept (c'est comme ça et pas autrement, « par définition »). Apprenez par cœur les définitions, et surtout comprenez-les.

La suite de ce chapitre comprend une unique session avec tout son matériel (vidéos, exercices), consacrée au positionnement du cours.

1.3 S1 : rappels

Supports complémentaires :
- Diapositives: rappels
- Vidéo

Nous commençons par un court rappel des notions de base que vous *devez* maîtriser avant d'aborder la suite de ce cours. Tout ce qui suit est détaillé dans le support sur les modèles que vous trouverez sur le site http://sql.bdpedia.fr. Reportez-vous à ce texte si vous avez des doutes sur vos acquis.

1.3.1 Bases de données et SGBD

Commençons par une vision d'ensemble des composants constituant un système relationnel et ses applications, vision illustrée par la Fig. 1.1. On distingue trois niveaux, soit, de droite à gauche, le niveau *physique*, le niveau *logique* et enfin le niveau *applicatif*. Le niveau physique est celui qui nous intéresse essentiellement dans ce cours ; le niveau logique est supposé connu ; le niveau applicatif ne nous concerne pas ou de manière très indirecte.

Le niveau physique comprend la *base de données* proprement dite, qui n'est rien d'autre qu'un ensemble de fichiers stockés sur un support persistant (un disque magnétique généralement). Ces fichiers contiennent des données dites *structurées*, par opposition à des systèmes stockant des documents sans forme prédéfinie (cas par exemple d'un moteur de recherche). Dans le cas des systèmes relationnels, les fichiers de la base contiennent
- soit la représentation binaire des tables relationnelles ; chaque table est constituée d'un ensemble d'"*enregistrements*, un par ligne ;
- soit des *index*, structures de données permettant d'accélérer les opérations sur les données, et notamment les recherches.

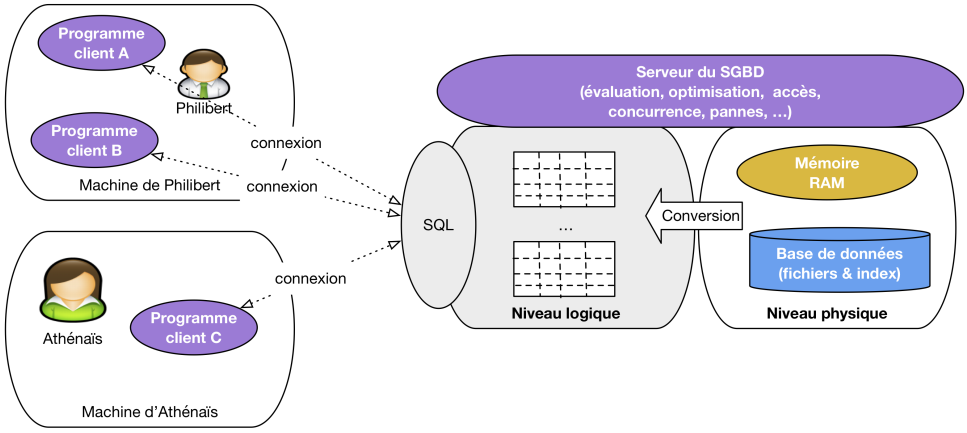

Fig. 1.1 – Les composants d'un système relationnel

La *persistance* du support de stockage garantit la préservation de la base de données indépendamment des applications qui y accèdent. Une base de données continue à exister même quand la machine qui l'héberge est arrêtée, et tout est fait pour qu'elle puisse se conserver à long terme, par des procédures avancées de protection

et de gestion des pannes. En contrepartie, les supports persistants étant beaucoup moins performants que les mémoires RAM, des stratégies sophistiquées d'accès sont nécessaires pour obtenir des bonnes performances.

Une base de données n'est jamais accessible directement pas une application car les fonctionnalitées évoquées ci-dessus (sécurité, performances) ne pourraient pas être assurées. Un système logiciel, le SGBD (système de gestion de bases de données) est chargé de prendre en charge *tous* les accès à la base. Le SGBD assure notamment

— la gestion des ressources physiques : lecture et écriture dans les fichiers, maintenance des index, transferts entre mémoire secondaire (le disque) et mémoire RAM ;
— l'exécution efficace des opérations requises par les applications : lectures et mises à jour ;
— la gestion ordonnée des accès concurrents ;
— la sécurisation des données et notamment la gestion des pannes.

Par ailleurs, le SGBD présente aux applications les données selon un modèle qui fait abstraction de tous les détails techniques de la représentation physique. Dans les SGBD relationnels, ce modèle comprend des *tables* constituées de *lignes*.

Le *niveau logique* est celui de la présentation des données selon ce modèle. La distinction entre niveau logique et niveau physique offre d'immenses avantages qui expliquent en grande partie le succès des systèmes relationnels et la très grande simplification qu'ils ont apporté à la gestion de données informatiques.

— le niveau logique offre une très grande simplication par rapport à la complexité du codage physique ; cette simplification évite à tous ceux aui accèdent à une base de se confronter à des problèmes d'ouverture de fichiers, de codage/décodage de données binaires, et d'algorithmes de parcours complexes ;
— le niveau logique permet la définition de langages d'interrogation et de manipulation de données simples et intuitifs ;
— le niveau logique est *indépendant* du niveau physique : il est possible de réorganiser ce dernier de fond en comble sans que cela affecte aucunement les applications ; on peut par exemple changer la base de machine, de support, la réorganiser entièrement, de manière complètement transparente.

L'indépendance logique / physique est l'atout maître des systèmes relationnels. Une conséquence pratique est qu'il est possible de distinguer deux rôles distincts dans l'utilisation d'un SGBD. Les *concepteurs / développeurs* ne voient que le niveau logique et les langages associés (SQL principalement). Il peuvent se concentrer sur la qualité fonctionnelle et applicative et n'ont pas à se soucier d'aspects non-fonctionnels comme les performances, la sécurité, la fiabilité ou les accès concurrents aux ressources. Les *administrateurs* sont, eux, concernés par le niveau physique et le réglage du système pour obtenir le comportement le plus satisfaisant possible.

C'est à ce second aspect qu'est consacré le support qui suit. Tout ce qui relève du niveau physique y est détaillé et expliqué. Les caractéristiques du modèle relationnel sont, elles, supposées connues et brièvement rappelées ci-dessous.

1.3.2 Le modèle relationnel

Les structures du niveau logique définissent une modélisation des données : on peut envisager par exemple des structures de graphe, d'arbre, de listes, etc. Le modèle relationnel se caractérise par une modélisation basée sur une seule structure, la table. Cela apporte au modèle une grande simplicité puisque toutes les données ont la même forme et obéissent aux même contraintes.

Pour rappeler les caractéritiques essentielles du modèle relationnel, nous allons nous appuyer sur un des exemples que nous allons traiter dans ce support de cours, celui d'une base stockant des informations sur

des films, leurs acteurs et réalisateurs. Une telle base comprend plusieurs tables. Voici un extrait de celle des artistes.

id	nom	prénom	année
130	Eastwood	Clint	1930
131	Hackman	Gene	1930
132	Scott	Tony	1930
133	Smith	Will	1968

Une table relationnelle est un ensemble de lignes, et chaque ligne est elle-même constituée d'une liste de valeurs. *Toutes les lignes ont la même structure*, et donc le même nombre de valeurs. Une ligne représente une « entité » (ici, chaque ligne représente un artiste) et chaque entité est décrite par un ensemble fixe d'attributs (le nom, le prénom). Les valeurs de chaque ligne sont donc les valeurs de ces attributs caractéristiques de l'entité représentée. La régularité de la structure permet de représenter toutes les valeurs d'un même attribut dans une colonne, et de leur attribuer un type fixe (une chaîne de caractères pour le nom, un entier pour l'année).

Un peu de vocabulaire : *table* et *ligne* sont des termes informels pour parler du contenu d'une base relationnelle. Ils sont assez peu précis car ils ne spécifient ni le niveau de représentation auquel on se place, ni la structure particulière de leur contenu. En contexte, on peut s'en satisfaire, mais pour lever quelques ambiguités on adoptera dans ce support de cours la terminologie suivante

— au niveau logique on préfèrera parler de *relation* pour désigner les tables et surtout de *nuplet* pour désigner les lignes ; un nuplet est une séquence de valeurs typées, correspondant chacune à un attribut précis ;
— au niveau physique on parlera d'*enregistrement* (*record*) pour désigner le codage binaire d'un nuplet, et de *collection* pour désigner un ensemble d'enregistrements.

Dans chaque table relationnelle on trouve un attribut particulier, la *clé primaire*. La valeur de la clé primaire permet d'identifier un unique nuplet dans la table. Elle permet donc également de référencer ce nuplet depuis une autre table. Souvent, la clé primaire est un simple numéro d'identification.

Les attributs, leur type, la clé primaire, sont des contraintes qui s'appliquent à chaque nuplet. Ces contraintes sont décrites dans un *schéma*, déclaré au moment de la création de la table. Voici la commande de création de la table *Artiste* :

```
create table Artiste  (idArtiste INTEGER NOT NULL,
                   nom VARCHAR (30) NOT NULL,
                   prénom VARCHAR (30) NOT NULL,
                   annéeNaissance INTEGER,
                   PRIMARY KEY (idArtiste));
```

On pourra représenter ce schéma par le résumé suivant, dans lequel on met **en gras** l'information essentielle, la clé primaire.

— Artiste (**idArtiste**, nom, prénom, annéeNaissance)

Voici une seconde table, celle des films.

id	titre	année	genre	idRéalisateur	codePays
20	Impitoyable	1992	Western	130	USA
21	Ennemi d'état	1998	Action	132	USA

On retrouve les mêmes caractéristiques que pour les artistes : chaque film est décrit par des valeurs d'attributs, et identifié par une clé primaire. On trouve également dans cette table une *clé étrangère* : l'attribut *idRéalisateur* prend pour valeur l'identifiant d'un artiste. Cet identifiant est la valeur d'une clé primaire dans la table *Artiste*. La clé étrangère est donc un référencement, dans un nuplet (ici un film) d'un autre nuplet. Ce mécanisme permet de savoir que, par exemple, le réalisateur du film *Impitoyable* est Clint Eastwood. La clé primaire de ce dernier (130) est également clé étrangère dans la table *Film*.

Il est important de noter que le référencement n'est pas de nature « physique ». Il n'y a pas de « pointeur » qui lie les deux nuplets. C'est uniquement par calcul, au moment de l'exécution des requêtes, que l'on va comparer les valeurs respectives des clés étrangère et primaire et effectuer le rapprochement. Ce principe du calcul à la place d'un codage « en dur » est conforme à celui de l'indépendance logique/physique évoqué ci-dessus, et a un fort impact sur les algorithmes d'évaluation de requêtes.

Les clés étrangères sont des contraintes, et comme telles décrites dans le schéma comme le montre la commande de création de la table *Film*.

```
create table Film  (idFilm integer NOT NULL,
              titre    varchar (80) NOT NULL,
              année    integer NOT NULL,
              genre varchar (20) NOT NULL,
              idRéalisateur    integer,
              codePays    varchar (4),
              primary key (idFilm),
              foreign key (idRéalisateur) references Artiste(idArtiste),
              foreign key (codePays) references Pays(code));
```

Vous devriez maîtriser sur la modélisation relationnelle pour aborder l'évaluation et l'exécution de requêtes. Voici, en résumé, le schéma de notre base des films.

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

Et pour compléter l'illustration des liens clé primaire / clé étrangère, voici un extrait de la table des rôles qui consiste ensentiellement en identifiants établissant des liens avec les deux tables précédentes. À vous de les décrypter pour comprendre comment toute l'information est représentée. Que peut-on dire de l'artiste 130 par exemple ? Peut-on savoir dans quels films joue Gene Hackman ? Qui a mis en scène *Impitoyable* ?

idFilm	idArtiste	nomRôle
20	130	William Munny
20	131	Little Bill
21	131	Bril
21	133	Robert Dean

Cette base est disponible en ligne à http://deptfod.cnam.fr/bd/tp.

1.3.3 Les langages

Un modèle, ce n'est pas seulement une ou plusieurs structures pour représenter l'information indépendamment de son format de stockage, c'est aussi un ou plusieurs langages pour interroger et, plus généralement, interagir avec les données (insérer, modifier, détruire, déplacer, protéger, etc.). Le langage permet de construire les commandes transmises au serveur.

Un langage relationnel sert à construire des expressions (les « requêtes ») qui s'appuient sur une base de données en entrée et fournissent une table en sortie. Deux langages d'interrogation, à la fois différents, complémentaires et équivalents ont été définis pour le modèle relationnel :

1. Un langage *déclaratif*, basé sur la logique mathématique.
2. Un langage *procédural*, et plus précisément *algébrique*, basé sur la théorie des ensembles.

Un langage est *déclaratif* quand il permet de spécifier le résultat que l'on veut obtenir, sans se soucier des opérations nécessaires pour obtenir ce résultat. Un langage algébrique, au contraire, consiste en un ensemble d'opérations permettant de transformer une ou plusieurs tables en entrée en une table - le résultat - en sortie.

Ces deux approches sont très différentes. Elles sont cependant parfaitement complémentaires. l'approche déclarative permet de se concentrer sur le raisonnement, l'expression de requêtes, et fournit une définition rigoureuse de leur signification. L'approche algébrique nous donne une boîte à outil pour calculer les résultats.

Le langage déclaratif : SQL

Le langage SQL, assemblant les deux approches est une syntaxe pratique pour le langage relationnel déclaratif (lequel est une variante de la logique des prédicats). Il est utilisé depuis les années 1970 dans tous les systèmes relationnels, et il paraît tellement naturel et intuitif que même des systèmes construits sur une approche non relationnelle tendent à reprendre ses constructions.

SQL exprime des requêtes comme des formules que doivent satisfaire les nuplets du résultat. Voici deux exemples :

```
select titre
from Film
where année = 2016
```

On note qu'il n'y a aucune référence à la méthode qui permet de calculer le résultat. Comme nous le verrons, il peut en exister plusieurs en fonction de l'organisation de la base, et c'est le système qui choisit la meilleure. C'est une illustration du principe d'indépendance logique / physique.

Le second exemple est une jointure :

```
select titre, prénom, nom
from Film as f, Artiste as a
where f.idRéalisateur = a.idArtiste
and  année = 2016
```

On assemble des nuplets partageant une propriété commune, ici l'identifiant de l'artiste, représenté comme clé primaire dans la table *Artiste* et comme clé étrangère dans la table *Film*. Ici encore, aucune procédure de calcul n'est indiquée, et ici encore le système a plusieurs choix et effectuera celui qui lui semble le meilleur.

Le langage procédural : l'algèbre

Comment le SGBD peut-il inférer une procédure de calcul à partir d'une requête SQL ? Par quelle démarche peut-il déterminer quel algorithme appliquer pour une requête SQL donnée ? La réponse est dans un langage intermédiaire, l'algèbre relationnelle. L'algèbre est un ensemble d'opérateurs donc chacun prend en entrée une ou deux tables et produit en sortie un table. Ces opérateurs sont :

— La sélection, dénotée σ
— La projection, dénotée π
— Le renommage, dénoté ρ
— Le produit cartésien, dénoté $\times$
— L'union, $\cup$
— La différence, $-$

En les *composant*, on construit des requêtes qui s'interprètent comme des séquences d'opérations à appliquer à la base. Il ne reste plus alors au système qu'à choisir le bon algorithme pour chaque opérateur, et la méthode d'évaluation (on parle de *plan d'exécution*) de la requête en découle.

L'algèbre a un pouvoir d'expression identique à celui du langage déclaratif, et toute requête SQL peut donc se transcrire en une expression algébrique. Voici l'expression correspondant à la première requête SQL ci-dessus.

$$\pi_{titre}(\sigma_{année=2016}(Film))$$

On compose une sélection (σ) pour trouver les films parus en 2016, suivi d'une projection (π) pour ne conserver que le titre.

La jointure s'exprime en algèbre ainsi :

$$\pi_{titre,prénom,nom}(\sigma_{année=2016}(Film) \bowtie_{idRéalisateur=idArtiste} Artiste)$$

Il reste à choisir l'algorithme pour la sélection (σ) et pour la jointure ($\bowtie$). Ce choix est dicté par des objectifs de performance, et constitue donc la base de processus *d'optimisation de requête*, qu'il est important de comprendre et auquel nous consacrons une partie sigificative de ce support.

Voici, en très résumé, ce que vous êtes censés connaître au moment d'aborder la suite. Les notions de niveaux logique et physique, les principes de conception des schémas relationnels, SQL et l'algèbre sont les fondements pour aborder les aspects systèmes des bases de données.

Vous pouvez vous tester avec le quiz qui suit avant d'entrer dans le cœur du sujet.

1.3.4 Quiz

CHAPITRE 2

Dispositifs de stockage

Une base de données est constituée, matériellement, d'un ou plusieurs *fichiers* 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 nettement supérieures, et le coût 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 possible 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 que celui d'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 session 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 session les principales techniques de gestion de la mémoire utilisées par un SGBD. La troisième session présente les principes d'organisation des fichiers de base de données.

2.1 S1 : Supports de stockage

Supports complémentaires :

— Diapositives: supports de stockage
— Vidéo sur les dispositifs de stockage

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 (Fig. 2.1), allant de la mémoire la plus petite mais la plus efficace à la mémoire la plus volumineuse mais la plus lente.

1. la *mémoire cache* est 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 mémoire persistante ; 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 peformances 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.

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 transférées 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.

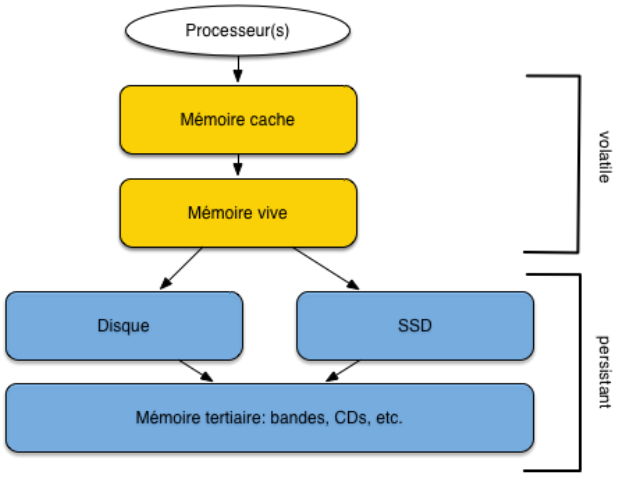

Fig. 2.1 – Hiérarchie des mémoires

Vocabulaire : Mais qu'est-ce qu'une « donnée » ?

Le terme de *donnée* désigne le codage d'une unité d'information. Dans le contexte de ce cours, « donnée » sera toujours synonyme de nuplet (ligne dans une table).

On parlera *d'enregistrement* pour désigner le codage binaire d'un nuplet, dans une perspective de stockage.

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 à 16 Go pour un ordinateur personnel, plusieurs dizaines de Go pour un serveur de données, plusieurs centaines pour de très gros serveurs). 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'un enregistrement, 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 (et principalement sur un disque), on accède à la donnée stockée à cette adresse.
— *Accès séquentiel* : on parcours la mémoire (et principalement un disque) dans un certain ordre en lisant les enregistrements 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'acès 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 *cache* (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 magnétiques 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.

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 enregistrements. 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) 4 096 octets (huit secteurs) ou 8 192 octets (seize secteurs).

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

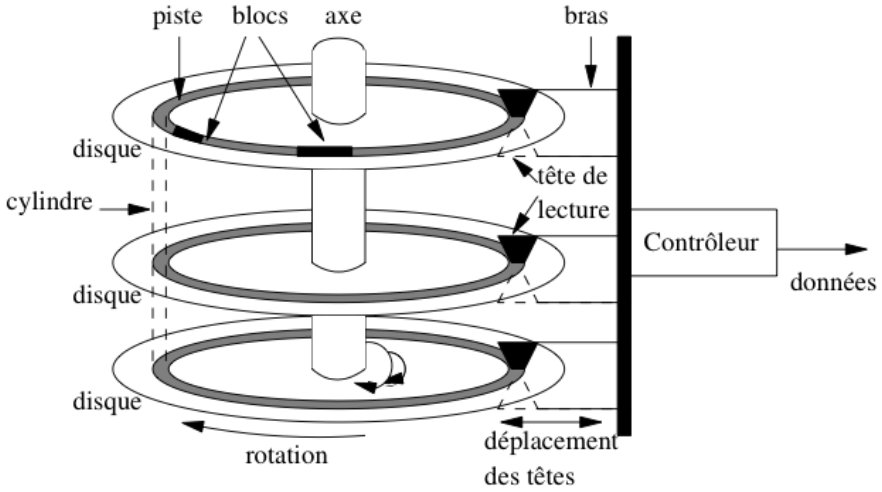

Fig. 2.2 – Fonctionnement d'un disque magnétique.

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.

Définition : bloc

Un *bloc* est une zone mémoire contigue de taille fixe 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 Fig. 2.2). Il y a autant de têtes de lectures que de disques (deux fois plus si les disques sont à double face) et toutes les têtes sont positionnées solidairement dans leur plan de déplacement. À tout moment, les pistes accessibles par les têtes sont donc les mêmes pour tous les disques de la pile, ce qui constitue une contrainte dont il faut savoir tenir compte quand on cherche à optimiser le placement des données.

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

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 donnée 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 2.2 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 mileu 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 $58115 \times 512 = 29$ Mégaoctets. 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 2.2. 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 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 magnétiqe 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 Quiz

— Pourquoi faut-il placer une base de données sur une mémoire persistante ?
— Qu'entend-on par *accès direct*
— Qu'entend-on par *accès aléatoire*
— Quelle affirmation sur les blocs est vraie :
— Pourquoi vaut-il mieux lire une collection d'enregistrements par un accès séquentiel plutôt que par un ensemble d'accès aléatoires.
— Quelle définition du *délai de latence* est correcte
— Quelle définition du *délai de positionnement* est correcte

2.2 S2 : Gestion des mémoires

Supports complémentaires :
— Diapositives: gestion des mémoires
— Vidéo sur la gestion des mémoires

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

La Fig. 2.3 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 appelerons *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).

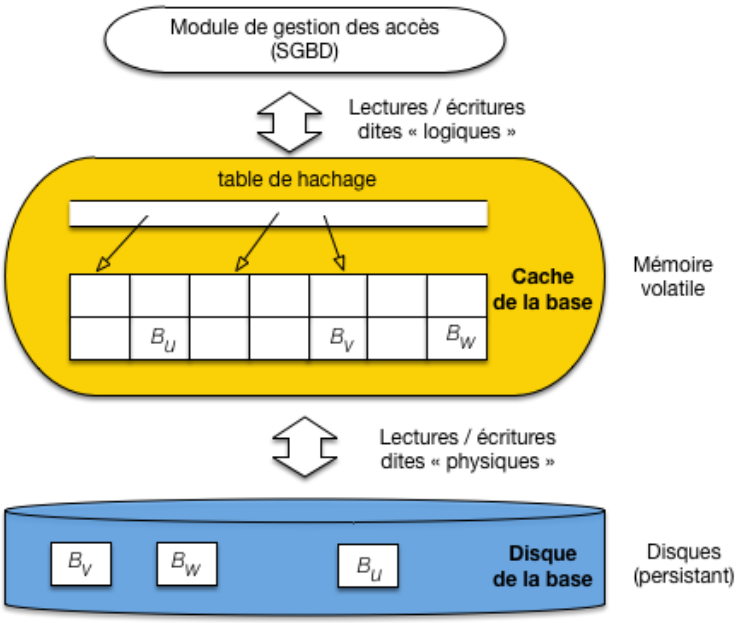

Fig. 2.3 – Le *cache* et le disque, ressources mémoires allouées au SGBD

Les enregistrements 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, et nous allons considérer comme acquis que ces requêtes comprennent *toujours* l'adresse du bloc. Cette section explique comment ces requêtes sont exécutées.

2.2.1 Les lectures

Comme le montre la Fig. 2.3, 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 :

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

La demande d'accès est appelée *lecture logique* : elle ignore si l'enregistrement est présent 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*, et donc un temps d'attente considérablement supérieur.

Note : Comment faire pour savoir si un bloc est ou non en *cache* ? Grâce à une table de hachage (illustrée sur la Fig. 2.3) qui pointe sur les blocs chargés en mémoire. L'accès à un bloc du cache, 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*. Une bonne organisation va minimiser le nombre de

lectures physiques par rapport au nombre de lectures logiques. On mesure cette efficacité avec un paramètre nommé 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 (tous les enregistrements demandés 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 lus avec une probabilité uniforme, mais en pratique certains blocs sont demandés beaucoup plus souvent que d'autres, et les écarts sont de plus variables dans le temps. Le *hit ratio* représente justement la capacité du système à stocker dans le cache les pages les plus lues pendant une période donnée.

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 lisent les blocs 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.

Supposons que 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 40 % des autres cas 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 un peu plus de 0,6. En effet les 200 Mo correspondant à 60 % des lectures vont finir par se trouver placées en cache ($200\ Mo\ =\ 333 \times 0,6$), 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.

En augmentant la mémoire cache au-delà de 333 Mo, on améliorera peu le *hit ratio* puisque les 40 % de lectures uniformes garderont peu de chances de trouver le bloc demandé en *cach*.

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'applatisse. 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 dans le *cache* le bloc contenant l'enregistrement, ou on le charge dans le *cache* s'il n'y est pas déjà ;
- on effectue la mise à jour sur l'enregistrement dans le *cache* ;

La situation est alors celle de la Fig. 2.4 : le bloc contenant l'enregistrement est marqué 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.

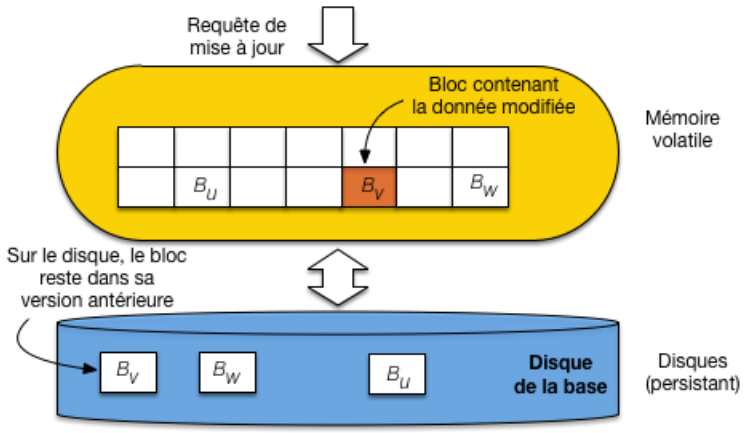

Fig. 2.4 – Gestion naïve des écritures : les blocs doivent être écrits en ordre aléatoire.

Il faut bien réaliser qu'un bloc peut contenir des *centaines* d'enregistrements, et que déclencher une écriture sur disque dès que *l'un* d'entre eux est modifié 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 Fig. 2.5. 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 d'écriture dans 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.

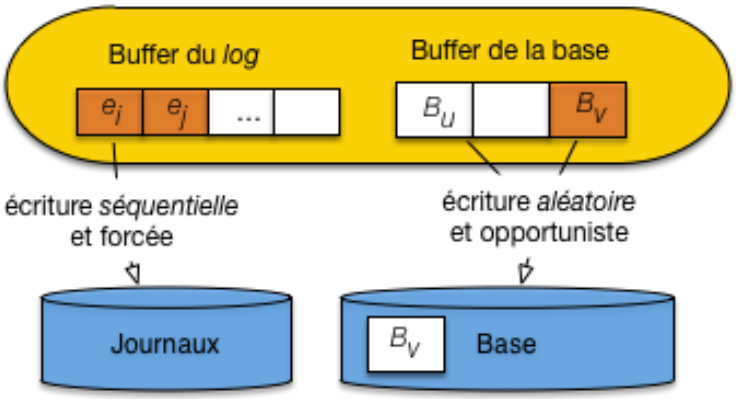

Fig. 2.5 – Gestion des écritures avec fichier journal

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.

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* enregistrements modifiés, 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 les 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 Fig. 2.6 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 rassemblés dans deux blocs qui sont consécutifs sur une même piste du disque.

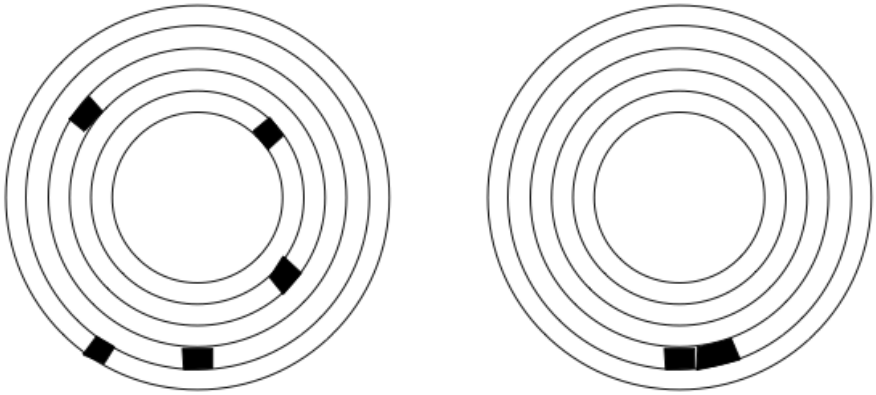

Fig. 2.6 – Mauvaise et bonne organisation sur un disque.

La lecture dans le premier cas implique 5 déplacements des têtes de lecture, et 5 délais de latence ce qui donne un temps de $5 \times (5.2 + 3) = 41$ 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'êtres 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 ensembles ;
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 concurramment. 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

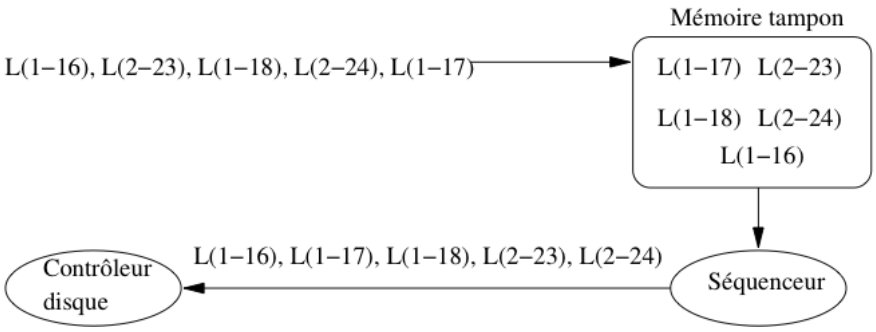

Fig. 2.7 – Séquencement des entrées/sorties

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.

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 Fig. 2.7 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 replacer sur le disque (opérations dite 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 Quiz

— Que contient le *cache* ?
— Un *cache* peut-il être plus grand que la base sur le disque ?
— Peut-il y avoir plus de lectures physiques de de lectures logiques ?
— Peut-il y avoir plus de lectures logiques de de lectures physiques ?
— Une mise à jour se fait :
— Quel est le danger d'effectuer une mise à jour dans le *cache* et pas sur le disque ?
— Parmi les arguments contre l'écriture immédiate d'un bloc contenant un enregistrement modifié, lequel vous semble faux ?
— Quel est le critère de proximité le plus précis parmi les suivants

2.3 S3 : Enregistrements, blocs et fichiers

Supports complémentaires :
— Diapositives: Enregistrements, blocs et fichiers
— Vidéo sur les enregistrements, blocs et fichiers

Pour le système d'exploitation, un fichier est une séquence d'octets sur un disque. Les fichiers gérés par un SGBD sont un peu plus structurés. Ils sont constitués de *blocs*, qui eux-même contiennent des *enregistrements* (*records* en anglais), lesquels 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 gestion de l'information : on ne descend jamais à une granularité plus fine.

2.3.1 Enregistrements

Un n-uplet dans une table relationnelle est constitué d'une liste d'attributs, chacun ayant un type. À ce n-uplet est représenté physiquement, sous forme binaire, par un enregistrement, constitué de *champs* (*field* en anglais). Chaque type d'attribut détermine la taillle du champ nécessaire pour stocker une instance du type. Le Tableau 2.3 donne la taille habituelle utilisée pour les principaux types de la norme SQL, étant entendu que les systèmes sont libres de choisir le mode de stockage.

Tableau 2.3 – 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 $\leq$ M
BIT VARYING	$< 2^8$
DATE	8
TIME	6
DATETIME	14

La taille d'un enregistrement 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 à `NULL`. Nous discutons des principaux aspects de la représentation des enregistrements dans ce qui suit.

Champs de tailles fixe et variable

Comme l'indique le Tableau 2.3, les types de la norme SQL peuvent être divisés en deux catégories : ceux qui peuvent être représentés par un champ une taille fixe, et ceux qui sont représentés par un champ de taille variable.

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

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

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

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 caratères "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.

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 *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 Fig. 2.8 montre cet enregistrement : notez qu'en lisant l'en-tête, on sait calculer l'adresse de l'enregistrement suivant.

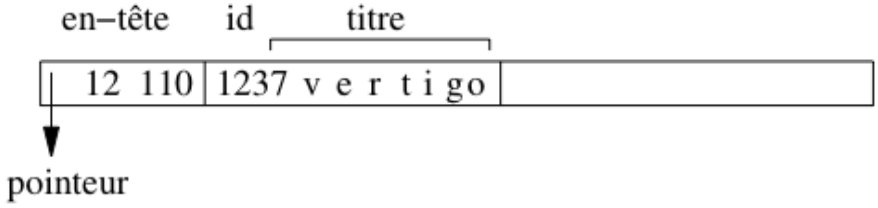

Fig. 2.8 – Représentation d'un enregistrement

2.3.2 Blocs

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

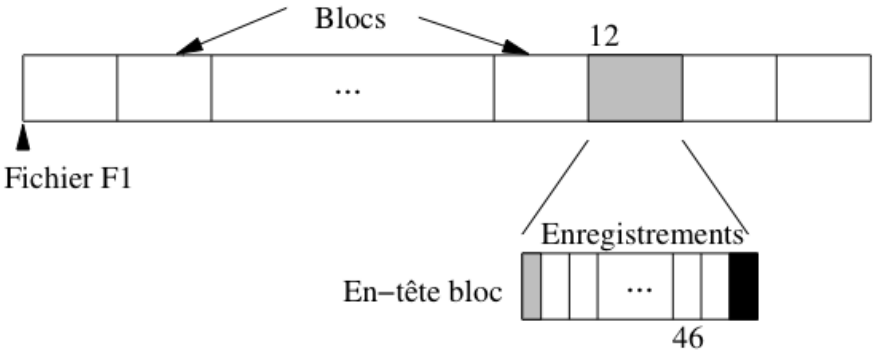

Fig. 2.9 – Stockage des enregistrements dans un bloc

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 $12 \times 47 = 564$. L'enregistrement 563 est donc l'avant-dernier du bloc, avec pour numéro interne le 46 (voir Fig. 2.9).

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 Fig. 2.10). Ce mécanisme d'indirection permet beaucoup de souplesse dans l'organisation et la réorganisation d'une base puisqu'il 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.

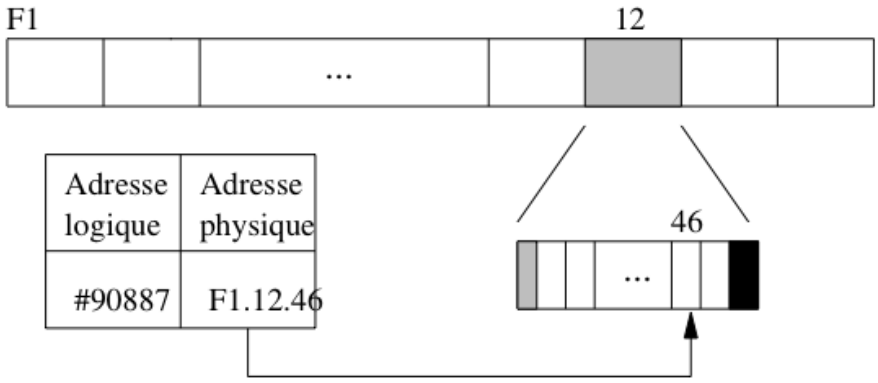

Fig. 2.10 – Adressage avec indirection

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 Fig. 2.11 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.

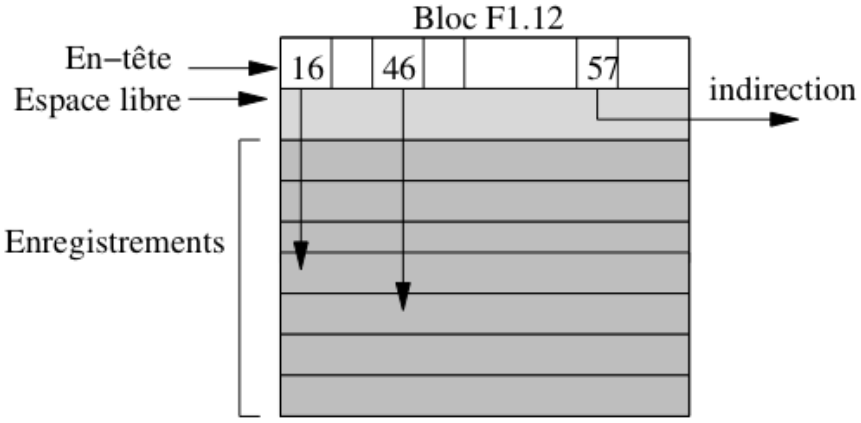

Fig. 2.11 – Combinaison adresse logique/adresse physique

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.

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.

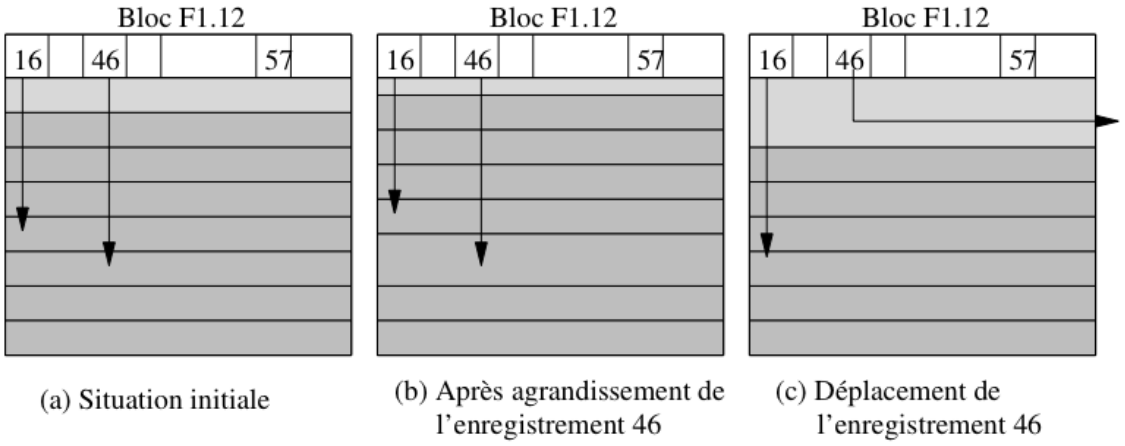

Fig. 2.12 – Mises à jour d'un enregistrement de taille variable

Considérons par exemple le scénario suivant, illustré dans la Fig. 2.12 :

- 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.

2.3.3 Fichiers

Les systèmes d'exploitation organisent les fichiers qu'ils gèrent dans une arborescence de *répertoires*. Chaque répertoire contient un ensemble de fichiers identifés 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.

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 quatres 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.

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 un 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 précèdent *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 rercherche la moitié droite ou gauche du fichier, selon le résultat de l'étape 2.

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

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

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

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

— 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.

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 Fig. 2.13). 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 Fig. 2.13, 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.

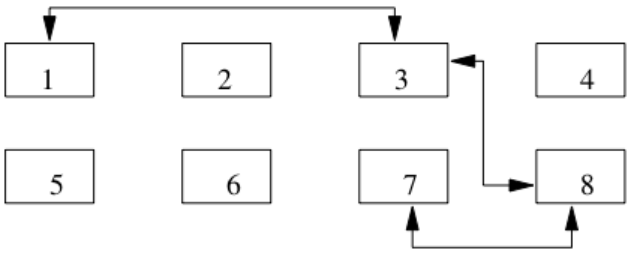

Fig. 2.13 – Gestion des blocs libres avec liste chaînée

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

La seconde solution repose sur une structure séparée des blocs du fichier. Il s'agit d'un répertoire qui donne, pour chaque page, un indicateur O/N indiquant s'il reste de l'espace, et un champ donnant le nombre d'octets (Fig. 2.14). Pour trouver un bloc avec une quantité d'espace libre donnée, il suffit de parcourir ce 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 élévée que

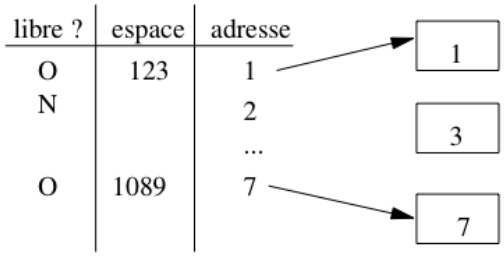

Fig. 2.14 – Gestion des blocs libres avec répertoire

celle du fichier lui-même, et on peut considérer que le temps d'accès au répertoire est négligeable comparé aux autres opérations.

2.3.4 Quiz

— Quelle est la différence entre un champ de type `varchar(25)` et un champ de type `varchar(250)` ?
— Je représente l'adresse d'un enregistrement par son numéro de bloc *B*, et son numéro interne au bloc *i* (schéma d'indirection, vu ci-dessus). Quelle réponse est vraie ?
— Un programmeur paresseux décide de simplifier la méthode d'insertion en insérant tous les nouveaux enregistrements dans le premier bloc du fichier. On connaît donc l'adresse du premier bloc, qui ne change jamais, et on ne connaît que ça. Quelle affirmation est vraie (en supposant que le système applique les méthodes de stockage vues précédemment) ?

    — Autre méthode d'insertion standard : on insère toujours dans le *dernier* bloc du fichier. On connaît donc l'adresse du dernier bloc, qui change de temps en temps, et on ne connaît que ça

— Je veux que mon fichier soit trié sur un champ, par exemple l'année du film. Quelles affirmations sont vraies ?

— Je veux stocker dans ma base, pour chaque film, son fichier vidéo MP4. Je choisis de créer un champ de type `BIT VARYING` pour le MP4. Quelle affirmation est vraie ?

2.4 Exercices

Exercice ex-lecture-disque : temps de lecture d'une base

Nous avons une base de 3 Tos.
— Quel est le temps minimal de lecture de la base complète sur un disque magnétique ?
— On veut lire 100 objets de 10 octets. Combien de temps cela prend-il s'ils sont dispersés sur un disque ?
— Même question s'ils sont en mémoire centrale

Exercice ex-seqaleatoire : lectures séquentielles et aléatoires sur un disque

On dispose d'une base de 3 Go constituée d'enregistrements dont la taille moyenne est 3 000 octets.

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

Vous pouvez prendre les valeurs du `tbl-memoires` pour les calculs.

Exercice ex-perfdisque : Spécifications d'un disque magnétique

Le Tableau 2.4 donne les spécifications partielles d'un disque magnétique. Répondez aux questions suivantes.

— Quelle est la capacité moyenne 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 ?
— Combien de temps faut-il pour transmettre le contenu d'une piste et combien de rotations peut effectuer le disque pendant ce temps ?

Tableau 2.4 – 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 moyen de secteurs par piste	40 000
Temps de positionnement moyen	10 ms
Vitesse de rotation	7 400 rpm
Déplacement de piste à piste	0,5 ms
Débit moyen	100 Mo / s

Exercice ex-deplmoyen : temps de positionnement moyen

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

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

On pourra utiliser les deux formules suivantes :

$$\Sigma_{i=1}^{n}(i) = \frac{n \times (n+1)}{2}$$

et

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

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

Exercice ex-delais : calcul latence et positionnement

Soit un disque de 5 000 cylindres tournant à 12 000 rotations par minute, avec un temps de déplacement entre deux pistes adjacentes égal à 0,002 ms et 500 secteurs de 512 octets par piste.

Quel est le temps moyen de lecture d'un bloc de 4,096 octets ? Calculer indépendamment le délai de latence, de positionnement et le temps de transfert. NB : on sait par l'exercice précédent que le nombre moyen de déplacements et 1/3 du nombre total de pistes.

Exercice ex-hitratio : 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.

Exercice ex-calculs-fichiers : quelques calculs

Soit la table de schéma suivant :

```
create table Personne (id integer not null,
          nom    varchar(40) not null,
          prenom varchar(40) not null,
          adresse varchar(70),
          dateNaissance date)
```

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

— Donner la taille maximale et la taille minimale d'un enregistrement. On suppose par la suite que tous les enregistrements ont une taille maximale.

— Quel est le nombre maximal d'enregistrements par bloc ?
— Quelle est la taille du fichier ?
— Quel est le temps moyen de recherche d'un enregistrement si les blocs sont distribués au hasard sur le disque.
— On suppose que le fichier est trié sur le nom. Quel est le temps d'une recherche dichotomique pour chercher une personne avec un nom donné ?

2.5 Atelier

Cet atelier va proposer, tout au long des chapitres du cours, d'étudier le stockage et l'optimisation d'une même base de données. Le schéma suivant, ainsi que ses statistiques, seront utilisés dans l'ensemble des exercices du cours.

— Cours (**Id_Cours**, Id_Enseignant, Intitule)
— Personne (**Id_Personne**, Nom, Prenom, Annee_Naissance)
— Salle (**Id_Salle**, Localisation, Capacite)
— Reservation (**Id_Salle, Id_Cours, Date, heure_debut, heure_fin**)

Voici la taille des différents types :

— Les identifiants (Id_…) et les nombres (ECTS, Capacite) sont codés sur 4 octets,
— les dates, années et heures sur 4 octets,
— Nom, Prenom, Intitule, Localisation sur 50 octets

Et voici les statistiques sur le contenu de la base :

— 2 200 tuples dans la table *Cours*,
— 124 000 tuples *Personne* (Contenant enseignants et auditeurs),
— 500 tuples *Salle* (Avec une capacite de 20 à 600),
— 514 000 tuples *Reservation*.

Le paramétrage du serveur de données est le suivant :

— Taille d'une adresse : 10 octets
— Le *cache* comprend 11 blocs en mémoire centrale,
— Un bloc fait 8 ko
— Un pourcentage d'espace libre (PCTFREE) de 10% est laissé dans chaque bloc

CHAPITRE 3

Structures d'index : 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 *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	(autres colonnes)
Vertigo	1958	…
Brazil	1984	…
Twin Peaks	1990	…
Underground	1995	…
Easy Rider	1969	…
Psychose	1960	…
Greystoke	1984	…
Shining	1980	…
Annie Hall	1977	…
Jurassic Park	1992	…
Metropolis	1926	…
Manhattan	1979	…
Reservoir Dogs	1992	…
Impitoyable	1992	…
Casablanca	1942	…
Smoke	1995	…

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 ($\lfloor 1000000/3 \rfloor$) 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.

3.1 S1 : Indexation de fichiers

Supports complémentaires :

— Diapositives: structures d'index
— Vidéo sur les structures d'index

3.1.1 Structure et contenu des index

Prenez n'importe quel livre de cuisine, de bricolage, d'informatique, ou autre sujet technique : il contient un *index*. Cet index présente une liste des termes considérées comme importants, classés en ordre alphabétique, et associées aux numéros des pages où on trouve un développement consacré à ce terme. On peut donc, avec l'index, accéder directement à la page (ou aux pages en général) contenant un terme donné.

Les index dans un SGBD suivent exactement les mêmes principes. On choisit dans une table un (au moins) ou plusieurs attributs, dont les valeurs constituent la *clé d'indexation*. Ces valeurs sont l'équivalent des termes indexant le livre. On associe à chaque valeur la liste des *adresse(s)* vers le (ou les enregistrements) correspondant à cette valeur : c'est l'équivalent des numéros de page. Et finalement, on trie alors cette liste selon l'ordre alphanumérique pour obtenir 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 dite « par dichotomie »). Les index des SGBD sont organisés pour appliquer exactement la même technique.

Quelques définitions

Voici la signification des termes propres à l'indexation.

- Une **clé** (d'indexation) est une liste (l'ordre est important) d'attributs d'une table. En toute rigueur, il faudrait toujours distinguer la clé (les noms d'attributs) de la valeur de la clé (celles que l'on trouve dans un enregistrement). On se passera de la distinction quand elle est claire par le contexte.
- Une **adresse** est un emplacement physique dans la base de données, qui peut être soit celle d'un bloc, soit un peu plus précisément celle d'un enregistrement dans un bloc. Reportez-vous au chapitre sur la stockage qui détaille comment est construite l'adresse d'un enregistrement.
- une **entrée** (d'index) est un enregistrement constitué d'une paire de valeurs. La première est la valeur de la clé, la seconde une adresse.
- Un **index** est un fichier structuré dont les enregistrements sont des *entrées*.

Continuons l'analogie en examinant le cas particulier d'un *dictionnaire*. Dans un dictionnaire, les mots sont placés dans l'ordre, alors que dans un livre classique ils apparaissent sans ordre prédéfini. On peut donc se servir d'un dictionnaire comme d'un index, et chercher par approximations successives (par dichotomie en termes algorithmiques) la page contenant le terme dont on cherche la définition. Cela facilite considérablement les recherches et permet de se passer d'un index. On pourrait malgré tout en créer un pour accélérer encore la recherche. Il serait alors pertinent de tirer parti de l'ordre existant sur les mots du dictionnaire. Une possibilité est de se s'intéresser, pour l'index, qu'au premier mot de chaque page. Imaginons que l'on trouve alors dans cet index les *entrées* suivantes :

- …
- ballon, page 56
- bille, page 57
- bulle, page 65,
- cable, page 72
- …

Comment peut-on utiliser un tel index pour trouver directement dans le dictionnaire un mot quelconaue, *même s'il ne figure pas dans l'index* ? Voici quelques réponses, toutes basées sur le fait que le dictionnaire est trié.

— si je cherche le mot *armée*, je *sais* qu'il est avant la page 56
— si je cherche le mot *crabe*, je *sais* qu'il est après la page 72
— si je cherche le mot *botte*, je *sais* qu'il est entre les pages 57 (incluse) et 65 (exclue)
— et enfin, encore plus précis, si je cherche le mot *belle*, je *sais* qu'il se trouve dans la page 56

Le dernier exemple nous montre comment un tel index sert une recherche d'un mot *m* en deux étapes. Grâce à l'index je trouve la page *p* associée au mot précédant immédiatement *m* ; puis j'accède à la page *p* et je cherche le mot par une recherche dichotomique locale.

Dans une base de données, l'équivalent de la page est le *bloc*, dont nous avons vu qu'il est entièrement en mémoire RAM ou pas du tout. Chercher localement dans un bloc en mémoire RAM prend un temps négligeable par rapport à l'accès éventuel à ce bloc sur le disque, de même que chercher dans la page d'un dictionnaire prend un temps négligeable pour un lecteur par rapport à un parcours des pages du dictionnaire.

Si les enregistrements d'un fichier sont triés sur la clé, alors on peut construire l'index sur la valeur de clé du premier enregistrement de chaque bloc. C'est l'équivalent de l'indexation du dictionnaire, et ce type d'index est dit *non dense*. En l'absence de tri il faut indexer toutes les valeurs de clé. C'est l'équivalent de l'index d'un livre de cuisine, et ce type d'index est dit *dense*.

Dans tous les cas l'index lui-même est trié sur la clé. C'est cette propriété qui garantit son efficacité.

3.1.2 Comment chercher avec un index

Un index est construit sur une *clé*, et sert à accélerer les recherches pour lesquelles cette clé sert de critère.

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. Reprenons le cas d'un livre de recettes de cuisine. On peut construire l'index sur le *nom* de la recette (c'est la clé). On peut utiliser cet index pour les recherches suivantes :

— par valeur de clé : on trouve directement la page pour « blanquette » ou « soufflé » ;
— par *intervalle* des valeurs de clé : si on cherche les recettes entre « daube » et « daurade aux navets », on trouvera un intervalle de pages ;
— par *préfixe* de la valeur de la clé : si je cherche toutes les recettes commençant par « dau », l'index me permet de trouver là encore l'intervalle des pages couvrant les recettes.

Le fait de pouvoir rechercher par intervalle ou par préfixe est une conséquence de l'ordre de l'index, trié sur les valeurs de clés. Il n'est pas possible en revanche de rechercher par *suffixe* : si je cherche les recettes terminant par « ette » (blanquette, piquette et vinaigrette), je n'ai pas vraiment d'autre solution que de parcourir la liste ou le livre entier.

Et, bien sûr (mais ça va mieux en le disant), l'index ne permet pas de chercher sur des valeurs autres que celles de la clé. Avec l'index précédent, je ne peux pas chercher par ingrédient : « veau », « œuf », « carotte ». Mais rien ne m'empêche de créer un *autre* index sur les ingrédients. On peut en fait créer autant d'index que l'on veut, sur autant de clés que l'on souhaite. Cela alourdit le contenu du livre (ou de la base de données), rend plus complexe sa production (c'est vrai aussi de la base), mais cela peut valoir le coup si chaque index est utilisé fréquemment.

En l'absence d'un index approprié, il n'existe qu'une solution possible : parcourir séquentiellement le livre (la table dans le cas d'une base de données) en examinant chaque page/bloc. 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 au pire 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).

Voilà pour les principes : tout ce qui compte pour la compréhension des structures d'index et de leur utilisation est dit ci-dessus, le reste est de l'ordre du détail technique.

3.1.3 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, et le fichier a globalement la même structure d'un dictionnaire. Dans ce cas particulier 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 mileu 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 entrées (voir définition ci-dessus) `[valeur, adresse]` où `valeur` désigne une valeur de la clé de recherche, et `adresse` 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 du *premier* enregistrement de chaque bloc.

Note : Souvenez-vous de la remarque ci-dessus pour les dictionnaires, et du petit exemple que nous avons donné d'un index contenant les premiers mots de chaque page : nous sommes *exactement* dans cette situation.

Cette information est suffisante pour trouver n'importe quel enregistrement.

La Fig. 3.1 montre un index non-dense sur le fichier des 16 films, la clé étant le titre du film, et l'adresse étant symbolisée par un cercle bleu. On suppose que chaque bloc du fichier de données (contenant les films) contient 4 enregistrements, ce qui donne un minimum de *quatre* blocs. Il suffit alors de *quatre entrées* `[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*.

Si on désigne par $c_1, c_2, \cdots, c_n$ 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* dont l'adresse est

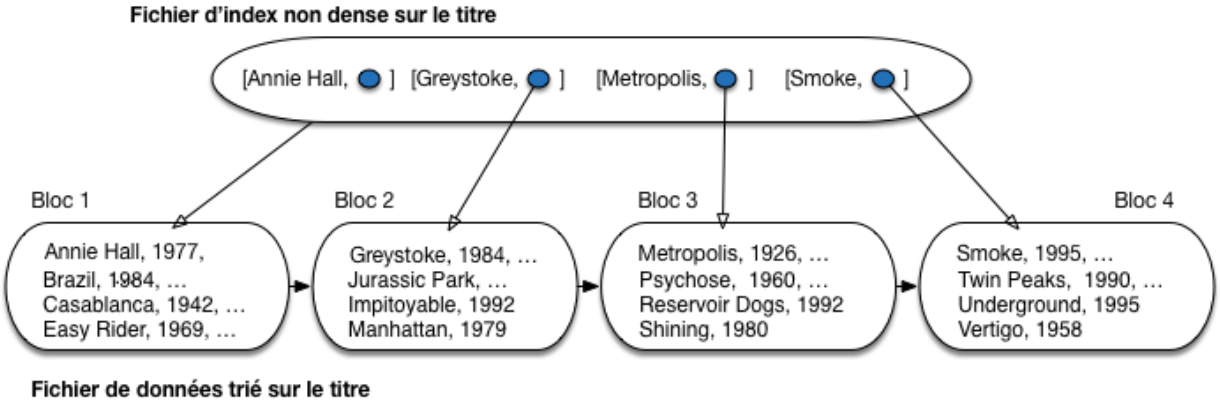

Fig. 3.1 – Un index non dense

donnée par l'index. 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 (les entrées) 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 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 la borne inférieure de l'intervalle. On accède alors au fichier grâce à cette adresse et il suffit de partir de cet emplacement et d'effectuer un parcours séquentiel pour obtenir tous les enregistrements cherchés. La recherche s'arrête quand on trouve un enregistrement donc la clé est supérieure à la borne supérieure de l'intervalle.

Exemple

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 Fig. 3.1 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 on néglige la recherche dans l'index, au parcours de la partie du fichier qui contient le résultat, soit $\frac{r}{b}$, où r désigne le nombre d'enregistrements du résultat, et b le nombre d'enregistrements dans un bloc. Ce coût est optimal (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.

3.1.4 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 Fig. 3.2 montre le même fichier contenant seize films, trié sur le titre, et indexé maintenant sur l'année de parution des films. On constate d'une part que toutes les années du fichier de données sont reportées dans l'index, ce qui accroît considérablement la taille de ce dernier, et d'autre part qu'à une même année sont associées plusieurs adresses correspondant aux films parus cette année là (l'index n'est pas *unique*).

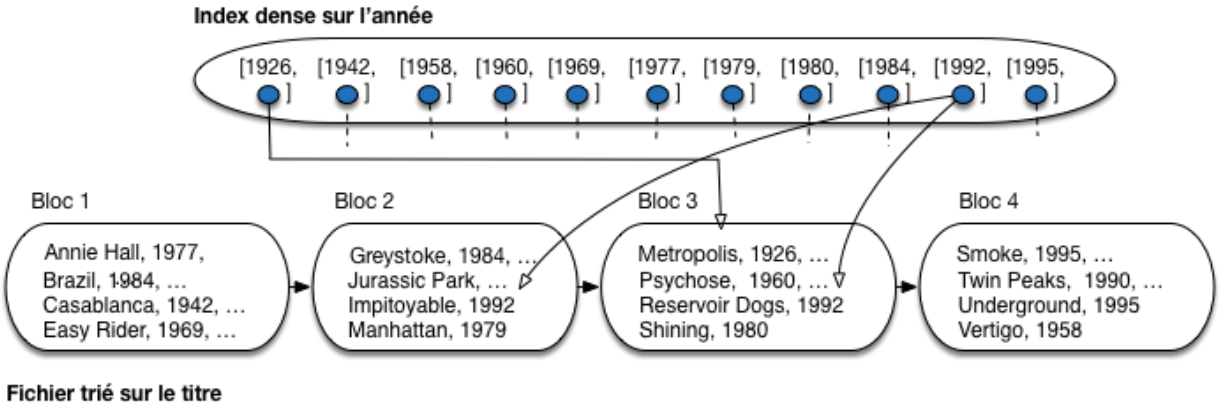

Fig. 3.2 – Un index dense (tous les liens ne sont pas représentés)

Exemple

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 et 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 serait 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. *Il ne peut y avoir qu'un seul index non-dense par fichier de données*.

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 adresses correspondant au critère de recherche. Par exemple, pour rechercher tous les films parus en 1992 dans l'index de la Fig. 3.2, on doit suivre les adresses pointant respectivement sur *Jurassic Park*, *Impitoyable* et *Reservoir Dogs*.

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 Fig. 3.2 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 enregistements, 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 aux blocs 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 des accès directs.

3.1.5 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, les principes des index non-denses* : on se contente d'indexer la première valeur de clé de chaque bloc.

Reprenons 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 Fig. 3.4. On a supposé, pour la clarté de l'illustration, qu'un bloc de l'index de premier niveau ne contient au plus que 3 entrées `[année, adr]`. Il faut donc quatre blocs pour ce premier niveau d'index.

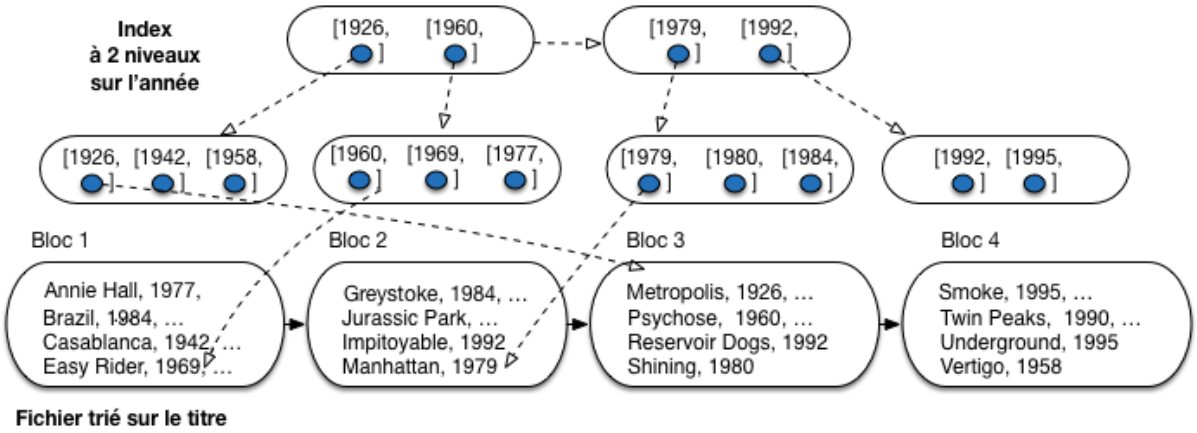

Fig. 3.3 – Index avec 2 niveaux : le second indexe le premier, qui indexe le fichier

L'index de second niveau est construit sur la clé du premier enregistrement de chaque bloc de l'index de premier niveau. On diminue donc le nombre d'entrées par 3 (nombre d'enregistrements par bloc, ou facteur de blocage) entre le premier et le second niveau. On y gagne en espace : sur notre exemple, la taille est diminuée par deux. Dans un cas réel, le facteur de blocage est de quelques centaines (on met plusieurs centaines d'entrées d'index dans un bloc de 4 096 octets), et le taux de réduction d'un niveau à un autre est donc considérable.

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.

Peut-on créer un troisième niveau, un quatrième niveau ? Oui, bien sûr. Quand s'arrête-t-on ? Et bien, si on se souvient que la granularité de lecture est le bloc, on peut dire que quand un niveau d'index tient dans un seul bloc, il est inutile d'aller plus loin.

Pour notre exemple, on peut créer un troisième niveau, illustré par la Fig. 3.4. Ce troisième niveau est constitué d'un seul bloc. Comme on ne peut pas faire un fichier plus petit, on arrête là.

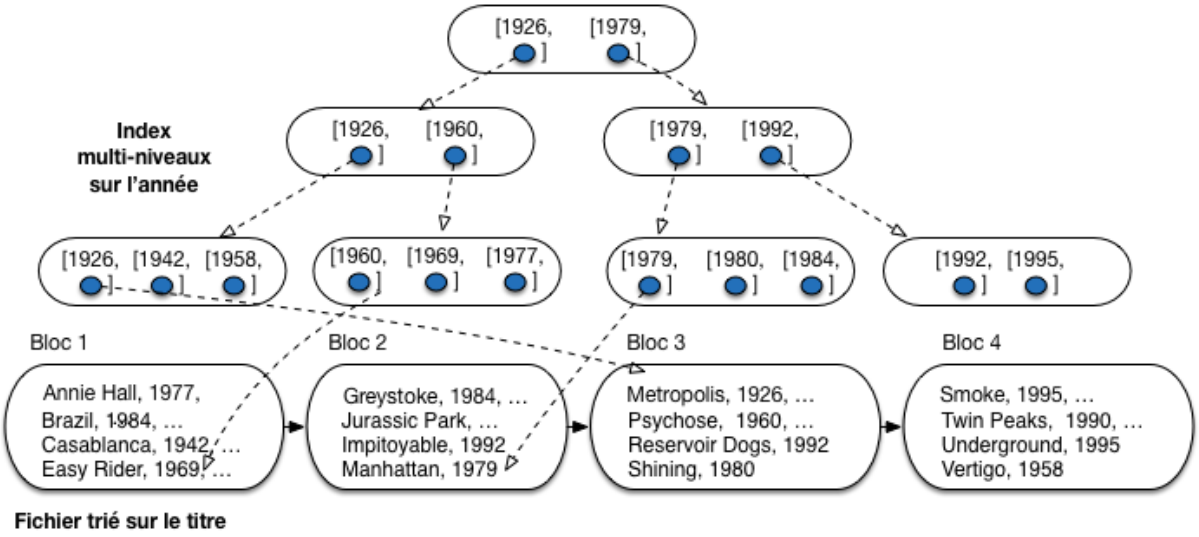

Fig. 3.4 – Index multi-niveaux

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 1980, avec l'index de la Fig. 3.4.

- Partant du troisième niveau d'index, on décide qu'il faut descendre vers la droite pour accéder au second bloc du deuxième niveau d'index.
- Le contenu de ce second bloc nous indique qu'il faut cette fois descendre vers la gauche, vers le troisième bloc du premier niveau d'index.
- On trouve dans ce troisième bloc l'entrée correspondant à 1980, avec les adresses des films à aller chercher dans le fichier de données.

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 compacts et triés. 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.

3.1.6 Quiz

1. Qu'appelle-t-on une *entrée* d'index ?
2. Combien d'index non denses peut-on créer sur un fichier de données
3. Quel est le type d'index adapté à un dictionnaire ?
4. La clé d'indexation de mon index est la paire (titre, année). Quelles requêtes sur la table des films permettent d'utiliser l'index (plusieurs réponses possibles).
5. Que reste-t-il à faire après avoir effectué une recherche dans un fichier d'index ?
6. Quel est l'inconvénient d'une recherche par intervalle avec un index ?
7. Pourquoi s'arrêter quand la racine d'un index multi-niveaux contient un seul bloc ?

3.2 S2 : L'arbre-B

Supports complémentaires :
— Diapositives: arbre B
— Vidéo consacrée à l'arbre B

L'arbre-B est une structure très proche des index présentés ci-dessus. La principale différence est que la granularité de construction de l'index n'est plus un *fichier* représentant un niveau monolithique, mais un *bloc*. Cette granularité minimale permet notamment une réorganisation efficace de l'arbre pour s'adapter aux évolutions du fichier de données indexé.

Commençons par une notion technique, celle *d'ordre* d'un arbre B.

Définition : Ordre d'un arbre B.

Si un bloc peut contenir au maximum n entrées, alors *l'ordre de l'arbre B* est $\lfloor \frac{n}{2} \rfloor$. L'ordre d'un arbre B correspond au nombre minimal d'entrées contenues dans chacun des blocs, propriété qui résulte de l'algorithme de construction que nous étudierons plus loin.

L'ordre d'un arbre dépend de la taille de la clé et se détermine simplement par le calcul suivant. Soit $||c||$ la taille de la clé à indexer (en prendra une moyenne pour les clés de taille variable), $||A||$ la taille d'une adresse et $||B||$ l'espace utile d'un bloc (en excluant l'entête), alors le nombre maximum d'entrées n est

$$\left\lfloor \frac{||B||}{||c|| + || + A||} \right\rfloor$$

Et l'ordre est $\lfloor \frac{n}{2} \rfloor$. Rappelons que la notation $\lfloor x \rfloor$ désigne la partie entière de x.

Reprenons l'exemple de notre fichier contenant un million de films. Admettons qu'une entrée d'index occupe 12 octets, soit 8 octets pour l'adresse, et 4 pour la clé (l'année du film). Chaque bloc contient 4 096 octets. On place donc $\lfloor \frac{4096}{12} \rfloor = 341$ entrées au maximum dans un bloc et l'ordre de cet index est $\lfloor \frac{341}{2} \rfloor = 170$.

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, car la plus facile à gérer (et la plus rapide) au moment des insertions. Nous allons indexer ce fichier avec un ou plusieurs arbres B.

3.2.1 Structure de l'arbe B

La Fig. 3.5 montre un arbre-B indexant notre collection de 16 films, avec pour clé d'indexation l'année de parution du film. L'index est organisé en blocs de taille égale, ce qui ajoute une souplesse supplémentaire à l'organisation en niveaux étudiée précédemment. En pratique un bloc peut contenir un grand nombre d'entrées d'index (plusieurs centaines, voir plus loin), mais pour la clarté de l'illustration nous supposerons que l'on peut stocker au plus 4 entrées d'index dans un bloc.

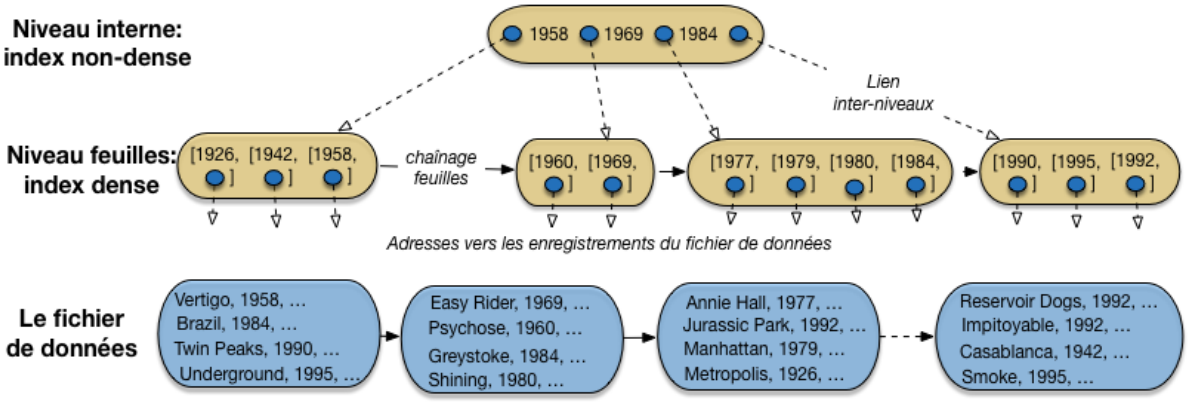

Fig. 3.5 – Un arbre B construit sur l'année des films

L'index a une structure arborescente constituée de plusieurs niveaux (en l'occurence seulement 2). Le niveau le plus bas, celui des *feuilles*, constitue un index *dense* sur le fichier de données. Le concept d'index dense a été présenté précédemment : toutes les valeurs de clé de la collection indexée y sont représentées. De plus, elles sont ordonnées, alors que l'on peut constater qu'il n'existe aucun ordre correspondant dans le fichier de données.

En revanche, contrairement aux fichiers d'index traditionnels, le niveau des feuilles de l'arbre B n'est pas constitué d'une séquence contiguë de blocs, mais d'une *liste chaînée* de blocs. Chaque bloc-feuille référence le bloc-feuille suivant, ce qui permet de parcourir ce niveau, en suivant le chaînage, sans avoir à remonter dans l'arbre. Cette organisation est moins efficace qu'un stockage continu (le passage d'un bloc à un autre peut entraîner un déplacement des têtes de lecture), mais permet la réorganisation dynamique de la liste pour y insérer de nouveaux blocs, comme nous le verrons plus loin.

Dans chaque bloc, au niveau des feuilles, on trouve des *entrées* d'index, soit une paire constituée d'une valeur de clé et d'une adresse vers un enregistrement du fichier de données.

Note : Comme plusieurs films ont pu paraître la même année, on peut trouver des entrées avec plusieurs adresses associées à une valeur de clé. On pourrait aussi trouver plusieurs entrées avec la même valeur de clé. Ce choix d'implantation ne remet pas en question la structure de l'index, que nous continuons à décrire.

Les niveaux de l'arbre B situés au-dessus des feuilles (sur notre exemple il n'y en a qu'un) sont les niveaux *internes*. Ils sont eux aussi constitués de blocs indépendants, mais sans chaînage associant les blocs d'un

même niveau. Chaque bloc interne sert d'index local pour se diriger, de bas en haut, dans la structure de l'arbre, vers la feuille contenant les valeurs de clé recherchées.

Regardons notre unique niveau interne, qui est également la racine de l'arbre. C'est un index *non dense*, avec des valeurs de clés triées, et des adresses qui s'interprètent de la manière suivante.

— dans le sous-arbre référencé par l'adresse située à gauche de *1958*, on ne trouve que des films parus *avant* (au sens large) 1958 ;
— dans le sous-arbre référencé par l'adresse située entre *1958* et *1969*, on ne trouve que des films parus *après* (au sens strict) 1958 et avant (au sens large) 1969 ;
— dans le sous-arbre référencé par l'adresse située entre *1969* et *1984*, on ne trouve que des films parus *après* (au sens strict) 1969 et avant (au sens large) 1984 ;
— enfin, la dernière adresse référence un sous-arbre contenant tous les films parus strictement après 1984.

Chaque bloc d'un nœud interne divise donc l'espace de recherche en 4. Si on a un niveau d'index, on va donc, en consultant un bloc de ce niveau, réduire par 4 la taille des données à explorer. Si on a deux niveaux, on réduit d'abord par 4, puis encore par 4, soit $4^2 = 16$. Si on a *k* niveaux, on réduit par 4^k. Si, de manière encore plus générale, on désigne par *n* le nombre d'entrées que l'on peut stocker dans un bloc, la « réduction » apportée par l'index devient n^k.

Sur notre exemple de 16 films, avec un seul niveau d'index, et très peu d'entrées par bloc, c'est évidemment assez peu spectaculaire, mais dans des conditions plus réalistes, on obtient une structure extrêmement efficace. Effectuons quelques calculs pour nous en convaincre.

Exemple : indexons un million de films sur l'année.

Reprenons l'exemple de notre fichier contenant un million de films. Admettons qu'une entrée d'index occupe 12 octets, soit 8 octets pour l'adresse, et 4 pour la clé (l'année du film). Chaque bloc contient 4 096 octets.

On place donc $\lfloor \frac{4096}{12} \rfloor = 341$ entrées (au maximum) dans un bloc. Comme le niveau des feuilles de l'abre B est dense, il faut $\lfloor \frac{1000000}{341} \rfloor = 2\,932$ blocs pour le niveau des feuilles.

Le deuxième niveau est *non dense*. Il comprend autant d'entrées que de blocs à indéxer, soit 2 932. Il faut donc $\lfloor \frac{2\,932}{341} \rfloor = 8$ blocs (au mieux). Finalement, un troisième niveau, constitué d'un bloc avec 8 entrées suffit pour compléter l'index.

Important : Le calcul précédent est valable dans le *meilleur* des cas, celui où chaque bloc est parfaitement plein. Mais alors quel est le *pire* des cas ? La réponse est que l'arbre B garantit que chaque bloc est *au moins* à moitié plein, et donc le pire des cas sur notre exemple serait un remplissage avec 170 entrées (la moitié de 341) par bloc. Cette (excellente) propriété de remplissage est une conséquence de la méthode de construction de l'index, présentée un peu plus loin.

Voici, réciproquement, une petite extrapolation montrant le nombre de films indexés en fonction du nombre de niveaux dans l'arbre (même remarque que précédemment : on calcule dans le meilleur des cas, en supposant qu'un bloc est toujours plein.)

— avec un niveau d'index (la racine seulement) on peut donc indexer 341 films ;
— avec deux niveaux la racine indexe 341 blocs d'index, référençant chacun 341 films, soit $341^2 = 116\,281$ films indexés au total ;
— avec trois niveaux on indexe $341^3 = 39\,651\,821$ films ;

— enfin avec quatre niveaux on indexe plus de 1 milliard de films.

Il y a donc une croissance très rapide, *exponentielle*, du nombre d'enregistrements 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 calcul précis est donné par la formule suivante (on suppose que les clés sont uniques, le lecteur peut faire lui-même l'extension de la formule au cas des index non uniques). On note $||T||$ la cardinalité d'une table T (nombre d'enregistrements), et k l'ordre de l'arbre B. Alors la hauteur h de l'arbre est donnée par

$$\lceil \log_k(||T||) \rceil$$

Il s'agit d'un calcul théorique qui peut, en pratique, être optimisé par diverses techniques, et notamment des méthodes de compression de valeurs de clé que nous ne présentons pas ici. La formule ci-dessus doit donc être considérée comme donnant un ordre de grandeur.

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 sur le titre, un chaîne de caractères occupant en moyenne quelques dizaines d'octets (disons, 20), on pourrait référencer $\lfloor \frac{4096}{20+8} \rfloor = 146$ films dans un bloc, et un index avec trois niveaux permettrait d'indexer $146^3 = 3\,112\,136$ films ! Du point de vue des performances, le choix d'une chaîne de caractères assez longue comme clé des enregistrements est donc assez défavorable.

3.2.2 Construction de l'arbre B

Voyons maintenant comment le système maintient un arbre-B sur un fichier (une table) sans organisation particulière : les enregistrements sont placés les uns après les autres dans l'ordre de leur insertion, dans une structure dite séquentielle. Nous construisons un arbre B sur le titre des films.

Nous avons donc deux fichiers (ou, de manière un peu plus abstraite, deux espaces de stockage indépendants, nommés *segments* dans ORACLE par exemple) : la table et l'index. Le premier a une structure séquentielle et contient les enregistrements représentant les lignes de la table. On va supposer pour la facilité de la présentation que l'on peut mettre 4 enregistrements par bloc. Le second fichier a une structure d'arbre B, et ses blocs contiennent les entrées de l'index. On va supposer pour l'illustration que l'on met au plus deux entrées par bloc. C'est tout à fait irréaliste, comme l'ont montré les calculs précédents, mais cela permet de bien comprendre la méthode.

La Fig. 3.6 montre la situation initiale, avec les deux structures, la seconde indexant la première, après l'insertion des deux premiers films. L'arbre est pour l'instant constitué d'un unique bloc avec deux entrées (il est donc déjà plein d'après notre hypothèse).

On insère ensuite *Twin Peaks*. Pas de problème pour l'enregistrement dans la table, qui vient se mettre à la suite des autres dans le fichier de données. Il faut également insérer l'entrée correspondante, et, d'après notre hypothèse, l'unique bloc de l'arbre B est plein (Fig. 3.7).

Il faut ajouter un nouveau bloc à l'index, en conservant la structure globale de l'arbre. L'ajout d'un bloc suit une procédure dite *d'éclatement* qui est illustrée sur la partie de droite de la Fig. 3.7. En voici les étapes

— les entrées du bloc trop plein sont triées sur la valeur de la clé
— l'entrée correspondant à la valeur *médiane* est placée dans un nouveau bloc, au niveau supérieur (ce nouveau bloc est donc un nœud *interne*)
— les entrées inférieures à la valeur médiane sont dans un bloc à gauche ; les entrées supérieures à la valeur médiane sont dans un bloc à droite

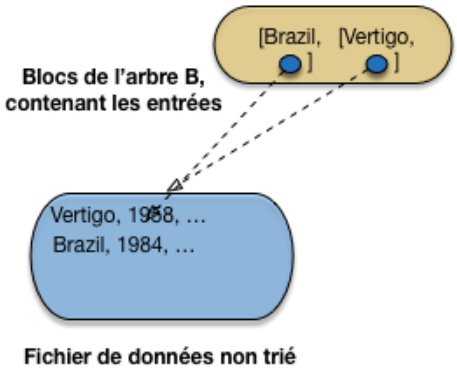

Fig. 3.6 – Début de la construction, 2 films seulement

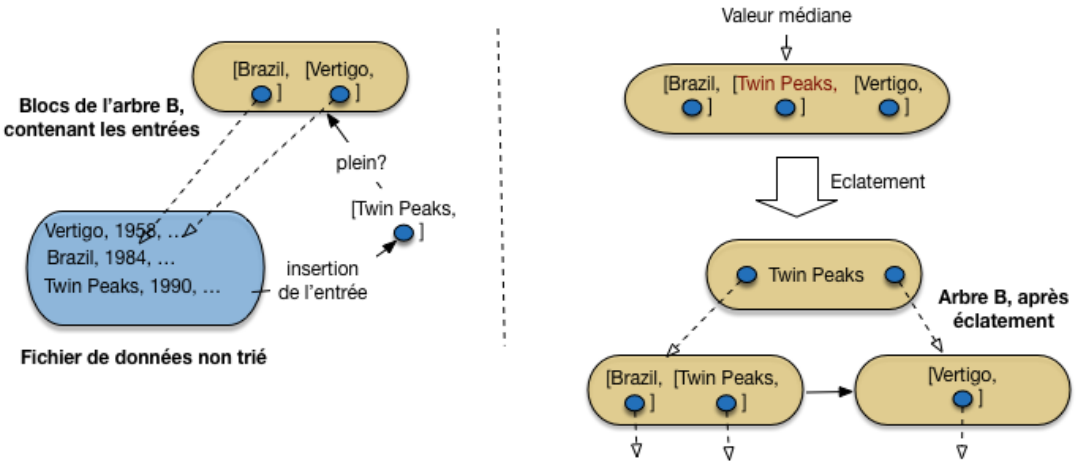

Fig. 3.7 – Après insertion d'un troisième film, et premier éclatement

On aboutit à l'arbre en bas à droite de la Fig. 3.7. C'est déjà un arbre B avec sa racine, qui est un nœud interne, et deux feuilles indexant de manière dense le fichier de données.

Important : Comme le niveau des feuilles est *dense*, on laisse *toujours*, quand on éclate une *feuille*, l'entrée de la valeur médiane dans le bloc de gauche, en plus d'insérer une entrée avec cette valeur médiane dans le bloc de niveau supérieur. Ici, la valeur *Twin Peaks* apparaît donc dans *deux entrées* : celle des feuilles référence l'enregistrement du film, celle du bloc interne référence la feuille de l'index.

Continuons avec les insertions de *Underground*, puis de *Easy Rider* (Fig. 3.8). Les enregistrements sont placés séquentiellement dans le fichier de données. Pour les entrées d'index, on doit déterminer dans quelle feuille on insère. Pour cela on part de la racine, et on suit les adresses comme si on recherchait un enregistrement pour la valeur de clé à insérer.

Dans notre cas, *Underground* étant supérieur dans l'ordre lexicographioque à *Twin Peaks*, va à droite, et *Easy Rider* va à gauche.

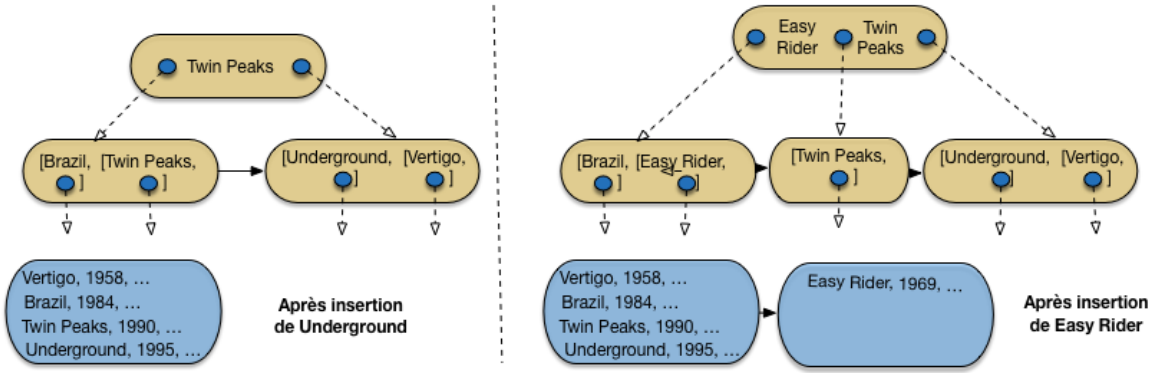

Fig. 3.8 – Après insertion de Underground, puis de Easy Rider

Underground vient donc prendre place dans la feuille de droite, qui ne déborde pas encore. En revanche, *Easy Rider* doit aller dans la feuille de gauche, qui devient trop pleine. Un *éclatement* a lieu, l'entrée correspondant à la valeur médiane (*Easy Rider*) est transmise au niveau supérieur qui indexe donc les trois blocs de feuilles.

On continue avec *Psychose*, puis *Greystoke* (Fig. 3.9). Tous deux vont dans le bloc contenant initialement *Twin Peaks*, ce qui entraîne un débordement. La valeur médiane, *Psychose*, doit donc être insérée dans la racine de l'arbre B.

Nous avons donc maintenant le cas d'un nœud interne qui déborde à son tour. On applique la même procédure d'éclatement, avec identification de la valeur médiane, et insertion d'une entrée avec cette valeur dans le bloc parent. Ici, on crée une nouvelle racine, en augmentant donc de 1 le nombre de niveaux de l'arbre. On obtient l'arbre de la Fig. 3.9.

Important : Quand on éclate un bloc *interne*, il est inutile de conserver la valeur médiane dans les blocs du niveau inférieur. Rappelons que l'indexation des blocs *internes* est non dense et qu'il n'est donc pas nécessaire de représenter toutes les valeurs de clés, contrairement aux feuilles.

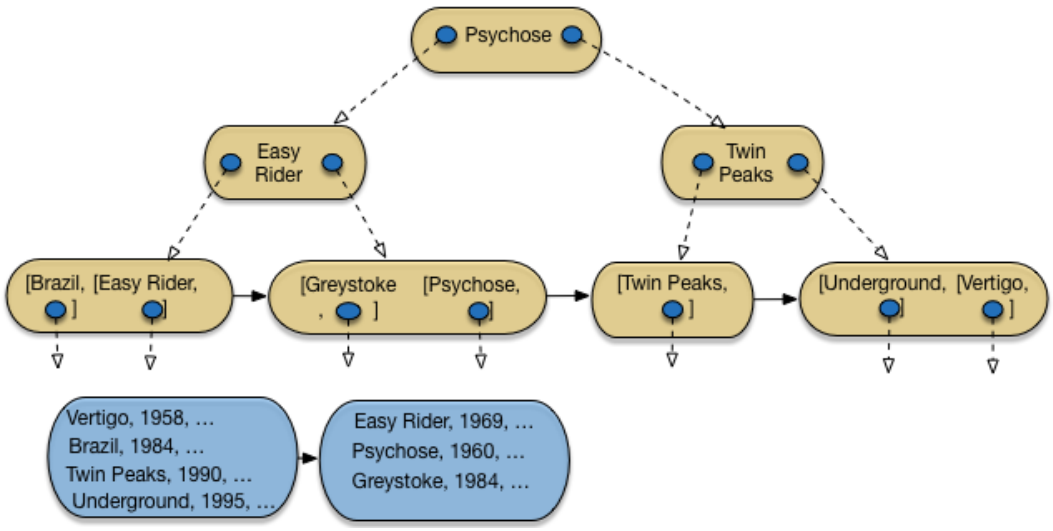

Fig. 3.9 – Après insertion de *Psychose* et *Greystoke*

Et ainsi de suite. La Fig. 3.10 montre l'arbre B après insertion de Shining et Annie Hall, et la Fig. 3.11 l'arbre après insertion de 12 films sur 16 : je vous laisse compléter avec les 4 films restant, soit *Reservoir Dogs*, *Impitoyable*, *Casablannca* et *Smoke*.

La méthode illustrée ici montre tout d'abord une propriété importante, déja évoquée : *chaque bloc de l'arbre B (sauf la racine) est au moins à moitié plein*. C'est une propriété obtenue par construction : un éclatement n'intervient que quand un bloc est plein, cet éclatement répartit les entrées en deux sous-ensembles de taille égale, et les deux blocs résultant d'un éclatement sont donc à moitié plein, avant de recevoir de nouvelles entrées avec les insertions ultérieures. On constate en pratique que le taux de remplissage est d'environ 70%.

La seconde remarque importante tient à l'aspect dynamique de la construction, illustrée ici. Une insertion dans un arbre B peut déclencher des réorganisations qui restent *locales*, en d'autres termes elles n'affectent qu'un partie minimale de la structure globale. C'est ce qui rend l'évolution de l'index fluide, sans nécessité d'introduire une coûteuse opération périodique de réorganisation globale.

3.2.3 Recherches avec un arbre-B

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

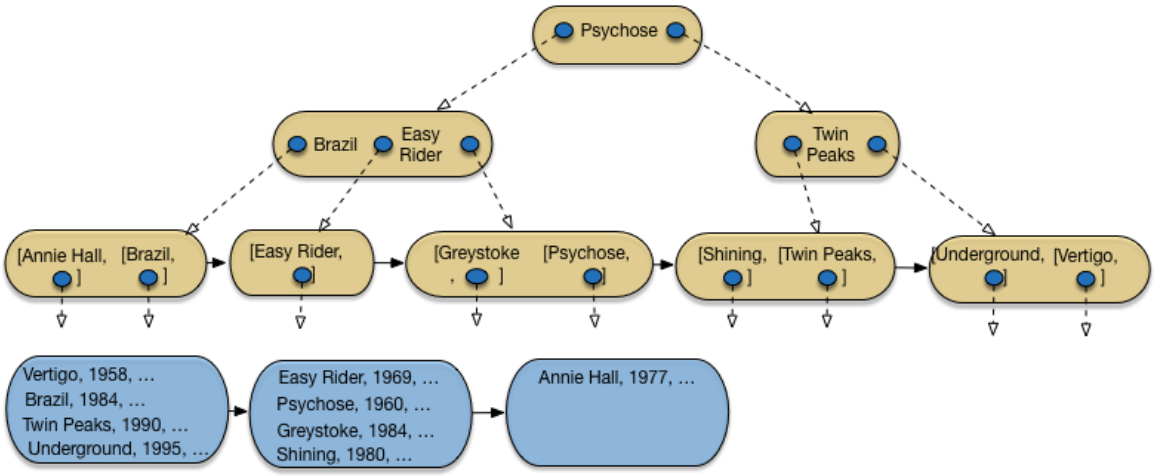

Fig. 3.10 – Après insertion de *Shining* et *Annie Hall*

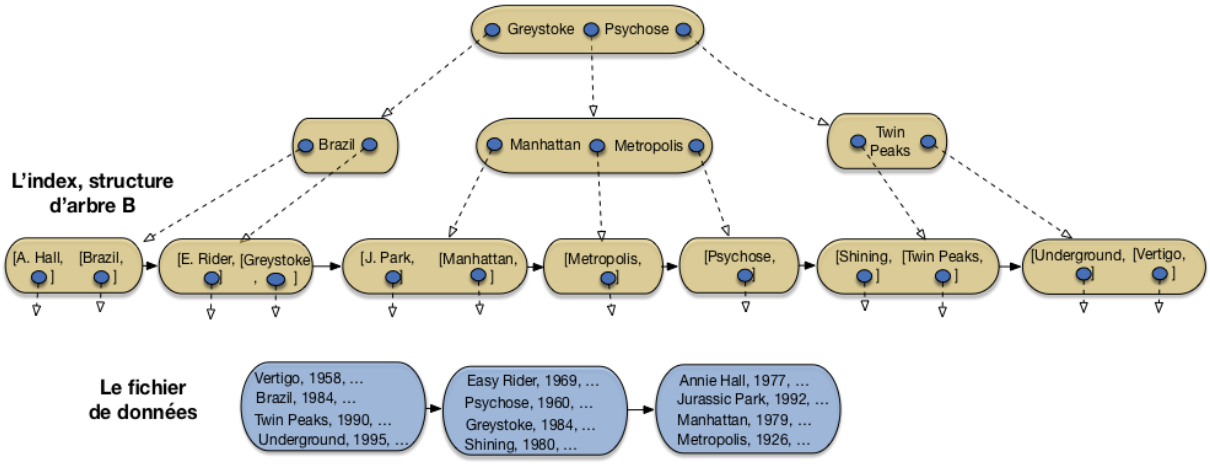

Fig. 3.11 – Après insertion de *Jurassic Park*, *Manhattan* et *Metropolis*

Recherche par clé

Prenons l'exemple suivant :

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

En l'absence d'index, la seule solution est de parcourir le fichier. Dans l'exemple de la Fig. 3.11, cela implique de lire inutilement 10 films avant de trouver *Manhattan* qui est en onzième position. L'index permet de trouver l'enregistrement beaucoup plus rapidement.

- on lit la racine de l'arbre : *Manhattan* étant situé dans l'ordre lexicographique entre *Easy Rider* et *Psychose*, on doit suivre le chaînage situé entre ces deux titres ;
- on lit le bloc interne intermédiaire : la feuille contenant *Manhattan* est dans l'arbre situé à gauche de l'entrée d'index *Manhattan*, on suit donc le chaînage de gauche ;
- on lit le bloc feuille dans lequel on trouve l'entrée *Manhattan* contenant l'adresse de l'enregistrement dans le fichier des données ;
- il reste à lire l'enregistrement.

Donc quatre lectures sont suffisantes : trois dans l'index, une dans le fichier de données. Plus généralement, le nombre de lectures (logiques) nécessaires pour une recherche par clé est égal au nombre de niveaux de l'arbre, plus une lecture (logique) pour accéder au fichier de données.

Important : Ce sont des lectures *logiques* par opposition aux lectures *physiques* qui impliquent un accès disque. En pratique, un index souvent utilisé réside en mémoire, et le parcours est très rapide.

Prenons le cas plus réaliste de notre fichier avec un million de film. Nous avons déjà calculé qu'il était possible de l'indexer avec un arbre B à trois niveaux. Quatre lectures (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 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. La création d'un index peut faire passer le temps de réponse dune requête de quelques secondes ou dizaines de secondes à quelques micro secondes.

Recherche par intervalle

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

```
select *
from  Film
where annee between 1960 and 1975
```

On peut utiliser l'index sur les années (Fig. 3.5) 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 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. On accède ainsi successivement aux valeurs 1969, 1977 (dans la troisième feuille) puis 1979. Arrivé à ce point, on sait que toutes les valeurs suivantes seront supérieures à 1979 et qu'il n'existe donc pas de film paru en 1975 dans la base de données. Toutes les adresses des films constituant le résultat de la requête ont été récupérées : il reste à lire les enregistrements dans le fichier des données.

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

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

Recherche par préfixe

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 (Fig. 3.11) 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` ». Nous y reviendrons dans le chapitre sur l'évaluation de requêtes.

3.2.4 Création d'un arbre B

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 (année)
```

Note : La création de l'index sur la clé primaire (et parfois sur la clé étrangère) est très utile pour vérifier la satisfaction des contraintes de clé ainsi que, nous le verrons, pour le calcul des jointures. Voir l'exercice *ex-arbreb6*.

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.

3.2.5 Propriétés de l'arbre B

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

Le coût des opérations avec l'arbre B est est logarithmique dans la taille des données, alors qu'une recherche sans index est linéaire. Mais au-delà de cette analyse, l'important est que cela correspond, en pratique, à des gains de performance considérables.

L'arbre B exploite bien l'espace, a de très bonnes performances, et se réorganise automatiquement et à coût minimal. Ces qualités expliquent qu'il soit systématiquement utilisé par tous les SGBD, notamment pour indexer la clé primaire des tables relationnelles.

3.2.6 Quiz

— Combien peut-on créer d'index en forme d'arbre B sur une table ?

— Indiquer quelles affirmations sont vraies

— Le chaînage des feuilles est utile pour (une seule réponse)

— La capacité maximale d'un bloc de mon arbre est de 10 entrées. Un bloc est toujours au moins à moitié plein (on va supposer que c'est vrai aussi pour la racine). Quel est le nombre d'enregistrements que je peux indexer avec un arbre à 2 niveaux ?

— À propos de l'éclatement, que peut-on dire ?

— Parmi les phrases suivantes, lesquelles décrivent correctement l'algorithme d'insertion d'une entrée dans un arbre B

— La recherche pour une valeur de clé, dans un arbre B

— Quel est l'inconvénient d'une recherche par intervalle avec un arbre B

3.3 Exercices

Exercice ex-dense-nondense : index dense ou non-dense

Soit un fichier de données tel que chaque bloc peut contenir 10 enregistrements. On indexe ce fichier avec un niveau d'index, et on suppose qu'un bloc d'index contient 100 entrées *[valeur, adresse]*.

Si n est le nombre d'enregistrements, donnez le nombre minimum de blocs en fonction de n pour un index dense et un index non-dense.

Exercice ex-construction : construction d'un arbre B

Soit la liste des départements suivants, à lire de gauche à droite et de bas en haut.

```
3  Allier; 36 Indre; 18 Cher; 75 Paris
39 Jura; 9  Ariège; 81 Tarn; 11 Aude
12 Aveyron; 25 Doubs; 73 Savoie; 55 Meuse;
15 Cantal; 51 Marne; 42 Loire; 40 Landes
14 Calvados; 30 Gard; 84 Vaucluse; 7 Ardèche
```

Questions :

— Construire, en prenant comme clé le numéro de département, un index dense à deux niveaux sur le fichier contenant les enregistrements dans l'ordre indiqué ci-dessus, en supposant 2 enregistrements par bloc pour les données, et 8 par bloc pour l'index.
— Construire un index non-dense sur le fichier trié par numéro, avec les mêmes hypothèses.
— Construire un arbre-B sur les numéros de département, en supposant qu'il y a au plus 4 entrées par bloc dans l'index, et en insérant les enregistrements dans l'ordre donné ci-dessus.
— Construire un arbre-B sur les noms de département, en supposant qu'il y a au plus 4 entrées par bloc dans l'index, et en insérant les enregistrements dans l'ordre donné ci-dessus.

Exercice ex-arbreb1 : propriétés d'un arbre B

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

— Combien faut-il de blocs ? Quelle est la taille du fichier ?
— Quelle est la taille d'un index de type arbre-B si la clé fait 32 octets et une adresse 8 octets ? Détaillez le calcul niveau par niveau.
— Même question si la clé fait 4 octets.
— Si on suppose qu'une lecture coûte 10 ms, quel est le coût moyen d'une recherche d'un enregistrement par clé unique, avec index et sans index dans le pire des cas ?

Exercice ex-arbreb2 : hauteur et efficacité d'un arbre B

On reprend les hypothèses précédentes, et on indexe maintenant le fichier avec un arbre-B dont chaque bloc peut contenir au maximum 100 entrées. Les feuilles de l'arbre contiennent des entrées référençant des enregistrements dans le fichier, et les nœuds internes contiennent des entrées référençant d'autres nœuds.

— Quel est l'ordre de cet arbre B et quel est sa hauteur théorique obtenue par la formule donnée en cours ?
— On suppose maintenant qu'un bloc d'arbre B est plein à 70% et contient donc 70 entrées pour un fichier de 1 000 000 d'enregistrements. En effectuant un calcul niveau par niveau, donnez (1) le nombre de blocs du niveau des feuilles, (2) le nombre minimal de blocs utilisés par la fichier et l'index, (3) le nombre de niveaux de l'arbre , (4) le nombre de lectures pour rechercher un enregistrement par sa clé.
— On effectue maintenant une recherche par intervalle ramenant 1 000 enregistrements. Décrivez la recherche et donnez le nombre de lectures dans le pire des cas.

Exercice ex-arbreb3 : encore des calculs sur l'arbre B

Un arbre B indexe un fichier de 300 enregistrements.

Dans un premier temps, on suppose que l'ordre de l'arbre est de 5. Chaque nœud stocke donc au plus 10 entrées. Quelle est la hauteur minimale de l'arbre et sa hauteur maximale ? (Un arbre constitué uniquement de la racine a pour hauteur 0).

Inversement, on ignore l'ordre de l'arbre mais on constate qu'il a deux niveaux. Quel est l'ordre maximal compatible avec cette constatation ? Et l'ordre minimal ?

Exercice ex-arbreb4 : indexation des séquences

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

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

Exercice ex-arbreb5 : index ou parcours séquentiel ?

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

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

— Donnez la fonction de n et m exprimant le nombre de lectures à effectuer pour cette recherche avec un parcours séquentiel.
— Donnez la fonction exprimant le le nombre de lectures à effectuer en utilisant l'index.
— À partir de quelle valeur de m la recherche séquentielle devient-elle préférable à l'utilisation de l'index, en supposant un temps d'accès uniforme pour chaque bloc ?
En déduire le pourcentage d'enregistrements concernés par la recherche à partir duquel le parcours séquentiel est préférable. On pourra simplifier les équations en éliminant les facteurs qui deviennent négligeables pour des grandes valeurs de n et de m.

Exercice ex-arbreb6 : utilité des index sur les clés primaires et étrangères

Soit les deux tables suivantes :

```
create table R (idR varchar(20) not null,
          primary key (idR));

create table S (idS int not null,
        idR varchar(20) not null,
          primary key (idS),
         foreign key idR references R);
```

Indiquez, pour les ordres SQL suivants, quels index peuvent améliorer les performances ou optimiser la vérification des contraintes `primary key` et `foreign key`.

```
select * from R where idR = 'Bou'
select * from R where idR like 'B%'
select * from R where length(idR) = 3
select * from R where idR like '_ou'
insert into S values (1, 'Bou')
select * from S where idS between 10 and 20
delete from R where idR like 'Z%'
```

3.4 Atelier

Nous reprenons la base de notre atelier et nous allons créer des index.

3.4.1 Arbre B

— Nous souhaitons créer un arbre B sur l'attribut `Identifiant` de la table `Salle`. Donner l'arbre d'ordre 2 après insertion des valeurs suivantes. Vous ferez apparaître les étapes d'éclatement :

```
100, 25, 72, 48, 10, 33, 58, 110, 40, 52, 115, 80, 5, 28, 49, 75
```

— Nous créons maintenant un index d'orbre 3 sur l'attribut `Capacité` de la table `Salle`, avec les valeurs suivantes : 20, 30, 40, 20, 25, 200, 300, 150, 40, 20, 20, 50, 30. Donnez l'arbre final.

— Lors de la création d'un index, on ne spécifie pas l'ordre de celui-ci. Il est calculé automatiquement en fonction de la taille de l'attribut indexé, de la taille d'une adresse, et de la taille d'un bloc (idem que pour les données). Calculer l'ordre de l'index sur l'attribut `Capacité`.

— Au vu de l'ordre d'un arbre B, on peut estimer la hauteur de cet index (permet d'estimer son coût de parcours). Donner la hauteur de l'index sur `Capacité`.

3.4.2 Index dense et non dense

— Quelle est la hauteur maximum d'un index *dense* sur l'attribut `id_Personne` de la table `Personne` ?

— Quelle est la hauteur maximum d'un index *non-dense* sur l'attribut `id_Personne` de la table `Personne` ?

— Supposons qu'il existe un index non-dense sur la table `Personne` pour l'attribut `id_Personne` . On souhaite maintenant ajouter un index sur l'attribut `Nom`, peut-il être un index non-dense ?

CHAPITRE 4

Structures d'index : le hachage

Les tables de hachage sont des structures très couramment utilisées en mémoire centrale pour organiser des ensembles et fournir un accès performant à ses éléments. Le hachage est également utilisé par les SGBD pour organiser de grandes collections de données sur mémoire persistante. Une technique intermédiaire, le *hachage hybride*, consiste à créer une seule structure de hachage dont les données sont partiellement en mémoire RAM et partiellement sur le disque. Le hachage hybride est notamment utilisé pour des algorithmes de jointures sophistiqués sur lesquels nous reviendrons.

Dans ce chapitre nous étudions les structures de hachage utilisées pour indexer de grandes collections de données. La plus simple, le *hachage statique* ne fonctionne correctement que pour des collections de tailles fixes, ce qui exclut des tables évolutives (le cas le plus courant). Le *hachage dynamique*, qui s'adapte à la taille de la collection indexée, est présenté en section 2. Il repose sur un répertoire (*directory*) dont la taille peut croître au point de devenir un problème. Enfin la troisième section introduit le *hachage linéaire*, une structure qui apporte toute l'efficacité du hachage tout en maintenant une taille de répertoire réduite.

4.1 S1 : le hachage statique

Supports complémentaires :

— Diapositives du hachage statique:
— Vidéo d'introduction au hachage

Nous commençons par rappeler les principes du hachage avant d'étudier les spécificités apportées par le stockage en mémoire secondaire.

4.1.1 Principes de base

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

Prenons l'exemple de notre ensemble de films, et organisons-le avec une table de hachage sur le titre. Pour simplifier, on va supposer que chaque fragment contient un seul bloc avec une capacité d'au plus quatre films. L'ensemble des 16 films occupe donc au moins 4 blocs. Pour garder un peu de souplesse dans la répartition qui n'est pas toujours uniforme, on va affecter 5 fragments à la collection de films, et on numérote ces fragments de 0 à 4.

Chaque fragment a une adresse, celle de son premier bloc. Il nous faut donc une structure qui associe le numéro du fragment à son adresse. Cette struture est le *répertoire* de la table de hachage. Le répertoire est un simple tableau à deux dimensions, avec les numéros dans une colonne et les adresses dans une autre. Le répertoire d'une structure de hachage est en principe très petit et doit tenir en mémoire RAM.

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

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

Si le premier critère est relativement facile à satisfaire, le second soulève quelques problèmes car l'ensemble des chaînes de caractères auxquelles on applique une fonction de hachage possède souvent des propriétés statistiques spécifiques. Dans notre exemple, l'ensemble des titres de film commencera souvent par « Le » ou « La » ce qui risque de perturber la bonne distribution du résultat si on ne tient pas compte de ce facteur. On sait définir des fonctions de hachage uniforme, à base d'un savant triturage de la représentation binaire de clés pour éliminer toute régularité. On va supposer ce point acquis : nos fonctions de hachage sont uniformes.

Nous allons utiliser un principe simple pour notre exemple, en considérant la première lettre du titre, et en lui affectant son rang dans l'alphabet. Donc *a* vaudra 1, *b* vaudra *2*, *i* vaudra 9, etc. Ensuite, pour se ramener à une valeur entre 0 et 4, on prendra simplement le reste de la division du rang de la lettre par 5 (« modulo 5 »). En résumé la fonction h est définie par :

$$h(titre) = rang(titre[0]) \, mod \, 5$$

La Fig. 4.1 montre la table de hachage obtenue avec cette fonction. Tous les films commençant par *a*, *f*, *k*, *p*, *u* et *z* sont affectés au bloc 1 ce qui donne, pour notre ensemble de films, *Annie Hall*, *Psychose* et *Underground*. les lettres *b*, *g*, *l*, *q* et *v* sont affectées au bloc 2 et ainsi de suite. Notez que la lettre *e* a pour rang 5 et se trouve donc affectée au bloc 0.

La Fig. 4.1 présente, outre les cinq fragments $f_0, \cdots, f_4$ stockant des films, le répertoire à cinq *entrées* permettant d'associer une valeur entre 0 et 4 à l'adresse d'un fragment sur le disque. Ce répertoire fournit une indirection entre l'identification *logique* du bloc et son emplacement physique, selon un mécanisme déjà rencontré dans la partie du chapitre *Dispositifs de stockage* consacrée aux techniques d'adressage de blocs. Comme déjà indiqué, on peut raisonnablement supposer que sa taille est faible et qu'il peut donc résider en mémoire principale, même pour de très grandes structures de hachage avec des milliers de fragments.

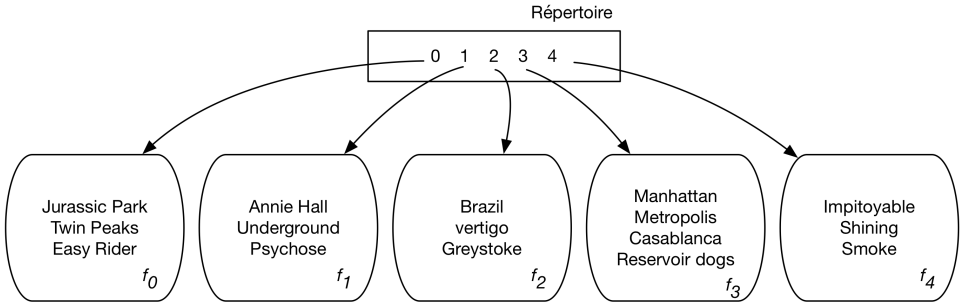

Fig. 4.1 – Exemple d'une table de hachage

On est assuré avec cette fonction d'obtenir toujours un chiffre entre 0 et 4, mais en revanche la distribution risque de ne pas être uniforme : si, comme on peut s'y attendre, beaucoup de titres commencent par la lettre *l*, le bloc 2 risque d'être surchargé. et l'espace initialement prévu s'avèrera insuffisant.

Dans le pire des cas, une fonction de hachage mal conçue affecte tous les enregistrements à la même adresse, et la structure dégénère vers un simple fichier séquentiel (cas d'une fonction renvoyant toujours à par exemple). Il faudrait utiliser un calcul beaucoup moins sensible à ce genre de biais ; prendre par exemple les 4 ou 8 premiers caractères de la chaînes, traiter ces caractères commes des entiers dont on effectue la somme, définir la fonction sur le résultat de cette somme.

4.1.2 Recherche dans une table de hachage

La structure de hachage permet les recherches par titre. Reprenons notre exemple favori :

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

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

— La structure n'occupe aucun espace disque additionnel aux données elles-mêmes, contrairement à l'arbre-B ;
— elle permet d'effectuer les recherches par clé par accès direct (calculé) au fragment susceptible de contenir les enregistrements.

Sauf exception (ici la recherche par la première lettre du titre), la recherche par préfixe n'est plus possible. La hachage ne permet pas non plus d'optimiser les recherches par intervalle, puisque l'organisation des enregistrements ne s'appuie pas sur l'ordre des clés. La requête suivante par exemple ne peut être résolule que par le parcours de tous les blocs de la structure, même si trois films seulement sont concernés.

```
select *
from  Film
where titre betseen 'Annie Hall' and 'Easy Rider'
```

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

4.1.3 Mises à jour

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

Cette situation est illustrée dans la figure Fig. 4.2. On a inséré un nouveau film, *Citizen Kane*. La valeur donnée par la fonction de hachage est 3, rang de la lettre "c" dans l'aplhabet, mais le bloc 3 est déjà plein.

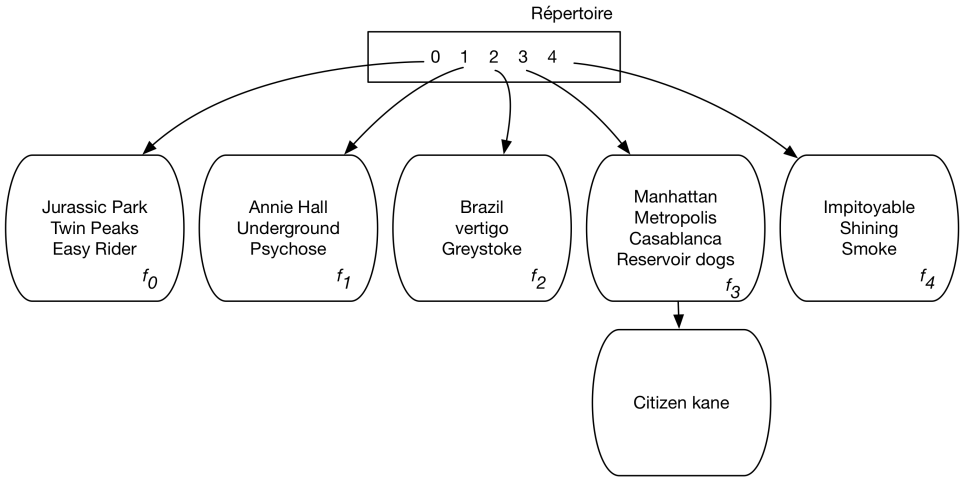

Fig. 4.2 – Table de hachage avec page de débordement

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

Il est très important de noter qu'il est impossible, avec les principes exposés ci-dessus, de modifier la structure de hachage en ajoutant des fragments et en modifiant le répertoire. Cela impliquerait en effet la modification de la fonction de hachage elle-même. On aurait donc une différence entre la fonction utilisée pour le stockage et celle utilisée pour la recherche.

Supposez que l'on ajoute un fragment, la fontion de hachage devient $modulo(rang, 6)$. La recherche du film "Impitoyable" avec cette fonction nous dirigerait vers le fragment 3 et on ne trouverait pas le film. Toute

modification de la fonction de hachage rend la structure obsolète, et il faut la reconstruire entièrement.

Autrement dit, ce type de hachage n'est pas *dynamique* et ne permet pas, d'une part d'évoluer parallèlement à la croissance ou décroissance des données, d'autre part de s'adapter aux déviations statistiques par rapport à la normale. Il faudrait reconstruire périodiquement la structure en fonction de son évolution. C'est un défaut majeur par rapport à la réorganisation dynamique de l'arbre B.

En résumé, les avantages et inconvénients du hachage statique, comparé à l'arbre-B, sont les suivantes :

— **Avantages**, : (1) recherche par accès direct, en temps constant ; (2) n'occupe pas d'espace disque.
— **Inconvénients** : (1) pas de recherche par intervalle ; (2) pas de dynamicité.

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

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

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

4.1.4 Quiz

— À quoi sert le *répertoire* de la structure de hachage ?
— Pourquoi la fonction de hachage doit-elle être uniforme ?
— Ma clé primaire est un idenfifiant séquentiel. Que peut-on dire du stockage de deux enregistrements consécutifs ?
— Quel est, dans le pire des cas, le coût d'une recherche par clé dans une structure de hachage statique ?

4.2 S2 : Hachage extensible

Supports complémentaires :

— Diapositives du hachage extensible:
— Vidéo sur le hachage extensible

Nous présentons tout d'abord le hachage extensible sur un exemple avant d'en donner une description plus générale. Dans un premier temps, la structure est tout à fait identique à celle que nous avons vue précédemment, à ceci près que le nombre d'entrées dans le répertoire est variable, et toujours égal à une puissance de 2.

Maintenant nous supposons donnée une fonction de hachage $h(c)$ qui s'applique à une valeur de clé c et dont le résultat est toujours un entier sur 4 octets, soit 32 bits. Cette fonction est immuable. Le tableau suivant donne les valeurs obtenues par application de cette fonction aux titres de nos films.

Titre	$h(titre)$
Vertigo	01110010
Brazil	1010010
Twin Peaks	11001011
Underground	01001001
Easy Rider	00100110
Psychose	01110011
Greystoke	10111001
Shining	11010011

Comme il n'y a que deux entrées, nous nous intéressons seulement au premier de ces 32 bits, qui peut valoir 0 ou 1. La figure Fig. 4.3 montre l'insertion des cinq premiers films de notre liste, et leur affectation à l'un des deux blocs. Le film *Vertigo* par exemple a pour valeur de hachage 01110010 qui commence par 0, et se trouve donc affecté à la première entrée.

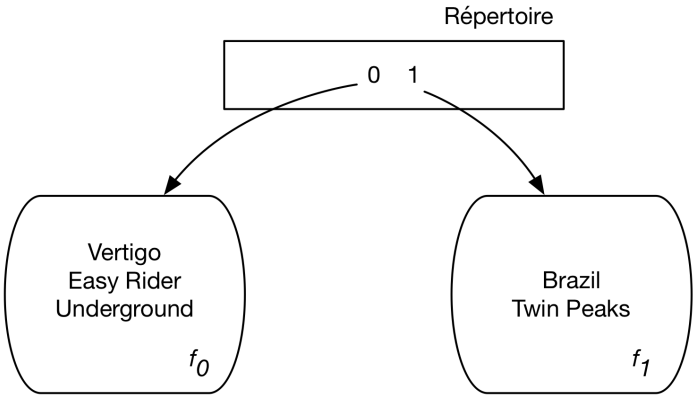

Fig. 4.3 – Hachage extensible avec 2 entrées

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

On va alors doubler la taille du répertoire pour la faire passer à quatre entrées, avec pour valeurs respectives 00, 01, 10, 11, soit les 2^2 combinaisons possibles de 0 et de 1 sur deux bits. Ce doublement de taille du répertoire entraine la réorganisation suivante (Fig. 4.4) :

En détail :

— les films de l'ancien fragment 0 sont répartis sur les fragments 00 et 01 en fonction de la valeur de leurs deux premiers bits : *Easy Rider* dont la valeur de hachage commence par 00 est placé dans le premier fragment, tandis que *Vertigo*, *Underground* et *Psychose*, dont les valeurs de hachage commencent par 01, sont placées dans le second fragment.

— les films de l'ancien fragment 1 n'ont pas de raison d'être répartis puisqu'il n'y a pas eu de débordement pour cette valeur, : *on va donc associer le même fragment aux deux entrées 10 et 11*.

Maintenant on insère *Greystoke* (valeur 10111001) et *Shining* (valeur) 11010011. Tous deux commencent par 10 et doivent donc être placés dans le troisième fragment qui déborde alors. Ici il n'est cependant pas

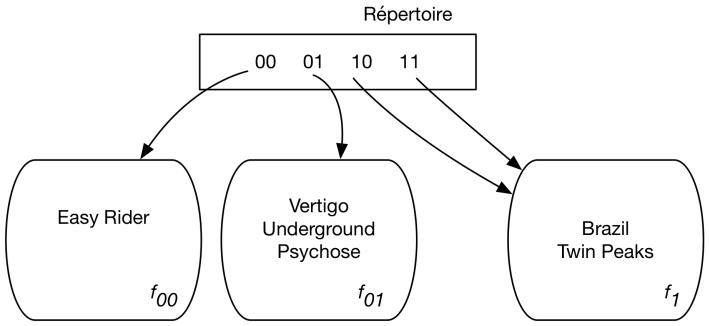

Fig. 4.4 – Doublement du répertoire dans le hachage extensible

nécessaire de doubler le répertoire puisqu'on est dans une situation où plusieurs entrées de ce répertoire pointent sur le même fragment.

On va donc allouer un nouveau fragment à la structure, et l'associer à l'entrée 11, l'ancien fragment restant associé à la seule entrée 10. Les films sont répartis dans les deux fragments, *Brazil* et *Greystoke* avec l'entrée 10, *Twin Peaks* et *Shining* avec l'entrée 11 (Fig. 4.5).

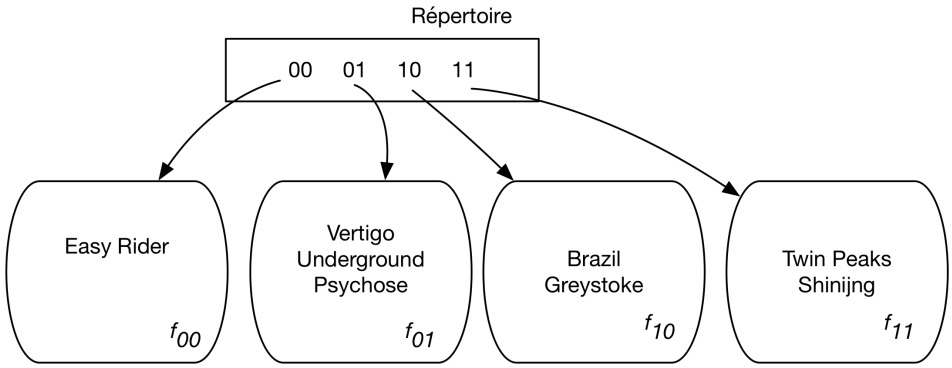

Fig. 4.5 – Jeu de pointeurs pour éviter de doubler le répertoire

En résumé, il n'y a que deux cas

- **Cas 1** : on insère dans un fragment plein, mais plusieurs entrées pointent dessus. On alloue alors un nouveau fragment, et on répartit les adresses du répertoire pour référencer les deux fragments.
- **Cas 2** : on insère dans un fragment plein, associé à une seule entrée. On double à nouveau le nombre d'entrées

La technique est simple et résout en partir le principal défaut du hachage, l'absence de dynamicité. L'inconvénient potentiel est que le répertoire tend à croître de manière exponentielle, ce qui peut soulever un problème à terme. Le hachage extensible reste par ailleurs une structure plaçante qui doit être complétée par l'arbre B pour des index secondaires.

4.2.1 Quiz

— Qu'est-ce qui caractérise le hachage extensible ?
— Quelle est la taille maximale du répertoire
— Combien ajoute-t-on de fragments quand l'un deux déborde
— Combien d'entrées du répertoire peuvent référencer le même fragment ?

4.3 S3 : hachage linéaire

Supports complémentaires :
— Diapositives du hachage linéaire:
— Vidéo de présentation

Le but du hachage linéaire est de maintenir une structure de hachage efficace quand le jeu de données est très dynamique, et en particulier quand il croît très rapidement. Cette maintenance implique une extension progressive du répertoire de hachage et de la fonction de hachage, ainsi que l'ajout de nouveaux fragments. L'apport du hachage linéaire est d'incrémenter à la fois le répertoire et les fragments *proportionnellement* aux besoins de stockage, et d'éviter le doublement systématique du répertoire.

Le point de départ du hachage linéaire est identique à celui du hachage extensible. Nous supposons donnée une fonction de hachage $h(c)$ qui s'applique à une valeur de clé c et dont le résultat est toujours un entier sur 4 octets, soit 32 bits. Le tableau suivant donne les valeurs que nous allons utiliser pour illustrer le hachage linéaire sur quelques-uns de nos films.

Titre	$h(titre)$
Vertigo	14
Brazil	43
Twin Peaks	25
Underground	20
Easy Rider	8
Psychose	33
Greystoke	17
Shining	16
Citizen Kane	44

La structure est celle d'une table de hachage classique, avec un répertoire dont chaque entrée référence un fragment. Nous prenons comme point de départ la situation de la Fig. 4.6 qui ressemble en tous points à celle du hachage extensible, à une (petite) exception près : un paramètre spécial, *l'indice de partitionnement* est ajouté à la structure. Appelons-le p. Sa valeur initiale est 0.

La fonction de hachage utilisée pour la structure de la Fig. 4.6 est $h(c)\ mod\ 2$, que nous noterons h_1. Plus généralement, on va considérer la suite de fonctions $h_0, h_1, h_2, \cdots$ définie par

$$h_i(c) = h(c)\ mod\ 2^i$$

En français : le résultat de $h_i(c)$ est le reste de la division de c par 2^i.

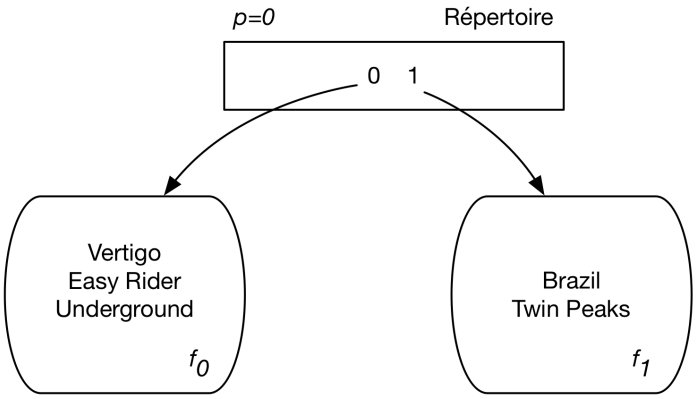

Fig. 4.6 – Structure initiale pour le hachage linéaire.

Quand un fragment f_i déborde, les actions suivantes sont effectuées.

- Un bloc de débordement est chaîné à f_i pour stocker le nouvel enregistrement.
- Le fragment f_p est éclaté en (f_p, f'_p), son contenu réparti dans les deux nouveaux fragments (f_p, f'_p), et *p* est incrémenté de 1.

Regardons ce qui se passe après insertion de Psychose, puis de Easy Rider. Tous les deux ont des valeurs impaires pour $h(c)$ et sont donc placés dans le fragment f_1 qui déborde. Parallèlement à ce débordement, le fragment f_0 est éclaté et son contenu réparti entre f_0 et un nouveau fragment f_2, comme le montre la Fig. 4.6.

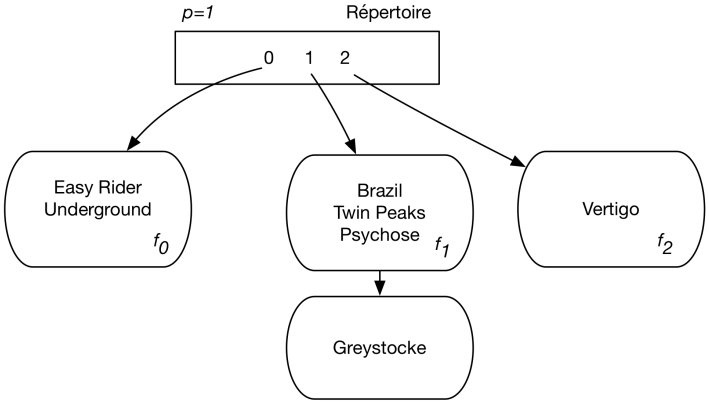

Fig. 4.7 – Après débordement de f_1 et éclatement de f_0

En éclatant le fragment f_0, on a utilisé comme fonction de répartition le successeur de h_1, h_2, et pris donc en compte le reste de la division par $2^2 = 4$. *Underground* et *Easy Rider*, dont les valeurs respectives de $h(c)$ sont 24 et 8, restent donc dans f_0, tandis que Vertigo (valeur 14, avec pour reste de la division 2) est déplacé dans f_2.

À ce stade, on constate donc que *deux fonctions de hachage cohabitent* : h_1 et h_2. Comment savoir laquelle utiliser ? Le critère est simplement déterminé par le paramètre *p*. Le hachage linéaire repose toujours sur une paire de fonctions (h_n, h_{n+1}). Initiallement, cette paire est (h_0, h_1), et comme *p=0*, h_0 s'applique à tous les

fragments. Au fur et à mesure de l'évolution de la structure suite à des éclatements, p est incrémenté et h_0 s'applique seulement aux fragments dont l'indice est supérieur ou égal à p, et h_1 à tous les autres.

Continuons notre exemple en insérant successivement Shining (valeur 16) puis Citizen Kane (valeur 48). Tous deux vont dans f_0 qui déborde. Il faut donc éclater le fragment désigné par la valeur courante de p, f_1, et incrémenter p. On se retrouve dans la situation de la Fig. 4.8.

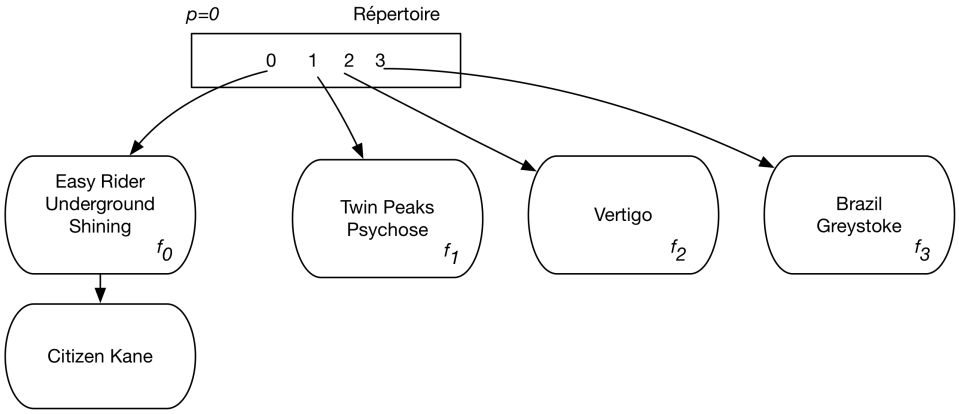

Fig. 4.8 – Après débordement de f_0 et éclatement de f_1

Que constate-t-on ? La structure a un nouveau bloc de débordement, mais celui de f_1 a disparu. Pourquoi ? Parce qu'en incrémentant p, la fonction h_1 s'applique maintenant à f_1, ce qui conduit à répartir les enregistrements initialement présents soit dans f_1 (si le modulo 4 de la clé est 1) ou le nouveau fragment f_3 (si le modulo 4 et 3).

Résumons. Dans la structure de hachage linéaire, quand un fragment déborde, un nouveau fragment est chaîné. On se retrouve dans la situation du hachage statique, le chaînage introduisant une indirection pénalisante pour les recherches. Si on s'en tenait là il n'y aurait aucun progrès. Mais le hachage linéaire va plus loin en incrémentant également le nombre de valeurs de hachage et effectuant un éclatement de l'un des fragments de la structure, *mais pas forcément de celui qui vient de déborder*. En fait, le débordement d'un fragment agit comme une sorte de signal que la structure doit évoluer et s'agrandir, et on le fait mais dans un ordre déterminé à l'avance. *C'est ce découplage entre la constatation d'un débordement d'un côté, et l'éclatement d'un fragment de l'autre*, qui constitue l'idée - brillante - fondant l'organisation du hachage linéaire.

On accepte donc d'avoir des blocs de débordement, mais uniquement de manière temporaire, car, à terme, le fragment qui a débordé et consiste en plusieurs blocs chaînés sera éclaté à son tour, et le chaînage disparaitra. On peut interpréter le principe comme étant celui d'une désynchronisation entre la croissance des données et le nécessaire éclatement des fragments.

On peut noter également qu'après cet éclatement, h_0 n'est plus utilisé. La paire de fonction requises pour la structure devient (h_1, h_2) et p est réinitialisé à 0 : on va recommencer une séquence d'éclatement des fragments, en partant de 0, dans l'ordre. Toutes les puissances de 2, on « décale » d'un niveau la paire de fonction de hachage, et on dispose de toutes les informations nécessaires pour gérer les insertions et les recherches. Le petit code suivant est à la base de l'identification du fragment contenant un enregistrement de clé c, la paire courante de fonctions étant (h_n, h_{n+1}).

```
$a := h_n(h(c));
if ($a < $p)  $a := h_{n+1}(h(c))
```

En clair : on applique d'abord h_n, en supposant que le fragment concerné n'a pas encore éclaté. On obtient une valeur de hachage a. Si a est inférieure à p, c'est :math :` h_{n+1}` qu'il faut utiliser.

4.3.1 Quiz

— Quelle affirmation est vraie pour le hachage linéaire ?
— Comment sait-on quelle fonction de hachage s'applique à une clé ?
— Pourquoi accepte-t-on de faire des chaînages dans le hachage linéaire ?

4.4 Exercices

Exercice ex-hach1 : construction d'un hachage statique

Reprenons une liste de 12 départements, à lire de gauche à droite et de bas en haut.

```
3  Allier; 36 Indre; 18 Cher; 75 Paris
39 Jura; 9  Ariège; 81 Tarn; 11 Aude
12 Aveyron; 25 Doubs; 73 Savoie; 55 Meuse;
```

La clé étant le numéro de département et on suppose qu'un bloc contient 5 enregistrements.
— Proposez une fonction de hachage et le nombre d'entrées du répertoire, puis construisez une structure de hachage statique en prenant les enregistrements dans l'ordre indiqué.

Exercice ex-hach2 : avec hachage extensible

Même exercice, mais avec une structure basée sur le hachage extensible. La fonction de hachage est la suivante : $h(nom) = i_1 i_2 \cdots i_4$ avec $i_j = 1$ si la lettre $nom[i_j]$ est en position impaire dans l'alphabet, et 0 sinon. Donc $f(Aude) = 1101$. Voici la liste des valeurs de hachage, en ne prenant que les 4 premiers bits.

```
Allier  1001    Indre   1000    Cher    1010    Paris   0101
Jura    0101    Ariège  1011    Tarn    0100    Aude    1101
Aveyron 1011    Doubs   0110    Savoie  1101    Meuse   1111
```

— On suppose toujours 5 enregistrements par bloc. Choisissez le nombre de bits initial de la structure de hachage en fonction du nombre de départements à indexer, et donnez la structure obtenue.
— Prenez les départements suivants, dans l'ordre indiqué (de gauche à droite, puis de haut en bas) et donnez les évolutions de la structure de hachage extensible.

```
Cantal  1100    Marne   1100    Loire   0110    Landes  0100
Calvados 1110   Gard    1110    Vaucluse 0111   Ardèche 0011
```

Exercice ex-hach3 : un peu de hachage linéaire

Expliquez l'évolution du hachage linéaiere en partant de la Fig. 4.8 et en insérant deux films, Metropolis donc le code de hachage est 49, et Manhattan donc le code est 5.

CHAPITRE 5

Moteurs de stockage

Ce chapitre propose un passage en revue de quelques-uns des principaux systèmes relationnels pour étudier la manière dont il gèrent le stockage des données et les paramètres qui permettent aux administrateurs de les ajuster. Cette étude est essentiellement destinée à montrer une mise en pratique concrète des principes généraux détaillés précédemment et ne prétend pas être une référence suffisante.

5.1 S1 : Oracle

Supports complémentaires :
— Vidéo sur le stockage oracle

Le système de représentation physique d'Oracle est riche et illustre assez complètement les principes exposés dans les chapitres précédents. Un système Oracle (une *instance* dans la documentation) stocke les données dans un ou plusieurs *fichiers*. Ces fichiers sont entièrement attribués au SGBD, qui est seul à organiser leur contenu. Ils sont divisés en *blocs* dont la taille – paramétrable – peut varier de 1K à 8K. Au sein d'un fichier des blocs consécutifs peuvent être regroupés pour former des *extensions* (« *extent* »). Enfin un ensemble d'extensions permettant de stocker un des objets physiques de la base (une table, un index) constitue un *segment*.

Il est possible de paramétrer, pour un ou plusieurs fichiers, le mode de stockage des données. Ce paramétrage comprend, entre autres, la taille des extensions, le nombre maximal d'extensions formant un segment, le pourcentage d'espace libre laissé dans les blocs, etc. Ces paramètres, et les fichiers auxquels ils s'appliquent, portent le nom de *tablespace*.

Nous revenons maintenant en détail sur ces concepts.

5.1.1 Fichiers et blocs

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

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

Les blocs ORACLE

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

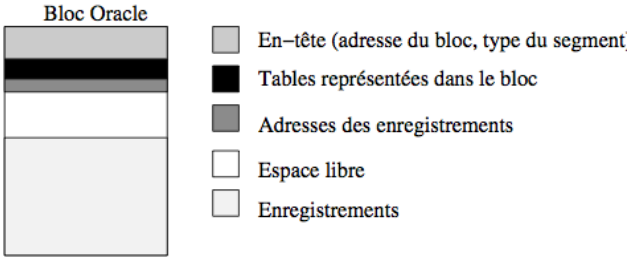

Fig. 5.1 – Structure d'un bloc Oracle

La structure d'un bloc est identique quel que soit le type d'information qui y est stocké. Elle est constituée des cinq parties suivantes (Fig. 5.1) :

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

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

Les paramètres `pctfree` et `pctused`

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

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

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

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

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

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

Enregistrements

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

Chaque attribut est précédé de la longueur de stockage. Dans Oracle les valeurs `NULL` sont simplement représentées par une longueur de 0. Cependant, si les *n* derniers attributs d'un enregistrement sont `NULL`, Oracle

se contente de placer une marque de fin d'enregistrement, ce qui permet d'économiser de l'espace.

Chaque enregistrement est identifié par un `ROWID`, comprenant plusieurs parties, dont, notamment :
- le numéro du bloc au sein du fichier ;
- le numéro de l'enregistrement au sein du bloc ;
- enfin l'identifiant du fichier.

Un enregistrement peut occuper plus d'un bloc, notamment s'il contient les attributs de type `LONG`. Dans ce cas Oracle utilise un *chaînage* vers un autre bloc. Un situation comparable est celle de l'agrandissement d'un enregistrement qui va au-delà de l'espace libre disponible. Dans ce cas Oracle effectue une *migration* : l'enregistrement est déplacé en totalité dans un autre bloc, et un pointeur est laissé dans le bloc d'origine pour ne pas avoir à modifier l'adresse de l'enregistrement (`ROWID`). Cette adresse peut en effet être utilisée par des index, et une réorganisation totale serait trop coûteuse. Migration et chaînage sont bien entendu pénalisants pour les performances.

Extensions et segments

Un segment est un ensemble de fragments de stockage (les « extensions, voir ci-dessous) pour un des types de données persistantes géré par Oracle. Il existe de nombreux types de segments, voici les principaux :
- les segments de données contiennent les enregistrements des tables, avec un segment de ce type par table ;
- les segments d'index contiennent les enregistrements des index ; il y a un segment par index ;
- les segments temporaires sont utilisés pour stocker des données pendant l'exécution des requêtes (par exemple pour les tris) ;
- les segments *rollbacks* contiennent les informations permettant d'effectuer une reprise sur panne ou l'annulation d'une transaction ; il s'agit typiquement des données avant modification, dans une transaction qui n'a pas encore été validée.

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

Le nombre de blocs dans une extension peut être spécifié par l'administrateur. Bien entendu des extensions de taille importante favorisent de bonnes performances, mais il existe des contreparties :
- si une table ne contient que quelques enregistrements, il est inutile de lui allouer une extension contenant des milliers de blocs ;
- l'utilisation et la réorganisation de l'espace de stockage peuvent être plus difficiles pour des extensions de grande taille.

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

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

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

réorganiser l'espace dans les fichiers existant.

5.1.2 Les *tablespaces*

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

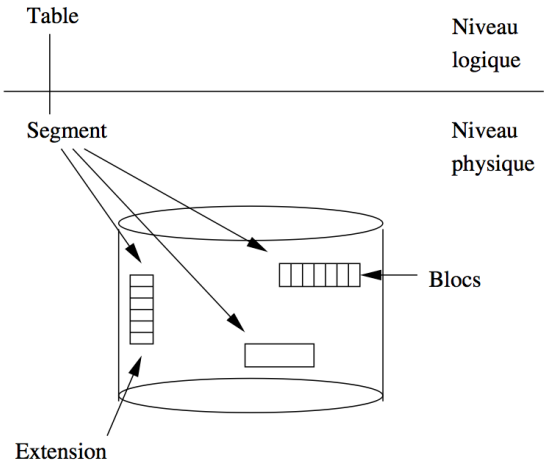

Fig. 5.2 – Organisation des *tablespaces* Oracle

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

— adaptation du mode de stockage en fonction d'un type de données particulier ;
— affectation d'un espace disque limité aux utilisateurs ;
— contrôle sur la disponibilité de parties de la base, par mise hors service d'un ou plusieurs *tablespaces* ;
— enfin – et surtout – répartition des données sur plusieurs disques afin d'améliorer les performances.

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

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

— la taille de l'extension initiale (par défaut 5 blocs) ;
— la taille de chaque nouvelle extension (par défaut 5 blocs également) ;

— le nombre maximal d'extensions, ce qui donne donc, avec la taille des extensions, le nombre maximal de blocs alloués à une table ou index ;
— la taille des extensions peut croître progressivement, selon un ratio indiqué par `pctincrease` ; une valeur de 50% pour ce paramètre indique par exemple que chaque nouvelle extension a une taille supérieure de 50% à la précédente.

Voici un exemple de création de *tablespace*.

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

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

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

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

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

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

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

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

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

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

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

```
ALTER TABLESPACE TB1 OFFLINE;
```

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

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

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

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

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

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

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

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

Comment inspecter les *tablespaces*

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

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

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

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

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

On obtient quelque chose qui ressemble à :

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

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

Avec un résultat :

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

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

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

Résultat :

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

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

5.1.3 Création des tables

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

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

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

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

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

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

5.2 S2 : MySQL

Supports complémentaires :
— Vidéo sur le stockage MySQL

Une importante spécificité de MySQL par rapport à d'autres SGBD est de proposer des moteurs de stockage différents et même de permettre leur cohabitation dans une même base de données. Il s'agit d'une souplesse assez exceptionnelle, car les moteurs de stockage ont des comportements spécifiques quant à la manière de conserver les données des tables et index, et sont donc plus ou moins adaptés selon les contextes d'utilisation.

Les deux moteurs étudiés ici sont MyISAM et InnoDB. Le premier est efficace et compact, mais ne gère pas les transactions, au contraire du second.

On peut choisir le moteur de stockage au moment de la création d'une table. La syntaxe est la suivante.

```
create table <nomTable> (...) engine <nomMoteur>
```

Nous avons choisi de ne pas parler des autres moteurs de stockage dont voici une brève description.

— `memory` (ou `HEAP`) qui permet de gérer des tables en mémoire principale et peut ponctuellement être utile pour stocker efficacement des tables temporaires.
— `Archive` qui compresse les données et permet de diminuer les coûts de stockage pour des volumes importants. En revanche, il n'accepte que les instructions `select` et `insert`. Il es utile surtout pour l'archivage.
— `Ndb` est un moteur de stockage dédié aux systèmes distribués (répartition dans une grappe de serveurs.
— *enfin, `Maria`* est le moteur de stockage de la version *Open source* nommée MariaDB depuis l'acquisition de MySQL par Oracle en 2010.

MySQL s'appuie sur les fichiers du système d'exploitation, soit complètement dans le cas de MyISAM qui associe une base à un répertoire et crée dans ce répertoire des fichiers pour chaque table, soit partiellement pour InnoDB qui stocke plusieurs tables dans un même fichier.

Chaque ligne d'une table relationnelle est stockée dans un enregistrement physique dans un fichier. Nous ne rentrerons pas dans le détail de la structure d'un enregistrement qui varie selon les moteurs de stockage. Un enregistrement contient des données dites de contrôle (date de création, de modification, liens vers d'autres versions, taille de l'enregistrement) qui servent au système en plus des valeurs des attributs.

5.2.1 MyISAM

MyISAM est le premier moteur de stockage de MySQL. Par défaut (autrement dit en l'absence de spécification d'un moteur) c'est lui qui stocke les tables. Le moteur MyISAM nécessite peu de volume : les données y sont stockées en tas, sans utiliser de blocs.

Le stockage en tas est très simple et non organisé : chaque nouvel enregistrement est placé soit dans le premier emplacement libre plus grand que lui, soit à la suite du dernier enregistrement. Quand un enregistrement est détruit, il libère une place qui peut être réutilisée ensuite à l'occasion de l'insertion d'un enregistrement de taille inférieure ou égale.

MyISAM n'utilise pas de *cache* pour les données et ne propose aucun mécanisme transactionnel. Une lecture (d'un enregistrement de données) est toujours physique (accès au disque) et une écriture est toujours faite immédiatement et remplace l'ancienne version de l'enregistrement. La modification est immédiatement visible des autres utilisateurs.

Pour chaque table MyISAM, trois fichiers sont créés par défaut dans le répertoire de la base de données :
— `<nomTable>.frm` : la description de la table `nomtable` ;
— `<nomTable>.MYD` : les données de la table `nomtable` ;
— `<nomTable>.MYI` : les index de la table `nomtable`, dont un index unique pour la clé primaire ;

Puisque MyISAM n'utilise pas de cache pour les données des tables, celles-ci restent toujours cohérentes sur le disque, même si le serveur s'arrête de manière inopinée. Le moteur MyISAM n'a donc pas besoin d'un journal des transactions (tout se tient), même s'il existe un journal de sauvegarde destiné aux reprises à chaud. Les seules incohérences qui peuvent apparaître seront dans les index, qui peuvent être réparés par reconstruction à partir des données.

En revanche, les index sont conservés dans un cache. Toute la stratégie d'évaluation des requêtes s'appuie sur l'hypothèse que les index seront utilisés. La présence des index en cache (partiellement ou totalement) est alors une garantie de très grande efficacité.

On peut modifier l'emplacement des fichiers au niveau de chaque table en positionnant deux paramètres lors de l'instruction `create table`. Ces paramètres permettent de répartir les entrées / sorties :
— `DATA DIRECTORY` : emplacement du fichier de données ;

— INDEX DIRECTORY, : emplacement du fichier des index.

Voici un exemple de répartition des données et des index.

```
CREATE TABLE Client
    (id_client INT AUTO_INCREMENT NOT NULL,
    prénom VARCHAR(50) NOT NULL,
    nom VARCHAR(50) NOT NULL,
    adresse VARCHAR(255) NOT NULL,
    ville   VARCHAR(60) NOT NULL,
    code_postal VARCHAR(10) NOT NULL,
    PRIMARY KEY (id_client)
 ) DATA DIRECTORY = '/disk1/Credit'
    INDEX DIRECTORY = '/disk2/index/Credit';
```

Quand on est sûr de disposer de deux disques physiques (attention, pas des volumes logiques), il est important, pour de gros volumes, de placer les index et les données sur des disques différents. Puisque le serveur accède à ces fichiers indépendamment, les accès s'en trouveront parallélisés. Si on ne maîtrise pas complètement la répartition physique des disques, un stockage par défaut fera l'affaire, éventuellement sur des périphériques RAID.

5.2.2 InnoDB

InnoDB est le moteur par défaut depuis la version 5.5. C'est un moteur de stockage qui apporte essentiellement deux fonctionnalités par rapport à MyISAM :

— un support complet pour les transactions ;

— la prise en compte des contraintes d'intégrité référentielle.

InnoDB utilise une technique très différente de MyISAM. Les données sont organisées par blocs, et InnoDB utilise un cache pour conserver en mémoire les plus utilisés. InnoDB est surtout un moteur transactionnel qui propose tous les mécanismes de validation, annulation et cohérence décrits par la norme SQL99 (voir chap-introconc). Le modèle de stockage et de transactionindex{transaction}s est assez nettement inspiré de celui d'ORACLE.

InnoDB propose deux options pour l'affectation des données aux fichiers de la base :

— un seul ensemble de fichiers, donné par le paramètre `innodb_data_file_path`. Toutes les tables et index seront répartis dans ces fichiers ;

— un fichier par table.

Dans tous les cas, un fichier `table}.frm` sera créé dans le sous-répertoire de la base. Comme pour MyISAM, il contient la description de la table. Les paramètres de stockage sont donnés à la création du fichier, et spécifiés dans le fichier de configuration. Il s'agit :

— de l'emplacement des fichiers de la base

— de leur taille

— et éventuellement d'une option d'extension automatique.

Par exemple on indiquera un fichier `mydata.ibd` de taille initiale de 100 Mo et qui peut s'étendre automatiquement jusqu'à 1 Go avec la spécification suivante, dans le fichier `my.cnf` :

```
innodb_data_file_path=mydata.ibd:100M:autoextend:1G
```

Quand on veut ajouter un fichier il suffit de le spécifier dans le fichier de configuration (seul le dernier fichier peut avoir l'option `autoextend`).

```
innodb_data_file_path=mydata.ibd:100M;mydata2.idb:100M:autoextend:1G
```

Si on choisit d'activer le paramètre `innodb_file_per_table`, la table et ses index sont stockés dans un fichier du sous-repertoire de la base. Le fichier est auto-extensible et porte le nom `table.ibd`}.

InnoDB organise les tables d'après l'index principal sur la clé primaire. Il s'agit d'une structure non-dense, donc *plaçante* : l'emplacement des enregistrements n'est pas libre, mais se détermine d'après la structure de l'index associé. Entre autres caractéristiques, le stockage est ordonné selon la clé primaire, ce qui peut considérablement améliorer la clause `order by` ou certains algorihmes de jointures car aucune indirection n'est nécessaire pour accéder aux données de la table.

Ce mode de stockage est un peu plus compliqué à gérer. Les insertions sont un peu plus lentes en moyenne que dans un stockage en tas. C'est un bon compromis si la base est plus lue qu'approvisionée en données nouvelles. Il est d'ailleurs proposé (au moins à titre d'option) dans la plupart des SGBD (dans Oracle par exemple il correspond aux `index-organized tables`).

Il est fortement déconseillé de modifier la clé primaire d'une ligne dans InnoDB car cela implique une réorganisation importante de la structure, y compris des index secondaires.

Le cache de données InnoDB utilise un mécanisme de liste LRU (*Least Recently Used*) pour gérer la montée et le recyclage des blocs en mémoire. Quand une donnée nécessaire est dans un bloc qui se trouve dans le cache, celle-ci est remise en tête de la liste. Si le bloc n'est pas dans la liste, il est lu sur le disque et monté en tête de la liste ; le bloc en fin de liste est alors supprimé.

Puisque InnoDB utilise un mécanisme de cache pour les données, ces dernières sont d'abord modifiées dans le cache, et se retrouvent alors incohérentes avec la version stockée sur disque. On pourrait penser à écrire immédiatement pour mettre les deux versions en concordance, mais cela entraînerait des entrées/sorties aléatoires très coûteuses. La stratégie « paresseuse » consiste donc à attendre que le bloc contenant la donnée arrive en bout de liste LRU et soit finalement écrit sur le disque. Dans l'intervalle entre la mise à jour et l'écriture, le disque est une image incohérente des données modifiées en cache.

Un mécanisme est donc nécessaire pour reconstruire cette cohérence en cas d'arrêt brutal du serveur prog{mysqld}. InnoDB conserve toutes les modifications des enregistrements dans un *journal des transactions*. Le chapitre *Reprise sur panne* détaille les mécanismes de reprise sur panne basés sur le journal des transactions.

Quel moteur choisir ?

Le moteur de stockage doit être choisi en fonction du type d'accès à la base de données, des besoins en performances et des exigences transactionnelles.

Pour les bases de données essentiellement consultées et chargées périodiquement par des traitements batches ou des transactions très simples voire atomiques, on choisira le moteur MyISAM. Il a un modèle de stockage simple et utilise peu d'espace disque, ce qui le rend performant. Les bases de ce type seront par exemple des catalogues, des annuaires ou des bases décisionnelles.

Pour les bases avec de fortes exigences transactionnelles, c'est-à-dire dont les traitements métiers comportent de nombreuses instructions de mise à jour avec une exigence forte de cohérence, on choisira le moteur InnoDB qui offre la gestion des transactions. Par exemple, les bases financières ou de gestion de stock nécessitent une cohérence qui justifie l'emploi de InnoDB.

Par ailleurs, si des requêtes effectuent de nombreuses jointures sur des tables de volume important, on pourra choisir le moteur InnoDB. En effet, le stockage en cluster sur la clé primaire et l'utilisation d'un cache de données permettent d'accélerer les jointures qui concernent beaucoup de lignes. Les traitements de reporting ou d'édition massives peuvent en bénéficier.

Pour toutes les tables qui stockent des données temporairement lors d'un traitement, on peut envisager le moteur `memory`. C'est souvent le cas pour les chargements et transformations des bases de données décisionnelles.

Pour les données qui ne seront plus modifiées, on pourra utiliser le moteur `archive` qui compresse les données. Il est conseillé d'écrire un traitement serveur spécifique pour archiver massivement les données qui doivent l'être. Par exemple, on pourra archiver chaque premier du mois les données du mois précédent.

Il est conseillé de stocker les données sur des systèmes de fichiers dédiés. Cela permet de mieux gérer l'espace disque. Pour cela on utilisera des liens symboliques. La répartition sur des disques distincts des données et des index (possible pour MyISAM, pas pour InnoDB), est tout à fait recommandée. Elle est indispensable dans le cas des fichiers journaux et des fichiers de données.

5.3 S3 : SQL Server

Pas de support écrit, mais présentation vidéo par Nicolas Travers

5.4 S4 : Postgres

Pas de support écrit, mais présentation vidéo par Nicolas Travers

CHAPITRE 6

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'évaluation de requêtes : le modèle d'exécution, les opérateurs algébriques, 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.

6.1 S1 : Modèle d'exécution : les itérateurs

Supports complémentaires :
- Diapositives: les itérateurs
- Vidéo de présentation du modèle d'exécution

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 resources matérielles nécessaires, et en particulier la mémoire RAM. Nous commençons par étudier en détail ce dernier aspect.

6.1.1 Matérialisation et pipelinage

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 naive 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 nuplets à l'application (Fig. 6.1).

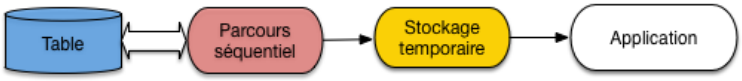

Fig. 6.1 – Exécution d'une requête avec matérialisation

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 (Fig. 6.2).

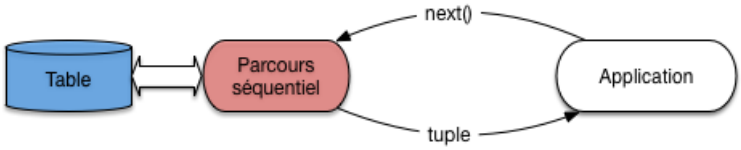

Fig. 6.2 – Exécution d'une requête avec pipelinage

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 premiers nuplets 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 nuplet produit, 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 nuplet alors que le plan d'exécution peut fournir 20 nuplets 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 les temps passés dans chaque phase *s'additionneraient*.

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 un mécanisme d'accès toujours identique, consistant à :

— ouvrir un curseur par exécution d'une requête ;
— avancer le curseur sur le résultat, nuplet par nuplet, 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 Fig. 6.2. L'application demande un nuplet 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.

6.1.2 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 * from T order by *
```

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 nuplet ; 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 reçues dès que possible.

6.1.3 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 nuplets résultats : elle alloue les resources 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 le nuplet 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
```

(suite sur la page suivante)

(suite de la page précédente)

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

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

```
function nextScan
{
   # Avançons sur le nuplet 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 plaçant *avant* le premier enregistrement du premièr bloc de T. Chaque appel à *nextScan* () retourne un enregistrement/nuplet. 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);
$nuplet = $curseur.next();

while [$nuplet != null ]
do
  # Traitement du nuplet
  ...
  # Récupération du nuplet suivant
  $nuplet = $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.

6.1.4 Quiz

Une application reçoit 100 000 nuplets, résultat d'une requête SQL. Le traitement par l'application de chaque nuplet prend 0,5 sec. Le SGBD met de son côté 0,2 secondes pour préparer chaque nuplet.

- En mode matérialisation, le temps de réponse est de
- En mode matérialisation, l'application finit de s'exécuter après
- En mode pipelinage, le temps de réponse est de
- En mode pipelinage, l'application finit de s'exécuter après
- Parmi les requêtes suivantes, quelles sont celles qui nécessitent un opérateur bloquant. (Aide : se demander s'il faut lire ou non toute la table avant de produire le premier nuplet du résultat).

6.2 S2 : les opérateurs de base

Supports complémentaires :

- Diapositives: les opérateurs de base
- Vidéo plans d'exécution (partie 1)
- Vidéo plans d'exécution (partie 2)

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 nuplets de la table sont examinés, dans l'ordre du stockage.
— *Accès par adresse*. Si on connaît l'adresse du nuplet, 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 ne connaît pas la notion d'adresse physique, cette 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.

6.2.1 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 limitée par le débit du disque.

6.2.2 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 *Structures d'index : 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 parcours les niveaux de l'index en partant de la racine
  $bloc = $I.racine();
  while [$bloc.estUneFeuille() = false]
```

(suite sur la page suivante)

(suite de la page précédente)

```
  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 nuplets. 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
 }
```

6.2.3 Accès par adresse

Quand on connait 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.

6.2.4 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 nuplets; $C est la condition de␣
↪sélection

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

  return $nuplet;
 }
```

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 la 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 explicitement dans le plan d'exécution.

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

6.2.5 Exécution de requêtes mono-tables

Reprenons 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 Fig. 6.3.

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 nuplet correspondant. Enfin, chaque nuplet 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 nuplets. 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 monotable, 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`), suivi 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

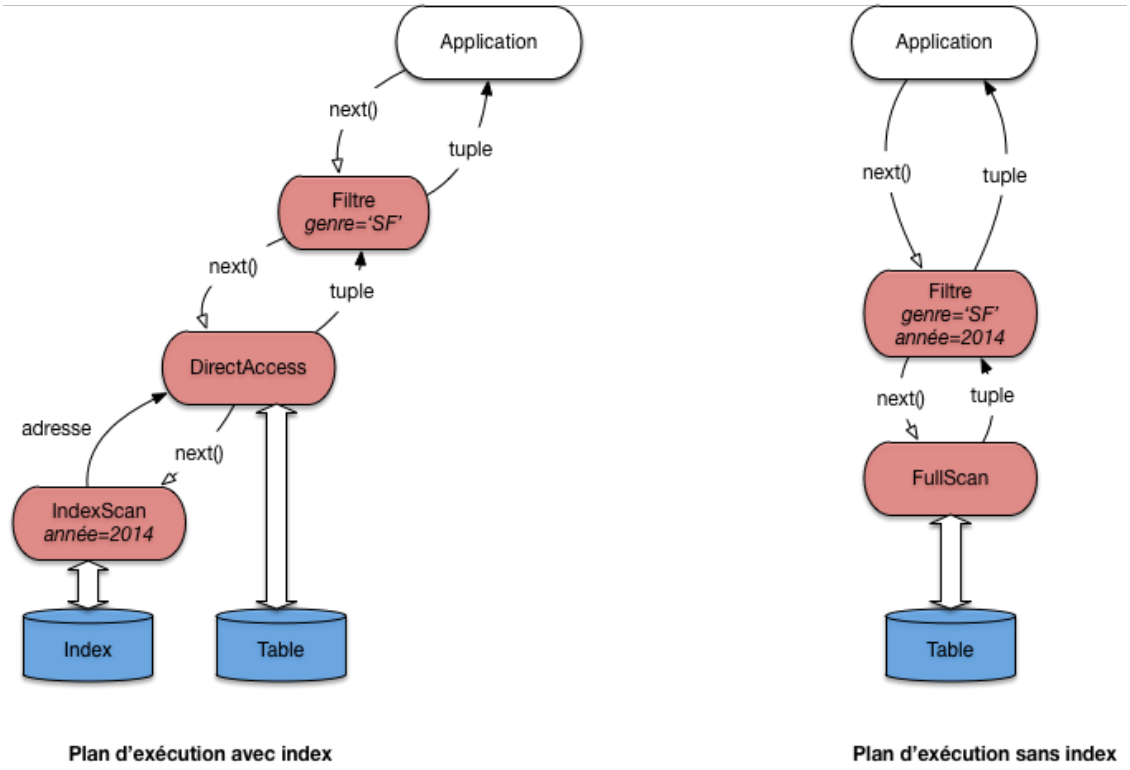

Fig. 6.3 – Deux plans d'exécution pour une requête monotable

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 nuplets 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.
- 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 Fig. 6.4. 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).

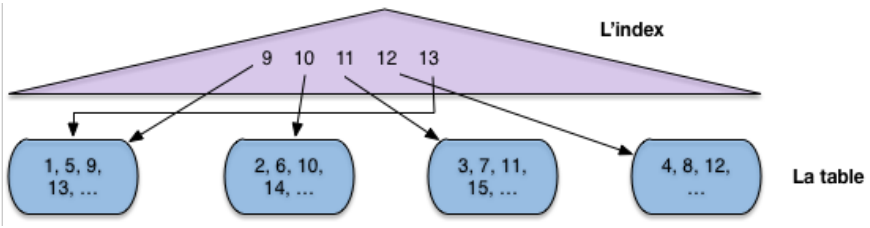

Fig. 6.4 – Recherche par intervalle avec index

L'index en revanche s'appuie sur l'ordre des clés (il s'agit ici typiquement d'un arbre B, voir chapitre *Structures d'index : 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 :

— 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 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 du moins celles qui semblent plausibles) celles qui vont donner le meilleur résutat. 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 Fig. 6.3 (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 Fig. 6.3, à droite.

6.2.6 Quiz

— L'opérateur de parcours d'index (`IndexScan`) fournit, à chaque appel `next()`

— On suppose qu'il existe un index sur l'année, quelles sont parmi les requêtes suivantes celles dont le plan d'exécution combine `IndexScan` et `DirectAccess` :

— Parmi les requêtes suivantes, quelles sont celles qui peuvent s'évaluer avec un plan d'exécution comprenant seulement un `IndexScan` :

— Dans le second plan d'exécution, celui basé sur un index, peut-on inverser les opérateurs d'accès direct et de filtre ?

— Reprenez l'exemple du cours d'un accès avec un index pour un intervalle [9,13]. Supposons maintenant que le fichier est trié sur la clé. Combien faudra-t-il lire de blocs dans le pire des cas, en supposant qu'un bloc contient 10 nuplets ?

— Si le fichier est trié sur la clé de l'index, est-il toujours préférable de passer par l'index pour une recherche par intervalle ?

6.3 S3 : Le tri externe

Supports complémentaires :
- Diapositives: le tri-fusion
- Algorithme de tri

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* tels 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.

6.3.1 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 (Fig. 6.5). On recommence avec les M blocs suivants du fichier, jusqu'à ce que tout le fichier ait été lu par fragments de M blocs,

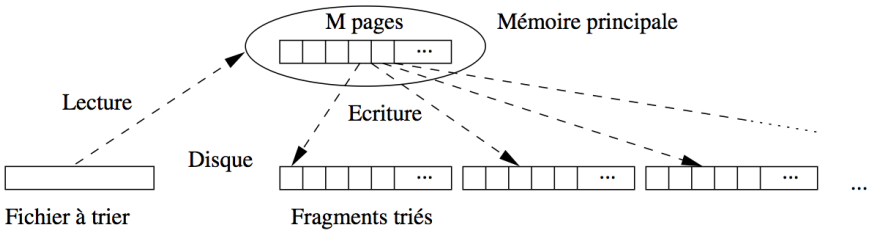

Fig. 6.5 – Algorithme de tri-fusion : phase de tri

À l'issue de cette phase on a $\lceil B/M \rceil$ fragments triès, 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.

6.3.2 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 Fig. 6.6 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.

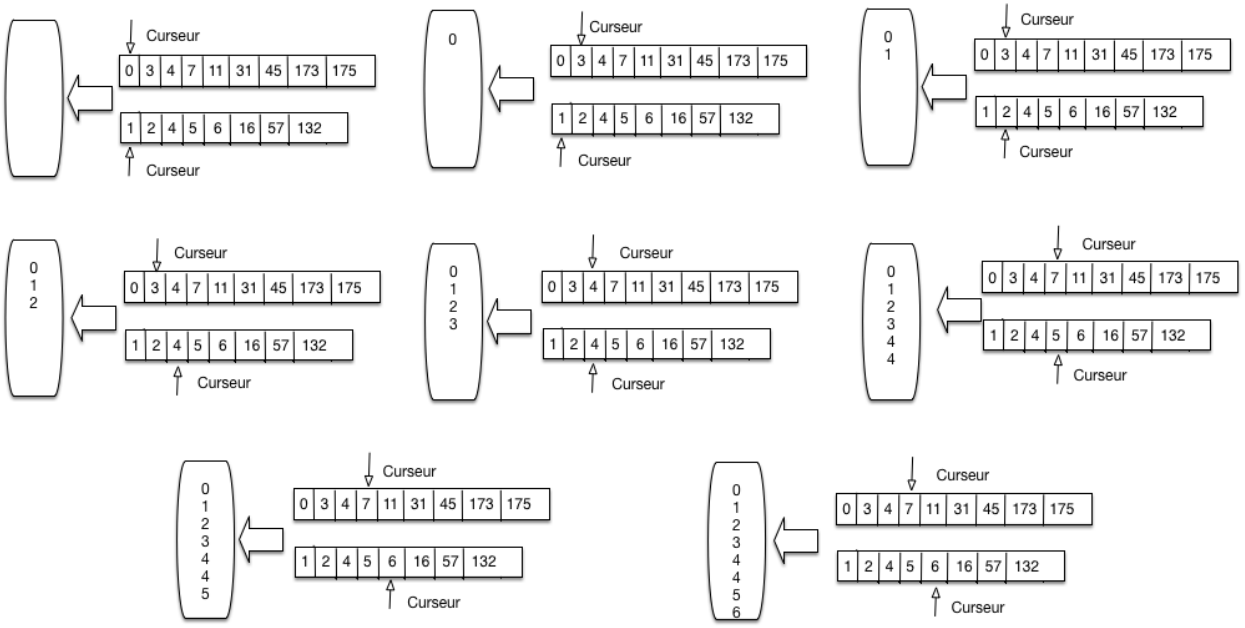

Fig. 6.6 – Parcours linéaire pour la fusion de listes triées

La Fig. 6.6 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 l1 et l2
function fusion
{
  # $l1 désigne la première liste
  # $l2 désigne la seconde liste
  $resultat = [];
  # Début de la fusion des listes
  while ($l1 != null and $l2 != null) do
    if ($l1.val == $l2.val) then
      # On a trouvé un doublon
      $resultat += $l1.val;
          $resultat += $l2.val;
      # Avançons sur les deux listes
      $l1 = $l1.next; $l2 = $l2.next;
    else if ($l1.val < $l2.val) then
      # Avançons sur l1
          $resultat += $l1.val;
      $l1 = $l1.next;
    else
       # Avançons sur l2
          $resultat += $l2.val;
      $l2 = $l2.next;
    fi
  done
}
```

Remarquons que :

— L'algorithme *fusion* se généralise (assez facilement) à plusieurs listes.

— Si on fusionne n listes de taille M, la liste résultante a une taille de $n \times M$.

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 Fig. 6.7 résume la phase de fusion sous la forme d'un arbre, chaque nœud (agrandi à droite) correspondant à une fusion de $M-1$ partitions.

Un exemple est donné dans la Fig. 6.7 sur un ensemble de films qu'on trie sur le nom du film. Il y a trois phases de fusion, à partir de 6 fragments initiaux que l'on regroupe 2 à 2.

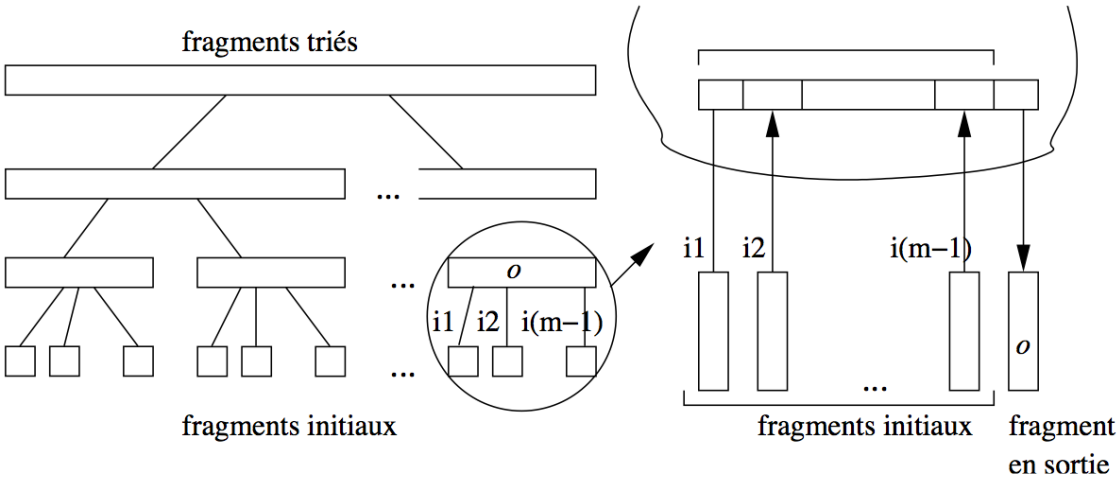

Fig. 6.7 – Algorithme de tri-fusion : la phase de fusion

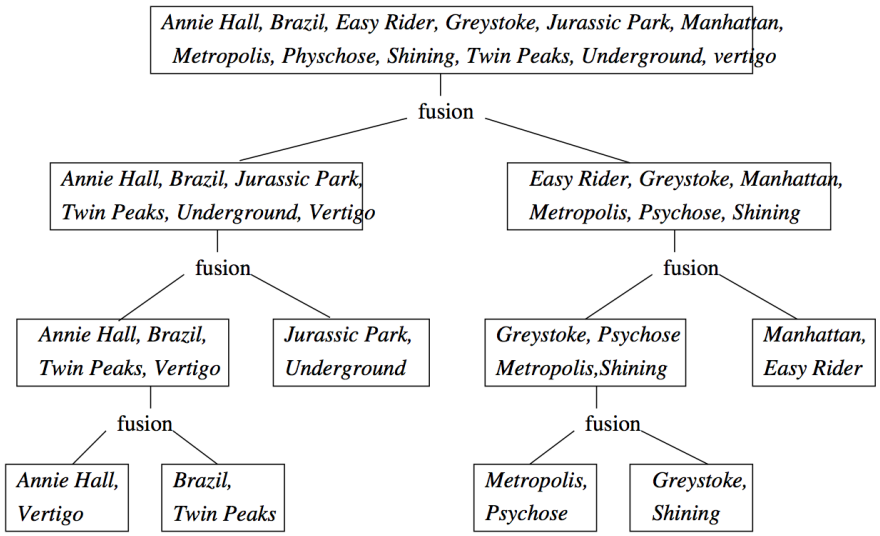

Fig. 6.8 – Algorithme de tri-fusion : un exemple

6.3.3 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, 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 $M > 307$Mo, tout le fichier peut être chargé et trié en mémoire. Une seule lecture suffit.
- Avec une mémoire $M = 2$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 $M = 1$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 la dernière 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 reprsente 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.

6.3.4 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 nuplets 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 nuplet 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.

6.3.5 Quiz

— Avec une mémoire de M blocs, quelle est la taille des fragments ?
— Avec une mémoire de $M + 1$ blocs, je peux fusionner M fragments. Quelle est la taille maximale de fichier obtenu par une étape de fusion ?
On en déduit la taille maximale d'un fichier triable avec une fusion.
— Je peux fusionner les fragments issus d'une fusion de fragments. Avec deux étapes de fusion, la taille maximale du fichier trié obtenu est de :
— Quelle est la formule permettant d'obtenir le nombre de fusions nécessaires pour trier un fichier de taille F :

6.4 S4 : Algorithmes de jointure

Supports complémentaires :
— Diapositives: algorithmes de jointure
— Vidéo algorithmes de jointure (partie 1)
— Vidéo algorithmes de jointure (partie 2)

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'est-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 compliqué.

```
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 d'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.

6.4.1 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 nuplet, 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 nuplet ;
— il reste à effectuer un accès direct, avec l'adresse *adr*, pour obtenir le second nuplet 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_réalisateur`. Pour chaque nuplet `film` obtenu durant ce parcours, on prend la valeur de `id_réalisateur` 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 un troisième, lui aussi déjà étudié : `DirectAccess`. La forme du programme qui effectue ce type de jointure est illustrée par la Fig. 6.9. 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.

L'opérateur `IndexedJoin` lui-même fait peu 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 nuplet
  $nuplet = $tScan.next();

  # On crée un opérateur de parcours d'index
  iScan = new IndexScan ();
  # On exécute le parcours d'index avec la clé étrangère
  $iScan.open ($nuplet.foreignKey);
```

(suite sur la page suivante)

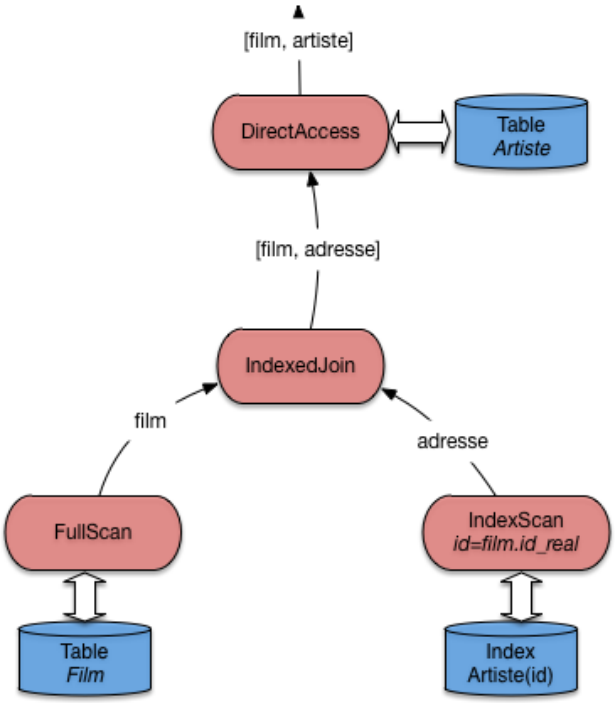

Fig. 6.9 – Algorithme de jointure avec index

(suite de la page précédente)

```
  # On récupère l'adresse
  addr = $iScan.next();

  # Et on renvoie la paire avec le nuplet et l'adresse
  return [$nuplet, $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 nuplets 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.

6.4.2 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'efectuer 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.

6.4.3 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 prédicats de jointure. Il consiste à énumérer tous les enregistrements dans le produit cartésien de R et S (en d'autres termes, toutes les paires possibles) et garde ceux qui satisfont le prédicat de jointure. La 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
  resultat = [];

  for nuplet1 in $L1
  do
   for nuplet2 in $L2
   do
     if (condition ($nuplet1, $nuplet2) = true) the
       $resultats[] = ($nuplet1, $nuplet2);
     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 nuplet de `L1` est comparé à chaque nuplet 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 Fig. 6.10, consiste à charger toutes les paires de blocs en mémoire, et à appliquer la fonction `JoinList` sur chaque paire.

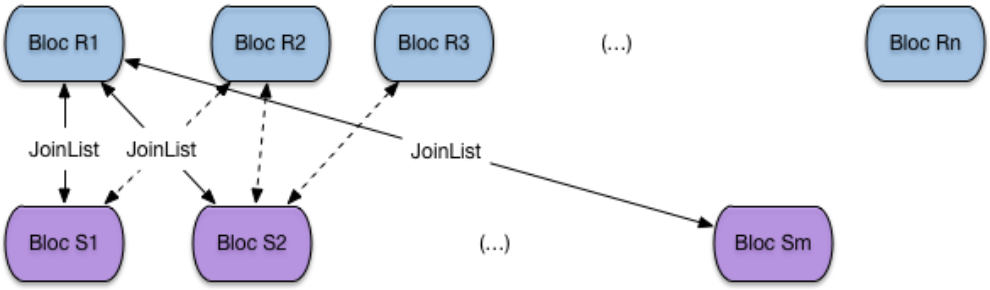

Fig. 6.10 – Boucle imbriquée sur les blocs

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 NestedLoopJoin
{
  # $R est la tablee dite "extérieure"
  # $S est la table dite "intérieure"
  # $condition est la condition de jointure
  resultat = [];

  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 (Fig. 6.11).

Avec cette solution (très fréquemment applicable en ces temps où la mémoire RAM est devenue très grosse), le coût est de $B_R + B_S$: une seule lecture des deux tables suffit. D'un coût quadratique dans les tailles des

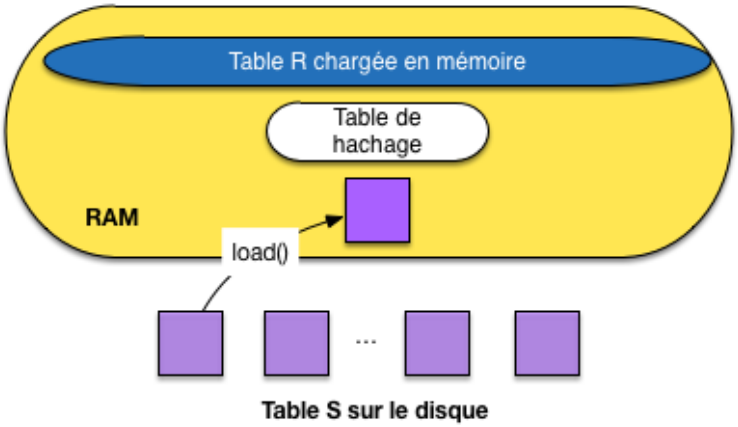

Fig. 6.11 – Boucle imbriquée avec chargement complet d'une table en RAM

relations, lorsqu'on n'a que 3 blocs, on est passé à un coût linéaire. Cet algorithme en devient très efficace et simple à implanter.

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).

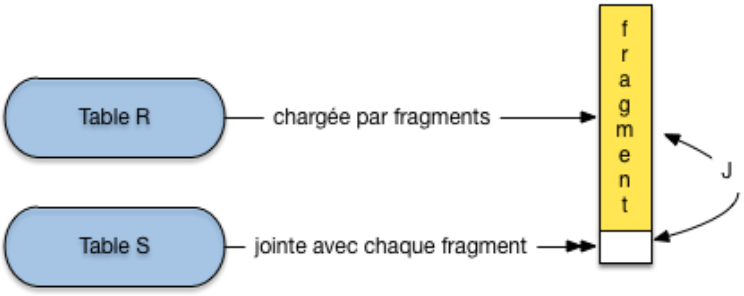

Fig. 6.12 – Boucle imbriquée avec chargement par fragments d'une table en RAM

Si R ne tient pas en mémoire car $B_R > M - 1$, il reste la version la plus générale de la jointure par boucles imbriquées (Fig. 6.12) : on découpe R en fragments de taille $M - 1$ blocs et on utilise la variante ci-dessus pour chaque groupe. R est lue une seule fois, groupe par groupe, S est lue $\lceil \frac{B_R}{M-1} \rceil$ fois. On obtient un coût final de :

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

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 $M = 251$ blocs.

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

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

— Et en prenant la table Film comme table extérieure :

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

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.

6.4.4 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 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 triés, 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'agorithme 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 nuplets fournis par les opérateurs
  $nupletR = $triR.next();
  $nupletS = $triS.next();

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

 return [$nupletR, $nupletS];
}
```

Le plan d'exécution typique d'une jointure par tri-fusion avec cet opérateur est illustré par la Fig. 6.13.

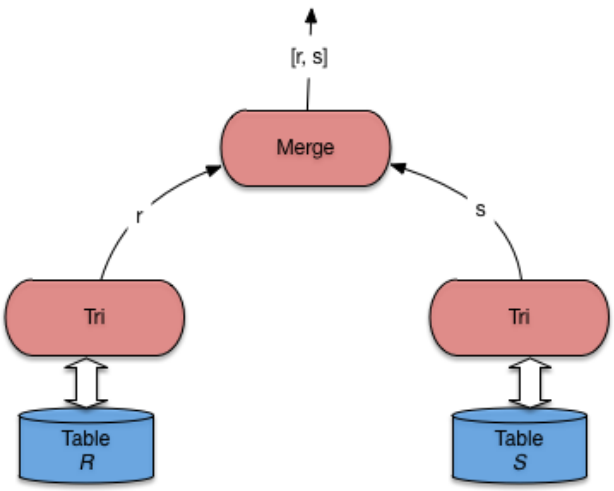

Fig. 6.13 – Plan d'exécution type pour la jointure par tri-fusion

La jointure s par tri-fusion est illustrée dans la Fig. 6.14.

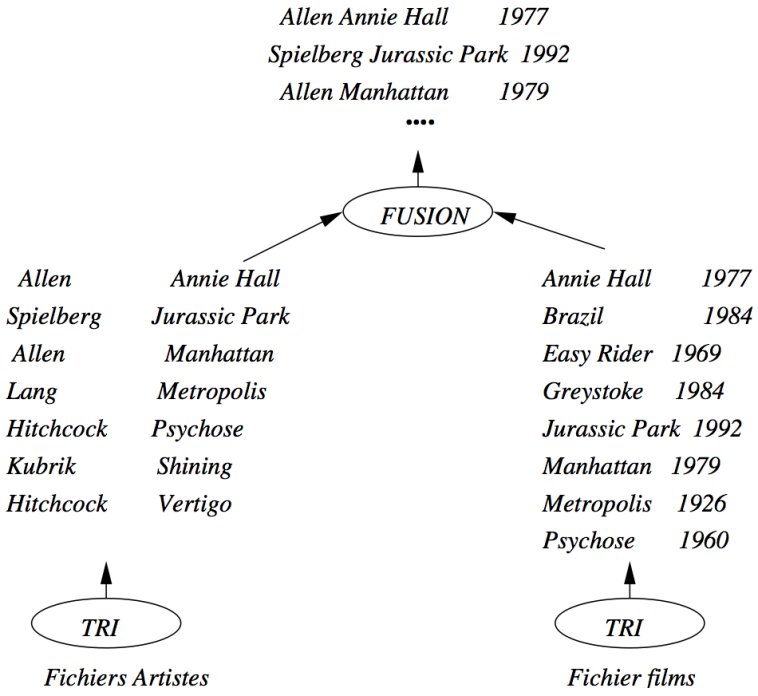

Fig. 6.14 – Exemple de jointure par tri-fusion

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).

6.4.5 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 fragements sont associés par paire, ce qui implique que l'on a la garantie qu'aucun nuplet de F_R^i ne joint avec un nuplet 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*.

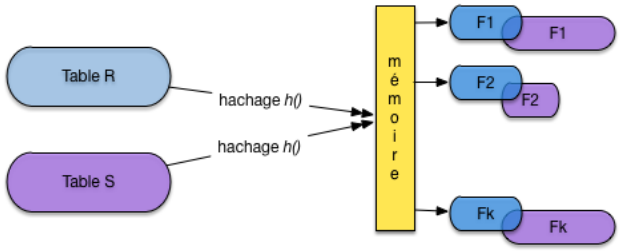

Fig. 6.15 – Première phase de la jointure par hachage : le partitionnement

---

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.

On peut alors passer à la seconde phase, dite de jointure. La remarque fondamentale ici est la suivante : si deux nuplets 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 sont fonctionnement. Lisez-le et relisez-le jusqu'à être convaincus que vous le comprennez.

La deuxième phase consiste alors pour $i = 1, ..., k$, 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 Fig. 6.16 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.

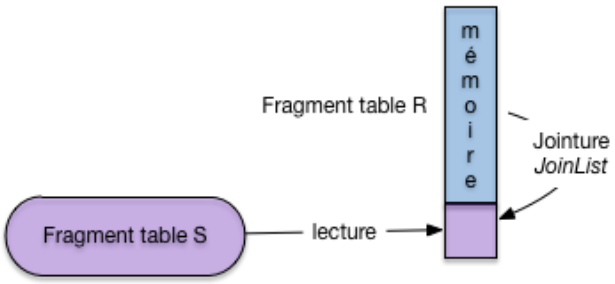

Fig. 6.16 – Première phase de la jointure par hachage : la jointure

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 $B_R + B_S$. 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. Il est bien adapté aux jointures déséquilibrées pour lesquelles une des tables est petite par rapport à lamémoire RAM disponible. 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 cet 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 nuplets 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.

La tendance est à l'utilisation de plus en plus fréquente de la joiunture par hachage qui remplace l'algorithme de tri-fusion qui était privilégié dans les premiers temps des SGBD relationnels. La taille atteinte par les mémoires RAM est sans doute le principal facteur explicatif de ce phénomène.

6.4.6 Quiz

— Pourquoi la clé primaire d'une table doit-elle être indexée (plusieurs réponses possibles) :

— Considérons les tables des employés et des départements suivantes. Les clés primaires sont indexées.

Enum	Nom	Dnum
E1	Benjamin	D1
E2	Philippe	D2
E3	Serge	D1

Dnum	Dnom
D1	INRIA
D2	CNAM

Pour une jointure avec index, combien de parcours d'index doit-on effectuer ?

— Supposons que l'attribut `Dnum` dans la table `Employé` soit indexé. Combien de parcours d'index devrait-on effectuer en prenant la table `Dept` comme table directrice (à gauche).
— Dans la requête suivante, peut-on appliquer la jointure par boucles imbriquées indexées, *f()* et *g()* étant des fonctions quelconques ?

```
select * from Emp, Dept where f(E.dnum) = g(Dept.dnum)
```

— Soit la jointure entre deux tables T(ABCD) et S(MNO) dans la requête suivante :

```
select * from T, S where T.A=S.M
```

À quels attributs faut-il appliquer la fonction de hachage pour la jointure ?
— Pourquoi 2 nuplets à joindre sont-ils forcément dans des fragments associés ?
— Peut-on appliquer la jointure par hachage à la requête suivante :

```
select * from T, S where T.A <= S.M
```

6.5 Exercices

Exercice ex-iter1 : définition d'itérateurs

— Définir sous forme de pseudo-code (`open()` et `next()`) un itérateur `min` qui renvoie le nuplet de sa source ayant la valeur minimale pour un attribut `att_min`.
— Définir un itérateur `distinct` qui élimine les doublons de sa source.

Exercice ex-iter2 : plans d'exécution

Donner des plans d'exécution pour les requêtes suivantes :
— Avec clause `order by`

```
select titre from Film order by annee
```

— Recherche d'un élément minimal

```
select min(annee) from Film
```

— Elimination des doublons

```
select distinct genre from Film
```

Exercice ex-opalgo1 : comprendre le tri externe

Soit un fichier de 10 000 blocs et une mémoire cache de 3 blocs. On trie ce fichier avec l'algorithme de tri-fusion.
— Combien de fragments sont produits pendant la première passe ?

— Combien d'étapes de fusion faut-il pour trier complètement le fichier ?
— Quel est le coût total en entrées/sorties ?
— Combien faut-il de blocs en mémoire faut-il pour trier le fichier en une fusion seulement.
— Répondre aux mêmes questions en prenant un fichier de 20 000 blocs et 5 blocs de mémoire *cache*.

Exercice ex-join1 : coût des jointures par boucles imbriquées

Soit deux relations R et S, de tailles respectives $|R|$ et $|S|$ (en nombre de blocs). On dispose d'une mémoire *mem* de taille M, dont les blocs sont dénotés $mem[1], mem[2], \ldots, mem[M]$.

— Donnez la formule exprimant le coût d'une jointure $R \bowtie S$, en nombre d'entrées/sorties, pour l'algorithme suivant :

```
$posR = 1   # On se place au début de R
while [$posR <= |R|] do
    Lire  R[$posR] dans $mem[1] # On lit les blocs 1 par 1
    $posS = 1  # On se place au début de S
    while ($posS <= |S|)  do
       Lire  S[$posS] dans $mem[2] # On lit les blocs 1 par 1
       # JoinList est l'algorithme donné en cours
       JoinList (mem[1], mem[2])
      done
      $posS = $posS + 1 # Bloc suivant de S
    done
   $posR = $posR + 1 # Bloc suivant de R
 done
```

— Même question avec l'algorithme suivant

```
$posR = 1   # On se place au début de R
while [$posR <= |R|] do
   Lire  R[p$osR..($posR+M-1)] dans $mem[1..M-1] # On lit M-1 blocs␣
↪de R
   $posS = 1
   while ($posS <= |S|) do
       Lire  S[$posS] dans $mem[2] # On lit les blocs 1 par 1
       # JoinList est l'algorithme donné en cours
      JoinList (mem[1], mem[2])
   done
   $posS = $posS + 1 # Bloc suivant de S
  done
  $posR = $posR + M # On lit les M blocs suivants de R
 done
```

— Quelle table faut-il prendre pour la boucle extérieure ? La plus petite ou la plus grande, ?

Exercice ex-joincost : coût des jointures

On suppose que $|R| = 10\,000$ blocs, $|S| = 1\,000$ et *M=51*. On a 10 enregistrements par bloc, `b` est la clé primaire de *S* et on suppose que pour chaque valeur de `R.a` on trouve en moyenne 5 enregistrements dans *R*. On veut calculer $\pi_{R.c}(R \bowtie_{a=b} S)$.

— Donnez le nombre d'entrée-sorties *dans le pire des cas* pour les algorithmes par boucles imbriquées de l'exercice *ex-join1*.

— Même question en supposant (a) qu'on a un index sur *R.a*, (b) qu'on a un index sur *S.b*, (c) qu'on a deux index, sachant que dans tous les cas l'index a 3 niveaux.

— Même question pour une jointure par hachage.

— Même question avec un algorithme de tri-fusion.

CHAPITRE 7

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 resources 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 goulots 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.

7.1 S1 : Introduction à l'optimisation et à l'évaluation

Supports complémentaires :

— Diapositives:
— Vidéo d'introduction à l'optimisation

Commençons par situer le problème. Nous avons une requête, exprimée en SQL, soumise au système. Comme vous le savez, SQL permet de déclarer un besoin, mais ne dit pas comment calculer le résultat. C'est au système de produire une forme opératoire, un programme, pour effectuer ce calcul (Fig. 7.1), . Notez que cette approche a un double avantage. Pour l'utilisateur, elle permet de ne pas se soucier d'algorithmique d'exécution. Pour le système elle laisse la liberté du choix de la meilleure méthode. C'est ce qui fonde l'optimisation, la liberté de déterminer la manière de répondre a un besoin.

Fig. 7.1 – Les requêtes SQL sont *déclaratives*

En base de données, le programme qui évalue une requête a une forme très particulière. On l'appelle *plan d'exécution*. Il a la forme d'un arbre constitué d'opérateurs qui échangent des données (Fig. 7.2). Chaque opérateur effectue une tâche précise et restreinte : transformation, filtrage, combinaisons diverses. Comme nous le verrons, un petit nombre d'opérateurs suffit a évaluer des requêtes, même très complexes. Cela permet au système de construire très rapidement, a la volée, un plan et de commencer a l'exécuter. La question suivante est d'étudier comment le système passe de la requête au plan.

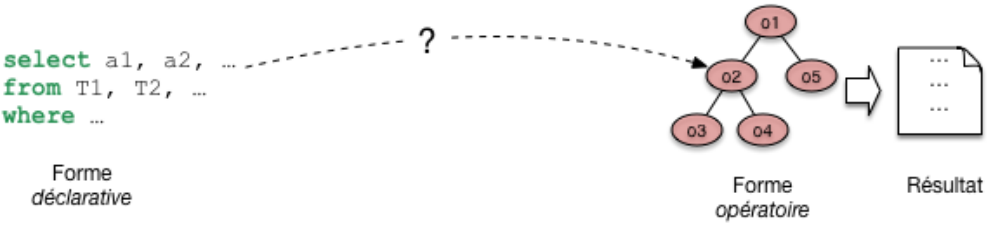

Fig. 7.2 – De la requête SQL au plan d'exécution.

Le passage de SQL a un plan s'effectue en deux étapes, que j'appellerai a) et b (Fig. 7.3). Dans l'étape a) on tire partie de l'équivalence entre SQL, ou une grande partie de SQL, avec l'algèbre. Pour toute requête on peut donc produire une expression de l'algèbre. Une telle expression est déjà une forme opérationnelle, qui nous dit quelles opérations effectuer. Nous l'appellerons plan d'execution logique. Une expression de l'algèbre peut se représenter comme un arbre, et nous sommes déjà proches d'un plan d'exécution. Il reste cependant assez abstrait.

Dans l'étape b) le système va choisir des opérateurs particulièrs, en fonction d'un contexte spécifique. Ce peut être là présence ou non d'index, la taille des tables, la mémoire disponible. Cette étape b) donne un plan

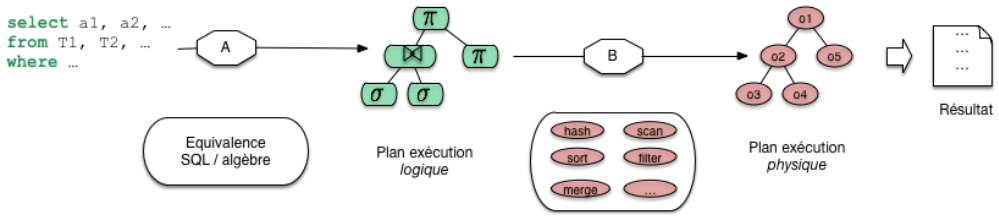

Fig. 7.3 – Les deux phases de l'optimisation

d'exécution *physique*, applicable au contexte.

Reste la question de l'optimisation. Il faut ici élargir le schéma : à chaque étape, a) ou b), plusieurs options sont possibles. Pour l'étape a), c'est la capacité des opérateurs de l'algèbre à fournir plusieurs expressions équivalentes. La Fig. 7.4 montre par exemple deux combinaisons possibles issues de la même requête sql. Pour l'étape b) les options sont liées au choix de l'algorithmique, des opérateurs à exécuter.

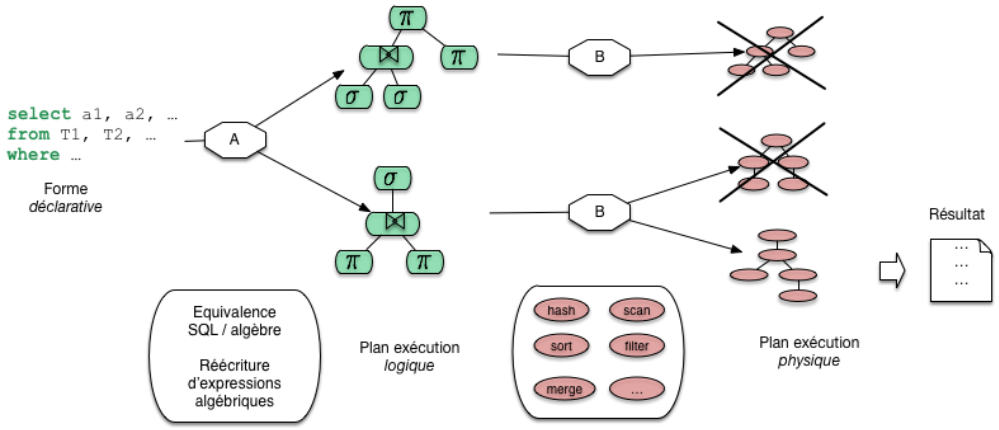

Fig. 7.4 – Processus général d'optimisation et d'évaluation

La Fig. 7.4 nous donne la perspective générale de cette partie du cours. Nous allons étudier les opérateurs, les plans d'exécution, les transformations depuis une requête SQL, et quelques critères de choix pour l'optimisation.

7.1.1 Quiz

— Dire qu'une requête est *déclarative*, c'est dire que (indiquer les phrases correctes) :
— Un plan d'exécution, c'est
— L'optimisation de requêtes, c'est
— Quelles sont les affirmations vraies parmi les suivantes ?

7.2 S2 : traitement de la requête

Supports complémentaires :
— Diapositives: traitement d'une requête

Cette section est consacrée à la phase de traitement permettant de passer d'une 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.

7.2.1 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 `Rôle` car il est important.

```
create table Rôle (id_acteur integer not null,
                   id_film integer not null,
                   nom_rôle 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, Rôle r, Artiste a
where  a.nom = 'Stewart' and a.prénom='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, Rôle 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 prénom='James')
```

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

```
select titre
from   Film f, Rôle r
where  f.id_film = r.id_film
and    f.annee = 1958
and    exists (select 'x'
               from Artiste a
               where nom='Stewart'
               and prénom='James'
               and r.id_acteur = a.id_acteur)
```

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

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

Que l'on peut aussi formuler en :

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

Dans les deux dernier 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 `Rôle` ?

La clé de `Rôle` 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 il 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 présente le risque 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.

7.2.2 Traduction et réécriture

Nous nous concentrons maintenant sur le traitement d'un bloc, étant entendu que ce traitement doit être effectué autant de fois qu'il y a de blocs dans une requête. Il comprend plusieurs phases. Tout d'abord une analyse syntaxique est effectuée, puis une traduction algébrique permettant d'exprimer la requête sous la forme d'un ensemble d'opérations sur les tables. Enfin l'optimisation consiste à trouver les meilleurs chemins d'accès aux données et à choisir les meilleurs algorithmes possibles pour effectuer ces opérations.

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, Rôle r
where  nom_rôle ='John Ferguson'
and    f.id = r.id_ilm
and    f.année = 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 :

$$\pi_{titre}(\sigma_{annee=1958 \wedge nom\_rle='\text{John Ferguson}'}(Film \bowtie_{id=id\_film} Rle)$$

Cette expression comprend des opérations unaires (un seul argument) et des opérations binaires. On peut la représenter sous la forme d'un arbre (Fig. 7.5), ou *Plan d'Exécution Logique* (PEL), représentant l'expression algébrique équivalente à la requête SQL. Dans l'arbre, les feuilles correspondent aux tables de l'expression algébrique, et les nœuds internes aux opérateurs. Un arc entre un nœud n et son nœud père p indique que l'"« opération p s'applique au résultat de l'opération n ».

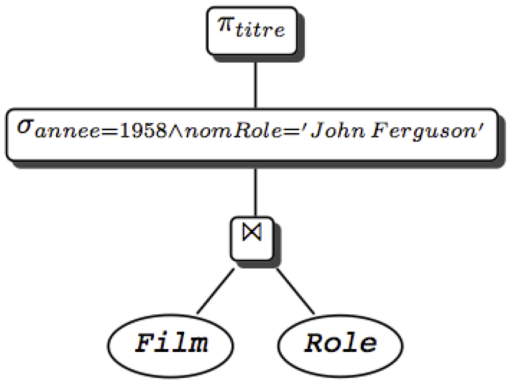

Fig. 7.5 – Expression algébrique sous forme arborescente

L'interprétation de l'arbre est la suivante. On commence par exécuter les opérations sur les feuilles (ici une jointure) ; sur le résultat, on effectue les opérations correspondant aux nœuds de plus haut niveau (ici une sélection), 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.

7.3 S3 : optimisation de la requête

Supports complémentaires :
- Diapositives: l'optimisation d'une requête
- Vidéo sur l'optimisation algébrique
- Vidéo sur les plans d'exécution (1)
- Vidéo sur les plans d'exécution (2)

7.3.1 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

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

— Commutativité de la sélection et de la jointure

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

— 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

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

— Distributivité de la projection sur l'union

$$\pi_{A_1 A_2 ... A_p}(R \cup S) \equiv \pi_{A_1 A_2 ... A_p}(R) \cup \pi_{A_1 A_2 ... A_p}(S)$$

Ces règles sont à la base du processus d'optimisation dont le principe est *d'énumérer* tous les plans d'exécution possibles. Par exemple la règle 3 permet de gérer finement l'affectation des sélections. En effet si la relation est indexée sur l'atttribut `B`, la règle justifie de filter sur `A` seulement les enregistrements satisfaisant le critère $B =' b'$ obtenus par traversée d'index. La commutatitivité 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 prédicats en plusieurs sélections à un prédicat (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.

$$\pi_{titre}(\sigma_{annee=1958 \wedge nom\_rle='John\ Ferguson'}(Film \bowtie_{id=id\_film} (Rle))$$

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

$$\pi_{titre}(\sigma_{annee=1958}(\sigma_{nom\_rle='John\ Ferguson'}(Film \bowtie_{id=id\_film} (Rle)))$$

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

$$\pi_{titre}(\sigma_{annee=1958}(Film) \bowtie_{id=id\_film} \sigma_{nom\_rle='John\ Ferguson'}(Rle))$$

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 du SGBD.

Cette transformation constitue la dernière étape de l'optimisation. Elle fait l'objet de la section suivante.

7.3.2 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.

$$\pi_{titre}(\sigma_{annee=1958}(Film) \bowtie_{id=id\_film} \sigma_{nom\_rle='John\ Ferguson'}(Rle))$$

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 `Rôle`, 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,

$$Film \bowtie_{id=id\_film} Rle$$

ou

$$Rle \bowtie_{id\_film=id} Film$$

Regardons pour quelle version nous pouvons utiliser un index avec l'algorithme `IndexNestedLoop`. Dans le premier cas, nous lisons des nuplets `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 nuplets 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.

$$\pi_{titre}(\sigma_{nom\_rle='John\ Ferguson'}(Rle) \bowtie_{id\_film=id} \sigma_{annee=1958}(Film))$$

Et, plus important, nous pouvons maintenant implanter ce plan avec l'algorithme de jointures imbriquées indexées, ce qui donne l'arbre de la Fig. 7.6.

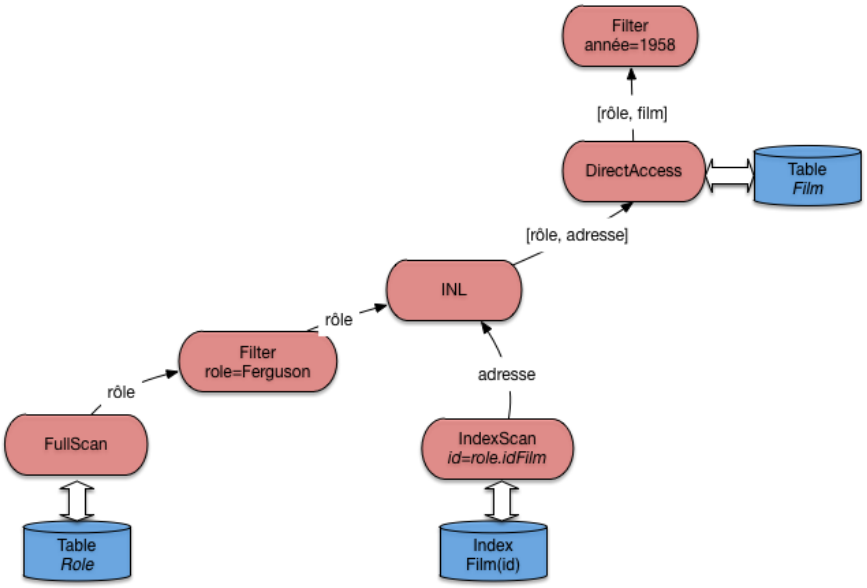

Fig. 7.6 – Le plan d'exécution « standard »

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 Fig. 7.7.

Ce plan est certainement le meilleur, du moins si on prend comme critère la minimisation du temps de réponse et de la mémoire utilisée. 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 possiblité 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.

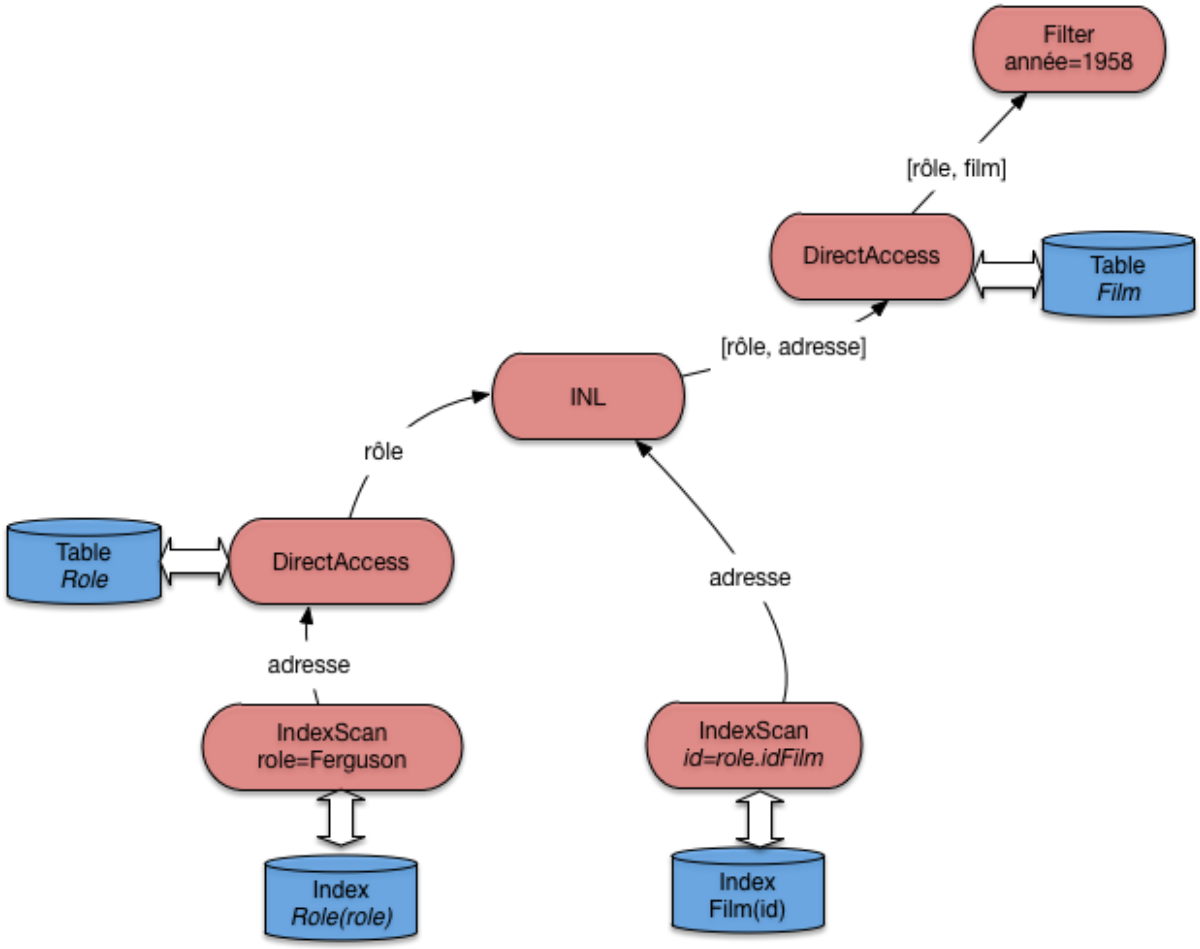

Fig. 7.7 – Le plan d'exécution avec deux index

Si, donc, la table `Rôle` 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 Fig. 7.8.

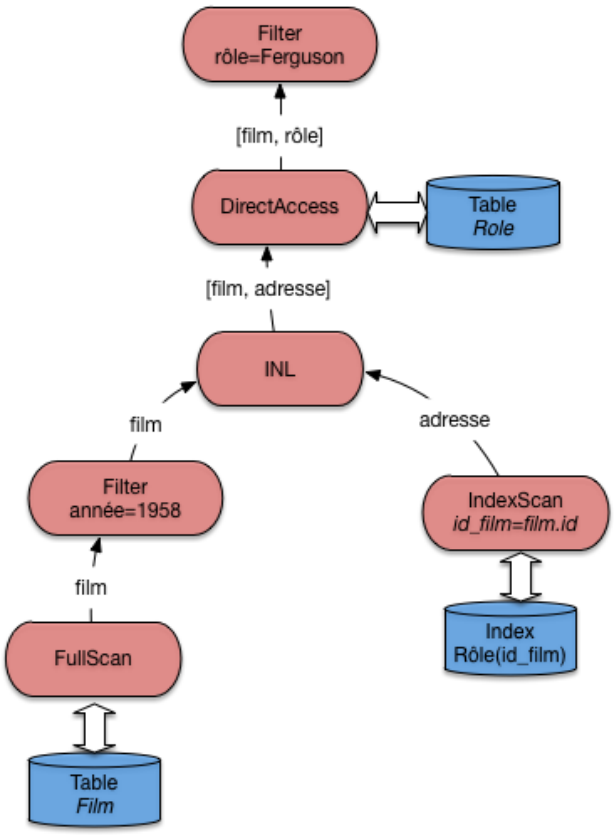

Fig. 7.8 – Le plan d'exécution avec index sur les clés étrangères

Ce plan est comparable à celui de la Fig. 7.6. Lequel des deux serait choisi par le système ? En principe, on choisirait comme table de gauche celle qui contient le *moins* de nuplets, 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 nuplets, 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 Fig. 7.9 illustre le cas où c'est l'agorithme de tri-fusion qui est choisi. La jointure par hachage est une alternative, sans doute préférable d'ailleurs si la mémoire RAM est suffisante.

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.

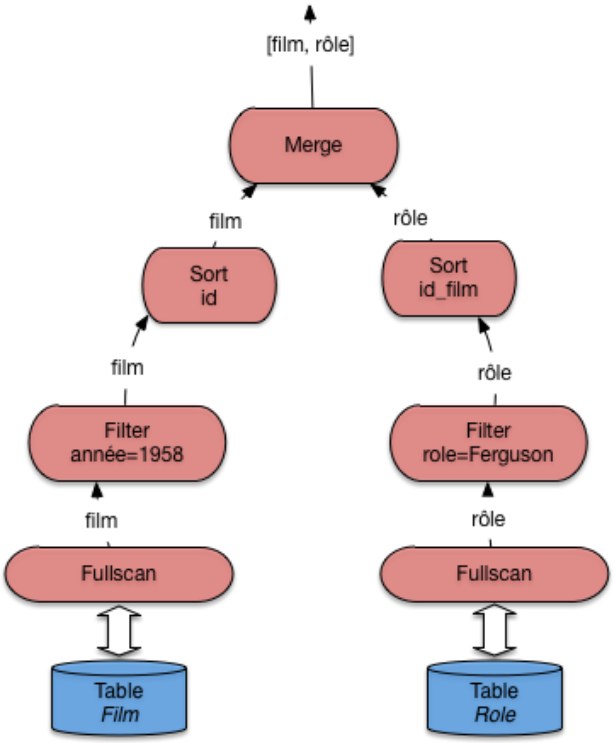

Fig. 7.9 – Le plan d'exécution en l'absence d'index

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

```
select *
from Film, Rôle, Artiste, Pays
where Pays.nom='Islande'
and   Film.id=Rôle.id_film
and   Rôle.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) :

$$Film \bowtie Rle \bowtie Artiste \bowtie Pays$$

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

$$Artiste \bowtie Pays \bowtie Rle \bowtie Film$$

Le plan d'exécution consistant en une lecture séquentielle suivi de boucles imbriquées indexées est donné sur la Fig. 7.10. 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.

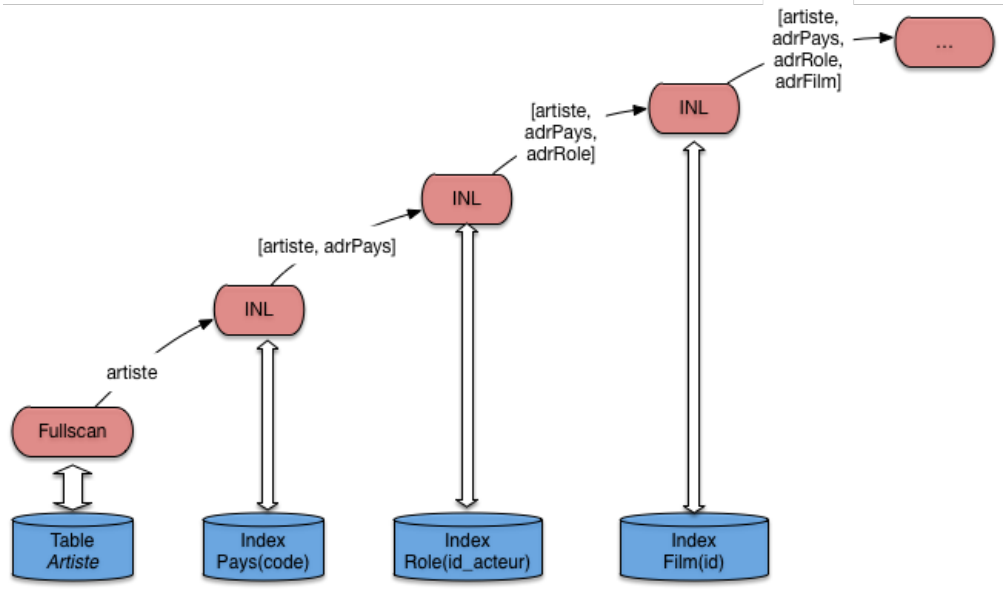

Fig. 7.10 – Le plan d'exécution avec algorithme de jointure indexée généralisé

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 Fig. 7.10) fournit un nuplet, et autant *d'adresses* de nuplets 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 Fig. 7.10 pour qu'il corresponde complètement à la requête.

7.3.4 Quiz

— La relation `Rôle(id_acteur, id_film, nom_rôle)` a pour clé primaire la paire d'attributs `(id_acteur, id_film)`. Sachant que le système construit un arbre B sur cette clé, laquelle parmi les requêtes suivantes ne peut pas utiliser l'index ?
En déduire pourquoi le bon ordre de jointure entre les tables Film et Rôle est celui exposé en cours.
— Dans le plan d'exécution de la requête suivante, comment pourrait–on éviter tout parcours séquentiel ?

```
select titre
from   Film f, Rôle r, Artiste a
where  a.nom = 'Stewart' and a.prénom='James'
and    f.id_film = r.id_film
and    r.id_acteur = a.idArtiste
and    f.annee = 1958
```

7.4 S4 : illustration avec Oracle

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).

Tous les GBD relationnel proposent un outil comparable d'exlication des plans d'exécution choisis. Les travaux pratique nous permettrons d'utiliser celui d'oracle, mais aussi celui de Postgres.

7.4.1 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.

7.4.2 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 (Fig. 7.11) avec les conventions utilisées jusqu'à présent.

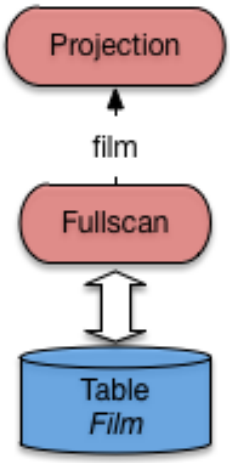

Fig. 7.11 – Plan Oracle pour une requête sans index

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 (Fig. 7.12).

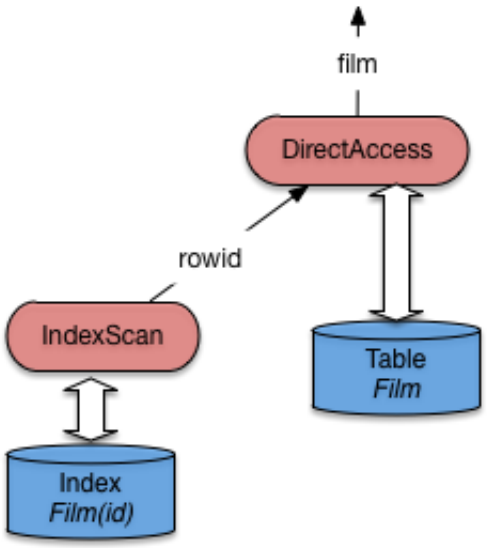

Fig. 7.12 – Plan Oracle pour une requête avec index

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 entre `Film` et `Artiste`.

```
explain plan
set statement_id='JoinIndex' for
select titre, nom, prénom
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 la figure suivante, très proche de celle du chapitre *Opérateurs et algorithmes*.

Ré-expliquons une nouvelle 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

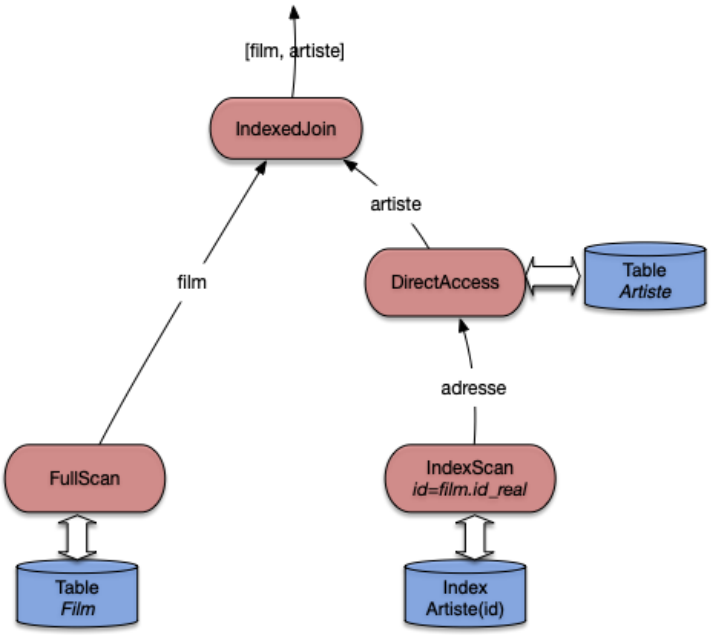

Fig. 7.13 – Plan Oracle pour une requête avec index

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.

On efffectue 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`. Le nœud de jointure récupère cet enregistrement et l'associe au film.

Note : Par rapport à la version de cet algorithme présenté précédemment, ORACLE choisit d'effectuer le `DirectAccess` immédiatement après le parcours d'index (alors que nous avons montré une version où il avait lieu après la jointure). Cela reste fondamentalement le même algorithme.

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 jouées 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_rôle
from   Rôle 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 nommé Pacino (avec des prénoms différents), et pour un artiste, on peut trouver plusieurs rôles (mais pas pour le même film). Tout cela résulte de la conception de la base.

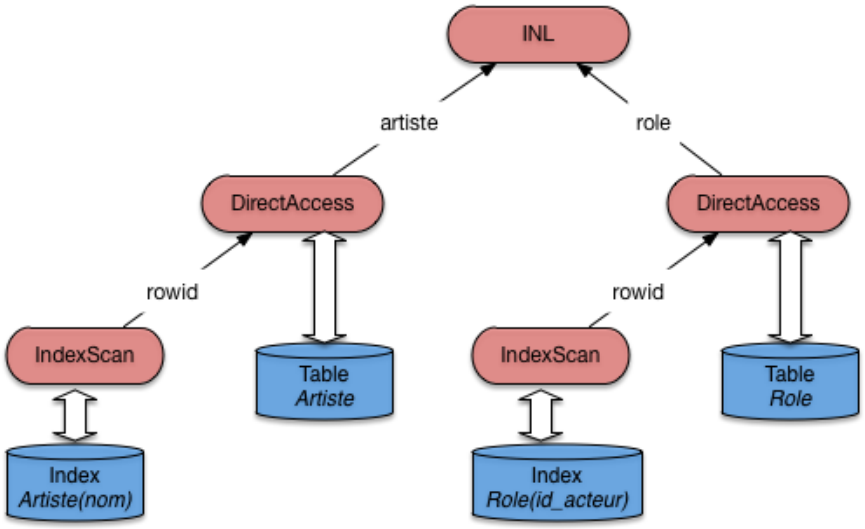

Fig. 7.14 – Plan Oracle pour une jointure et sélection avec index

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, prénom
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
```

(suite sur la page suivante)

(suite de la page précédente)

```
1 MERGE JOIN
  2 SORT JOIN
    3 TABLE ACCESS FULL ARTISTE
  4 SORT JOIN
    5 TABLE ACCESS FULL FILM
```

L'arbre de la Fig. 7.15 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.

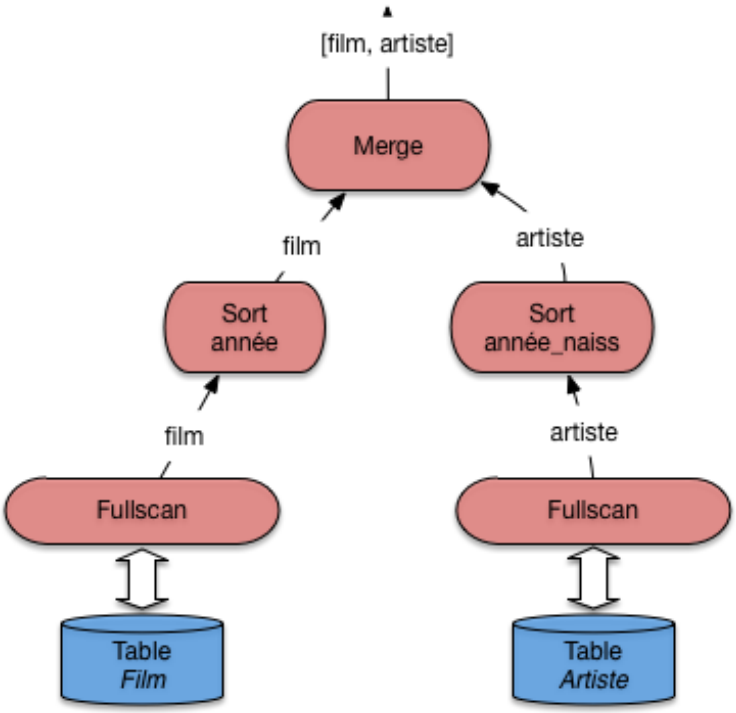

Fig. 7.15 – Plan Oracle pour une jointure sans index

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.

7.5 Exercices

Les exercices sont essentiellement des études de plan d'exécution, pour lesquels nous utilisons la syntaxe expliquée dans le contexte d'Oracle. Vous pouvez également produire des diagrammes assez facilement avec https://www.lucidchart.com si vous préférez.

Exercice ex-planex1 : définir des plans d'exécution

Donner le meilleur plan d'exécution pour les requêtes suivantes, en supposant qu'il existe un index sur la clé primaire de la table `idFilm`. Essayez d'identifier les requêtes qui peuvent s'évaluer uniquement avec l'index. Inversement, identifier celles pour lesquelles l'index est inutile.

```
select * from Film
where  idFilm = 20 and titre = 'Vertigo';

select * from   Film
where  idFilm = 20 or titre = 'Vertigo';

select COUNT(*) from Film;

select MAX(idFilm) from Film;
```

Exercice ex-planex2 : encore des plans d'exécution

Soit le schéma relationnel :
— Journaliste (**jid**, nom, prénom)
— Journal (**titre**, rédaction, id_rédacteur)
La table `Journaliste` stocke les informations (nom, prénom) sur les journalistes (`jid` est le numéro d'identification du journaliste). La table `Journal` stocke pour chaque rédaction d'un journal le titre du journal (titre), le nom de la rédaction (rédaction) et l'id de son rédacteur (`rédacteur_id`). Le titre du journal est une clé . On a un arbre B dense sur la table `Journaliste` sur l'attribut `jid`, nommé `Idx-Journaliste-jid`.

On considère la requête suivante :

```
select nom
from Journal, Journaliste
where titre='Le Monde'
and jid=id_redacteur
and prénom='Jean'
```

Questions :
— Voici deux expressions algébriques :

$$\pi_{nom}(\sigma_{titre='Le\ Monde' \wedge prenom='Jean'}(Journaliste \bowtie_{jid=redacteur\_id} Journal))$$

et

$$\pi_{nom}(\sigma_{prenom='Jean'}(Journaliste) \bowtie_{jid=redacteur\_id} \sigma_{titre='Le\ Monde'}(Journal))$$

Les deux expressions retournent-elles le même résultat (sont-elles équivalentes) ? Justifiez votre réponse en indiquant les règles de réécriture que l'on peut appliquer.

— Une expression vous semble-t-elle meilleure que l'autre si on les considère comme des plans d'exécution ?

— Donner le plan d'exécution physique basé sur la jointure par boucles imbriquées indexées, sous forme arborescente ou sous forme d'une expression EXPLAin, et expliquez en détail ce plan.

Exercice ex-planex3 : toujours des plans d'exécution

Soit la base d'une société d'informatique décrivant les clients, les logiciels vendus, et les licences indiquant qu'un client a acquis un logiciel.

— Société (**id**, intitulé)

— Logiciel (**id**, nom)

— Licence (**idLogiciel, idSociété**, durée)

Bien entendu un index unique est créé sur les clés primaires. Pour chacune des requêtes suivantes, donner le plan d'exécution qui vous semble le meilleur.

```
select intitulé
from  Société, Licence
where durée = 15
and   id = idSociete;

select intitule
from  Société, Licence, Logiciel
where nom='EKIP'
and   Société.id = idSociete
and   Logiciel.id = idLogiciel;

select intitule
from  Société, Licence
where Société.id = idSociete
and   idLogiciel in (select id from Logiciel
                     where nom='EKIP')

select intitule
from  Société s, Licence c
where s.id = c.idSociete
and exists (select * from Logiciel l
            where nom='EKIP' and c.idLogiciel=l.idLogiciel)
```

Exercice ex-optim1 : plans d'exécution Oracle

On prend les tables suivantes, abondamment utilisées par Oracle dans sa documentation :
— `Emp (empno, ename, sal, mgr, deptno)`
— `Dept (deptno, dname, loc)`
La table `Emp` stocke des employés, la table `Dept` stocke les départements d'une entreprise. La requête suivante affiche le nom des employés dont le salaire est égal à 10000, et celui de leur département.

```
select  e.ename, d.dname
from    emp e, dept d
where   e.deptno = d.deptno
and     e.sal = 10000
```

Voici des plans d'exécution donnés par Oracle, qui varient en fonction de l'existence ou non de certains index. Dans chaque cas expliquez ce plan.

— Index sur `Dept(deptno)` et sur `Emp(Sal)`.

```
0 SELECT STATEMENT
    1 NESTED LOOPS
        2 TABLE ACCESS BY ROWID EMP
            3 INDEX RANGE SCAN EMP_SAL
        4 TABLE ACCESS BY ROWID DEPT
            5 INDEX UNIQUE SCAN DEPT_DNO
```

— Index sur `Emp(sal)` seulement.

```
0 SELECT STATEMENT
    1 NESTED LOOPS
        2 TABLE ACCESS FULL DEPT
        3 TABLE ACCESS BY ROWID EMP
            4 INDEX RANGE SCAN EMP_SAL
```

— Index sur `Emp(deptno)` et sur `Emp(sal)`.

```
0 SELECT STATEMENT
    1 NESTED LOOPS
        2 TABLE ACCESS FULL DEPT
        3 TABLE ACCESS BY ROWID EMP
            4 and-EQUAL
                5 INDEX RANGE SCAN EMP_DNO
                6 INDEX RANGE SCAN EMP_SAL
```

— Voici une requête légèrement différente.

```
select e.ename
from emp e, dept d
where  e.deptno = d.deptno
and    d.loc = 'Paris'
```

On suppose qu'il n'y a pas d'index. Voici le plan donné par Oracle.

```
0 SELECT STATEMENT
    1 MERGE JOIN
        2 SORT JOIN
            3 TABLE ACCESS FULL DEPT
        4 SORT JOIN
            5 TABLE ACCESS FULL EMP
```

Indiquer quel(s) index on peut créer pour obtenir de meilleures performances (donner le plan d'exécution correspondant).

— Que pensez-vous de la requête suivante par rapport à la précédente ?

```
select e.ename
from emp e
where  e.deptno in (select d.deptno
            from Dept d
            where d.loc = 'Paris')
```

Voici le plan d'exécution donné par Oracle, :

```
0 SELECT STATEMENT
    1 MERGE JOIN
        2 SORT JOIN
            3 TABLE ACCESS FULL EMP
        4 SORT JOIN
            5 VIEW
                6 SORT UNIQUE
                    7 TABLE ACCESS FULL DEPT
```

Qu'en dites vous ?

— Sur le même schéma, voici maintenant la requête suivante.

```
select *
from Emp e1 where sal in (select salL
                from Emp e2
                where e2.empno=e1.mgr)
```

Cette requête cherche les employés dont le salaire est égal à celui de leur patron. On donne le plan d'exécution avec Oracle (outil EXPLAin) pour cette requête dans deux cas : (i) pas d'index, (ii) un index sur le salaire et un index sur le numéro d'employé.
Expliquez dans les deux cas ce plan d'exécution (éventuellement en vous aidant d'une représentation arborescente de ce plan d'exécution).

— Pas d'index.

```
0 FILTER
  1 TABLE ACCESS FULL EMP
  2 TABLE ACCESS FULL EMP
```

— Index sur empno et index sur sal.

```
0 FILTER
    1 TABLE ACCESS FULL EMP
        2 and-EQUAL
            3 INDEX RANGE SCAN I-EMPNO
            4 INDEX RANGE SCAN I-SAL
```

— Dans le cas où il y a les deux index (salaire et numéro d'employé), on a le plan d'exécution suivant :

```
0 FILTER
    1 TABLE ACCESS FULL EMP
    2 TABLE ACCESS ROWID EMP
        3 INDEX RANGE SCAN I-EMPNO
```

Expliquez-le.

Exercice ex-optim2 : encore des plans d'exécution Oracle

Soit le schéma d'une base, et une requête

```
create table TGV (
    NumTGV integer,
    NomTGV varchar(32),
    GareTerm varchar(32));

create table Arret (
    NumTGV integer,
    NumArr integer,
    GareArr varchar(32),
    HeureArr varchar(32));

select NomTGV
from TGV, Arret
where TGV.NumTGV = Arret.NumTGV
and GareTerm = 'Aix';
```

Voici le plan d'exécution obtenu :

```
0 SELECT STATEMENT
    1 MERGE JOIN
        2 SORT JOIN
            3 TABLE ACCESS FULL ARRET
        4 SORT JOIN
            5 TABLE ACCESS FULL TGV
```

— Que calcule la requête ?

— Que pouvez-vous dire sur l'existence d'index pour les tables `TGV` et `Arret` ? Décrivez en détail le plan d'exécution : quel algorithme de jointure a été choisi, quelles opérations sont effectuées et dans quel ordre ?

— On fait la création d'index suivante :

```
CREATE INDEX  index arret_numtgv ON Arret(numtgv);
```

— L'index créé est-il dense ? unique ? Quel est le plan d'exécution choisi par Oracle ? Vous pouvez donner le plan avec la syntaxe ou sous forme arborescente. Expliquez en détail le plan choisi.

— On ajoute encore un index :

```
CREATE INDEX tgv_gareterm on tgv(gareterm);
```

Quel est le plan d'exécution choisi par Oracle ? Expliquez le en détail.

CHAPITRE 8

Travaux pratiques : optimisation

8.1 Atelier en ligne : plans d'exécution

Ces travaux pratiques consistent à exécuter des requêtes sur de véritables bases de données gérées par le système PostgreSQL, et à interpéter le plan d'exécution de ces requêtes.

Vous diposez pour cela d'un outil en ligne qui vous permet d'entrer des requêtes SQL, de les exécuter, et de consulter le résultat et, surtout, le plan d'exécution. Les exercices consistent en deux parties

— étant donnée une question posée sur la base, plusieurs requêtes SQL sont proposées ; vous devez trouvez celle(s) qui exprime(nt) correctement la question - il peut y en avoir plusieurs, équivalentes ;
— vous pouvez alors copier une des requêtes SQL correctes dans la fenêtre d'entrée de l'outil en ligne, et inspecter le plan d'exécution ; des questions vous sont posées sur l'interprétation de ce plan.

Le schéma de la base ne comprend que trois tables : Artiste contient des personnalités du cinéma, acteurs/actrices ou réalisateur/réalisatrice ; Film contient des films, chaque film étant lié à son/sa réalisateur/réalisatrice ; enfin la table Rôle indique quels acteurs ont tourné dans quels films. Ce sont les tables qui ont servi de support aux exemples du cours.

Voici leur schéma :

```
CREATE TABLE Artiste (
   id integer,
  nom varchar(30),
  prenom varchar(30),
  annee_naissance integer,
  primary key (id)
  )

  CREATE TABLE Film (
        id integer,
```

(suite sur la page suivante)

(suite de la page précédente)

```
        titre varchar(50),
        annee integer
        id_realisateur integer,
        genre varchar(30),
        resume text,
        code_pays varchar(4),
        version integer,
        primary key (id),
        foreign key (id_realisateur) references Artiste(id),
        foreign key (code_pays) references Pays(code)
        )

        CREATE TABLE Role (
           id_film integer,
           id_acteur integer,
           nom_role varchar(60),
           primary key (id_film, id_acteur) ,
           foreign key (id_film) references Film(id),
           foreign key (id_acteur) references Artiste(id)
        )
```

Sur ce schéma, trois bases ont été créées.

— La première, nommée Minus, contient quelques centaines de films et artistes
— La seconde, nommée Magnus, contient quelques millions de films et d'acteurs - elle a été obtenue en dupliquant les données de la base Minus, ne vous étonnez donc pas de trouver beaucoup de fois le même titre ou le même nom : nous cherchons ici un volume suffisant pour étudier comment le plan d'exécution d'une requête est adapté par PostgreSQL par rapport à celui de la base Minus.
— Enfin, la troisième, nommée Magnindex, a le même contenu que Magnus, mais des index supplémentaires on été créés.

Pour chaque requête, vous êtes invités à étudier le plan d'exécution produit par PostgreSQL sur chaque base. La variation de ce plan correspond, comme expliqué en cours, à la prise en compte du contexte (volumétrie et présence d'index) par l'optimiseur de PostgreSQL. À vous d'interpréter ces variations et de réponde pertinemment aux questions posées.

Voici un exemple commenté d'interrogation et d'analyse du plan d'exécution.

8.1.1 Un exemple

La question posée est la suivante : *Donnez tous les titres des films parus après (au sens large) l'an 2000.* Parmi les requêtes suivantes, laquelle n'exprime pas cette question ?

Souvenez-vous : il peut y avoir plusieurs requêtes SQL équivalentes mais différentes syntaxiquement. À vous de jouer : si vous ne trouvez pas la bonne réponse, il est clairement nécessaire de vous lancer dans une sérieuse révision SQL avant d'aller plus loin.

Maintenant, vous pouvez copier/coller une des bonnes requêtes dans le formulaire ci-dessous, et l'exécuter. Vous obtiendrez un échantillon du résultat et, surtout, le plan d'exécution du SGBD (PostgreSQL).

8.1.2 L'interprétation du plan

En appliquant la requête à la base Minus, vous devriez obtenir un plan d'exécution de la forme

```
Parcours séquentiel de film (temps de réponse:0.00 ; temps d'exécution:7.10 ;
            nombre de nuplets:20 ; mémoire allouée:15)
  filter: (annee >= 2000)
```

Que nous dit PostgreSQL ? Que la table `Film` est parcourue séquentiellement, en appliquant un filtre sur l'année. De plus, pour chaque opérateur du plan d'exécution, PostgreSQL a l'amabilité de nous fournir une estimation du coût d'exécution :

— le temps de réponse est le temps mis pour obtenir le premier nuplet ;
— le temps d'exécution est le temps mis pour exécuter l'ensemble de la requête.

L'unité des valeurs affichées pour ces mesures est arbitraire et dépend des capacités du serveur : l'intérêt est de les comparer pour comprendre l'ordre de grandeur de l'optimisation obtenue. Grossièrement, il s'agit du nombre de blocs auxquels Postgres doit accéder pour satisfaire la requête (faites une recherche « Postgres explain » pour en savoir - un peu - plus).

Postgres nous donne également une estimation du nombre de nuplets ramenés par la requête et la taille moyenne de chaque nuplet dans le résultat (un titre, donc).

Première requêtes

Vous devriez maintenant pouvoir répondre aux questions suivantes sans aucun problème :

8.1.3 Et en changeant de base

Nous reprenons maintenant la même requête, mais vous allez l'exécuter en changeant la base et la sélectivité de la requête : essayez d'abord avec Minus, puis avec Magnus, puis avec Magnindex.

Commençons par chercher les films parus après 2000. Le résultat a peu d'intérêt et vous montre seulement de nombreuses réplications d'un même film pour Magnus et Magnindex. Regardez surtout le temps et de réponse et de temps d'exécution tels qu'ils sont évalués par Postgres, pour les bases Magnus et Magnindex qui ont la même volumétrie mais une organisation physique différente : Magnindex a plus d'index.

Ordonner, grouper, dé-dupliquer

Déterminez les requêtes qui vont introduire un opérateur bloquant dans le plan d'exécution.

Nous allons pouvoir le vérifier avec Postgres. Copier/coller une des bonnes requêtes dans le formulaire ci-dessous, et exécutez-le. Vous obtiendrez le résultat dans un onglet et le plan d'exécution du SGBD (PostgreSQL) dans un autre. Puis répondez aux questions qui suivent :

Requêtes avec ou sans index

Voici un ensemble de requêtes. Indiquez celles pour lesquelles il est possible d'utiliser un index. Rappelons que toutes les clés primaires sont indexées par un arbre B, que la clé primaire de Film est l'attribut id, et que la clé primaire de Rôle est la paire (id_film, id_acteur).

Nous allons pouvoir le vérifier avec Postgres. Exécutez les requêtes ci-dessus pour consulter le plan d'exécution de Postgres et vérifier si ce dernier utilise ou non l'index. Pour chaque requête, regardez si le plan est le même pour la base Minus et la base Magnus. Puis répondez aux questions qui suivent :

Algorithmes de jointure

Copier/coller une des bonnes requêtes dans le formulaire ci-dessous, et l'exécuter. Vous obtiendrez le résultat dans un onglet et le plan d'exécution du SGBD (PostgreSQL) dans un autre. Puis répondez aux questions qui suivent :

CHAPITRE 9

Transactions

Quand on développe un programme *P* accédant à une base de données, on effectue en général plus ou moins 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 entrainant l'arrêt de l'application ;
— 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 précédant 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 resources (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.

9.1 S1 : Transactions

Supports complémentaires :

— Diapositives: la notion de transaction
— Fichier de commandes pour tester les transactions sous MySQL
— Vidéo sur la notion de transaction

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.

9.1.1 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 INTEGER NOT NULL,
                     nom VARCHAR(255) NOT NULL,
                     nb_places_reservees INTEGER NOT NULL,
                     solde INTEGER NOT NULL,
                     primary key (id_client));
create table Spectacle (id_spectacle INTEGER NOT NULL,
                        nb_places_offertes INTEGER NOT NULL,
                        nb_places_libres INTEGER 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 reservation  */

create or replace procedure Reservation (v_id_client INTEGER,
                                         v_id_spectacle INTEGER,
                                          nb_places INT) AS
-- Déclaration des variables
v_client Client%ROWTYPE;
v_spectacle Spectacle%ROWTYPE;
```

(suite sur la page suivante)

(suite de la page précédente)

```
v_places_libres INTEGER;
v_places_reservees INTEGER;
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 reservation
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 reervees 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 (Fig. 9.1).

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 connaît que la séquence des instructions qui lui sont explicitement destinées, autrement dit les ordres de lecture ou d'écriture, les `commit` et les `rollback`.

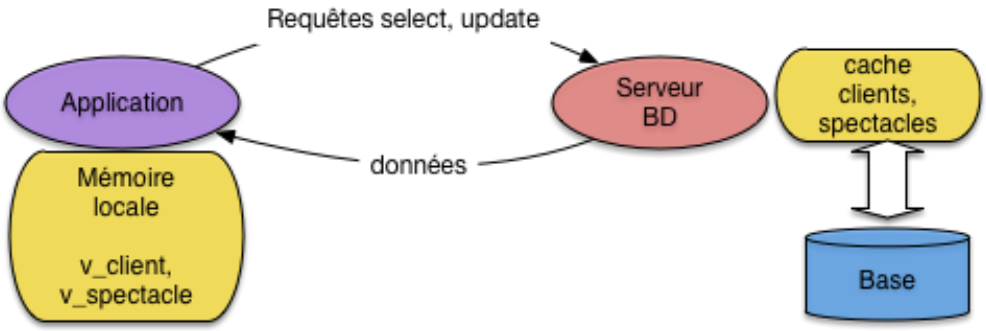

Fig. 9.1 – Le processus client et le processus serveur pendant une transaction

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 des transactions *en série* (retenez le terme, il est important) ;
- 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, se concluant par* `commit` ou `rollback`.

9.1.2 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. C'est même le cas général pour une base de données à laquelle accèdent simultanément plusieurs applications. Si elles manipulent les *mêmes* données, on peut aboutir à un entrelacement des lectures et écritures par le serveur, potentiellement générateur d'anomalies (Fig. 9.2).

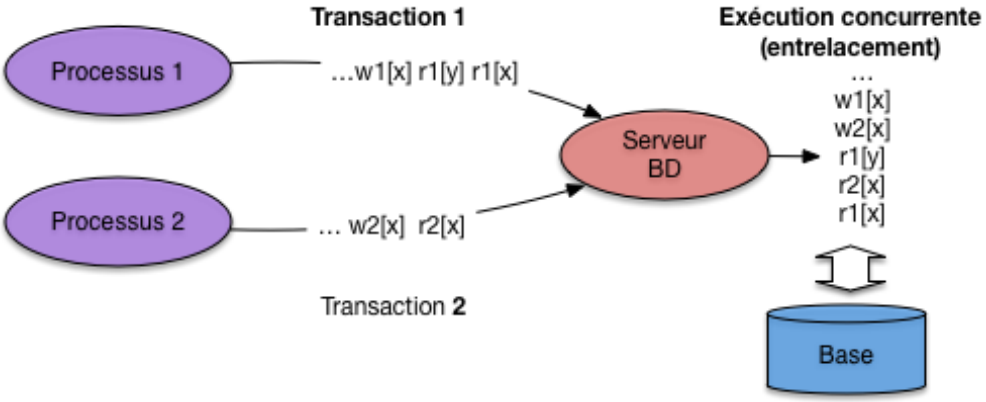

Fig. 9.2 – Exécution concurrentes engendrant un entrelacement

Chaque processus est identifié par un numéro unique qui est soumis au SGBD avec chaque ordre SQL effectué par ce processus (c'est l'identifiant de session, obtenu au moment de la connexion). 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 nuplets –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 nuplet (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 metre 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 lequelles 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*.

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

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_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 Fig. 9.3 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

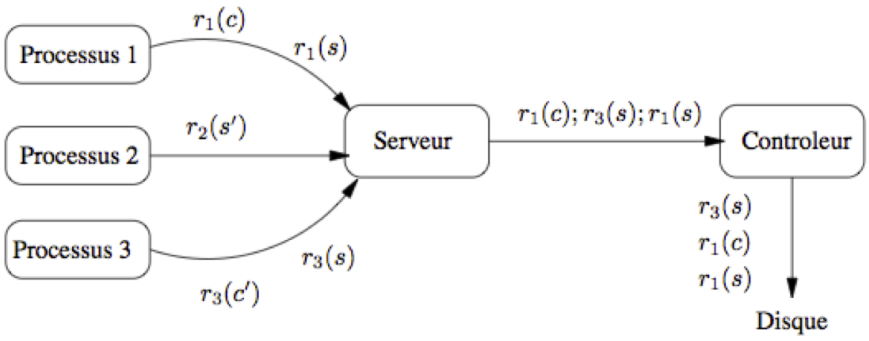

Fig. 9.3 – Processus soumettant des transactions

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 réordonnancer l'exécution s'il estime que la cohérence en dépend. Sur l'exemple de la Fig. 9.3, le contrôleur exécute, dans l'ordre $r_1(s)$, $r_1(c)$ puis $r_3(s)$.

9.1.3 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.

Reprenons 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 –déséquilibre 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 solidairement (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.

9.1.4 Quiz

9.2 S2 : Pratique des transactions

Supports complémentaires :

— Lien vers l'application de travaux pratiques en ligne
— Vidéo expliquant le fonctionnement de l'application « transactions »,
— Fichier de commandes pour tester les transactions sous un SGBD (MySQL, Oracle)

Pour cette session, nous vous proposons une mise en pratique consistant à constater directement les notions de base des transactions en interagissant avec un système relationnel. Vous avez deux possibilités : effectuer chez vous les commandes avec un système que vous avez installé localement (MySQL, Postgres, Oracle ou autre), ou utiliser l'application en ligne que nous mettons à disposition à l'adresse indiquée ci-dessus.

Quel que soit votre choix, n'hésitez pas à expérimenter et à chercher à comprendre le fonctionnement que vous constatez.

9.2.1 L'application en ligne « Transactions »

Ce TP est basé sur l'utilisation d'une base de données. Dans un premier temps, nous expliquons le contenu de la fenêtre d'interaction avec cette base.

Important : Vous pouvez à tout moment réinitialiser l'ensemble de la base en appuyant sur le bouton « Réinitialiser » en haut à droite de la fenêtre. Avant d'exécuter cette réinitialisation, vous devez sélectionner le niveau d'isolation parmi les 4 possibles : SERIALIZABLE, REPEATABLE READ, READ COMMITTED et READ UNCOMMITTED. Pour l'instant, conservez le niveau d'isolation par défaut, nous y reviendrons ultérieurement.

L'application est divisée en deux parties semblables, l'une à gauche, l'autre à droite, représentant deux sessions distinctes connectées à une même base. Ce dispositif permet de visualiser les interactions entre deux processus clients accédant de manière concurrente à des mêmes données. Nous détaillons maintenant les informations présentes dans chaque transaction.

Structure et contenu de la base

Le contenu de la base représente des clients achetant des places sur des vols. À peu de choses près, ce n'est qu'une variante de notre procédure de réservation. Son schéma est constitué de deux tables :

- Client(Id, Nom, Solde, Places)
- Vol (Id, Intitulé, Capacité, Réservations)

La table `Client` indique pour chaque client son nom, le solde de son compte et le nombre de places qu'il a réservées. Dans cette base de données, il y a deux clients : C1 et C2. La table `Vol` indique pour un vol donné la destination, la capacité et le nombre de réservations effectuées. Dans cette base, il n'y a qu'un seul vol V1. Il s'agit bien entendu d'un schéma très simplifié, destiné à étudier les mécanismes transactionnels.

Important : Sur cette base on définit la cohérence ainsi : **la somme des places réservées par les clients doit être égale à la somme des places réservées pour les vols.**

Vous noterez qu'à l'initialisation de la base, elle est dans un état cohérent. Chaque transaction, prise individuellement, préserve la cohérence de la base. Nous verrons que les mécanismes d'isolation ne garantissent pas que cela reste vrai en cas d'exécution concurrente.

L'affichage des tables montre, à tout moment, l'état de la base visible par une session. Nous affichons deux tables car chaque session peut avoir une vision différente, comme nous allons le voir.

Variables

En dessous des tables sont indiquées des variables utilisées par les transactions, soit comme paramètres, soit pour y stocker le résultat des requêtes. Par convention, le nom d'une variable est préfixé par « : ». Au départ, toutes les valeurs sont inconnues, sauf `:billet` qui représente le nombre de billets à réserver.

Actions

Quatre actions, qui correspondent à des requêtes utilisant les valeurs courantes des variables sont disponibles. Le code de chaque requête s'affiche lorsque vous passez la souris sur le bouton. Certaines requêtes peut être utilisées depuis la session S1 ou S2 selon qu'on les exécute à gauche ou à droite.

— Requête `select V1 into`

```
select capacité, réservations, tarif
into :capacité, :réservations, :tarif
from Vol where id=V1
```

Cette requête permet de stocker la capacité du vol et le nombre actuel de réservations dans les variables de la transaction.

— Requête `Select C1 into` :

```
select solde into :solde
from Client where id=C1
```

Cette requête permet de stocker le solde du client C1. Cette requête n'est accessible que depuis la transaction T1.

— Requête `select C2 into`

```
select solde into :solde
from Client where id=C2
```

Cette requête permet de stocker le solde du client C2. Cette requête n'est accessible que depuis la transaction T2.

— Requête `Update V1` :

```
update Vol set réservations=:réservations+billets
where id=V1 C
```

Cette requête permet de mettre à jour le nombre de réservations dans la table Vol, sur la base de la valeur des variables `:réservations` et `:billets` Cette requête est disponible depuis T1 et T2.

— Requête `Update C1` :

```
update Client set solde=:solde-:billets*:tarif
where id=C1
```

Cette requête permet de mettre à jour le solde du client C1 qui vient d'acheter les billets. Cette requête n'est disponible que depuis T1.

— Requête `Update C2`

```
update Client set solde=:solde-:billets*:tarif
where id=C2
```

Cette requête permet de mettre à jour le solde du client C2 qui vient d'acheter les billets. Cette requête n'est disponible que depuis T2.

— `Commit` : permet d'accepter toutes les mises à jour de la transaction.
— `Rollback` : permet d'annuler toutes les mises à jour depuis le dernier commit.

Une transaction se compose d'une suite d'actions, se terminant soit par un `commit`, soit par un `rollback`. Remarquez bien que si l'on exécute, dans l'ordre suggéré, les lectures puis les mises à jour, on obtient une transaction correcte qui préserve la cohérence de la base. Rien ne vous empêche d'effectuer les opérations dans un ordre différent si vous souhaitez effectuer un test en dehors de ceux donnés ci-dessous.

Historique

La liste des commandes effectuées, ainsi que les réponses du SGBD s'affichent au fur et à mesure de leur exécution. Celles-ci disparaissent lorsqu'on réinitialise l'application.

Nous vous invitons maintenant à regarder la vidéo et en reproduire vous-même le déroulement.

9.2.2 Quelques expériences avec l'interface en ligne

Voici quelques manipulations à faire avec l'interface en ligne. Vous pouvez suivre au préalable la vidéo indiquée en début de session, mais pratiquer vous-mêmes sera plus concret pour assimiler les principales leçons.

Leçon 1 : isolation des transactions

Effectuez une transaction de réservation à gauche (deux sélections, deux mises à jour), mais ne validez pas encore. Vous devriez constater que :

— les mises à jour sont *visibles* par la transaction qui les a effectuées,
— elles sont en revanche *invisibles* par toute autre transaction.

C'est la notion *d'isolation des transactions*. L'isolation est *complète* quand le résultat d'une transaction ne dépend pas de la présence ou non d'autres transactions. Nous verrons que l'isolation complète a des conséquences potentiellement pénalisantes et que les systèmes ne l'adoptent pas par défaut.

Leçon 2 : `commit` et `rollback`

Effectuez un `rollback` de la transaction en cours (à gauche). Vous devriez constater :

— que la base revient à son état initial ;
— que `commit` ou `rollback` n'ont plus aucun effet : nous sommes au début d'une nouvelle transaction ;
— pour la transaction de droite, rien n'a changé, tout se passe comme si celle de gauche n'avait pas existé.

Maintenant, lisez le vol à droite et constatez qu'il a zéro réservations. Recommencez la transaction à gauche, et effectuez un `commit`. Vous constatez que :

— il n'est plus possible d'annuler par `rollback`.
— les mises à jour sont *toujours* invisibles par l'autre transaction ;

Le fait de ne pas pouvoir annuler correspond à la notion de *durabilité* : le `commit` valide les données de manière définitive. Elles intègrent ce que nous appellerons *l'état de la base*, autrement dit l'ensemble des données validées, visibles par toutes les transactions.

Le fait que la transaction de droite ne puisse toujours pas voir les mises à jour est plus surprenant. C'est encore un effet de *l'isolation* : la transaction de droite a débuté avant celle de gauche, et elle continue durant toute son existence à voir la base telle qu'elle existait *au moment où elle a débuté*. C'est ce qu'indique le mode `repeatable read` : les mêmes lectures renvoient *toujours* le même résultat.

Commencez une autre transaction à gauche : il suffit de faire `commit` ou `rollback`. Cette fois le nouvel état de la base apparaît.

Leçon 3 : les écritures concurrentes posent des verrous

Réinitialisez, toujours en mode `repeatable read`. Maintenant déroulez deux transactions en parallèle selon l'alternance suivante :

— on effectue les deux sélections à gauche, puis une mise à jour du vol ;
— on effectue les mêmes opérations à droite ;

On constate que la transaction de droit est bloquée sur la tentative d'écriture. *Il est impossible d'effectuer deux écritures concurrentes du même nuplet*. Il deviendrait en effet alors impossible de gérer correctement les possibilités de `commit` et de `rollback`. Un verrou est donc posé par le système sur un nuplet modifié par une transaction, et ce verrou empêche toute modification par une autre transaction.

Les verrous entraînent une mise en attente. Ils sont conservés jusqu'à la fin de la transaction qui les a posés (`commit` ou `rollback`). Effectuez un `rollback` à gauche, et constatez que la transaction de droite est libérée.

La leçon, c'est que les mises à jour peuvent potentiellement bloquer les autres transactions. Il faut de préférence les effectuer le plus tard possible dans une transaction pour limiter le temps de rétention.

Leçon 4 : isolation incomplète = incohérence possible

Réinitialisez, toujours en mode `repeatable read`. Maintenant déroulez deux transactions en parallèle selon l'alternance suivante :

— on effectue les deux sélections à gauche ;
— on effectue les deux sélections à droite ;
— on effectue les deux mises à jour à droite, et on valide ;
— on effectue les deux mises à jour à gauche, et on valide ;

On a effectué deux transactions indépendantes : l'une qui réserve 2 billets, l'autre 5. À l'arrivée, vous constaterez que les clients ont bien réservé 2+5=7 billets, mais que seulement 2 billets ont été réservés pour le vol.

La base est maintenant incohérente, alors que chaque transaction, individuellement, est correcte. *On constate un défaut de concurrence, dû à un niveau d'isolation incomplet*.

La leçon, c'est qu'un niveau d'isolation non maximal (c'est le cas ici) mène potentiellement (pas toujours, loin de là) à une incohérence. De plus cette incohérence est à peu près inexplicable, car elle ne survient que dans une situation très particulière.

Leçon 5 : isolation complète = blocages possibles

Si on veut assurer la cohérence, il faut donc choisir le mode d'isolation maximal, c'est-à-dire `serializable`. Réinitialisez en choisissant ce mode, et refaites la même exécution concurrente que précédemment.

Cette fois, on constate que le système se bloque et que l'une des transactions est rejetée. C'est la dernière leçon : en cas d'imbrication forte des opérations (ce qui encore une fois ne peut survenir que très rarement), on rencontre un risque *d'interblocage*.

En mode `serializable` le verrouillage est plus strict que dans le mode par défaut : *la lecture d'un nuplet par une transaction bloque les tentatives d'écriture par une autre transaction*. Les risques d'être mis en attente sont donc bien plus élevés. C'est la raison pour laquelle les systèmes ne choisissent pas ce mode par défaut.

Vous pouvez continuer à jouer avec la console, en essyant d'interpréter les résultats en fonction des remarques qui précèdent. Nous revenons en détail sur le fonctionnement d'un système transactionnel concurrent dans les prochaines sessions.

9.2.3 Mise en pratique directe avec un SGBD

Vous pouvez également pratiquer les transactions avec MySQL, Postgres ou Oracle. Un fichier de commandes vous est fourni ci-dessus. 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
---------- ------------------- ----------
         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 nuplet 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
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_OFFERTES 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 et 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_OFFERTES 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.

9.2.4 Quiz

9.3 S3 : effets indésirables des transactions concurrentes

Supports complémentaires :

— Diapositives: anomalies transactionnelles

— Vidéo sur les 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 nuplet (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 nuplet dans le fichier et on le transmet au processus ;

— quand il s'agit d'une instruction de mise à jour, on écrit directement le nuplet 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 dus 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.

9.3.1 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 :numref :*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.

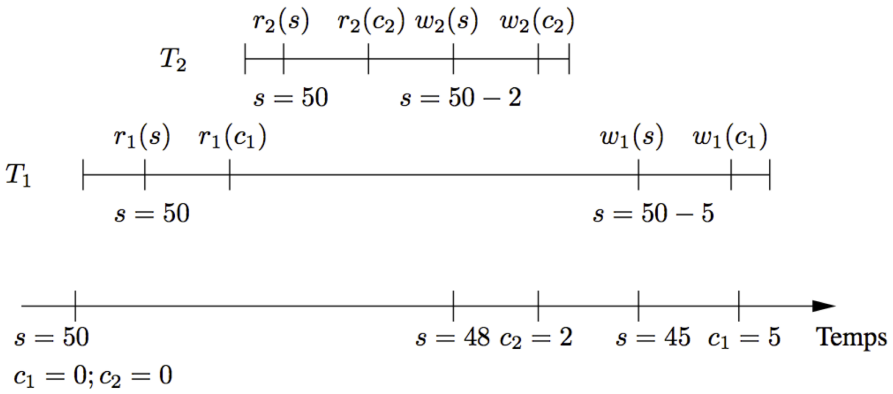

Fig. 9.4 – Exécution concurrente de T_1 et T_2

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 :

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

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 Fig. 9.5 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.

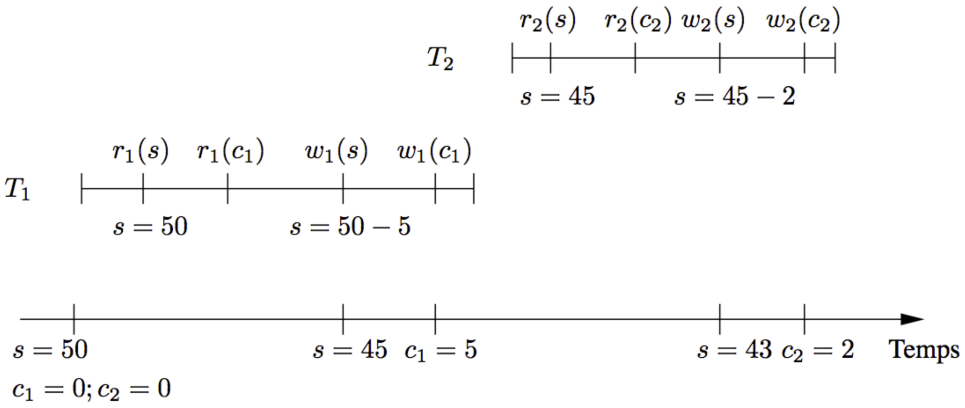

Fig. 9.5 – Exécution en série de T_1 et T_2

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*.
- 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 il faut imaginer l'intérêt de débloquer rapidement de longues transactions qui ne rentrent en concurrence que sur une petite partie des nuplets qu'elles manipulent.

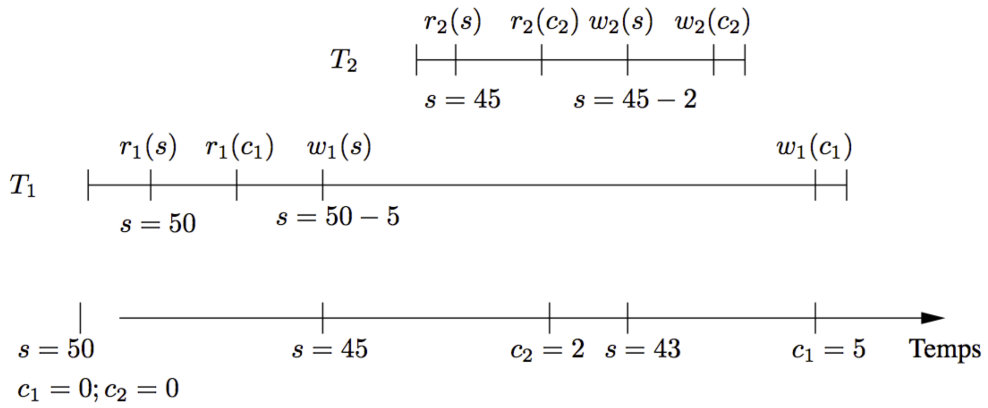

Fig. 9.6 – Exécution concurrente correcte de T_1 et T_2

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 nuplets différent d'une seconde exécution de *q* effectuée un peu plus tard, parce certain nuplets ont disparu ou sont apparus dans l'intervalle (on parle de *nuplets 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.

```
Lire tous les clients et effectuer la somme des places prises
Procédure Contrôle()
Début
  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 $r_c(c_1) \ldots r_c(c_n) r_c(s)$ d'une lecture des *n* clients $\{c_1, \ldots, c_n\}$ 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.

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

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 Fig. 9.7 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.

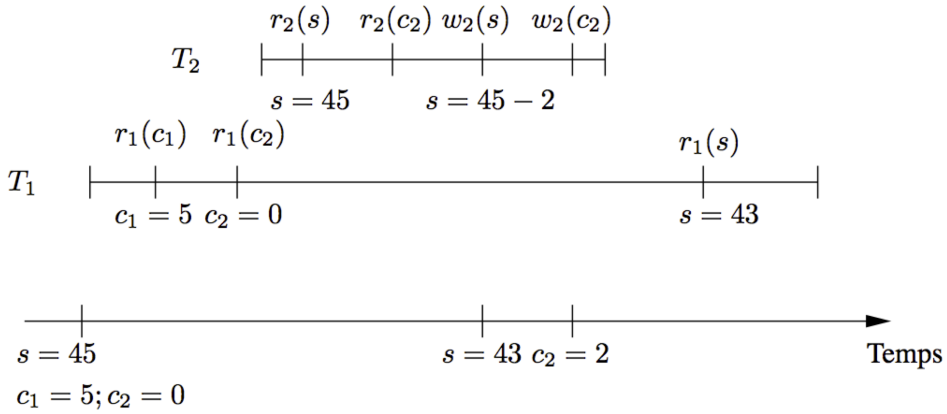

Fig. 9.7 – Exécution concurrente d'un contrôle et d'une réservation

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 peux 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 nuplet 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 *nuplets fantômes*.

Les lectures non répétables et nuplets 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 nuplet à nuplet, avec un temps de traitement de chaque nuplet qui peut alonger considérablement le temps seulement consacré au parcours du résultat.

9.3.2 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 au 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 nuplet *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 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 nuplets 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 nuplet *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 nuplet 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 nuplets en cours de modification.

Ecriture sale

Imaginons qu'une transaction *T* ait modifé un nuplet *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 « fluiditéé 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.

9.3.3 Quiz

9.4 S4 : choisir un niveau d'isolation

Supports complémentaires :
- Diapositives: niveaux d'isolation
- Vidéo sur les 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 :

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

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.

9.4.1 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 nuplet 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 nuplet qu'elle avait déjà lu auparavant, alors que ce nuplet a été modifié entre temps par une autre transaction T_2 ;
- *Tuples fantômes* : une transaction T_1 lit un nuplet qui a été créé par une transaction T_2 *après* le début de T_1.

Tableau 9.1 – Niveaux d'isolation de la norme SQL

	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 des interactions avec d'autres utilisateurs.

Avec le mode `read committed`, on ne peut lire que les nuplets 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 nuplets peuvent apparaître s'ils ont étés insérés par d'autres transactions (les fameux « nuplets 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.

9.4.2 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]*. 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 nuplets 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 la transaction. En contrepartie le niveau de verrouillage est faible, ce qui évite les bloquages.

9.4.3 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]*. 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 nuplets 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`.

Note : Vous pouvez répéter ce déroulement avec notre interface en ligne.

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;
+--------------+--------------------+------------------+--------+
```

(suite sur la page suivante)

(suite de la page précédente)

```
| 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 |
```

(suite sur la page suivante)

(suite de la page précédente)

```
+-------------+--------------------+------------------+-------+
|           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.

9.4.4 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 spectacles) 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/InoDB.

Note : Vous pouvez aussi choisir le niveau sérialisable et reproduire le scénario avec l'application en ligne

---

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 | solde |
+-----------+----------------------+--------+
|         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 partagées 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.

9.4.5 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 réserver chaque nuplet lu en vue d'une prochaine modification.

Verrouillage des tables

Reprenons 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 nuplet (le client ou le spectacle) sera suivie par la mise à jour de ce même nuplet. Le système pose alors un *verrou exclusif* qui réserve l'accès au nuplet, 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'excéute.

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 nuplet 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 dont 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 solde=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.

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 nuplets sélectionnés par un ordre `SELECT`. Ces nuplets 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 nuplets 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 verrouiler 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 déjà posé 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.

9.4.6 Quiz

9.5 Exercices

Exercice Ex-transaction-1 : je comprends (et j'explique) les mécanismes transactionnels

Répondez aux questions suivantes clairement et concisément

— Pascal veut modifier un nuplet. Il s'aperçoit qu'il est mis en attente parce que ce nuplet est déjà en cours de modification par Julie. Comment lui expliqueriez-vous la justification de cette mise en attente ? À quel moment prendra-t-elle fin ?

— Justin a écrit un programme en mode `serializable` qui crée les fiches de paie de tout le personnel. Il vous annonce qu'il a décidé de n'effectuer qu'une seule fois l'ordre `commit`, quand toutes les fiches sont calculées. Comment lui expliqueriez-vous les inconvénients de ce choix ?

— Alphonsine lit un nuplet dans la table des voyageurs et s'aperçoit qu'il manque l'adresse. Elle demande à Barnabé d'effectuer la correction, et il le fait immédiatement. Pourtant, Alphonsine ne voit toujours pas l'adresse, même en réaffichant sans cesse son écran. La tension monte entre les deux. Que feriez-vous pour les aider ?

— Timothée veut surveiller les nouvelles réservations de billets d'avion. Il écrit une transaction qui effectue une boucle de lecture sur la table des vols et affiche le total des billets réservés. Or, le résultat affiché est toujours le même, alors que des billets sont bel et bien réservés. Que conseilleriez-vous à Timothée ?

Exercice Ex-transaction-2 : la sérialisabilité

Supposons une table *T(id, valeur)*, et la procédure suivante qui copie la valeur d'une ligne vers la valeur d'une autre :

```
/* Une procédure de copie */
create or replace procedure Copie (id1 INT, id2 INT) AS
 -- Déclaration des variables
```

(suite sur la page suivante)

(suite de la page précédente)

```
  val INT;
  begin
    -- On recherche la valeur de id1
    select valeur into val from T where id = id1

    -- On copie dans la ligne id2
    update T set valeur = val where id = id2

    -- Validation
    commit;
 end;
/
```

On prend deux transactions *Copie(A, B)* et *Copie(B,A)*, l'une copiant du nuplet *A* vers le nuplet *B* et l'autre effectuant la copie inverse. Initialement, la valeur de *A* est *a* et la valeur de *B* est *b*.

— Qu'est-ce qui caractérise une exécution concurrente correcte de ces deux transactions ?
Aide : une exécution est correcte si le résultat peut être obtenu par une exécution en série (exécution sérialisable). Etudiez les exécutions en série et déduisez-en la propriété générale du résultat.

— Voici une exécution concurrente de *Copie(A, B)* et *Copie(B,A)*

$$r_1(A)r_2(B)w_1(B)w_2(A)$$

En supposant qu'elle s'exécute sans contrôle de concurrence, avec une valeur initiale de *A* à *a* et une valeur de *B* à *b*, quel est l'état de la base à la fin ?

Conclusion : cette exécution est-elle sérialisable ?

— On se met en mode sérialisable. Dans ce mode, une transaction est mise en attente si elle tente de modifier un nuplet qui est en cours de lecture par une autre transaction.
Que se passe-t-il en mode sérialisable pour l'exécution concurrente précédente ?

Exercice Ex-transaction-3 : sérialisabilité et mode autocommit

On dispose d'une table $T (id, valeur)$. Initialement toutes les valeurs sont différentes. Voici une procédure qui échange les valeurs de 2 nuplets.

```
create or replace procedure Echange (id1 INT, id2 INT) AS
   -- Déclaration des variables
   val1, val2 integer;
   begin
       -- On recherche la valeur de id1 et de id2
       select valeur into val1 from T where id = id1
       select valeur into val2 from T where id = id2
```

(suite sur la page suivante)

(suite de la page précédente)

```
      -- On échange les valeurs
      update T set valeur = val1 where id = id2
      update T set valeur = val2 where id = id1

   end;
```

On est en mode `autocommit` : un `commit` a lieu après chaque requête SQL. Expliquez dans quel scénario l'exécution concurrente de deux procédures d'échange peut aboutir à ce que deux nuplets aient la même valeur.

Exercice Ex-transaction-4 : la sérialisabilité, suite

Supposons qu'un hôpital gère la liste de ses médecins dans une table (simplifiée) *Docteur(nom, garde)*, chaque médecin pouvant ou non être de garde. On doit s'assurer qu'il y a toujours au moins deux médecins de garde. La procédure suivante doit permettre de placer un médecin au repos en vérifiant cette contrainte.

```
/* Une procédure de gestion des gardes */

create or replace procedure HorsGarde (nomDocteur VARCHAR) AS

 -- Déclaration des variables
 val nb_gardes;

 begin
   -- On calcule le nombre de médecin de garde
   select count(*) into nb_gardes from Docteur where garde = true

   if (nb_gardes > 2) then
      update Docteur set garde = false where nom = nomDocteur;
      commit;
   endif
end;
/
```

En principe, cette procécure semble très correcte (et elle l'est). Supposons que nous ayons trois médecins, Philippe, Alice, et Michel, désignés par *p*, *a* et *m*, tous les trois de garde. Voici une exécution concurrente de deux transactions T_1 = *HorsGarde("Philippe")* et T_2 = *HorsGarde("Michel")*.

$$r_1(p)r_1(a)r_1(m)r_2(p)r_2(a)r_2(m)w_1(p)w_2(m)$$

Questions :

- Quel est, avec cette exécution, le nombre de médecins de garde constatés par T_1 et T_2
- Quel est le nombre de médecins de garde à la fin, quand T_1 et T_2 ont validé ?
- Au vu des réponses qui précèdent, expliquer pourquoi cette exécution concurrente n'est pas sérialisable (aide : définir la propriété de la base qui est respectée si les exécutions ont lieu en série, et

constater qu'elle est violée ici).

— Mettons-nous en mode sérialisable, dans lequel l'écriture d'un nuplet est bloquée si une autre transaction a lu ce même nuplet (et ce, jusqu'à la fin de cette autre transaction). Que va-t-il se passer dans ce mode avec notre exécution concurrente ?

9.6 Atelier : réservons des places pour Philippe

Le but de cet atelier est de simuler des exécutions concurrentes avec un utilitaire de commandes SQL. Les instructions données ci-dessous ont été testées avec MySQL, elles fonctionnent probablement avec tout système relationnel, au prix éventuel de quelques ajustements.

9.6.1 Préparation

Il faut utiliser une application cliente qui permet de soumettre des commandes SQL au serveur. Pour MySQL, vous disposez de l'utilitaire `mysql`, ou du client graphique *MySQL workbench*, ou encore du client web phpMyAdmin. À vous de l'installer, de créer la base et le compte utilisateur.

Comme le support de cours montre comment réserver un même spectacle pour deux clients, nous allons simplement étudier une variante : réservation de deux spectacles pour le même client ! En d'autres termes, il s'agit d'une simple transposition de ce qui a été abondamment démontré ci-dessus, la mise en pratique en plus.

Voici tout d'abord les commandes de création des tables

```
create table Client (id_client integer not null,
                     nom varchar(30) not null,
                     nb_places_reservees integer not null,
                     solde integer not null,
                     primary key (id_client))
                      ;

create table Spectacle (id_spectacle integer not null,
                        titre varchar(30) not null,
                        nb_places_offertes integer not null,
                        nb_places_libres integer not null,
                        tarif decimal(10,2) not null,
                        primary key (id_spectacle))
                      ;
```

Insérez des données dans la base (et refaites cette insertion quand vous souhaitez la réinitialiser). Vous pouvez exécuter en bloc les commandes suivantes :

```
set autocommit = 0;
delete from Client;
delete from Spectacle;
```

(suite sur la page suivante)

(suite de la page précédente)

```
insert intoClient values (1, 'Philippe', 0, 2000);
insert intoClient values (2, 'Julie', 0, 350);
insert intoSpectacle values (1, 'Ben hur', 250, 50, 50);
insert intoSpectacle values (2, 'Tartuffe', 120, 30, 30);
commit;
```

Vous pouvez alors ouvrir deux connexions simultanées à votre base. La première chose à faire est de s'assurer que l'on n'est pas en mode `autocommit` (mode où chaque mise à jour se termine par un `commit`, autrement dit un mode anti-transactionnel). Pour MySQL la commande est :

```
set autocommit = 0;
```

Nous appellerons les deux connexions Session1 et Session2. Le second réglage est le niveau d'isolation. Choisissez une des commandes ci-dessous.

```
set session transaction isolation level read uncommitted;
set session transaction isolation level read committed;
set session transaction isolation level repeatable read;
set session transaction isolation level serializable;
```

Et pour finir, il faudra, pour simuler une réservation, exécuter pas à pas les commandes de sélection et de mise à jour. Voici quelques exemple à reprendre et adapter.

```
select * from Client where id_client=1;
select * from Spectacle where id_spectacle=2;

update Client set nb_places_reservees = 0 + 2 where id_client=2;
update Spectacle set nb_places_libres = 50 - 2 where id_spectacle=1;
```

9.6.2 Déroulement

Le but est de réserver des places pour Philippe : 2 places pour Ben Hur, et 4 pour Tartuffe. On a donc deux exécutions de la procédure de réservation, l'une à dérouler dans la session 1, l'autre dans la session 2.

- Commencez par écrire les requêtes effectuées par chaque transaction
- Effectuez les deux transactions l'une après l'autre (en série). Quel est l'état de la base à la fin ? Cela vous satisfait-il ?
- Maintenant, en mode `read committed` ou `repeatable read`, déroulez des deux transactions dans l'ordre suivant : la transaction 1 fait ses lectures ; la transaction 2 fait ses lectures ; la transaction 1 fait ses écritures, la transaction 2 fait ses écritures.
  Quel est l'état de la base à la fin ? Conclusion ?
- Recommencez en mode `serialisable`.

Vous devriez arriver à des conclusions déjà largement détaillées dans le cours. À vous de faire le lien !

CHAPITRE 10

Contrôle de concurrence

Le contrôle de concurrence est l'ensemble des méthodes mises en œuvre par un serveur de bases de données pour assurer le bon comportement des transactions, et notamment leur *isolation*. Les autres propriétés désignées par l'acronyme ACID (soit la durabilité et l'atomicité) sont garanties par des techniques de reprise sur panne que nous étudierons ultérieurement.

Les SGBD utilisent essentiellement deux types d'approche pour gérer l'isolation. La première s'appuie sur un mécanisme de versionnement des mises à jour successives d'un nuplet. On parle de contrôle multiversion ou « d'isolation par cliché » (*snapshot isolation* en anglais). C'est une méthode satisfaisante jusqu'au niveau d'isolation *repeatable read* car elle impose peu de blocages et assure une bonne fluidité. En revanche elle ne suffit pas à garantir une isolation totale, de type *serializable* (sauf à recourir à des algorithmes sophistiqués qui ne semblent pas être encore adoptés dans les systèmes).

La seconde approche a recours au verrouillage, en lecture et en écriture, et garantit la sérialisabilité des exécutions concurrentes, grâce à un algorithme connu et utilisé depuis très longtemps, le *verrouillage à deux phases* (*two-phases locking*). Le verrouillage impacte négativement la fluidité des exécutions, certaines transactions devant être mises en attente. Il peut parfois même entraîner un rejet de certaines transactions. C'est le prix à payer pour éviter toute anomalie transactionnelle.

10.1 S1 : isolation par versionnement

Supports complémentaires :

— Diapositives: isolation par versionnement
— Vidéo sur l'isolation par versionnement

Nous avons déjà évoqué à plusieurs reprises le fait qu'un SGBD gère, à certains moments, plusieurs versions d'un même nuplet. C'est clairement le cas pendant le déroulement d'une transaction, pour garantir la pos-

sibilité d'effectuer des *commit/rollback*. Le versionnement est utilisé de manière plus générale pour garantir l'isolation, au moins jusqu'au niveau *repeatable read*. C'est ce qui est détaillé dans cette session.

Important : Dans le contexte d'une base relationnelle, le mot « donnée » dans ce qui suit désigne toujours un nuplet dans une table.

10.1.1 Versionnement et lectures « propres »

Comme vous avez dû le constater pendant la mise en pratique, par exemple avec notre interface en ligne, 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 :

Définition : image avant et image après

Le système maintient, pour tout nuplet en cours de modification par une transaction, deux versions de ce nuplet :

- une version *après* la mise à jour, que nous appellerons *l'image après* de ce nuplet ;
- une version *avant* la mise à jour, que nous appellerons *l'image avant* de ce nuplet.

Ces deux images correspondent à deux versions successives du même nuplet, stockées dans deux espaces de stockage séparés (nous étudierons ces espaces de stockage dans le chapitre consacré à la reprise sur panne). Le versionnement est donc d'abord une conséquence de la nécessité de pouvoir effectuer des *commit* ou des *rollback*. Il s'avère également très utile pour gérer *l'isolation* des transactions, grâce à l'algorithme suivant.

Algorithme : lectures propres et cohérentes

Soit deux transactions T et T'. Leur isolation, basée sur ces deux images, s'effectue de la manière suivante.

- Chaque fois que *T* effectue la mise à jour d'un nuplet, la version courante est d'abord copiée dans l'image avant, puis remplacée par la valeur de l'image après fournie par *T*.
- Quand *T* effectue la lecture de nuplets qu'elle vient de modifier, le système doit lire dans l'image après pour assurer une vision cohérente de la base, reflétant les opérations effectuées par *T*.
- En revanche, quand c'est une autre transaction *T"* qui demande la lecture d'un nuplet en cours de modification par *T*, il faut lire dans l'image avant pour éviter les effets de lectures sales.

Cet algorithme est illustré par la Fig. 10.1. L'espace « après » et « avant » sont distingués (attention, il s'agit d'une représentation simplifiée d'espaces de stockage dont l'organisation est assez complexe : voir chapitre *Reprise sur panne*). Dans l'espace « après » on trouve les versions les plus récentes des nuplets de la base : ceux qui on fait l'objet d'un *commit* sont en vert, ceux qui sont en cours de modification sont en rouge.

L'image avant contient la version précédente de chaque nuplet en cours de modification. En l'occurrence, e'_2 est en rouge dans l'image après car il représente une version du nuplet e_2 en cours de modification par T_1. La version précédente est en vert dans l'image avant.

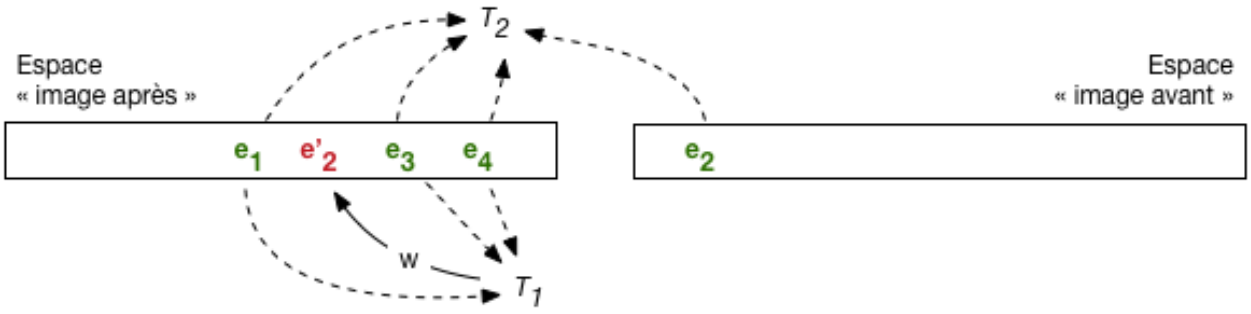

Fig. 10.1 – Lectures et écritures avec image après et image avant

Pour les lectures : T_1 lira e'_2 qu'elle est en train de modifier, dans l'image après, pour des raisons de *cohérence*, alors que T_2 (ou n'importe quelle autre transaction) lira la version e_2 dans l'image avant pour éviter une lecture sale.

Note : On peut se poser la question du nombre de paires image après/avant nécessaires. Que se passe-t-il par exemple si T_2 demande la mise à jour d'un nuplet déjà en cours de modification par T_1 ? Si cette mise à jour était autorisée, il faudrait créer une troisième version du nuplet, l'image avant de T_2 étant l'image après de T_1 (tout le monde suit ?). La multiplication des versions rendrait la gestion des `commit` et `rollback` extrêmement complexe, voire impossible : comment agir dans ce cas pour effectuer un *rollback* de la transaction T_1 ? Il faudrait supprimer l'image après de T_1, et donc l'image avant de T_2, Il n'est pas nécessaire d'aller plus loin pour réaliser que ce casse-tête est insoluble. 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.

10.1.2 Lectures répétables

Le mécanisme décrit ci-dessus est suffisant pour assurer un niveau d'isolation de type *read committed*. En effet, toute transaction autre que celle effectuant la modification d'un nuplet doit lire l'image avant, qui est nécessairement une version ayant fait l'objet d'un *commit* (pour les raisons exposées dans la note qui précède). En revanche, ce même mécanisme ne suffit pas pour le niveau *repeatable read*, comme le montre la Fig. 10.2.

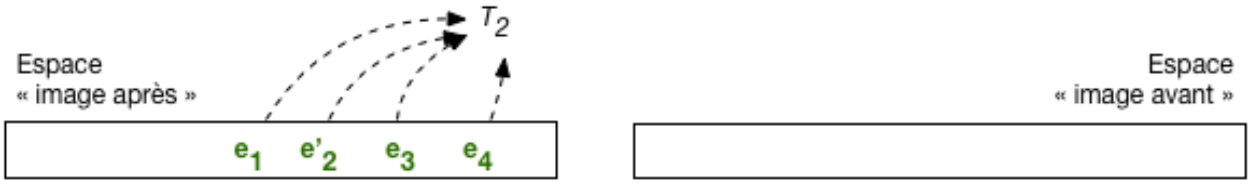

Fig. 10.2 – Lecture non répétable après validation par T_1

Cette figure montre la situation après que T_1 a validé. La version e'_2 fait maintenant partie des nuplets ayant fait l'objet d'un *commit* et T_2 lit cette version, qui est donc différente de celle à laquelle elle pouvait accéder avant le *commit* de T_1. La lecture est « non répétable », ce qui constitue un défaut d'isolation puisque T_2 constate, en cours d'exécution, l'effet des mises à jour d'une transaction concurrente.

Les *lectures non répétables* sont dues au fait qu'une transaction T_2 lit un nuplet *e* qui a été modifié par une transaction T_1 *après* le début de T_2. Or, quand T_1 modifie *e*, il existe avant la validation deux versions de *e* : l'image avant et l'image après. Il suffirait que T_2 continue à lire l'image avant, même après que T_1 a validé, pour que le problème soit résolu. En d'autres termes, ces images avant peuvent être vues, au-delà de leur rôle dans le mécanisme des *commit/rollback*, 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.

Un peu de réflexion suffit pour se convaincre qu'il n'est pas suffisant de conserver une seule version de l'image avant, mais qu'il faut conserver toutes celles qui existaient au moment où la transaction la plus ancienne 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 l'algorithme suivant :

Algorithme : lectures répétables

- chaque transaction T_i se voit attribuer, quand elle débute, une estampille temporelle τ_i ; 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 nuplet *e* est de même estampillée par le moment τ_e de sa validation ;
- quand T_i doit lire un nuplet *e*, le système regarde dans l'image après. Si *e* a été modifié par T_i ou si son estampille est inférieure à τ_i, le nuplet peut être lu puisqu'il a été validé avant le début de T_i, sinon le système recherche dans l'image avant la version de *e* validée et immédiatement antérieure à τ_i.

La seule extension nécessaire par rapport à l'algorithme précédent est la non-destruction des images avant, même quand la transaction qui a modifié le nuplet valide par `commit`. L'image avant contient alors toutes les versions successives d'un nuplet, 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 Fig. 10.3 illustre le mécanisme de lecture cohérente et répétable. Les nuplets de l'image après sont associés à leur estampille temporelle : 6 pour e_1, 25 pour e_2'', etc. On trouve dans l'image avant les versions antérieures de ces nuplets : e_2' avec pour estampille 14, e_4 avec pour estampille 7, e_2, estampille 5.

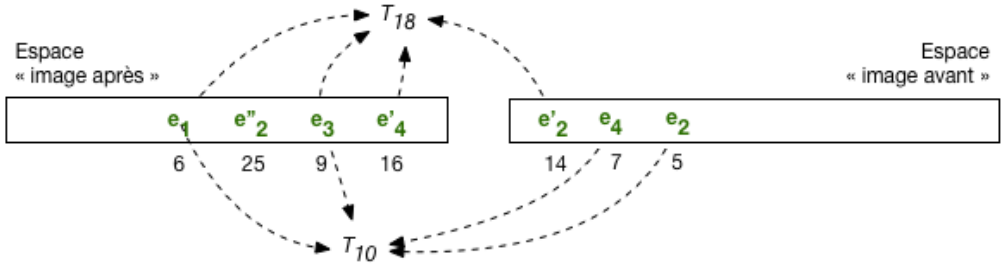

Fig. 10.3 – Lectures répétables avec image avant

La transaction T_{18} a débuté à l'instant 18. Elle lit, dans l'image après, les nuplets dont l'estampille est inférieure à 18 : e_1, e_3, e_4'. En revanche elle doit lire dans l'image avant la version de e_2 dont l'estampille est inférieure à 18, soit e_2'.

La transaction T_{10} a débuté à l'instant 10. Le même mécanisme s'applique. On constate que T_{10} doit remonter jusqu'à la version de e_2 d'estampille 5 pour effectuer une lecture cohérente.

L'image avant contient 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. L'image avant peut donc être vue comme un conteneur des « clichés » de la base pris au fil des mises à jour successives. Toute transaction ne lit que dans le cliché « valide » au moment où elle a débuté.

Certaines de ces versions n'ont plus aucune chance d'être utilisées : ce sont celles pour lesquelles il existe une version plus récente et antérieure à tous les débuts de transaction en cours. Cette propriété permet au SGBD d'effectuer un nettoyage (*garbage collection*) des versions devenues inutiles.

On peut imaginer la difficulté (et donc le coût) pour le système de cette garantie de lecture répétable. Il faut, pour une transaction donnée effectuant une lecture, remonter la chaîne des versions successives de chaque nuplet jusqu'à trouver la version faisant partie de l'état de la base au moment où la transaction a débuté. Le niveau *read committed* apparaît beaucoup plus simple à garantir, et donc probablement plus efficace. L'isolation a un prix, qui résulte de structures de données et d'algorithmes (relativement) sophistiqués.

10.1.3 Quiz

— Quelle affirmation sur l'image avant et l'image après est exacte ?

— Dans quel mode a-t-on besoin de conserver plus d'une version dans l'image avant ?

— Jusqu'à quand doit-on garder une version e_i estampillée à l'instant i

10.2 S2 : la sérialisabilité

Supports complémentaires :
— Diapositives: la sérialisabilité
— Vidéo sur la sérialisabilité

Nous en arrivons maintenant à l'isolation complète des transactions, garantie par le niveau *serializable*. La sérialisabilité est le critère utime de correction pour l'exécution concurrente de transactions. Voici sa définition :

Définition de la sérialisabilité

Soit *H* une exécution concurrente de *n* transactions $T_1, \cdots T_n$. Cette exécution est *sérialisable* si et seulement si, quel que soit l'état initial de la base, il existe un ordonnancement *H"* de $T_1, \cdots T_n$ tel que le résultat de l'exécution de *H* est équivalent à celui de l'exécution *en série* des transactions de *H"*.

En d'autres termes : si *H*, constitué d'une imbrication des opérations de $T_1, \cdots T_n$, est sérialisable, alors le résultat aurait pu être obtenu par une exécution *en série* des transactions, pour au moins un ordonnancement.

Au cours d'une exécution *en série*, chaque transaction est seule à accéder à la base au moment où elle se déroule, et l'isolation est, par définition, totale.

Relisez bien cette définition jusqu'à l'assimiler et en comprendre les détails. Le but du contrôle de concurrence va consister à n'autoriser que les exécutions concurrentes sérialisables, en retardant si nécessaire l'exécution de certaines des transactions.

Notez que la définition ci-dessus est de nature *déclarative* : elle nous donne le sens de la notion de sérialisabilité, mais ne nous fournit aucun moyen pratique de vérifier qu'une exécution est sérialisable. On ne peut pas en effet se permettre de vérifier, à chaque étape d'une exécution concurrente, s'il existe un ordonnancement donnant un résultat équivalent. Il nous faut donc des conditions plus facile à mettre en œuvre : elles reposent sur la notion de *conflit* et sur le *graphe de sérialisabilité*.

10.2.1 Conflits et graphe de sérialisation

La notion de base pour tester la sérialisabilité est celle de *conflits* entre deux opérations

---

Définition : conflit entre opérations d'une exécution concurrente.

Deux opérations $p_i[x]$ et $q_j[y]$, provenant de deux transactions distinctes T_i et T_j ($i \neq j$), sont *en conflit* si et seulement si elles portent sur le même nuplet ($x = y$), et *p* ou (non exclusif) *q* est une écriture.

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

Exemple.

Reprenons une nouvelle fois l'exemple des mises à jour perdues.

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

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 ;
— $r_2(s)$ et $w_1(s)$ 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.

Les conflits permettent de définir une relation entre les transactions d'une exécution concurrente.

Définition.

Soit *H* une exécution concurrente d'un ensemble de transactions $T = \{T_1, T_2, \cdots, T_n\}$. Alors il existe une relation $\lhd$ sur cet ensemble, définie par :

$$T_i \lhd T_j \Leftrightarrow \exists p \in T_i, q \in T_j, p \text{ est en conflit avec q et p} <_{\text{H}} \text{q}$$

où $p <_H q$ indique que p apparaît avant q dans H.

Dans l'exemple 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.

10.2.2 Condition de sérialisabilité

La condition sur la sérialisabilité s'exprime sur le graphe de la relation $(T, \lhd)$, dit *graphe de sérialisation* :

Théorème de sérialisabilité.

Soit H une exécution concurrente d'un ensemble de transactions $\mathcal{T}$. Si le graphe de $(\mathcal{T}, \lhd)$ est acyclique, alors H est sérialisable.

La Fig. 10.4 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.

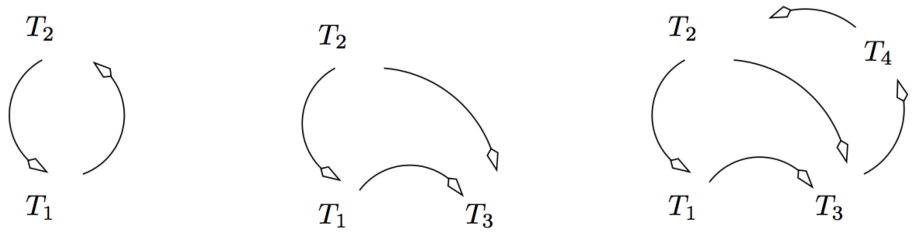

Fig. 10.4 – Exemples de graphes de sérialisation

Un algorithme de contrôle de concurrence a donc pour objectif d'éviter la formation d'un cycle dans le graphe de sérialisation. C'est une condition pratique qu'il est envisageable de vérifier pendant le déroulement d'une exécution. Il existe en fait deux grandes familles de contrôle de concurrence :

— les algorithmes dits « optimistes » surveillent les conflits en intervenant au minimum sur le déroulement des transactions, et rejettent une transaction quand un cycle apparaît ;
— les algorithmes dits « pessimistes » effectuent des verrouillages et blocages pour tenter de prévenir l'apparition de cycle dans le graphe de sérialisation.

Les sessions qui suivent présentent deux algorithmes très répandus : le contrôle multiversions, optimiste, dont la version de base (que nous présentons) ne garantit pas totalement la sérialisabilité, et le verrouillage à deux phases, plutôt de nature pessimiste, le plus utilisé quand la sérialisabilité stricte est requise.

Note : (Parenthèse pour ceux qui veulent tout savoir). Dans l'idéal, la condition sur le graphe des conflits caractériserait *exactement* les exécutions sérialisables et ontrouverait un « si et seulement si » dans l'énoncé du théorème.. Ce n'est pas tout à fait le cas. Tester qu'une exécution a un graphe des conflits cyclique (appelons ces exécutions *conflit sérialisables*) est une condition *suffisante* mais pas *nécessaire*. Certaines exécutions (très rares) peuvent être sérialisables avec un graphe de conflits cyclique. En d'autres termes, un système qui fonctionne avec le graphe des conflits rejette (un peu) trop de transactions.

L'alternative serait d'effectuer un ensemble de vérifications complexes, dont je préfère ne pas vous donner la liste. Les exécutions qui satisfont ces vérifications sont appelées *vue sérialisables* dans les manuels traitant de manière approfondie de la concurrence. La vue-sérialisabilité est une condition nécessaire *et* suffisante, et caractérise exactement la sérialisabilité.

Tester la vue-sérialisabilité est très complexe, beaucoup plus complexe que le test sur le graphe des conflits. Les systèmes appliquent donc la seconde et tout le monde est content, les scientifiques et les praticiens. Fin de la parenthèse.

10.2.3 Quiz

— Supposons une table *T(id, valeur)*, et la procédure suivante qui copie la valeur d'une ligne vers la valeur d'une autre :

```
/* Une procédure de copie */

create or replace procedure Copie (id1 INT, id2 INT) AS

 -- Déclaration des variables
 val INT;

 BEGIN
   -- On recherche la valeur de id1
   SELECT * INTO val FROM T WHERE id = id1

   -- On copie dans la ligne id2
   UPDATE T SET valeur = val WHERE id = id2

   -- Validation
   commit;
 END;
/
```

On prend deux transactions *Copie(A, B)* et *Copie(B,A)*, l'une copiant de la ligne *A* vers la ligne *B* et l'autre effectuant la copie inverse. Initialement, la valeur de *A* est *a* et la valeur de *B* est *b*. Qu'est-ce qui caractérise une exécution sérialisable de ces deux transactions ?

— Voici une exécution concurrente de deux transactions de réservation.

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

Quelles sont les opérations en conflit ?

— Voici une exécution concurrente de *Copie(A, B)* et *Copie(B,A)*

$$r_1(v_1)r_2(v_2)w_1(v_2)w_2(v_1)$$

Est-elle sérialisable, d'après les conflits et le graphe ?

10.3 S3 : Contrôle de concurrence multi-versions

Supports complémentaires :
- Diapositives: contrôle de concurrence multiversions
- Vidéo sur le contrôle de concurrence multiversions

Le contrôle de concurrence multiversions (*isolation snapshot*) permet une gestion relativement simple de l'isolation, au prix d'un verrouillage minimal. Même s'il ne garantit pas une sérialisabilité totale, c'est une solution adoptée par de nombreux systèmes transactionnels (par seulement des SGBD d'ailleurs). La méthode s'appuie essentiellement sur un versionnement des données (voir ci-dessus), et sur des vérifications de cohérence entre les versions lues et celles corrigéees par une même transaction.

L'algorithme tire 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 lisaient la base, dès le début de la transaction, donc dans l'état t_0. Cette remarque réduit considérablement les cas possibles de conflits et surtout de cycles entre conflits.

10.3.1 Les possibilités de conflit

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 (de fait, T_2 et T_1 s'exécutent en série), et il n'y a pas de risque de cycle (Fig. 10.5).

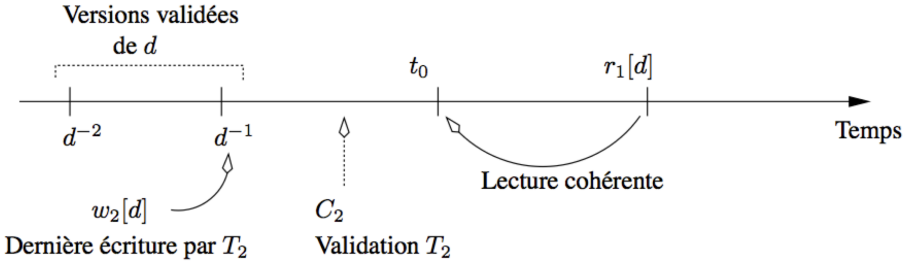

Fig. 10.5 – Contrôle de concurrence multi-versions : conflit avec les écritures précédentes

Le second cas est celui qui peut poser problème. Notez tout d'abord qu'une nouvelle lecture de d par T_1 n'introduit pas de cycle puisque toute lecture s'effectue à t_0. En revanche, si T_1 cherche à écrire d après l'écriture $w_2[d]$, alors un conflit cyclique apparaît (Fig. 10.6).

Le contrôle de concurrence peut alors se limiter à vérifier, au moment de l'écriture d'un nuplet d par une transaction *T*, qu'aucune transaction *T"* n'a modifié d entre le début de T et l'instant présent. Si on autorisait la modification de d par *T*, un cycle apparaîtrait dans le graphe de sérialisation. En d'autres termes, *une mise*

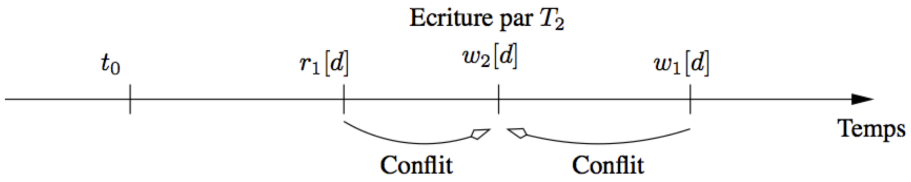

Fig. 10.6 – Contrôle de concurrence multi-versions : conflit avec une écriture d'une autre transaction.

*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*.

10.3.2 L'algorithme

Rappelons que pour chaque transaction T_i on connaît son estampille temporelle de début d'exécution τ_i ; et pour chaque version d'un nuplet *e* son estampille de validation τ_e. Le contrôle de concurrence multi-versions s'appuie sur la capacité pour une transaction de *verrouiller* une version, auquel cas aucune autre transaction ne peut y accéder jusqu'à ce que les verrous soient levés par le *commit* ou le *rollback* de la transaction verrouillante.

Algorithme de contrôle de concurrence multiversions

- toute lecture $r_i[e]$ lit la plus récente version de *e* telle que $\tau_e \leq \tau_i$; aucun contrôle ou verrouillage n'est effectué ;
- **en cas d'écriture** $w_i[e]$**,**
  - si $\tau_e \leq \tau_i$ et aucun verrou n'est posé sur *e* : T_i pose un verrou exclusif sur *e*, et effectue la mise à jour ;
  - si $\tau_e \leq \tau_i$ et un verrou est posé sur *e* : T_i est mise en attente ;
- si $\tau_e > \tau_i$, T_i est rejetée.
- au moment du `commit` d'une transaction T_i, tous les enregistrements modifiés par T_i obtiennent une nouvelle version avec pour estampille l'instant du `commit`.

Avec cette technique, on peut ne pas poser de verrou au moment des opérations de lecture, ce qui est souvent présenté comme un argument fort par rapport au verrouillage à deux phases qui sera étudié plus loin. En revanche les verrous 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 $\tau_1 = 100$, $\tau_2 = 120$. On va considérer également 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 c_2, sans verrouiller ;

— T_2 veut modifier *s* : l'estampille de *s* est inférieure à $\tau_2 = 120$, 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 sur *s* et on effectue la modification :
— T_2 modifie c_2, avec pose d'un verrou ;
— 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 $\tau_s > \tau_1$, donc que *s* a été modifié après le début de T_1. Le contrôleur doit donc rejeter T_1 sous peine d'autoriser un cycle et donc d'obtenir une exécution non sérialisable.

On obtient le rejet de l'une des deux transactions avec un contrôle *à postériori*, d'où l'expression « approche optimiste » exprimant l'idée que la technique choisit de laisser faire et d'intervenir seulement quand les conflits cycliques interviennent réellement.

10.3.3 Limites de l'algorithme

L'algorithme de contrôle multiversions est réputé efficace, plus efficace que le traditionnel verrouillage à deux phases. La comparaison est cependant biaisée car le contrôle multiversions ne garantit pas la sérialisabilité dans tous les cas, comme le montre l'exemple très simple qui suit. On reprend la procédure de copie d'une ligne à l'autre dans la table *T*, et l'exécution concurrente de deux transactions issues de cette procédure.

$$r_1(v_1)r_2(v_2)w_1(v_2)w_2(v_1)$$

Vous devriez déjà être convaincu que cette exécution n'est pas sérialisable. Si on la soumet à l'algorithme de contrôle de concurrence multi-version, un rejet de l'une des transactions devrait dont survenir. Or, il est facile de vérifier que

— La valeur v_1 est lue, pas de contrôle ni de verrouillage.
— La valeur v_2 est lue, pas de contrôle ni de verrouillage.
— La valeur v_2 est modifiée sans obstacle, car aucune mise à jour de v_2 n'a eu lieu depuis le début de l'exécution.
— Même chose pour v_1.

Donc, tout se déroule sans obstacle, et la non-sérialisabilité n'est pas détectée. À la fin de l'exécution, les valeurs de *A* et *B* diffèrent alors qu'elles devraient être égales. Cet algorithme n'est pas d'une correction absolue, même s'il détecte la plupart des situations non-sérialiables, y compris notre exemple prototypique des mises à jour perdues.

Des travaux de recherche ont proposé des améliorations garantissant la sérialisabilité stricte, mais elles ne semblent pas encore intégrées aux systèmes qui s'appuient sur la solution éprouvée du verrouillage à deux phases, présenté ci-dessous.

10.4 S4 : le verrouillage à deux phases

Supports complémentaires :
— Diapositives: verrouillage à deux phases
— Vidéo sur le verrouillage à deux phases

L'algorithme de verrouillage à deux phases (que nous simplifierons en 2PL pour *2 phases locking*) est le plus ancien, et toujours le plus utilisé des méthodes de contrôle de concurrence assurant la sérialisabilité stricte. Il a la réputation d'induire beaucoup de blocages, voire d'interblocages, ainsi que des rejets de transactions. Comme nous l'avons indiqué à de très nombreuses reprises, aucune solution n'est idéale et il faut faire un choix entre le risque d'anomalies ponctuelles et imprévisibles, et des blocages et rejets tout aussi ponctuels et imprévisibles mais assurant la correction des exécutions concurrentes.

L'algorithme lui-même est relativement simple. Il s'appuie sur des méthodes de verrouillage qui sont présentées en premier lieu.

10.4.1 Verrouillage

Le 2PL est basé sur le *verrouillage* des nuplets lus ou mis à jour. L'idée est simple : chaque transaction désirant lire ou écrire un nuplet doit auparavant obtenir un verrou sur ce nuplet. Une fois obtenu (sous 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.

Le 2PL gère deux types de verrous :

— les *verrous partagés* autorisent la pose d'autres verrous partagés sur le même nuplet.
— 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 nuplet. 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*), et les 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 nuplet. 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 nuplet tant que ce nuplet n'est pas verrouillé exclusivement par une autre transaction. Enfin, si une transaction est la seule à détenir un verrou partagé sur un nuplet, elle peut « promouvoir » ce verrou en 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 ne peut pas s'exécuter car le verrou correspondant ne peut pas être posé, 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 (nuplet ou même table) (cf. chapitre d'introduction à la concurrence).

Tous les SGBD proposent un verrouillage au niveau du nuplet, et privilégient les verrous partagés tant que cela ne remet pas en cause la correction des exécutions concurrentes. Un verrouillage au niveau du nuplet est considéré comme moins pénalisant pour la fluidité, puisqu'il laisse libres d'autres transactions d'accéder à tous les autres nuplets 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 nuplet, et valide à la fin, on

va poser un verrou sur chaque nuplet 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 nuplets 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. Ces méthodes ne sont pas abordées ici : à vous de voir, une fois les connaissances fondamentales acquises, comment gérer au mieux votre application avec les outils proposés par votre SGBD.

10.4.2 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 *ordonnanceur* qui reçoit les opérations émises par les transactions et les traite selon l'algorithme suivant :

— L'ordonnanceur reçoit $p_i[x]$ et consulte le verrou déjà posé sur *x*, $ql_j[x]$, s'il existe.

1. 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.
2. 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 sont sérialisables. 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 nuplet 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.

10.4.3 Quelques exemples

Prenons pour commencer 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 ne respecte pas les deux phases du 2PL, et 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 valide puis relâche les verrous sur *x* et *y* ;

— 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 a 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*. Le graphe de sérialisation est cyclique.

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_2 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. Cette méthode reste cependant peu sûre et ne dispense pas de se mettre en mode sérialisable pour garantir la correction des exécutions concurrentes.

10.4.4 Quiz

— Une transaction T_1 a lu un nuplet x et posé un verrou partagé. Quelles affirmations sont vraies ?
— On reprend la procédure de copie et l'exécution concurrente de deux transactions.

$$r_1(v_1)r_2(v_2)w_1(v_2)w_2(v_1)$$

A votre avis, en appliquant un 2PL :
— On exécute en concurrence deux transactions de réservation, par le même client mais pour deux spectacles différents :

$$r_1(s_1)r_1(c)r_2(s_2)r_2(c)w_2(s_2)w_2(c)C_2w_1(s_1)w_1(c)C_1$$

Quelle opération à votre avis entraînera le premier blocage d'une transaction ?

10.5 Exercices

Exercice ex-grapheserial : Graphe de sérialisabilité et équivalence des exécutions

Identifiez les conflits et construisez les graphes de sérialisabilité pour les exécutions concurrentes suivantes. Indiquez les exécutions sérialisables et vérifiez s'il y a des exécutions équivalentes.
— $H_1 : w_2[x]\ w_3[z]\ w_2[y]\ c_2\ r_1[x]\ w_1[z]\ c_1\ r_3[y]\ c_3$
— $H_2 : r_1[x]\ w_2[y]\ r_3[y]\ w_3[z]\ c_3\ w_1[z]\ c_1\ w_2[x]\ c_2$
— $H_3 : w_3[z]\ w_1[z]\ w_2[y]\ w_2[x]\ c_2\ r_3[y]\ c_3\ r_1[x]\ c_1$

Exercice ex-multiversions1 : application du contrôle multiversion à l'anomalie des lectures non répétables

On reprend l'exemple d'une imbrication de la procédure de réservation et de la procédure de contrôle (voir chapitre précédent).

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

Appliquer le contrôle multi-versions. Que constate-t-on ?

Exercice ex-multiversions2 : les limites du contrôle de concurrence multi-versions

Supposons qu'un hôpital gère la liste de ses médecins dans une table (simplifiée) *Docteur(nom, garde)*, chaque médecin pouvant ou non être de garde. On doit s'assurer qu'il y a toujours au moins deux médecins de garde. La procédure suivante doit permettre de placer un médecin au repos en vérifiant cette contrainte.

```
/* Une procédure de gestion des gardes */

create or replace procedure HorsGarde (nomDocteur VARCHAR) AS

 -- Déclaration des variables
 val nb_gardes;

 BEGIN
   -- On calcule le nombre de médecin de garde
   SELECT count(*) INTO nb_gardes FROM Docteur WHERE nom = nomDocteur;

   IF (nb_gardes > 2) THEN
      UPDATE Docteur SET garde = false WHERE nom = nomDocteur;
      COMMIT;
   ENDIF
 END;
/
```

En principe, cette procécure semble très correcte (et elle l'est). Supposons que nous ayons trois médecins, Philippe, Alice, et Michel, désignés par *p*, *a* et *m*, tous les trois de garde. Voici une exécution concurrente de deux transactions T_1 = *HorsGarde("Philippe")* et T_2 = *HorsGarde("Michel")*.

$$r_1(p)r_1(a)r_1(m)r_2(p)r_2(a)r_2(m)w_1(p)w_2(m)$$

Questions :

- Quel est avec cette exécution le nombre de médecins de garde constatés par T_1 et T_2
- Quel est le nombre de médecins de garde à la fin ?
- Conclusion ? Vous pouvez vérifier la sérialisabilité (conflits, graphes) et appliquer le contrôle de concurrence multi-versions pour vérifier s'il détecte et prévient les anomalies de cette exécution.

Exercice ex-debitcredit : le cas des débits/crédits

Les trois programmes suivants peuvent s'exécuter dans un système de gestion bancaire. `Débit()` diminue le solde d'un compte *c* avec un montant donné *m*. Pour simplifier, tout débit est permis (on accepte des découverts). `Crédit()` augmente le solde d'un compte *c* avec un montant donné *m*. Enfin `Transfert()` transfère un montant *m* à partir d'un compte source *s* vers un compte destination *d*. L'exécution de chaque programme démarre par un `Start` et se termine par un `Commit` (non montrés ci-dessous).

```
Débit (c:Compte;    |  Crédit (c:Compte;    |  Transfert (s,d:Compte;
       m:Montant)   |           m:Montant)   |              m:Montant)
begin               |  begin                 |  begin
  t := Read(c);     |    t := Read(c);       |    Débit(s,m);
  Write(c,t-m);     |    Write(c,t+m);       |    Crédit(d,m);
end                 |  end                   |  end
```

Le système exécute en même temps les trois opérations suivantes :

1. un transfert de montant 100 du compte A vers le compte B
2. un crédit de 200 pour le compte A
3. un débit de 50 pour le compte B

— Écrire les transactions T_1, T_2 et T_3 qui correspondent à ces opérations. Montrer que l'histoire H suivante :

$$r_1[A]r_3[B]w_1[A]r_2[A]w_3[B]r_1[B]c_3w_2[A]c_2w_1[B]c_1$$

est une exécution concurrente de T_1, T_2 et T_3.

— Mettre en évidence les conflits dans H et construire le graphe de sérialisation de cette exécution. H est-elle sérialisable ?

— Quelle est l'exécution $H"$ obtenue à partir de H par verrouillage à deux phases ?

— Si au début le compte A avait un solde de 100 et B de 50, quel sera le solde des deux comptes après la reprise si une panne intervient après l'exécution de :math :*w_{1}[B]* ?

Exercice ex-2pl1 : verrouillage à deux phases

Le programme suivant s'exécute dans un système de gestion de commandes pour les produits d'une entreprise. Il permet de commander une quantité donnée d'un produit qui se trouve en stock. Les paramètres du programme représentent respectivement la référence de la commande (c), la référence du produit (p) et la quantité commandée (q).

```
function Commander
  $c: référence de la commande
  $p: référence du produit
  $q: quantité commandée

  Lecture prix produit $p
  Lecture du stock $s$ du produit $p
  if ($q > $s) then
    rollback
  else
     Mise à jour stock de $p
    Enregistrement dans $c du total de facturation
    commit
  fi
```

Notez que le prix et la quantité de produit en stock sont gardés dans des enregistrements différents.

— Lesquelles parmi les transactions suivantes peuvent être obtenues par l'exécution du programme ci-dessus ? Justifiez votre réponse.
  — $T_1 : r[x]r[y]R$
  — $T_2 : r[x]R$
  — $T_3 : r[x]w[y]w[z]C$
  — $T_4 : r[x]r[y]w[y]w[z]C$
— Dans le système s'exécutent en même temps trois transactions : deux commandes d'un même produit et le changement du prix de ce même produit. Montrez que l'histoire *H* ci-dessous est une exécution concurrente de ces trois transactions et expliquez la signification des enregistrements qui y interviennent.

$$r_1[x]r_1[y]w_2[x]w_1[y]c_2r_3[x]r_3[y]w_1[z]c_1w_3[y]w_3[u]c_3$$

— Vérifiez si *H* est sérialisable en identifiant les conflits et en construisant le graphe de sérialisation.
— Quelle est l'exécution obtenue par verrouillage à deux phases à partir de *H* ? Quel prix sera appliqué pour la seconde commande, le même que pour la première ou le prix modifié ?

Exercice ex-medecindegarde : les médecins de garde et le contrôle de concurrence

On reprend les transactions de gestion de docteurs et de leurs gardes (voir section *S3 : Contrôle de concurrence multi-versions*). On a l'exécution concurrente suivante des deux transactions cherchant à lever la garde de Philippe et de Michel.

$$r_1(p)r_1(a)r_1(m)r_2(p)r_2(a)r_2(m)w_1(p)w_2(m)$$

Questions :

— Trouver les conflits
— En déduire (argumenter) que cette exécution concurrente n'est pas sérialisable
— Appliquer l'algorithme de contrôle multiversions
— Appliquer l'algorithme 2PL.

10.6 Références

Le sujet du contrôle de concurrence est complexe et a donné lieu à de très nombreux travaux. C'est, au final, une des très grandes réussites des systèmes relationnels. Si vous voulez aller plus loin :

— le livre de Jim Gray, *Transaction Processing : Concepts and Techniques*, paru en 1992, est *la* référence
— Cet excellent résumé est facilement accessible : https://wiki.postgresql.org/wiki/Serializable
— Le contrôle multi-versions amélioré pour atteindre le niveau sérialisable est ici : http://www.cs.nyu.edu/courses/fall09/G22.2434-001/p729-cahill.pdf. Quelques exemples du chapitre sont repris de cet article.

CHAPITRE 11

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, d'une défaillance logicielle, ou des pannes affectant les disques Par souci de simplicité on va distinguer deux types de panne (quelle que soit la cause).

- *Panne légère* : affecte la RAM du serveur de données, pas les disques
- *Panne lourde* : affecte un disque

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.

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 relationnele, 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.

11.1 S1 : introduction

Supports complémentaires :
— Diapositives: introduction
— Vidéo d'introduction

11.1.1 L'état de la base

Commençons par définir une première notion très importante, *l'état de la base*.

Définition : l'état de la base ?

On définit l'état de la base à un instant t commme l'état résultant de l'ensemble des transactions validées à l'instant t.

Pour assurer sécurité des données (face à une panne légère au moins), il est impératif que l'état de la base soit stocké sur support persistant, à tout instant. Une première règle simple est donc :

Règle 1

L'état de la base doit toujours être stocké sur disque

Pour le dire autrement, aucune donnée validée ne devrait être en mémoire RAM et pas sur le disque, car elle serait perdue en cas de panne.

11.1.2 Garanties transactionnelles

Souvenons-nous maintenant des propriétés des transactions.
— *Durabilité (et atomicité)* : quand le système rend la main après un `commit` (*acquittement*), *toutes* les modifications de la transaction intègrent l'état de la base et deviennent *permanentes*.
— *Recouvrabilité (et atomicité)* : tant qu'un `commit` n'a pas eu lieu, *toutes* les modifications de la transaction doivent pouvoir être *annulées* par un `rollback`.

Nous avons déjà rencontré les notions de versions d'un nuplet en cours d'exécution d'une transaction, et nous les avons appelées *image avant* et *image après*. Rappelons leur définition :

Définition : image avant et image après

L'image après d'une transaction T désigne la nouvelle valeur des nuplets modifiés par T. *L'image avant* désigne l'ancienne valeur de ces nuplets (*avant* modification).

L'exécution d'une transaction peut donc se décrire de la manière suivante, illustrée par la Fig. 11.1. Au départ, l'état de la base est stocké sur disque. Une partie de cet état correspond à l'image avant des nuplets qui vont

être modifiés par la transaction. Au cours de la transaction, l'*image après* est constituée, et la transaction a, à chaque instant, deux possibilités :

— un `commit`, et l'image après remplace l'image avant dans l'état de la base ;
— un `rollback`, et l'état de la base est restauré comme initialement, avec l'image avant.

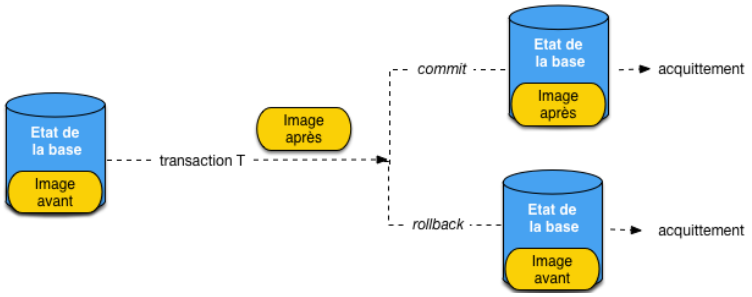

Fig. 11.1 – Problématique de la reprise sur panne en termes d'état de la base, image avant et image après.

Le moment où la transaction effectue un `commit` ou un `rollback` est crucial. Chacune de ces instructions est « acquitté » quand le serveur de données rend la main au processus client. Pour garantir le `commit`, la condition suivante doit être respectée.

Règle 2

L'image après doit être sur le disque avant l'acquittement du `commit`.

Si ce n'était pas le cas et qu'une panne survienne juste après l'acquittement, mais avant l'écriture de l'image après sur le disque, cette dernière serait en partie ou totalement perdue, et la garantie de durabilité ne serait pas assurée.

Maintenant, pour garantir le `rollback`, il faut pouvoir trouver sur le disque, après une panne, les données modifiées par la transaction dans l'état où elles étaient au moment où la transaction a débuté. La condition suivante doit être respectée.

Règle 3

L'image avant doit être sur le disque jusqu`à l'acquittement du `commit`.

On peut résumer la difficulté ainsi : jusqu'au `commit`, c'est *l'image avant* qui fait partie de l'état de la base. Au `commit`, *l'image après* remplace *l'image avant* dans l'état de la base. Il faut assurer que ce remplacement s'effectue de manière `atomique` (« tout ou rien ») ? La suite montre que ce n'est pas facile.

11.1.3 Quiz

Une panne légère est une panne qui affecte :

Quels sont les liens entre la reprise sur panne (RP) et les propriétés transactionnelles présentées dans les chapitres précédents ?

Indiquez, parmi les affirmations suivantes, celles qui sont **incorrectes** :

Une transaction T débutant à t0 et s'exécutant en isolation totale voit à un instant t :

11.2 S2 : mise à jour différée, immédiate et opportuniste

Supports complémentaires :
- Diapositives: architecture
- Vidéo sur l'architecture

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.

La Fig. 11.2 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.

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.

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.

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 Fig. 11.3 illustre la situation pour le bloc B_v.

La question qui se pose alors, par rapport à la reprise sur panne, est de savoir quand il faut écrire un bloc modifié, et l'imopact qu'à cette stratégie d'écriture sur la gestion de la reprise. Nous allons étudier trois possibilités : écriture immédiate, écriture différée, et écriture opportuniste.

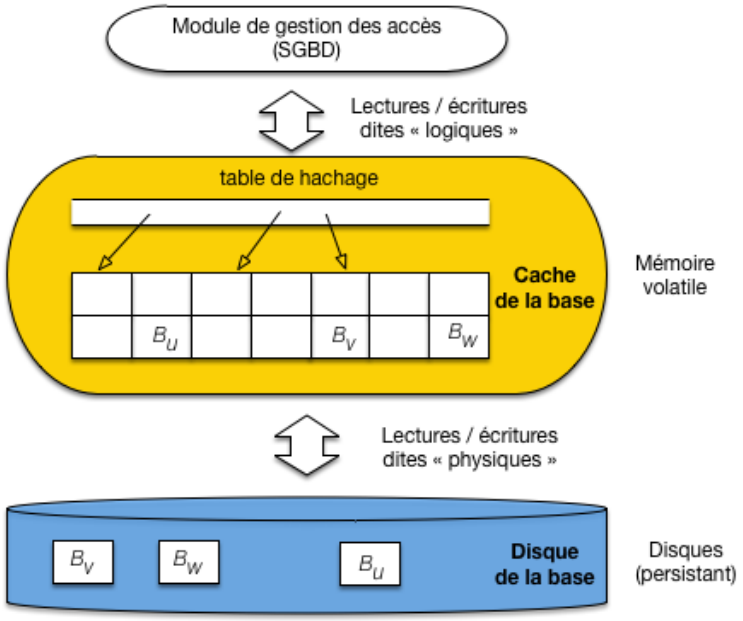

Fig. 11.2 – Le *cache* et le disque, ressources mémoires allouées au SGBD

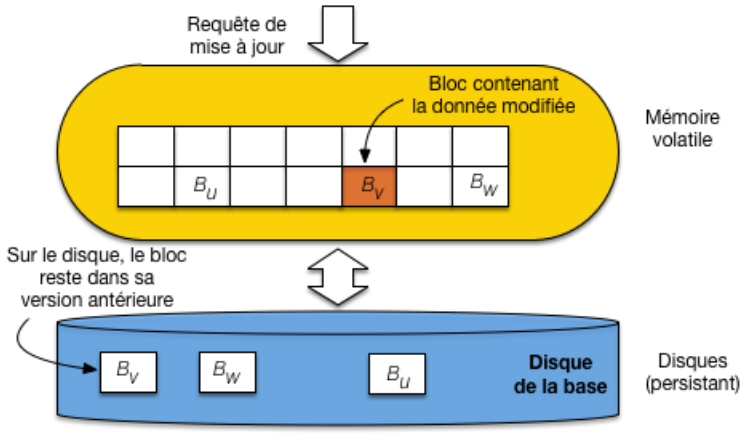

Fig. 11.3 – Mise à jour dans le *cache* : faut-il écrire sur le disque ou pas ?

11.2.1 Ecritures immédiates

La stratégie d'écriture immédiate synchronise un bloc modifié avec son image dans le cache dès qu'une mise à jour est effectuée (Fig. 11.4). Cela garantit que l'image après est sur le disque, mais écrase l'image avant et risque donc de rendre impossible un `rollback`.

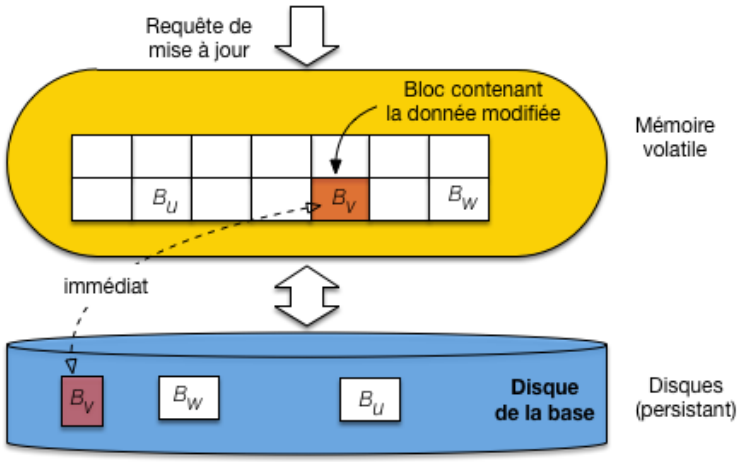

Fig. 11.4 – Ecriture immédiate : le *cache* et le disque sont synchrones

L'écriture immédiate a un autre inconvénient : le coût d'écriture d'un bloc pour chaque mise à jour d'un nuplet. Pour des applications qui font beaucoup de modifications, les performances risquent d'être sévèrement affectées.

11.2.2 Ecritures différées

Les écritures différées fonctionnent à l'inverse des écritures immédiates : on garde dans le cache tous les blocs modifiés, et on s'interdit de les écrire tant que les modifications ne font pas l'objet d'un `commit` (Fig. 11.5).

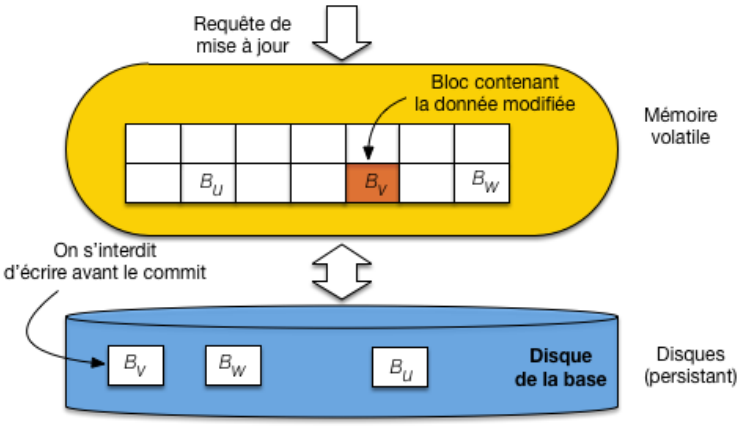

Fig. 11.5 – Ecriture différée : On *interdit* l'écriture d'un bloc modifié avant le `commit`

Cette stratégie est en apparence plus satisfaisante pour la reprise sur panne : en cas de panne, la mémoire

RAM contient l'image après qui doit justement être effacée, et le disque contient l'image avant qui doit être conservée. Elle a l'inconvénient de mobiliser potentiellement beaucoup de mémoire RAM en présence de grandes transactions qui font beaucoup de modifications. De plus, certains détails techniques sont compliqués : comment faire par exemple si un bloc contient des données modifiées par deux transactions, et que la première valide alors que la seconde annule ?

11.2.3 Ecritures opportunistes

Enfin, la troisième stratégie est celle des écritures opportunistes. Dans ce cas, un bloc modifié en mémoire n'est pas écrit immédiatement, mais il peut l'être à un moment totalement indépendant du déroulement de la transaction, si le système l'estime nécessaire (Fig. 11.6).

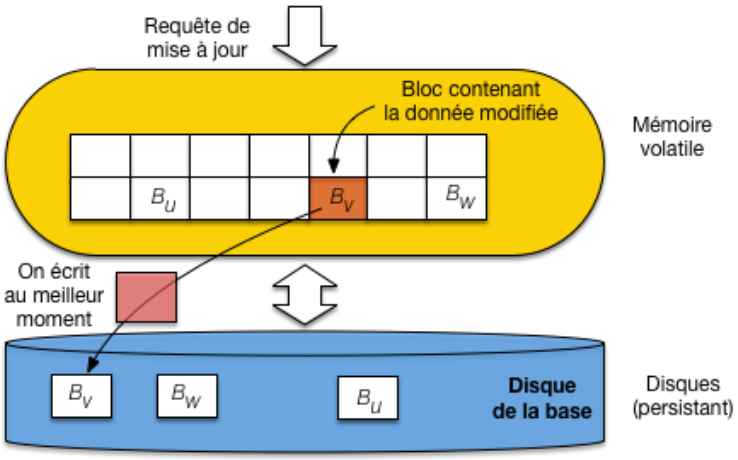

Fig. 11.6 – Ecriture opportuniste : on attend la meilleure opportunité pour synchroniser le buffer et le disque.

C'est le mode le plus satisfaisant pour les performances, puisque le système peut choisir l'opprtunité offerte par un moment favorable pour écrire un bloc, en subissant un minimum de contraintes. C'est en apparence une stratégie très défavorable à la reprise sur panne puisque l'image après est partiellement en mémoire, partiellement sur le disque, et que l'image avant est partiellement effacée.

Nous verrons dans la prochaine session qu'aucune de ces stratégies n'est à elle seule suffisante pour concevoir et implanter un algorithme fiable de reprise sur panne.

11.2.4 Quiz

En mode opportuniste, quelle affirmation est exacte ?

En mode différé, quelle affirmation est exacte ?

En mode immédiat, quelle affirmation est exacte ?

11.3 S3 : une approche simpliste

Supports complémentaires :
- Diapositives: une approche simpliste
- Vidéo sur l'approche simpliste

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 épinglés 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.

11.3.1 Quiz

Le point de commit est une notion qui désigne :

En mode opportuniste, je peux garantir :

En mode immédiat, je peux garantir :

En mode différé, je peux garantir :

11.4 S4 : journal des transactions

Supports complémentaires :
— Diapositives: le journal des transactions
— Vidéo sur 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 dite *write-ahead*

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 Fig. 11.7 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.

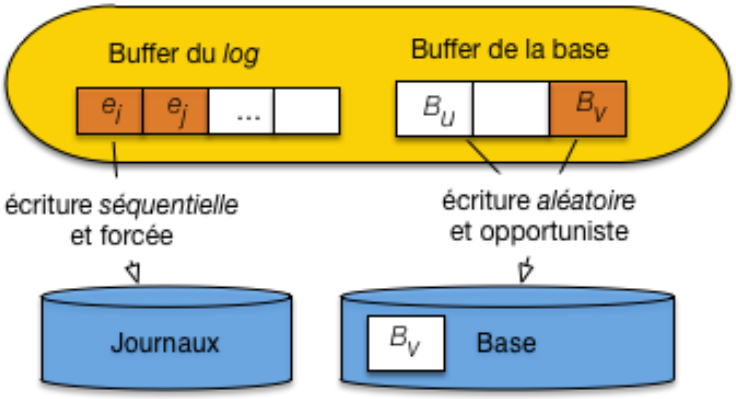

Fig. 11.7 – Gestion des écritures avec fichier journal

11.4.1 Quiz

Quelle affirmation, relatives au contenu du *log*, est-elle *inexacte* ?

La règle du point de commit vise à garantir :

Dans quel mode n'est-t-il pas nécessaire d'appliquer la règle du *write-ahead logging* ?

Quelle affirmation sur le fonctionnement du log est-elle fausse ?

11.5 S5 : Algorithmes de reprise sur panne

Supports complémentaires :

— Diapositives: faire et défaire
— Vidéo sur les algorithmes redo/undo

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 par 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).

11.5.1 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.

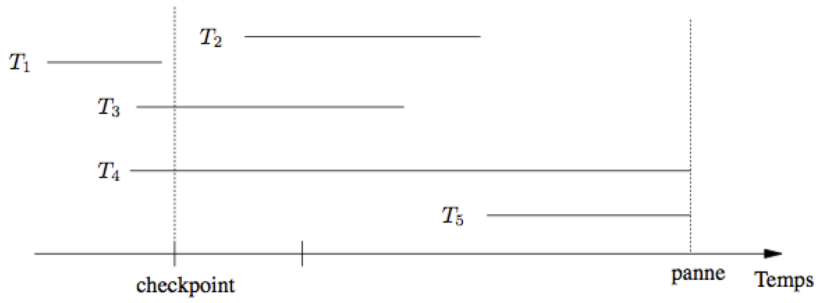

Fig. 11.8 – Reprise sur panne après un *checkpoint*

La Fig. 11.8 montre un exemple, avec un *checkpoint* survenant après la validation de *T_1*. Toutes les mises à jour de *T_1* 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 *T_2* 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.

11.5.2 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 épingler 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 !

11.5.3 Avec mise à jour immédiates ou opportunistes

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 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 modifiées) intervienne naturellement pour qu'elles soient placées sur le disque.

11.5.4 Quiz

Dans quel mode ne faut-il pas faire un Redo ?

11.6 S6 : pannes de disque

Supports complémentaires :

— Diapositives: pannes de disque
— Vidéo sur les pannes de disque

11.6.1 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).

La panne du disque journal est très grave car il devient alors impossible de reconstituer correctement la base. Il se peut notamment que des transactions en cours d'exécution aient déjà écrit sur le disque de la base. L'image avant est alors seulement dans le journal, se trouve donc perdue, et il devient impossible d'annuler la transaction. Forcer les transactions en cours à effectuer un `commit`, et ce au moment d'une panne de disque, n'est pas une solution viable.

Il faut à tout prix protéger le journal, soit en utilisant des systèmes RAID de disques redondants, soit en le répliquant dans un site sûr.

En cas de panne du disque de la base, il faut en fait effectuer une reprise sur panne à partir des journaux, en appliquant les Redo et Undo à la dernière sauvegarde disponible.

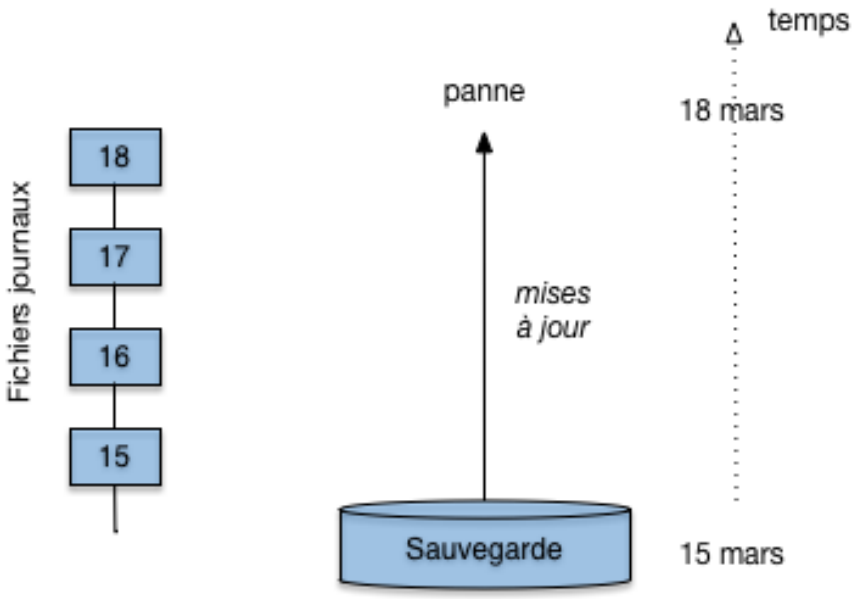

Fig. 11.9 – Reprise à froid avec une sauvegarde et des fichiers *log*

La Fig. 11.9 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* doit être refaites). C'est vrai seulement pour des reprises à chaud, après coupure de courant. En cas de perte d'une 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 modifiées, répartis dans (au pire) *n* pages, il faut écrire ces *n* pages sur le disque, souvent sans contiguité, ce qui est très coûteux.
En revanche ces *n* enregistrements peuvent être regroupés dans une 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 `rolback` 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 un 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*.

11.6.2 Quiz

11.7 Exercices

Exercice ex-rp1 : la reprise sur panne à la main

Soit le fichier journal suivant (les écritures les plus anciennes sont en haut).

```
start(T1)
write (T1, x, 10, 20)
commit(T1)
checkpoint
start(T2)
write(T2, y, 5, 10)
start(T4)
write(T4, x, 20, 40)
start(T3)
write(T3, z, 15, 30)
write(T4, u, 100, 101)
commit(T4)
```

(suite sur la page suivante)

(suite de la page précédente)

```
write (T2, x, 40, 60)
..................... panne!
```

Questions.

— Indiquer la reprise sur panne avec l'algorithme Undo/Redo
— Donner le comportement de la variante avec mise à jour différée (No-Undo/Redo). Y a-t-il des informations inutiles dans le journal ?

Exercice ex-rp2 : la reprise sur panne à la main

Spécifiez un algorithme de Redo qui effectue le parcours du *log* dans l'ordre inverse des insertions, et s'arrête dès que possible.

Exercice ex-rp3 : petit questionnaire

Indiquez la bonne réponse aux questions suivantes (en les justifiant).

— Pendant une reprise sur panne les opérations doivent être, :
  — commutatives
  — associatives
  — idempotentes
  — distributives.
— Dans un protocole de reprise sur panne avec mise à jour différée, quelles sont les opérations nécessaires :
  — Undo
  — Redo
  — Undo et Redo
  — aucune des deux.
— Dans le cas d'un algorithme avec mises à jour différées, que doit-on conserver dans le log, ?
  — la valeur avant mise à jour
  — la valeur après mise à jour
  — les valeurs avant et après mise à jour
  — uniquement les `start(T)` et `commit(T)`

Exercice ex-rp4 : pourquoi dans l'ordre inverse ?

Donner un exemple illustrant la nécessité d'effectuer un Redo dans l'ordre inverse de l'exécution (donner les entrées du fichier journal, et expliquer le déroulement de l'annulation).

CHAPITRE 12

Annales des examens

12.1 Examen blanc du 20 janvier 2020 (sans concurrence)

12.1.1 Stockage et indexation

On veut stocker un fichier F avec 120000 enregistrements d'une taille fixe de 100 octets par article.

Questions :

— De combien de blocs a-t-on besoin au minimum pour stocker tout le fichier si la taille d'un bloc est 8192 octets, sachant que chaque bloc contient un entête de 150 octets et qu'un enregistrement ne peut pas chevaucher 2 blocs ?

— On suppose maintenant que F est indexé par un index non dense sur `A` et un index dense sur un autre champ `B`. On peut stocker 100 entrées dans un bloc d'index et aucune place libre n'est laissée dans les blocs. Combien d'entrées y a-t-il dans les feuilles de l'index non dense ? dans les feuilles de l'index dense ?

— Supposons que l'index non dense a deux niveaux, racine comprise, et que seule la racine est en mémoire. On se place dans le pire des cas où il faut une lecture physique pour lire une feuille d'index et une autre pour lire un bloc du fichier.
On cherche les enregistrements pour lesquels la valeur de l'attribut `A` est comprise entre P1A3 et P3G5. On suppose qu'il y a moins de 100 enregistrements tels que `A` commençe par P. Combien de lectures coûte la recherche par l'index dans le pire des cas ?

— Même question avec l'index dense, pour une recherche sur l'attribut `B`.

12.1.2 Index et optimisation

Soit les tables relationnelles suivantes (les attributs qui forment une clé sont en gras) :

- Produits (**code**, marque, desig, descr). NB : code : identifiant du produit, marque : marque du produit, desig : désignation du produit, descr : description.
- PrixFour (**code, nom**, prix). NB : code : code du produit, nom : nom du fournisseur.
- NoteMag (**code, titre**, note). NB : code : code du produit, titre : titre du magazine, note : note entre 1 et 10 du produit dans le magazine)

Voici une instance de la table NoteMag.

Code	Titre	Note
“A345”	“HIFI”	8
“P123”	“Audio Expert”	6
“X254”	“HIFI”	7
“K783”	“Son & Audio”	3
“P345”	“HIFI”	6
“P512”	“Audio Expert”	8
“L830”	“Audio Expert”	8
“M240”	“”HIFI””	6

La table occupe plusieurs blocs dont chacun peut contenir au maximum 3 n-uplets.

- Construire un arbre B+ sur l’attribut `code`, avec 4 entrées par bloc au maximum.
- Est-il utile d’indexer l’attribut `titre` ? Pourquoi ?
- Soit la requête suivante :

```
select *
from NoteMag
where code between 'A000' and 'X000';
```

  Pour évaluer cette requête, on suppose que le tampon de lecture ne peut contenir qu’un seul bloc et l” index tient en mémoire. Dans ce cas, est-ce qu’il est préférable d’utiliser l’index ou de parcourir la table séquentiellement ? Pourquoi ?

On donne ci-dessous une requête SQL et le plan d’exécution fourni par Oracle :

```
select desig, marque, prix
from Produits, PrixFour, NoteMag
where Produits.code=PrixFour.code
and Produits.code=NoteMag.code
and note > 8;
```

Plan d’exécution :

```
0 SELECT STATEMENT
   1 MERGE JOIN
      2 SORT JOIN
         3 NESTED LOOPS
            4 TABLE ACCESS FULL NOTEMAG
            5 TABLE ACCESS BY INDEX ROWID PRODUITS
```

(suite sur la page suivante)

(suite de la page précédente)

```
            6 INDEX UNIQUE SCAN A34561
     7 SORT JOIN
        8 TABLE ACCESS FULL PRIXFOUR
```

Questions :

- Existe-t-il un index ? sur quel(s) attribut(s) de quel(s) table(s) ?
- Algorithme de jointure : Expliquer en détail le plan d'exécution (accès aux tables, sélections, jointure, projections)
- Ajout d'index : On crée un index sur l'attribut `note` de la table `NoteMag`. Expliquez les améliorations en terme de plan d'exécution apportées par la création de cet index.

12.2 Examen blanc juin 2020

On prend pour exemple une base de données qui sert à la gestion d'un site web de diffusion de la musique en *streaming* :

- Abonné (id, nom, prénom, typeAbonnement, dateDébut)
- Artiste (id, nom, prénom, nationalité)
- Album (id, nom, idArtiste, annéeSortie)
- Chanson (id, nom, idAlbum)
- Ecoute (idAbonné, idChanson, date, téléchargé)

L'abonnement est sans engagement : seulement la date de début est nécessaire pour un abonnement en cours. Le type d'un abonnement peut être : "free" (financé par la publicité), "normal" et "premium" (donne le droit de télécharger la musique en local pour une écoute hors connexion). L'attribut "téléchargé" vaut vrai ou faux. Un abonné peut avoir écouté une chanson plusieurs fois (à des dates différentes).

Le SGBD crée un index sur les clés primaires, mais pas sur les clés étrangères.

12.2.1 Questions sur le schéma (3 points)

- Pour la table Ecoute, identifiez les clés primaires et étrangères.
- Donnez la commande SQL pour créer la table Ecoute.
- Une recherche par index est-elle possible si on interroge la table Ecoute sur l'id d'une chanson (justifier) ?
- Une recherche par index est-elle possible si on interroge la table Ecoute sur l'id d'un abonné (justifier) ?

12.2.2 Stockage et indexation (4 points)

On insère successivement les enregistrements suivants dans la table Artiste, selon cet ordre :

Id	Nom	Prénom	Nationalité
1	Moustaki	Georges	française
2	Modja	Inna	malienne
3	LeForestier	Maxime	française
4	Vian	Boris	française
5	DePalmas	Gérald	française
6	June	Valérie	américaine
7	Higelin	Jacques	française
8	Berger	Michel	française
9	Goldman	Jean-Jacques	française
10	Mitchell	Eddy	française
11	Katerine	Philippe	française
12	Estefan	Gloria	cubaine
13	Marley	Bob	jamaicaine
14	Azrié	Abed	française

On met en place un index sur l'attribut `nom` de cette table.

— Construire l'arbre B d'ordre 2 (4 entrées max par bloc) correspondant à cet ensemble d'enregistrements, en respectant l'ordre d'insertion. Donner les étapes de construction intermédiaires importantes (celles qui produisent un changement de la structure de l'arbre).

— On considère que cet arbre B est stocké à raison d'un nœud par bloc sur disque. On recherche les noms des artistes dont l'initiale du nom se trouve entre "B" et "I" (inclus). Combien de blocs disque doivent être chargées au minimum pour répondre à cette requête en utilisant l'arbre B+ ? Justifier.

12.2.3 Optimisation (7 points)

Soit la requête suivante :

```
select Album.nom
from Artiste, Album
where Artiste.id=Album.idArtiste
and Artiste.nom='Moustaki'
```

Questions :

— Donnez le plan d'exécution sous la forme de votre choix, en supposant que les seuls index sont ceux sur les clés primaires

— Qu'est-ce qui change si on crée l'index sur le nom des artistes ?

Soit maintenant le plan d'exécution suivant :

```
0 SELECT STATEMENT
    1*   MERGE JOIN
    2      SORT JOIN
    3*        NESTED LOOPS
```

(suite sur la page suivante)

(suite de la page précédente)

```
4*          TABLE ACCESS FULL           Ecoute
5           TABLE ACCESS BY ROWID       Chanson
6              INDEX RANGE SCAN         IDX-Chanson_ID
7       SORT JOIN
8          TABLE ACCESS FULL            Album

1 - access(Chanson.id_album=Album.id)
3 - access(Ecoute.id_chanson=Chanson.id)
4 - access(date=29/05/2013)
```

Questions :

— Donnez la requête correspondante
— Expliquez ce plan, en indiquant notamment quels index existent, et lesquels n'existent pas
— Quels index pouvez-vous ajouter pour optimiser cette requête, et quel est le plan d'exécution correspondant ?

12.2.4 Concurrence (6 points)

Soit l'exécution concurrente suivante :

$$H = r_2[x]r_3[x]w_1[y]r_3[y]w_3[y]r_1[z]w_2[y]c_1w_3[z]w_2[z]c_3c_2$$

Questions

— Donner la liste des conflits de H
— Donner le graphe de sérialisation de H. Que pouvez-vous déduire de ce graphe ?
— Donner l'exécution finale obtenue par application de l'algorithme de verrouillage à deux phases. Donner le détail du déroulement de l'algorithme.
— Que se passerait-il si on ne posait que des verrous exclusifs ?

12.3 Examen juin 2022

12.3.1 Stockage et indexation (6 points)

Voici le contenu d'un fichier `animaux` :

```
(jaguar, 17), (chameau, 22), (gnou, 1), (girafe, 13), (chat, 3),
(lion, 8), (dauphin, 40), (zèbre, 11), (hamster, 6), (hyène,9),
(licorne, 2), (babouin, 12), (piranha,55), (saumon, 82), (bar,44),
(anguille, 43), (gorille,98), (tigre,76), (requin, 56), (escargot,45).
```

On suppose que l'on peut placer 2 enregistrements par bloc.

— On veut construire un index non-dense sur le premier attribut. Que faut-il faire au préalable ? Donnez les différents niveaux de l'index.
— Construire un arbre B sur le second attribut, en supposant 2 enregistrements et trois pointeurs par bloc *au maximum*.

— On dispose des deux index ci-dessus. Quel est le nombre d'entrées/sorties *dans le pire des cas* pour la recherche des animaux dont le nom commence par un "g", et pour la recherche des animaux dont l'identifiant est compris entre 40 et 50 (prendre en compte les accès à l'index *et* au fichier).
— On veut trier ce fichier (tel qu'il est donné dans l'énoncé) avec seulement 3 blocs, toujours en supposant qu'on peut placer deux enregistrements par bloc. Décrivez le déroulement de l'algorithme de tri-fusion, et donnez le nombre total d'entrées/sorties, sans compter l'écriture finale du fichier trié.

12.3.2 Jointures et optimisation (9 points)

On considère trois relations $R(\mathbf{a}, b, d)$, $S(\mathbf{c}, d, e)$ et $T(\mathbf{e}, f)$ dont les clés primaires sont respectivement a, c et e. R contient 200 000 enregistrements, S 20 enregistrements et T 500 enregistrements. Des index sont créés sur les clés primaires, et on suppose que pour chaque relation, y compris celles qui sont calculées par une jointure, on stocke 10 enregistrements par bloc.

On suppose que tous les index sont toujours en mémoire principale (donc pas d'entrée/sortie pour les accès aux index). On dispose de 20 blocs en mémoire pour traiter les jointures. Les jointures se font sur les attributs de même nom.

— Quel est le nombre maximal d'enregistrements dans $S \bowtie T$ (indiquez également la condition pour que ce nombre maximal soit atteint) ? Quelle est la clé de la relation obtenue ?
— Mêmes questions pour $R \bowtie S \bowtie T$.
— Décrire le fonctionnement de l'algorithme par boucles imbriquées pour calculer $S \bowtie T$, en exploitant au mieux la mémoire disponible. Indiquez le coût en entrées/sorties pour cet algorithme.
— Même question l'algorithme par boucles imbriquées *indexées*.
— Décrire un plan d'exécution pour calculer $R \bowtie S \bowtie T$et évaluer son coût (nombre de blocs lus).
— Une application a inséré des enregistrements dans S qui en contient maintenant 5 000. Le SGBD gère un histogramme qui indique que la sélectivité de l'attribut d est 5 (autrement dit, une sélection sur R ou S pour une valeur de d ramène 5% des enregistrements). Quelle taille peut-on estimer pour $R \bowtie S$?
— Je n'ai toujours que 20 blocs en mémoire. Décrire l'algorithme de jointure par hachage pour $R \bowtie S$ et évaluer son coût.

12.3.3 Concurrence (6 points)

On considère le système d'information d'un institut de sondage, avec les tables relationnelles suivantes (les attributs ou combinaisons d'attributs qui forment une clé unique sont **en gras**).

— Personne (**numPers**, nom, sexe, numCat)
— Question (**numQ**, description)
— Avis (**numQ, numPers**, reponse)

L'exécution suivante est reçue par le système de l'institut de sondage :

$$H : r_1[x]r_2[y]w_1[x]r_3[y]r_2[x]w_3[y]r_2[z]C_1r_3[z]w_2[z]C_2w_3[z]C_3$$

Répondez aux questions suivantes sur H.

— Parmi les programmes qui s'exécutent dans le système, il y a `ModifierAvis(nomPers, numQuestion, nouvelle_réponse)`, qui modifie la réponse donnée par la personne `nomPers` à la question `numQuestion` en `nouvelle\_réponse`
Si les enregistrements de H sont des nuplets des relations de la base de données, montrez (en justifiant votre réponse) quelles transactions de H pourraient provenir de `ModifierAvis`.

— Vérifiez si H est sérialisable en identifiant les conflits et en construisant le graphe de sérialisation.
— Montrer qu'il existe des lectures sales, et expliquez les conséquences possibles.
— Quelle est l'exécution obtenue par verrouillage à deux phases à partir de H ?
— (**Question bonus, pour 2 points**). Quelle est l'exécution obtenue avec l'algorithme de concurrence par versionnement ? On suppose que toutes les transactions débutent au même moment.

CHAPITRE 13

Indices and tables

— genindex
— modindex
— search