Cours de bases de données

Philippe Rigaux

13 juin 2001

Table des matières

Chapitre 1

Introduction

Sommaire

Ce cours s'adresse aux étudiants du cycle A du CNAM et a pour objectif l'étude des principes des SGBD relationnels et la mise en pratique de ces principes. Le contenu du cours est essentiellement le suivant :

1. **Conception d'un schéma relationnel**. Il s'agit de savoir définir un schéma relationnel complet et correct, comprenant des tables, des contraintes, des vues.

2. **Langages d'interrogation et de manipulation**. L'accent est mis sur SQL et ses fondements, et sur l'intégration de SQL avec un langage de programmation comme le C.

De plus, le cours comprend une introduction aux problèmes de concurrence d'accès, dont la connaissance est nécessaire aux développeurs d'applications basées sur des SGBD. Des travaux pratiques avec le SGBD ORACLE permettent de mettre en oeuvre les techniques étudiées en cours.

L'accent est donc plutôt mis sur les notions de base (qu'est-ce qu'un SGBD, qu'une base de données, qu'un langage d'interrogation) et leur application pratique. Il demandé d'avoir acquis à la fin du cours les connaissances nécessaires à l'utilisation d'un SGBD par un informaticien non-spécialiste. : création d'un schéma, insertion, mise-à-jour, destruction, interrogation de données, et comprehension des mécanismes de concurrence intervenant dans la gestion d'un SGBD. En revanche, tout ce qui relève de la compréhension des mécanismes internes d'un SGBD (représentation physique, évaluation de requêtes) ou des fondements théoriques du modèle relationnel n'est pas abordé ici.

Ce document est un support de cours : il ne prétend certainement pas être exhaustif ni traiter en détail tous les sujets abordés. L'assistance au cours proprement dit, ainsi qu'aux travaux dirigés et aux travaux pratiques est fortement recommandée. Il existe de plus un site WEB qui donne des renseignements complémentaires, les horaires des cours, les solutions de certains exercices, etc. Voici l'adresse :

`http://sikkim.cnam.fr/~rigaux/bdpi.html`

Pour ceux qui veulent en savoir plus, il existe une riche bibliographie dont voici quelques éléments recommandables :

Ouvrages en français

1. Carrez C., *Des Structures aux Bases de Données*, Masson

2. Gardarin G., *Maîtriser les Bases de Données: modèles et langages*, Eyrolles

3. Marcenac, P., *SGBD relationnels, Optimisation des performances*, Eyrolles.

Ouvrages en anglais

1. Melton J. et A.R. Simon, *Understanding SQL, A Complete Guide*, Morgan Kaufmann, 1993.

2. Ullman J.D., *Principles of Database and Knowledge-Base Systems*, 2 volumes, Computer Science Press

3. Date C.J., *An Introduction to Database Systems*, Addison-Wesley

Le premier chapitre (correspondant au premier cours) est une (rapide) présentation de tous les thèmes présentés en détails dans ce cours. On peut le lire comme une mise en perspective générale de l'ensemble de l'enseignement.

Chapitre 2

Présentation générale

Sommaire

Ce chapitre présente un panorama général de la problématique des bases de données.

2.1 Données, Bases de données et SGBD

La première chose à faire est d'établir quelques points de terminologie. Qu'est-ce qu'une donnée ? C'est une information quelconque comme, par exemple : voici une personne, elle s'appelle Jean. C'est aussi une relation entre des informations : Jean *enseigne* les bases de données. Des relations de ce genre définissent des *structures*. Une base de données est un ensemble, en général volumineux, de telles informations, avec une caractéristique essentielle : on souhaite les mémoriser de manière permanente. D'où la définition :

Definition 2.1 *Une Base de données est un* gros ensemble *d'informations* structurées *mémorisées sur un support* permanent.

On peut remarquer qu'une organisation consistant en un (ou plusieurs) fichier(s) stockés sur mémoire secondaire est conforme à cette définition. Un ensemble de fichiers ne présentant qu'une complexité assez faible, il n'y aurait pas là matière à longue dissertation. Malheureusement l'utilisation directe de fichiers soulève de très gros problèmes :

1. *Lourdeur d'accès aux données*. En pratique, pour chaque accès, même le plus simples, il faudrait écrire un programme.
2. *Manque de sécurité*. Si tout programmeur peut accéder directement aux fichiers, il est impossible de garantir la sécurité et l'intégrité des données.
3. *Pas de contrôle de concurrence*. Dans un environnement où plusieurs utilisateurs accèdent aux même fichiers, des problèmes de concurrence d'accès se posent.

D'où le recours à un logiciel chargé de gérer les fichiers constituant une base de données, de prendre en charge les fonctionnalités de protection et de sécurité et de fournir les différents types d'interface nécessaires à l'accès aux données. Ce logiciel (le SGBD) est très complexe et fournit le sujet principal de

ce cours. En particulier, une des tâches principales du SGBD est de masquer à l'utilisateur les détails complexes et fastidieux liés à la gestion de fichiers. D'où la définition.

Definition 2.2 *Un Système de Gestion de Bases de Données (SGBD) est un logiciel de* haut niveau *qui permet de manipuler les informations stockées dans une base de données.*

La complexité d'un SGBD est essentiellement issue de la diversité des techniques mises en oeuvre, de la multiplicité des composants intervenant dans son architecture, et des différents types d'utilisateurs (administrateurs, programmeurs, non informaticiens, ...) qui sont confrontés, à différents niveaux, au système. Voici quelques exemples illustrant tous les cas de figure qu'il faudrait envisager dans un cours exhaustif :

- Les modèles de données : entité-relation, réseau, hiérarchique, relationnel, orienté-objet, modèles sémantiques.
- Les langages de requêtes : fondements théoriques (logiques du premier ordre, du point fixe, algèbres diverses) et les langages comme SQL, SQL3, Datalog, OQL, etc.
- Les techniques de stockage : sur disque (optique), sur bande.
- L'organisation des fichiers : index, arbre-B, hachage, ...
- L'architecture : centralisé, distribué, sur d'autres bases accessibles par réseau.
- Les techniques d'évaluation et d'optimisation de requêtes.
- La concurrence d'accès et les techniques de reprise sur pane.

Pour mettre un peu d'ordre dans tout cela, on peut se raccrocher à une architecture standard conforme à la plus grande partie des SGBD existant, et offrant l'avantage de bien illustrer les principales caractéristiques d'un SGBD.

Cette architecture distingue trois niveaux correspondant d'une part à trois représentations équivalentes de l'information, d'autre part aux champs d'interventions respectifs des principaux acteurs. Pour ces derniers, nous utiliserons la terminologie suivante :

- *Utilisateur naïf :* du non spécialiste des SGBD au non informaticien.
- *Concepteur et programmeur d'application* : à partir des besoins des différents utilisateurs, écrit l'application pour des utilisateurs "naïfs".
- *Utilisateur expert :* informaticien connaissant le fonctionnement interne d'un SGBD et chargé d'administrer la base.

Chaque niveau du SGBD remplit (réalise) un certain nombre de fonctions :

- *Niveau physiques* : gestion sur mémoire secondaire (fichiers) des données, du schéma, des index ; Partage de données et gestion de la concurrence d'accès ; Reprise sur pannes (fiabilité) ; Distribution des données et interopérabilité (accès aux réseaux).
- *Niveau logique* : Définition de la structure de données : Langage de Description de Données (LDD) ; Consultation et Mise à Jour des données : Langages de Requêtes (LR) et Langage de Manipulation de Données (LMD) ; Gestion de la confidentialité (sécurité) ; Maintien de l'intégrité ;
- *Niveau externe* : Vues ; Environnement de programmation (intégration avec un langage de programmation) ; Interfaces conviviales et Langages de 4e Génération (L4G) ; Outils d'aides (e.g. conception de schémas) ; Outils de saisie, d'impression d'états.

En résumé, un SGBD est destiné à gèrer un gros volume d'informations, persistantes (années) et fiables (protection sur pannes), partageables entre plusieurs utilisateurs et/ou programmes et manipulées indépendamment de leur représentation physique.

2.2 Que doit-on savoir pour utiliser un SGBD ?

L'utilisation d'un SGBD suppose de comprendre (et donc de savoir utiliser) les fonctionnalités suivantes :

1. *Définition du schéma de données* en utilisant les *modèles de données* du SGBD.
2. *Opérations sur les données* : recherche, mises-à-jour, etc.
3. *Partager les données* entre plusieurs utilisateurs. (Mécanisme de *transaction*).
4. *Optimiser les performances*, par le réglage de l'organisation physique des données. Cet aspect relève plutôt de l'administration et ne sera évoqué que dans l'introduction.

Reprenons dans l'ordre ces différents points.

2.2.1 Définition du schéma de données

Un schéma est simplement la description des données contenues dans la base. Cette description est conforme à un *modèle de données* qui propose des outils de description (structures, contraintes et opérations). En fait, dans un SGBD, il existe plusieurs modèles plus ou moins abstraits des mêmes objets, e.g. :

- Le modèle conceptuel : la description du système d'information
- Le modèle logique : interface avec le SGBD
- Le modèle physique : fichiers.

Ces différents modèles correspondent aux niveaux dans l'architecture d'un SGBD. Prenons l'exemple du modèle conceptuel le plus courant : le modèle Entité/Association. C'est essentiellement une description très abstraite qui présente les avantages suivants :

- l'analyse du monde réel
- la conception du système d'information
- la communication entre différents acteurs de l'entreprise

En revanche, il ne propose pas d'opérations. Or définir des structures sans disposer d'opérations pour agir sur les données stockées dans ces structures ne présente pas d'intérêt pratique pour un SGBD. D'où, à un niveau inférieur, des modèles dits "logiques" qui proposent :

1. Un *langage de définition de données (LDD)* pour décrire la structure, incluant des contraintes.
2. Un *langage de manipulation de données (LMD)* pour appliquer des opérations aux données.

Ces langages sont *abstraits* : le LDD est indépendant de la représentation physique des données, et le LMD est indépendant de l'implantation des opérations. On peut citer une troisième caractéristique : oute les structures et les opérations, un modèle logique doit permettre d'exprimer des *contraintes d'intégrité* sur les données. Exemple :

```
nom character 15, not null;
âge integer between 0 and 120;
débit = crédit;
...
```

Bien entendu, le SGBD doit être capable de garantir le respect de ces contraintes.

Quand on conçoit une application pour une BD, on tient compte (plus ou moins consciemment) de cette architecture en plusieurs niveaux. Typiquement : (1) On décide la structure logique, (2) on décide la structure physique, (3) on écrit les programmes d'application en utilisant la structure logique, enfin (4) Le SGBD se charge de transcrire les commandes du LMD en instructions appropriées appliquées à la représentation physique.

Cette aproche offre de très grands avantages qu'il est important de souligner. Tout d'abord elle ouvre l'utilisation des SGBD à de utilisateurs non-experts : les langages proposés par les modèles logiques sont plus simples, et donc plus accessibles, que les outils de gestion de fichiers. Ensuite, on obtient une caractéristique essentielle : *l'indépendance physique*. On peut modifier l'implantation physique sans modifier les programmes d'application. Un concept voisin est celui *d'indépendance logique* : on peut modifier les programmes d'application sans toucher à l'implantation.

Enfin le SGBD décharge l'utilisateur, et en grande partie l'administrateur, de la lourde tâche de contrôler la sécurité et l'intégrité des données.

2.2.2 Les opérations sur les données

Il existe 4 opérations classiques (ou *requêtes*) :

1. La *création* (ou *insertion*).
2. La *modification* (ou *mise-à-jour*).
3. La *destruction*.
4. La *recherche*.

Ces opérations correspondent à des commandes du LMD. La plus complexe est la *recherche* en raison de la variété des critères.

Pour l'utilisateur, une bonne requête a les caractéristiques suivantes. Tout d'abord elle s'exprime facilement : l'idéal serait de pouvoir utiliser le langage naturel, mais celui-ci présente trop d'ambiguités. Ensuite le langage ne devrait pas demander d'expertise technique (syntaxe compliquée, structures de données, implantation particulière ...). Il est également souhaitable de ne pas attendre trop longtemps (à charge pour le SGBD de fournir des performances acceptables). Enfin , et peut-être surtout, la réponse doit être fiable.

Une bonne partie du travail sur les SGBD consiste à satisfaire ces besoins. Le résultat est ce que l'on appelle un *langage de requêtes*, et constitue à la fois un sujet majeur d'étude et une caractéristique essentielle de chaque SGBD. Le langage le plus répandu à l'heure actuelle est SQL.

2.2.3 Optimisation

L'optimisation (d'une requête) s'appuie sur *l'organisation physique des données*. Les principaux types d'organisation sont les fichiers séquentiels, les index (denses. secondaires, arbres B) et le regroupement des données par hachage.

Un module particulier du SGBD, *l'optimiseur*, tient compte de cette organisation et des caractéristiques de la requête pour choisir le meilleur séquencement des opérations.

2.2.4 Concurrence d'accès

Plusieurs utilisateurs doivent pouvoir accéder en même temps aux mêmes données. Le SGBD doit savoir :

– Gérer les conflits si les deux font des mises-à-jour.

– Offrir un mécanisme de retour en arrière si on décide d'annuler des modifications en cours.

– Donner une image cohérente des données si l'un fait des requêtes et l'autre des mises-à-jour.

Le but : éviter les blocages, tout en empéchant des modifications anarchiques.

2.3 Le plan du cours

Le cours comprend trois parties consacrées successivement à la conception d'une base de données relationnelles, aux langages de requêtes relationnels, enfin à la pratique d'un SGBD relationnel.

Conception d'un schéma relationnel

Le cours présente d'abord la technique classique de conception à l'aide du *modèle entité/association*, suivie de la transcription du schéma obtenu dans le modèle relationnel. On obtient un moyen simple et courant de créer des schémas ayant de bonnes propriétés. Les concepts de 'bon' et de 'mauvais' schémas sont ensuite revus plus formellement avec la théorie de la *normalisation*.

Langages relationnels

Les langages d'interrogation et de manipulation de données suivants sont présentés : *l'algèbre relationnelle* qui fournit un petit ensemble d'opérateurs permettant d'exprimer des requêtes complexes et *le langage SQL*, norme SQL2.

Pratique d'un SGBD relationnel

Cette partie reprend et étend les sujets précédents et développe leur mise en pratique dans l'environnement d'un SGBD relationnel. Elle comprend :

1. Une revue complète du langage de définition de données SQL2 pour la création de *schémas relationnels*, incluant l'expression de contraintes, les vues et les *triggers*.
2. Une introduction au développement d'applications avec SQL.
3. Une introduction à la *concurrence d'accès* et à ses implications pratiques.
4. Une série de travaux pratiques et d'exercices avec le SGBD Oracle.

Première partie

Modèles et langages

Chapitre 3

Le modèle Entité/Association

Sommaire

Ce chapitre présente le modèle Entité/Association (E/A) qui est utilisé à peu près universellement pour la *conception* de bases de données (relationnelles principalement). La conception d'un schéma correct est essentielle pour le développement d'une application viable. Dans la mesure où la base de données est le fondement de tout le système, une erreur pendant sa conception est difficilement récupérable par la suite. Le modèle E/A a pour caractéristiques d'être simple et suffisamment puissant pour représenter des structures relationnelles. Surtout, il repose sur une représentation graphique qui facilite considérablement sa compréhension.

Le modèle E/A souffre également de nombreuses insuffisances : la principale est de ne proposer que des *structures*. Il n'existe pas d'opération permettant de manipuler les données, et pas (ou peu) de moyen d'exprimer des contraintes. Un autre inconvénient du modèle E/A est de mener à certaines ambiguités pour des schémas complexes.

La présentation qui suit est délibérement axée sur l'utilité du modèle E/A dans le cadre de la conception d'une base de données. Ajoutons qu'il ne s'agit pas de *concevoir* un schéma E/A (voir un cours sur les systèmes d'information), mais d'être capable de le comprendre et de l'interpréter. Dans tout ce chapitre nous prenons l'exemple d'une base de données décrivant des films, avec leur metteur en scène et leurs acteurs, ainsi que les cinémas où passent ces films. Nous supposerons également que cette base de données est accessible sur le Web et que des internautes peuvent noter les films qu'ils ont vus.

3.1 Principes généraux

La méthode permet de distinguer les *entités* qui constituent la base de données, et les *associations* entre ces entités. Ces concepts permettent de donner une structure à la base, ce qui s'avère indispensable. Nous commençons par montrer les problèmes qui surviennent si on traite une base relationnelle comme un simple fichier texte, sans se soucier de lui donner une structure correcte.

3.1.1 Bons et mauvais schémas

Considérons une table *FilmSimple* stockant des films avec quelques informations de base, dont le metteur en scène. Voici une représentation de cette table.

titre	année	nomMES	prénomMES	annéeNaiss
Alien	1979	Scott	Ridley	1943
Vertigo	1958	Hitchcock	Alfred	1899
Psychose	1960	Hitchcock	Alfred	1899
Kagemusha	1980	Kurosawa	Akira	1910
Volte-face	1997	Woo	John	1946
Pulp Fiction	1995	Tarantino	Quentin	1963
Titanic	1997	Cameron	James	1954
Sacrifice	1986	Tarkovski	Andrei	1932

Même pour une information aussi simple, il est facile d'énumérer tout un ensemble de problèmes potentiels. Tous ou presque découlent d'un grave défaut de la table ci-dessus : *il est possible de représenter la même information plusieurs fois*.

Anomalies lors d'une insertion

Rien n'empêche de représenter plusieurs fois le même film. Pire : il est possible d'insérer plusieurs fois le film *Vertigo* en le décrivant à chaque fois de manière différente, par exemple en lui attribuant une fois comme réalisateur Alfred Hitchcock, puis une autre fois John Woo, etc.

Une bonne question consiste d'ailleurs à se demander ce qui distingue deux films l'un de l'autre, et à quel moment on peut dire que la même information a été répétée. Peut-il y avoir deux films différents avec le même titre par exemple ? Si la réponse est non, alors on devrait pouvoir assurer qu'il n'y a pas deux lignes dans la table avec la même valeur pour l'attribut `titre`. Si la réponse est oui, il reste à déterminer quel est l'ensemble des attributs qui permet de caractériser de manière unique un film.

Anomalies lors d'une modification

La redondance d'information entraîne également des anomalies de mise à jour. Supposons que l'on modifie l'année de naissance de Hitchcock pour la ligne *Vertigo* et pas pour la ligne *Psychose*. On se retrouve alors avec des informations incohérentes.

Les mêmes questions que précédemment se posent d'ailleurs. Jusqu'à quel point peut-on dire qu'il n'y a qu'un seul réalisateur nommé Hitchcock, et qu'il ne doit donc y avoir qu'une seule année de naissance pour un réalisateur de ce nom ?

Anomalies lors d'une destruction

On ne peut pas supprimer un film sans supprimer du même coup son metteur en scène. Si on souhaite, par exemple, ne plus voir le film *Titanic* figurer dans la base de données, on va effacer du même coup les informations sur James Cameron.

3.1.2 La bonne méthode

Une bonne méthode évitant les anomalies ci-dessus consiste à ;

1. être capable de représenter individuellement les films et les réalisateurs, de manière à ce qu'une action sur l'un n'entraîne pas systématiquement une action sur l'autre ;
2. définir une méthode *d'identification* d'un film ou d'un réalisateur, qui permette d'assurer que la même information est représentée une seule fois ;
3. préserver le lien entre les films et les réalisateurs, mais sans introduire de redondance.

Commençons par les deux premières étapes. On va d'abord distinguer la table des films et la table des réalisateurs. Ensuite on décide que deux films ne peuvent avoir le même titre, mais que deux réalisateurs peuvent avoir le même nom. Afin d'avoir un moyen d'identifier les réalisateurs, on va leur attribuer un numéro, désigné par `id`. On obtient le résultat suivant, les identifiants (ou *clés*) étant en gras.

titre	année
Alien	1979
Vertigo	1958
Psychose	1960
Kagemusha	1980
Volte-face	1997
Pulp Fiction	1995
Titanic	1997
Sacrifice	1986

La table des films

id	nomMES	prénomMES	annéeNaiss
1	Scott	Ridley	1943
2	Hitchcock	Alfred	1899
3	Kurosawa	Akira	1910
4	Woo	John	1946
5	Tarantino	Quentin	1963
6	Cameron	James	1954
7	Tarkovski	Andrei	1932

La table des réalisateurs

Premier progrès : il n'y a maintenant plus de redondance dans la base de données. Le réalisateur Hitchcock, par exemple, n'apparaît plus qu'une seule fois, ce qui élimine les anomalies de mise à jour évoquées précédemment.

Il reste à représenter le lien entre les films et les metteurs en scène, sans introduire de redondance. Maintenant que nous avons défini les identifiants, il existe un moyen simple pour indiquer quel est le metteur en scène qui a réalisé un film : associer l'identifiant du metteur en scène au film. On ajoute un attribut `idMES` dans la table *Film*, et on obtient la représentation suivante.

titre	année	*idMES*
Alien	1979	1
Vertigo	1958	2
Psychose	1960	2
Kagemusha	1980	3
Volte-face	1997	4
Pulp Fiction	1995	5
Titanic	1997	6
Sacrifice	1986	7

La table des films

id	nomMES	prénomMES	annéeNaiss
1	Scott	Ridley	1943
2	Hitchcock	Alfred	1899
3	Kurosawa	Akira	1910
4	Woo	John	1946
5	Tarantino	Quentin	1963
6	Cameron	James	1954
7	Tarkovski	Andrei	1932

La table des réalisateurs

Cette représentation est correcte. La redondance est réduite au minimum puisque seule la clé identifiant un metteur en scène a été déplacée dans une autre table (on parle de *clé étrangère*). On peut vérifier que toutes les anomalies que nous avons citées ont disparu.

Anomalie d'insertion. Maintenant que l'on sait quelles sont les caractéristiques qui identifient un film, il est possible de déterminer au moment d'une insertion si elle va introduire ou non une redondance. Si c'est le cas on doit interdire cette insertion.

Anomalie de mise à jour. Il n'y a plus de redondance, donc toute mise à jour affecte l'unique instance de la donnée à modifier.

Anomalie de destruction. On peut détruire un film sans affecter les informations sur le réalisateur.

Ce gain dans la qualité du schéma n'a pas pour contrepartie une perte d'information. Il est en effet facile de voir que l'information initiale (autrement dit, avant décomposition) peut être reconstituée intégralement. En prenant un film, on obtient l'identité de son metteur en scène, et cette identité permet de trouver *l'unique* ligne dans la table des réalisateurs qui contient toutes les informations sur ce metteur en scène. Ce processus de reconstruction de l'information, dispersée dans plusieurs tables, peut s'exprimer avec SQL.

La modélisation avec un graphique Entité/Association offre une méthode simple pour arriver au résultat ci-dessus, et ce même dans des cas beaucoup plus complexes.

3.2 Le modèle E/A : Présentation informelle

Un schéma E/A décrit l'application visée, c'est-à-dire une *abstraction* d'un domaine d'étude, pertinente relativement aux objectifs visés. Rappelons qu'une abstraction consiste à choisir certains aspects de la réalité perçue (et donc à éliminer les autres). Cette sélection se fait en fonction de certains *besoins* qui doivent être précisément définis.

Par exemple, pour notre base de données *Films*, on n'a pas besoin de stocker dans la base de données l'intégralité des informations relatives à un internaute, ou à un film. Seules comptent celles qui sont importantes pour l'application. Voici le schéma décrivant cete base de données *Films* (figure 3.1). Sans entrer dans les détails pour l'instant, on distingue

1. des *entités*, représentées par des rectangles, ici *Film*, *Artiste*, *Internaute* et *Pays* ;
2. des *associations entre entités* représentées par des liens entre ces rectangles. Ici on a représenté par exemple le fait qu'un artiste *joue* dans des films, qu'un internaute *note* des films, etc.

Chaque entité est caractérisée par un ensemble d'attributs, parmi lesquels un ou plusieurs forment l'identifiant unique (en gras). Comme nous l'avons exposé précédemment, il est essentiel de dire ce qui caractérise de manière unique une entité, de manière à éviter la redondance d'information.

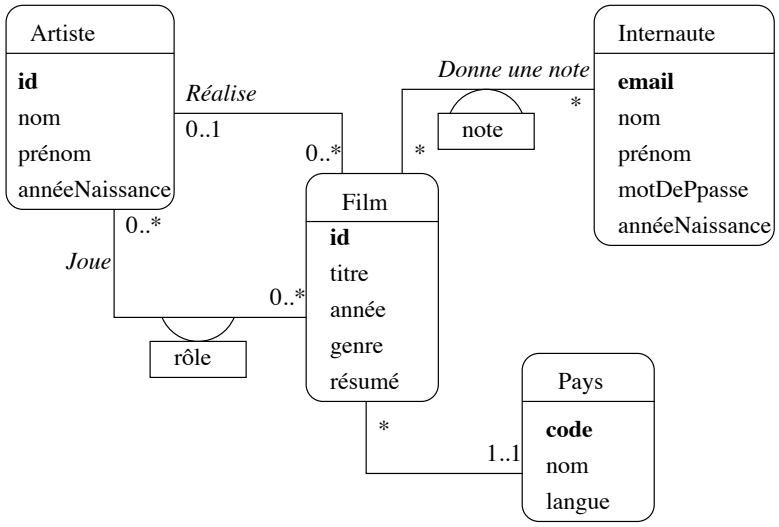


FIG. 3.1 – *Le schéma de la base de données Films*

Les associations sont caractérisées par des *cardinalités*. La notation « 0..* » sur le lien *Réalise*, du côté de l'entité *Film*, signifie qu'un artiste peut réaliser plusieurs films, ou aucun. La notation « 0..1 » du côté *Artiste* signifie en revanche qu'un film ne peut être réalisé que par au plus un artiste. En revanche dans l'association *Donne une note*, un internaute peut noter plusieurs films, et un film peut être noté par plusieurs internautes, ce qui justifie l'a présence de « 0..* » aux deux extrêmités de l'association.

Le choix des cardinalités est *essentiel*. Ce choix est aussi parfois discutable, et constitue donc l'aspect le plus délicat de la modélisation. Reprenons l'exemple de l'association *Réalise*. En indiquant qu'un film est réalisé par *un seul* metteur en scène, on s'interdit les – rares – situations où un film est réalisé par plusieurs personnes. Il ne sera donc pas possible de représenter dans la base de données une telle situation. Tout est ici question de choix et de compromis : est-on prêt en l'occurrence à accepter une structure plus complexe (avec « 0..* » de chaque côté) pour l'association *Réalise*, pour prendre en compte un nombre minime de cas ?

Les cardinalités sont notées par deux chiffres. Le chiffre de droite est la *cardinalité maximale*, qui vaut en général 1 ou *. Le chiffre de gauche est la cardinalité minimale. Par exemple la notation « 0..1 » entre

Artiste et *Film* indique qu'on s'autorise à ne pas connaître le metteur en scène d'un film. Attention : cela ne signifie pas que ce metteur en scène n'existe pas. Une base de données, telle qu'elle est décrite par un schéma E/A, n'est qu'une vision partielle de la réalité. On ne doit surtout pas rechercher une représentation exhaustive, mais s'assurer de la prise en compte des besoins de l'application.

La notation « 1..1 » entre *Film* et *Pays* indique au contraire que l'on doit toujours connaître le pays producteur d'un film. On devra donc interdire le stockage dans la base d'un film sans son pays.

Les cardinalités minimales (également appelées « contraintes de participation ») sont moins importantes que les cardinalités maximales, car elles ont un impact moindre sur la structure de la base de données et peuvent plus facilement être remises en cause après coup. Il faut bien être conscient de plus qu'elles ne représentent qu'un choix de conception, souvent discutable. Dans la notation UML que nous présentons ici, il existe des notations abrégées qui donnent des valeurs implicites aux cardinalités minimales :

1. La notation « * » est équivalente à « 0..* » ;
2. la notation « 1 » est équivalente à « 1..1 ».

Outre les propriétés déjà évoquées (simplicité, clarté de lecture), évidentes sur ce schéma, on peut noter aussi que la modélisation conceptuelle est totalement indépendante de tout choix d'implantation. Le schéma de la figure 3.1 ne spécifie aucun système en particulier. Il n'est pas non plus question de type ou de structure de données, d'algorithme, de langage, etc. En principe, il s'agit donc de la partie la plus stable d'une application. Le fait de se débarrasser à ce stade de la plupart des considérations techniques permet de se concentrer sur l'essentiel : que veut-on stocker dans la base ?

Une des principales difficultés dans le maniement des schémas E/A est que la qualité du résultat ne peut s'évaluer que par rapport à une demande qui est souvent floue et incomplète. Il est donc souvent difficile de valider (en fonction de quels critères ?) le résultat. Peut-on affirmer par exemple que :

1. *toutes* les informations nécessaires sont représentées ;
2. qu'un film ne sera *jamais* réalisé par plus d'un artiste ;
3. qu'il n'y aura *jamais* deux films avec le même titre.

Il faut faire des choix, en connaissance de cause, en sachant toutefois qu'il est toujours possible de faire évoluer une base de données, quand cette évolution n'implique pas de restructuration trop importante. Pour reprendre les exemples ci-dessus, il est facile d'ajouter des informations pour décrire un film ou un internaute ; il serait beaucoup plus difficile de modifier la base pour qu'un film passe de un, et un seul, réalisateur, à plusieurs. Quant à changer la clé de *Film*, c'est une des évolutions les plus complexes à réaliser. Les cardinalités et le choix des clés font vraiment partie des des aspects décisifs des choix de conception.

3.3 Le modèle

Le modèle E/A, conçu en 1976, est à la base de la plupart des méthodes de conception. La syntaxe employée ici est celle de la méthode UML, reprise à peu près à l'identique de celle de la méthode OMT. Il existe beaucoup d'autres notations, dont celle de la méthode MERISE principalement utilisée en France. Ces notations sont globalement équivalentes. Dans tous les cas la conception repose sur deux concepts complémentaires, *entité* et *association*.

3.3.1 Entités, attributs et identifiants

Il est difficile de donner une définition très précise des entités. Les points essentiels sont résumés ci-dessous.

Definition 3.1 (Entité) *On désigne par* entité *tout objet* identifiable *et* pertinent *pour l'application*.

Comme nous l'avons vu précédemment, la notion *d'identité* est primordiale. C'est elle qui permet de distinguer les entités les unes des autres, et donc de dire qu'une information est redondante ou qu'elle ne l'est pas. Il est indispensable de prévoir un moyen technique pour pouvoir effectuer cette distinction entre entités au niveau de la base de données : on parle *d'identifiant* ou de *clé*.

La pertinence est également essentielle : on ne doit prendre en compte que les informations nécessaires pour satisfaire les besoins. Par exemple :

1. le film *Impitoyable* ;
2. l'acteur *Clint Eastwood* ;

sont des entités pour la base *Films*.

La première étape d'une conception consiste à identifier les entités utiles. On peut souvent le faire en considérant quelques cas particuliers. La deuxième est de regrouper les entités en ensembles : en général on ne s'intéresse pas à un individu particulier mais à des groupes. Par exemple il est clair que les films et les acteurs sont des ensembles distincts d'entités. Qu'en est-il de l'ensemble des réalisateurs et de l'ensemble des acteurs ? Doit-on les distinguer ou les assembler ? Il est certainement préférable de les assembler, puisque des acteurs peuvent aussi être réalisateurs.

Attributs

Les entités sont caractérisées par des *propriétés* : le titre (du film), le nom (de l'acteur), sa date de naissance, l'adresse, etc. Ces propriétés sont dénotées *attributs* dans la terminologie du modèle E/A. Le choix des attributs relève de la même démarche d'abstraction qui a dicté la sélection des entités : il n'est pas question de donner exhaustivement toutes les propriétés d'une entité. On ne garde que celles utiles pour l'application.

Un attribut est désigné par un *nom* et prend ses valeurs dans un domaine énumérable comme les entiers, les chaînes de caractères, les dates, etc. On peut considérer un nom d'atribut A comme une fonction définie sur un ensemble d'entités E et prenant ses valeurs dans un domaine D. On note alors $A(e)$ la valeur de l'attribut A pour une entité $e \in E$.

Considérons par exemple un ensemble de films $\{f_1, f_2, \ldots f_n\}$ et les attributs *titre* et *année*. Si f_1 est le film *Impitoyable*, tourné par Clint Eastwood en 1992, on aura :

$$\textit{titre}\ (f_1) = \text{Impitoyable}\ ;\ \textit{année}\ (f_1) = 1992$$

Il est très important de noter que selon cette définition un attribut prend une valeur et une seule. On dit que les attributs sont *atomiques*. Il s'agit d'une restriction importante puisqu'on ne sait pas, par exemple, définir un attribut *téléphones* d'une entité *Personne*, prenant pour valeur les numéros de téléphone d'une personne. Certaines méthodes admettent (plus ou moins clairement) l'introduction de constructions plus complexes :

1. les *attributs multivalués* sont constitués d'un *ensemble* de valeurs prises dans un même domaine ; une telle construction permet de résoudre le problème des numéros de téléphones multiples ;
2. les *attributs composés* sont constitués par agrégation d'autres atributs ; un attribut `adresse` peut par exemple être décrit comme l'agrégation d'un code postal, d'un numéro de rue, d'un nom de rue et d'un nom de ville.

Nous nous en tiendrons pour l'instant aux attributs atomiques qui, au moins dans le contexte d'une modélisation orientée vers un SGBD relationnel, sont suffisants.

Types d'entités

Il est maintenant possible de décrire un peu plus précisément les entités par leur *type*.

Definition 3.2 (Type d'entité) *Le type d'une entité est composé des éléments suivants :*

1. son nom ;

2. *la liste de ses attributs avec, – optionnellement – le domaine où l'attribut prend ses valeurs : les entiers, les chaînes de caractères ;*

3. *l'indication du (ou des) attribut(s) permettant d'identifier l'entité : ils constituent la* clé.

On dit qu'un entité e est une *instance* de son type E. Enfin, un ensemble d'entités $\{e_1, e_2, \ldots e_n\}$ instance d'un même type E est une *extension* de E.

Il reste à définir plus précisément la notion de clé.

Definition 3.3 (Clé) *Soit E un type d'entité et A l'ensemble des attributs de E. Une* clé *de E est un sous-ensemble* minimal *de A permettant d'identifier de manière unique une entité parmi n'importe quelle extension de E.*

Prenons quelques exemples pour illustrer cette définition. Un internaute est caractérisé par plusieurs attributs : son email, son nom, son prénom, la région où il habite. L'email constitue une clé naturelle puisqu'on ne trouve pas, en principe, deux internautes ayant la même adresse électronique. En revanche l'identification par le nom seul paraît impossible puisqu'on constitureait facilement un ensemble contenant deux internautes avec le même nom. On pourrait penser à utiliser la paire `(nom,prénom)`, mais il faut utiliser avec modération l'utilisation d'identifiants composés de plusieurs attributs, quoique possible, peut poser des problèmes de performance et complique les manipulations par SQL.

Il est possible d'avoir plusieurs clés pour un même ensemble d'entités. Dans ce cas on en choisit une comme *clé primaire*, et les autres comme *clés secondaires*. Le choix de la clé (primaire) est déterminant pour la qualité du schéma de la base de données. Les caractéristiques d'une bonne clé primaire sont les suivantes :

– sa valeur est connue pour toute entité ;

– on ne doit jamais avoir besoin de la modifier ;

– enfin, pour des raisons de performance, sa taille de stockage doit être la plus petite possible.

Il n'est pas toujours évident de trouver un ensemble d'attributs satisfaisant ces propriétés. Considérons l'exemple des films. Le choix du titre pour identifier un film serait incorrect puisqu'on aura affaire un jour ou l'autre à deux films ayant le même titre. Même en combinant le titre avec un autre attribut (par exemple l'année), il est difficile de garantir l'unicité.

Dans la situation, fréquente, où on a du mal à déterminer quelle est la clé d'une entité, on crée un identifiant « abstrait » indépendant de tout autre attribut. On peut ainsi ajouter dans le type d'entité *Film* un attribut **id**, corespondant à un numéro séquentiel qui sera incrémenté au fur et à mesure des insertions. Ce choix est souvent le meilleur, dès lors qu'un attribut ne s'impose pas de manière évidente comme clé. Il satisfait notamment toutes les propriétés énoncées précédemment (on peut toujours lui attribuer une valeur, il ne sera jamais nécessaire de la modifier, et elle a une représentation compacte).

On représente graphiquement un type d'entité comme sur la figure 3.2 qui donne l'exemple des types *Internaute* et *Film*. L'attribut (ou les attributs s'il y en a plusieurs) formant la clé sont en gras.

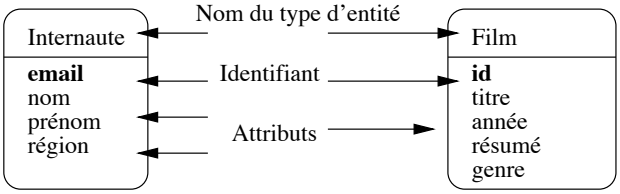


FIG. 3.2 – *Représentation des types d'entité*

Il est essentiel de bien distinguer *types d'entités* et *entités*. La distinction est la même qu'entre *schéma* et *base* dans un SGBD, ou entre *type* et *valeur* dans un langage de programmation.

3.3.2 Associations binaires

La représentation (et le stockage) d'entités indépendantes les unes des autres est de peu d'utilité. On va maintenant décrire les *relations* (ou *associations*) entre des ensembles d'entités.

Definition 3.4 *Une association binaire entre les ensembles d'entités E_1 et E_2 est un ensemble de couples (e_1, e_2), avec $e_1 \in E_1$ et $e_2 \in E_2$.*

C'est la notion classique de relation en théorie des ensembles. On emploie plutôt le terme d'association pour éviter toute confusion avec le modèle relationnel. Une bonne manière d'interpréter une association entre des ensembles d'entités est de faire un petit graphe où on prend quelques exemples, les plus généraux possibles.

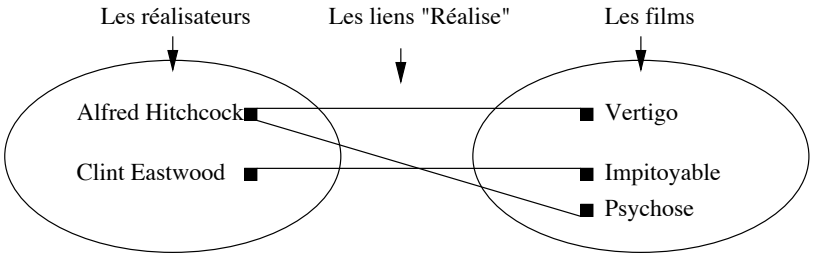


FIG. 3.3 – *Association entre deux ensembles.*

Prenons l'exemple de l'association représentant le fait qu'un réalisateur met en scène des films. Sur le graphe de la figure 3.3 on remarque que :

1. certains réalisateurs mettent en scène plusieurs films ;
2. inversement, un film est mis en scène par au plus un réalisateur.

La recherche des situations les plus générales possibles vise à s'assurer que les deux caractéristiques ci-dessus sont vraies dans tout les cas. Bien entendu on peut trouver 1% des cas où un film a plusieurs réalisateurs, mais la question se pose alors : doit-on modifier la structure de notre base, pour 1% des cas. Ici, on a décidé que non. Encore une fois on ne cherche pas à représenter la réalité dans toute sa complexité, mais seulement la partie de cette réalité que l'on veut stocker dans la base de données.

Ces caractéristiques sont essentielles dans la description d'une association entre des ensembles d'entités.

Definition 3.5 (Cardinalité) *Soit une association (E_1, E_2) entre deux types d'entités. La cardinalité de l'association pour $E_i, i \in \{1, 2\}$, est une paire [min, max] telle que :*

1. *Le symbole* max *(*cardinalité maximale*) désigne le nombre* maximal *de fois où une une entité e_i de E_i peut intervenir dans l'association.*

   *En général, ce nombre est 1 (au plus une fois) ou n (plusieurs fois, nombre indeterminé), noté par le symbole « * ».*

2. *Le symbole* min *(*cardinalité minimale*) désigne le nombre* minimal *de fois où une une entité e_i de E_i peut intervenir dans la relation. En général, ce nombre est 1 (au moins une fois) ou 0.*

Les cardinalités maximales sont plus importantes que les cardinalités minimales ou, plus précisément, elles s'avèrent beaucoup plus difficiles à remettre en cause une fois que le schéma de la base est constitué. On décrit donc souvent une association de manière abrégée en omettant les cardinalités minimales. La notation « * », en UML, est l'abréviation de « 0..* », et « 1 » est l'abréviation de « 1..1 ». On caractérise

également une association de manière concise en donnant les cardinalités maximales aux deux extrêmités, par exemple « 1:n » (association de un à plusieurs) ou « n:n » (association de plusieurs à plusieurs).

Les cardinalités minimales sont parfois désignées par le terme « contraintes de participation ». La valeur 0 indique qu'une entité peut ne pas participer à l'association, et la valeur 1 qu'elle doit y participer.

Insistons sur le point suivant : *les cardinalités n'expriment pas une vérité absolue, mais des choix de conception*. Elles ne peuvent être déclarés valides que relativement à un besoin. Plus ce besoin sera exprimé précisément, et plus il sera possible d'appécier la qualité du modèle.

Il existe plusieurs manières de noter une association entre types d'entités. Nous utilisons ici la notation de la méthode UML, qui est très proche de celle de la méthode OMT. En France, on utilise aussi couramment – de moins en moins... – la notation de la méthode MERISE que nous ne présenterons pas ici.

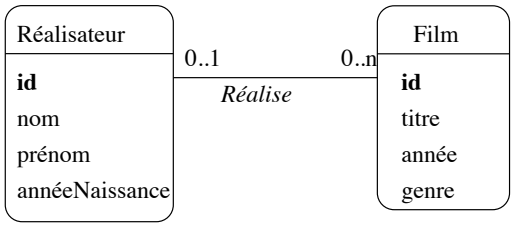


FIG. 3.4 – *Représentation de l'association.*

Dans la notation UML, on indique les cardinalités aux deux extrêmités d'un lien d'association entre deux types d'entités T_A et T_B. Les cardinalités pour T_A sont placées à l'extrémité du lien allant de T_A vers T_B et les cardinalités pour T_B sont l'extrémité du lien allant de T_A vers T_B. Pour l'association entre *Réalisateur* et *Film*, cela donne l'association de la figure 3.4. Cette association se lit *Un réalisateur réalise zéro, un ou plusieurs films*, mais on pourrait tout aussi bien utiliser la forme passive avec comme intitulé de l'asscoiation *Est réalisé par* et une lecture *Un film est réalisé par au plus un réalisateur*. Le seul critère à privilégier dans ce choix des termes est la clarté de la représentation.

Prenons maintenant l'exemple de l'association (*Acteur*,*Film*) représentant le fait qu'un acteur joue dans un film. Un graphe basé sur quelques exemples est donné dans la figure 3.5. On constate tout d'abord qu'un acteur peut jouer dans plusieurs films, et que dans un film on trouve plusieurs acteurs. Mieux : Clint Eastwood, qui apparaissait déjà en tant que metteur en scène, est maintenant également acteur, et dans le même film.

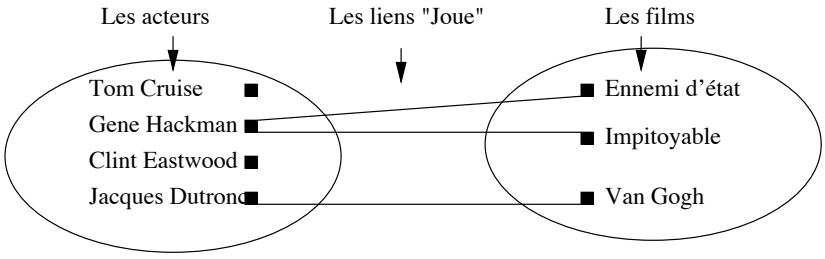


FIG. 3.5 – *Association (*Acteur,Film*)*

Cette dernière constatation mène à la conclusion qu'il vaut mieux regrouper les acteurs et les réalisateurs dans un même ensemble, désigné par le terme plus général « Artiste ». On obtient le schéma de la figure 3.6, avec les deux associations représentant les deux types de lien possible entre un artiste et un film : il peut jouer dans le film, ou le réaliser. Ce « ou » n'est pas exclusif : Eastwood joue dans *Impitoyable*, qu'il a aussi réalisé.

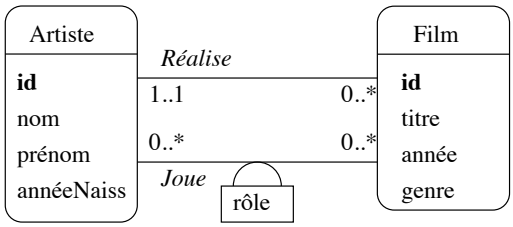


FIG. 3.6 – *Associations entre* Artiste *et* Film.

Dans le cas d'associations avec des cardinalités multiples de chaque côté, on peut avoir des attributs qui ne peuvent être affectés qu'à l'association elle-même. Par exemple l'association *Joue* a pour attribut le rôle tenu par l'acteur dans le film (figure 3.6).

Rappelons qu'un attribut ne peut prendre qu'une et une seule valeur. Clairement, on ne peut associer `rôle` ni à *Acteur* puisqu'il a autant de valeurs possibles qu'il y a de films dans lesquels cet acteur a joué, ni à *Film*, la réciproque étant vraie également. Seules les associations ayant des cardinalités multiples de chaque côté peuvent porter des attributs.

Quelle est la clé d'une association ? Si l'on s'en tient à la définition, une association est un *ensemble* de couples, et il ne peut donc y avoir deux fois le même couple (parce qu'on ne trouve pas deux fois le même élément dans un ensemble). On a donc :

Definition 3.6 (Clé d'une association) *La clé d'une association (binaire) entre un type d'entité E_1 et un type d'entité E_2 est le couple constitué de la clé c_1 de E_1 et de la clé c_2 de E_2.*

En pratique cette contrainte est souvent trop contraignante car on souhaite autoriser deux entités à être liées plus d'une fois dans une association. Imaginons par exemple qu'un internaute soit amené à noter à plusieurs reprises un film, et que l'on souhaite conserver l'historique de ces notations successives. Avec une association binaire entre *Internaute* et *Film*, c'est impossible : on ne peut définir qu'un seul lien entre un film donné et un internaute donné.

Le problème est qu'il n'existe pas de moyen pour distinguer des liens multiples entre deux mêmes entités. Le seul moyen pour effectuer une telle distinction est d'introduire une entité discriminante, par exemple la date de la notation. On obtient alors une association ternaire dans laquelle on a ajouté un type d'entité *Date* (figure 3.7).

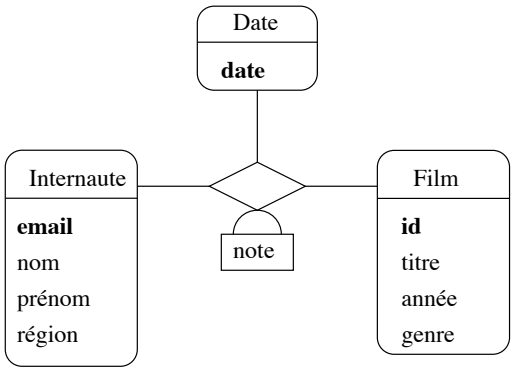


FIG. 3.7 – *Ajout d'une entité* Date *pour conserver l'historique des notations*

Un lien de cette association réunit donc une entité *Film*, une entité *Internaute* et une entité *Date*. On peut identifier un tel lien par un triplet *(id, email, date)* constitué par les clés des trois entités constituant le lien.

Comme le montre la figure 3.8, il devieent alors possible, pou un même internaute, de noter plusieurs fois le même film, pourvu que ce ne soit pas à la même date. Réciproquement un internaute peut noter des films différents le même jour, et un même film peut être noté plusieurs fois à la même date, à condition que ce ne soit pas par le même internaute.

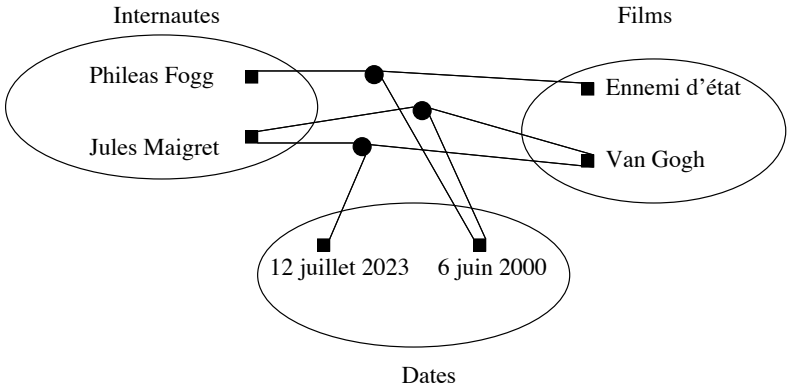


FIG. 3.8 – *Graphe d'une association ternaire*

Même si cette solution est correcte, elle présente l'inconvénient d'introduire une entité assez artificielle, *Date*, qui porte peu d'information et vient alourdir le schéma. En pratique on s'autorise une notation abrégée en ajoutant un attribut `date` dans l'association, et en le soulignant pour indiquer qu'il fait partie de la clé, *en plus du couple des clés des entités* (voir figure 3.9).

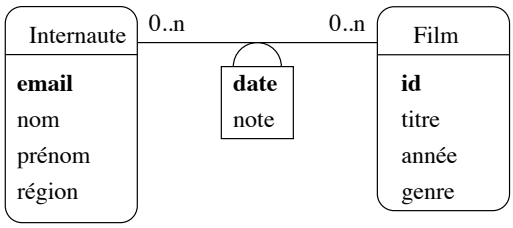


FIG. 3.9 – *Notation abrégée d'une association avec un type d'entité* Date

Nous reviendrons plus longuement sur les associations ternaires par la suite.

3.3.3 Entités faibles

Jusqu'à présent nous avons considéré le cas d'entités *indépendantes* les unes des autres. Chaque entité, disposant de son propre identifiant, pouvait être considérée isolément. Il existe des cas où une entité ne peut exister qu'en étroite association avec une autre, et est identifiée relativement à cette autre entité. On parle alors *d'entité faible*.

Prenons l'exemple d'un cinéma, et de ses salles. On peut considérer chaque salle comme une entité, dotée d'attributs comme la capacité, l'équipement en son Dolby, ou autre. Il est diffcilement imaginable de représenter une salle sans qu'elle soit rattachée à son cinéma. C'est en effet au niveau du cinéma que l'on va trouver quelques informations générales comme l'adresse ou le numéro de téléphone.

Il est possible de représenter le lien en un cinéma et ses salles par une association classique, comme le montre la figure 3.10.a. La cardinalité « 1..1 » force la participation d'une salle à un lien d'association avec un et un seul cinéma. Cette représentation est correcte, mais présente un inconvénient : on doit créer

un identifiant artificiel **id** pour le type d'entité *Salle*, et numéroter toutes les salles, *indépendamment du cinéma auquel elles sont rattachées*.

On peut considérer qu'il est beaucoup plus naturel de numéroter les salles par un numéro interne à chaque cinéma. La clé d'identification d'une salle est alors constituée de deux parties :

1. la clé de *Cinéma*, qui indique dans quel cinéma se trouve la salle ;
2. le numéro de la salle au sein du cinéma.

En d'autres termes, l'entité salle ne dispose pas d'une identification absolue, mais d'une identification *relative* à une autre entité. Bien entendu cela force la salle a toujours être associée à un et un seul cinéma.

La représentation graphique des entités faibles avec UML est illustrée dans la figure 3.10.b. La salle est associée au cinéma avec une association qualifiée par l'attribut **no** qui sert de discriminant pour distinguer les salles au sein d'un même cinéma. Noter que la cardinalité du côté *Cinéma* est implicitement « 1..1 ».

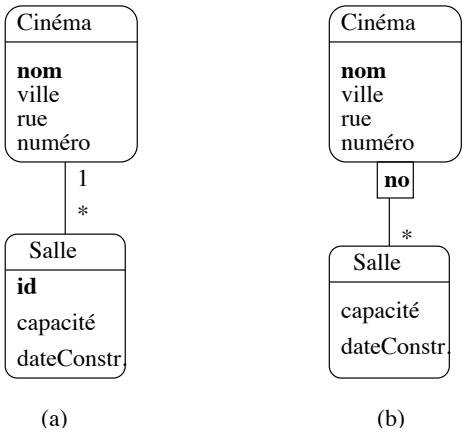


FIG. 3.10 – *Modélisation du lien* Cinéma-Salle *(a) sous la forme d'une association classique (b) avec une entité faible*

L'introduction d'entités faibles n'est pas une nécessité absolue puisqu'on peut très bien utiliser une association classique. La principale différence est que, dans le cas d'une entité faible, on obtient un identification composée qui est souvent plus pratique à gérer, et peut également rendre plus faciles certaines requêtes.

La présence d'un type d'entité faible B associé à un type d'entité A implique également des contraintes fortes sur les créations, modifications et destructions des instances de B et A car on doit toujours s'assurer que la contrainte est valide. Concrètement, en prenant l'exemple de *Salle* et de *Cinéma*, on doit mettre en place les mécanismes suivants :

1. Quand on insère une salle dans la base, on doit toujours l'associer à un cinéma ;
2. quand un cinéma est détruit, on doit aussi détruire toutes ses salles ;
3. quand on modifie la clé d'un cinéma, il faut répercuter la modification sur toutes ses salles.

Pour respecter les règles de destruction/création énoncées, on doit mettre en place une stratégie. Nous verrons que les SGBD relationnels nous permettent de spécifier de telles stratégies.

3.3.4 Associations généralisées

On peut envisager des associations entre plus de deux entités, mais elles sont plus difficiles à comprendre, et surtout la signification des cardinalités devient beaucoup plus ambigue. La définition d'une association n-aire est une généralisation de celle des associations binaires.

Definition 3.7 *Une association n-aire entre n types d'entités $E_1, E_2, \ldots E_n$ est un ensemble de n-uplets $(e_1, e_2, \ldots, e_n)$ où chaque e_i appartient à E_i.*

Même s'il n'y a en principe pas de limite sur le degré d'une association, en pratique on ne va jamais au-delà d'une association entre trois entités.

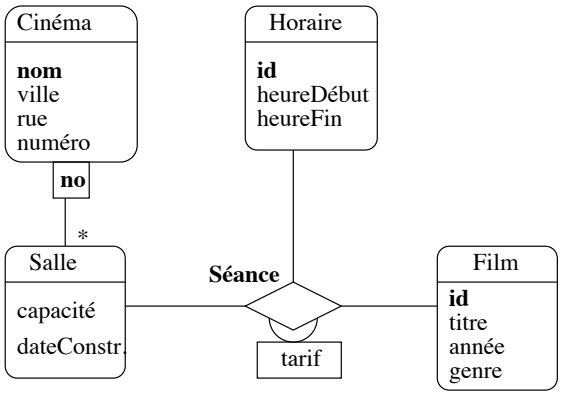


FIG. 3.11 – *Association ternaire représentant les séances*

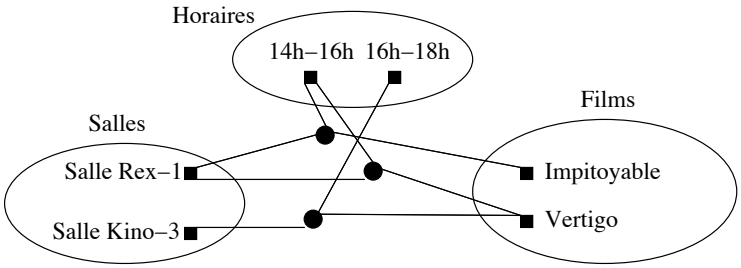


FIG. 3.12 – *Graphe d'une association ternaire*

Nous allons prendre l'exemple d'une association permettant de représenter la projection de certains films dans des salles à certains horaires. Il s'agit d'une association ternaire entre les types d'entités *Film*, *Salle* et *Horaire* (figure 3.11). Chaque instance de cette association lie un film, un horaire et une salle. La figure 3.12 montre quelques-unes de ces instances.

Bien que, jusqu'à présent, une association ternaire puisse être considérée comme une généralisation directe des associations binaires, en réalité de nouveaux problèmes sont soulevés.

Tout d'abord les cardinalités sont, implicitement, « 0..* ». Il n'est pas possible de dire qu'une entité ne participe qu'une fois à l'association. Il est vrai que, d'une part la situation ce présente rarement, d'autre part cette limitation est due à la notation UML qui place les cardinalités à l'extrémité opposée d'une entité.

Plus problématique en revanche est la détermination de la clé. Qu'est-ce qui identifie un lien entre trois entités ? En principe, la clé est le triplet constitué des clés respectives de la salle, du film et de l'horaire constituant le lien. On aurait donc le n-uplet *[nomCinéma, noSalle, idFilm, idHoraire]*. Une telle clé est

assez volumineuse, ce qui risque de poser des problèmes de performance. De plus elle ne permet pas d'imposer certaines contraintes comme, par exemple, le fait que dans une salle, pour un horaire donné, il n'y a qu'un seul film. Comme le montre la figure 3.12, il est tout à fait possible de créer deux liens distincts qui s'appuient sur le même horaire et la même salle.

Ajouter une telle contrainte, c'est signifier que la clé de l'association est en fait le couple *(nomCinéma, noSalle, idHoraire]*. Donc c'est un sous-ensemble de la concaténation des clés, ce qui semble rompre avec la définition donnée précédemment. On peut évidemment compliquer les choses en ajoutant une deuxième contrainte similaire, comme « connaissant le film et l'horaire, je connais la salle ». Il faut ajouter une deuxième clé *[idFilm, idHoraire]*. Il n'est donc plus possible de déduire automatiquement la clé comme on le faisait dans le cas des associations binaires. Plusieurs clés deviennent possibles : on parle de *clé candidates*.

Les associations de degré supérieur à deux sont donc difficiles à manipuler et à interpréter. Il est *toujours* possible de remplacer cette association par un type d'entité. Pour cela on suit la règle suivante :

Règle 3.1 *Soit A une association entre les types d'entité $\{E_1, E_2, \ldots, E_n\}$. La transformation de A en type d'entité s'effectue en trois étapes :*

1. *On attribue un identifiant autonome à A.*
2. *On crée une association A_i de type '1:n' entre A et chacun des E_i.* La contrainte minimale, du côté de A, est toujours à 1.

L'association précédente peut être transformée en un type d'entité *Séance*. On lui attribue un identifiant `idSeance` et des associations '1-n' avec *Film*, *Horaire* et *Salle*. Voir figure 3.13.

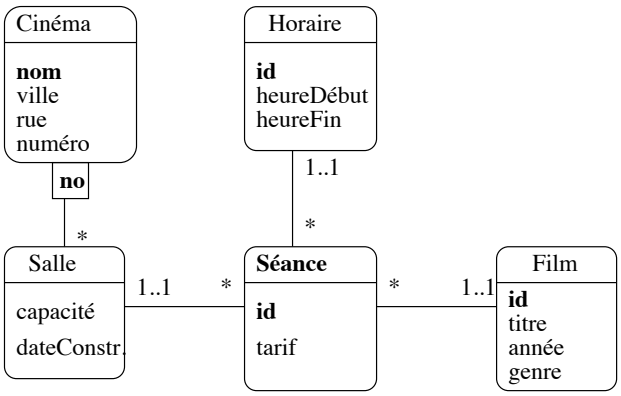


FIG. 3.13 – *L'association* Séance *transformée en entité*

3.4 Avantage et inconvénients du modèle E/A

Le modèle Entité/Association est *simple* et *pratique*.

1. Il n'y a que 3 concepts : *entités*, *associations* et *attributs*.
2. Il est approprié à une représentation graphique intuitive, même s'il existe beaucoup de conventions.
3. Il permet de modéliser rapidement des structures pas trop complexes.

Il y a malheureusement plusieurs inconvénients. Tout d'abord il est *non-déterminisme* : il n'y a pas de règle absolue pour déterminer ce qui est entité, attribut ou relation. Exemple : est-il préférable de représenter

le metteur en scène (MES) comme un attribut de *Film* ou comme une association avec *Artiste* ? Réponse : comme une association ! Les arguments sont les suivants :

1. On connaît alors non seulement le nom, mais aussi toutes les autres propriétés (prénom, âge, ...).
2. L'entité MES peut-être associée à beaucoup d'autres films : on permet le *partage* de l'information.

Autre exemple : est-il indispensable de gérer une entité *Horaire* ? Réponse : pas forcément ! Arguments :

1. *Pour*. Permet de normaliser les horaires. Plusieurs séances peuvent alors faire référence au même horaire (gain de place, facilité de mise à jour, cohérence, ...)
2. *Contre*. On alourdit le schéma inutilement : un horaire est propre à une séance. On peut le représenter comme un attribut de *Séance*.

Enfin on a vu qu'une association pouvait être transformée en entité. Un des principaux inconvénients du modèle E/A reste sa pauvreté : il est difficile d'exprimer des contraintes d'intégrité, des structures complexes. Beaucoup d'extensions ont été proposées, mais la conception de schéma reste en partie matière de bon sens et d'expérience. On essaie en général :

1. de se ramener à des associations entre 2 entités : au-delà, on a probablement intérêt a transformer l'association en entité ;
2. d'éviter toute redondance : une information doit se trouver en un seul endroit ;
3. enfin – et surtout – de privilégier la simplicité et la lisibilité, notamment en ne représentant que ce qui est strictement nécessaire.

Pour en savoir plus

Le modèle E/A est utilisé dans la plupart des méthodes d'analyse/conception : OMT, CASE, MERISE, etc. La syntaxe varie, mais on retrouve toujours les mêmes éléments fondamentaux. Pour en savoir (beaucoup) plus : J. Rumbauch *et al*, *La méthode OMT*.

Dans le cadre des bases de données, le modèle E/A est utilisé dans la phase de conception. Il permet de spécifier la structure des informations qui vont être contenues dans la base et d'offrir une représentation abstraite indépendante du modèle logique qui sera choisi ensuite. Le modèle E/A à cepedant l'inconvénient majeur de ne pas proposer d'opérations sur les données.

3.5 Exercices

Exercice 3.1 *On vous donne un schéma E/A (figure 3.14) représentant des visites dans un centre médical. Répondez aux questions suivantes* en fonction des caractéristiques de ce schéma *(autrement dit, indiquez si la situation décrite est représentable, indépendamment de sa vraissemblance).*

1. *Un patient peut-il effectuer plusieurs visites ?*
2. *Un médecin peut-il recevoir plusieurs patients dans la même consultation ?*
3. *Peut-on prescrire plusieurs médicaments dans une même consultation ?*
4. *Deux médecins différents peuvent-ils prescrire le même médicament ?*

Exercice 3.2 *Le second schéma (figure 3.15) représente des rencontres dans un tournoi de tennis.*

1. *Peut-on jouer des matchs de double ?*
2. *Un joueur peut-il gagner un match sans y avoir participé ?*

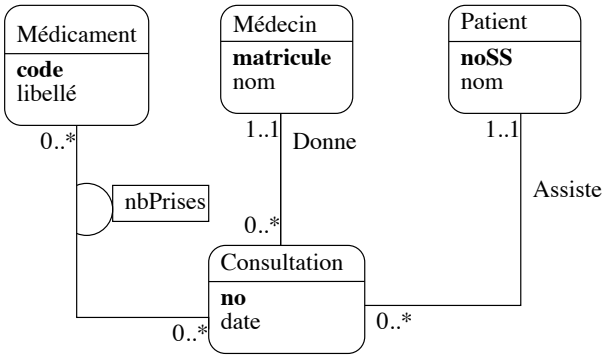


FIG. 3.14 – *Centre médical*

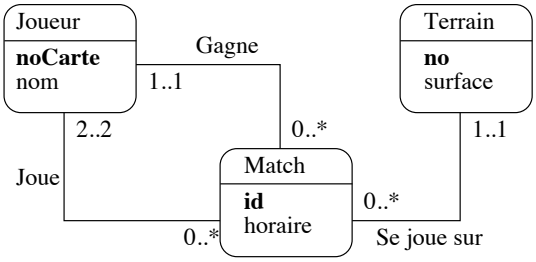


FIG. 3.15 – *Tournoi de tennis*

3. *Peut-il y avoir deux matchs sur le même terrain à la même heure ?*
4. *Connaissant un joueur, peut-on savoir sur quels terrains il a joué ?*

Exercice 3.3 *Voici le schéma E/A (figure 3.16) du système d'information (très simplifié) d'un quotidien.*

1. *Un article peut-il être rédigé par plusieurs journalistes ?*
2. *Un article peut-il être publié plusieurs fois ?*
3. *Peut-il y avoir plusieurs articles sur le même sujet dans le même numéro ?*
4. *Connaissant une article, est-ce que je connais le journal où il est paru ?*

Exercice 3.4 *Voici (figure 3.17) le début d'un schéma E/A pour la gestion d'une médiathèque. La spécification des besoins est la suivante : un disque est constitué d'un ensemble de plages. Chaque plage contient un oeuvre et une seule, mais une oeuvre peut s'étendre sur plusieurs plages (Par exemple une symphonie en 4 mouvements). De plus, pour chaque plage, on connaît les interprêtes.*

1. *Complétez le modèle de la figure 3.17, en ajoutant les cardinalités.*
2. *On suppose que chaque interprète utilise un instrument (voix, piano, guitare, etc) et un seul sur une plage. Où placer l'attribut « Instrument » dans le modèle précédent ?*
3. *Transformez l'association « Joue » dans la figure 3.17 en entité. Donnez le nouveau modèle, sans oublier les cardinalités.*
4. *Introduisez maintenant les entités* Auteur *(d'une oeuvre) et* Editeur *d'un disque dans le schéma. Un disque n'a qu'un éditeur, mais une oeuvre peut avoir plusieurs auteurs.*

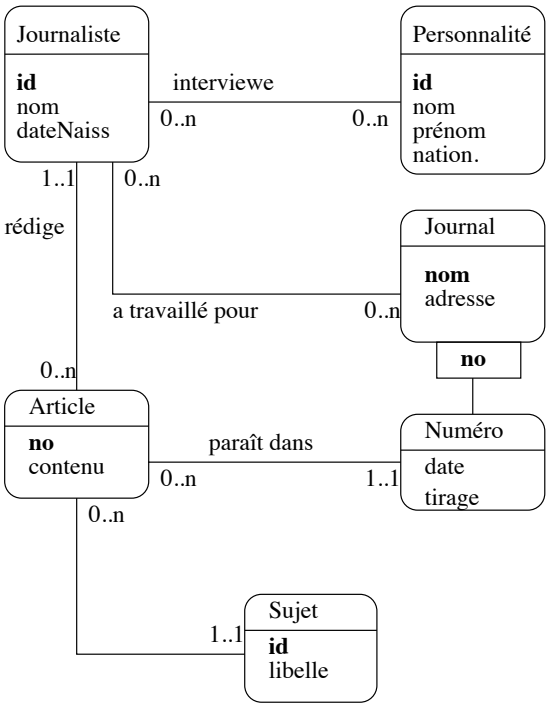


FIG. 3.16 – *Système d'information d'un quotidien*

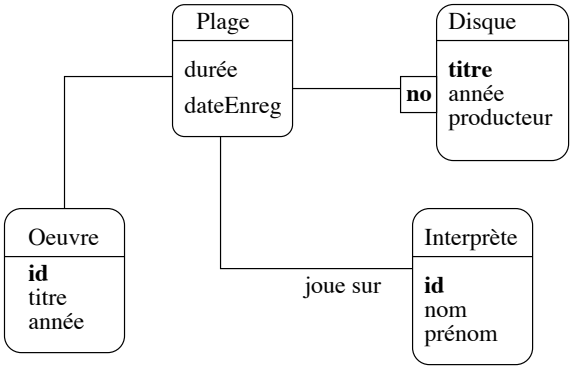


FIG. 3.17 – *Contenu d'un disque*

Exercice 3.5 *La figure 3.18 montre la modélisation de séances de cours sous forme d'une association ternaire. Noter que l'horaire fait partie de la clé (on aurait pu le représenter comme une entité supplémentaire).*

La notion de « cours » regroupe les notions de cours magistral, enseignement dirigé et travaux pratiques, pour une UV donnée. Par exemple l'UV "Base de données" comprend un cours, un ED et un TP.

1. *Donner une représentation, sous forme de graphe ou de tableau, de l'instance de l'association correspondant aux enseignements (cours, EDs, TPs) de l'UV "Base de données".*

2. *Comment exprimer les contraintes suivantes : (i) un professeur ne donne pas deux cours en même temps, (ii) Pour une salle, un cours, un horaire, il y a un seul professeur.*

3. *Transformer l'association en entité, selon la règle vue en cours.*

Exercice 3.6 *Voici quelques tableaux (figure 3.19, 3.20, 3.21) représentant des associations entre entités. Pour chacun,*

1. *Donner une représentation sous forme de graphe.*
2. *Donner le schéma E/A avec les cardinalités correspondant aux exemples donnés.*

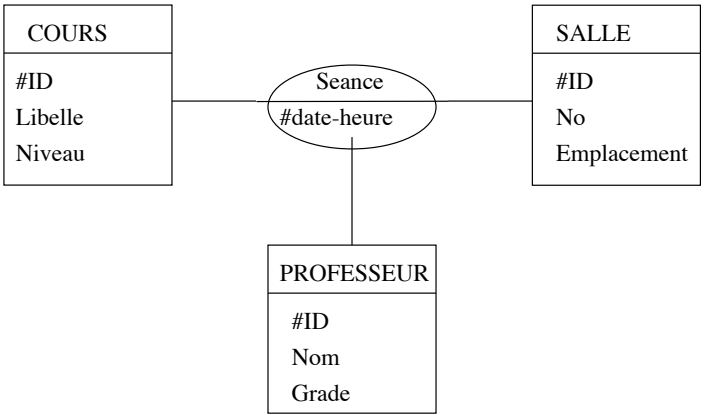


FIG. 3.18 – *Séances de cours*

SOCIETE	DIRECTEUR
Tresys	Charlus
Fungus	Morel
Demona	Saint-Loup
Faribole	Charlus

FIG. 3.19 – *Association SOCIETE/DIRECTEUR*

ORDINATEUR	UTILISATEUR
PC124	Charlus
MAC04	Morel
MAC03	Saint-Loup
PC02	Morel
MAC03	Charlus

FIG. 3.20 – *Association ORDINATEUR/UTILISATEUR*

ORDINATEUR	DISQUES
PC124	dsk09
MAC04	dsk08
MAC04	dsk05
PC124	dsk11
PC02	dsk04

FIG. 3.21 – *Association ORDINATEUR/DISQUES DURS*

Chapitre 4

Le modèle relationnel

Sommaire

Un *modèle de données* définit un mode de représentation de l'information selon trois composantes :

1. Des *structures de données*.
2. Des *contraintes* qui permettent de spécifier les règles que doit respecter une base de données.
3. Des *opérations* pour manipuler les données, en interrogation et en mise à jour.

Les deux premières composantes relèvent du *Langage de Définition de Données* (DDL) dans un SGBD. Le DDL est utilisé pour décrire le *schéma* d'une base de données. La troisième composante (opérations) est la base du *Langage de Manipulation de Données* (DML) dont le représentant le plus célèbre est SQL.

Dans le contexte des bases de données, la principale qualité d'un modèle de données est d'être indépendant de la représentation physique. Cette indépendance permet de séparer totalement les tâches respectives des administrateurs de la base, chargés de l'optimisation de ses performances, et des développeurs d'application ou utilisateurs finaux qui n'ont pas à se soucier de la manière dont le système satisfait leurs demandes.

Le modèle relationnel, venant après les modèles hiérarchique et réseau, offre une totale indépendance entre les représentations logique et physique. Ce chapitre présente la partie du modèle relative à la définition et à la création des tables, ce qui constitue l'essentiel du schéma.

4.1 Définition d'un schéma relationnel

Un des grands avantages du modèle relationnel est sa très grande simplicité. Il n'existe en effet qu'une seule structure, la *relation*. Une relation peut simplement être représentée sous forme de *table*, comme sur la figure 4.1. Une relation a donc un *nom* (*Film*) et se compose d'un ensemble de colonnes désignées par

titre	année	genre
Alien	1979	Science-Fiction
Vertigo	1958	Suspense
Volte-face	1997	Thriller
Pulp Fiction	1995	Policier

FIG. 4.1 – *Une relation*

un *nom d'attribut*. Dans chaque colonne on trouve des valeurs d'un certain *domaine* (chaînes de caractères, nombres). Enfin on constate que chaque ligne (ou *tuple*) correspond à une entité (ici des films).

Un schéma relationnel est constitué d'un ensemble de *schémas de relations* qui décrivent, à l'aide des élements présentés informellement ci-dessus (domaines, attributs, noms de relation) le contenu d'une relation. Le schéma de la relation de la figure 4.1 est donc :

Film (titre: string, année: number, genre : string)

Il existe un langage de définition pour créer une relation dans un SGBDR (voir section 4.3), mais nous nous contenterons pour l'instant de la description ci-dessus. Voici maintenant quelques précisions sur la terminologie introduite ci-dessus.

Domaines

Un *domaine de valeurs* est un ensemble d'instances d'un type élémentaire. Exemple : les entiers, les réels, les chaînes de caractères, etc. La notion de 'type élémentaire' s'oppose à celle de type structuré : il est interdit en relationnel de manipuler des valeurs instances de graphes, de listes, d'enregistrements, etc. En d'autres termes le système de types est figé et fourni par le système.

Attributs

Les *attributs* nomment les colonnes d'une relation. Il servent à la fois à indiquer le contenu de cette colonne, et à la référencer quand on effectue des opérations. Un attribut est toujours associé à un *domaine*. Le nom d'un attribut peut apparaître dans plusieurs schémas de relations.

Schéma de relation

Un schéma de relation est simplement un nom suivi de la liste des attributs, chaque attribut étant associé à son domaine. La syntaxe est donc :

$$R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$$

où les A_i sont les noms d'attributs et les D_i les domaines. *L'arité* d'une relation est le nombre de ses attributs.

On peut trouver dans un schéma de relation plusieurs fois le même domaine, mais une seule fois un nom d'attribut. Le domaine peut être omis en phase de définition.

Instance d'une relation

Une *instance d'une relation* R, ou simplement *relation* se définit mathématiquement comme un sous-ensemble *fini* du produit cartésien des domaines des attributs de R. Rappelons que le produit cartésien $D_1 \times \ldots \times D_n$ entre des domaines $D_1, \ldots, D_n$ est l'ensemble de *tous* les tuples $(v_1, \ldots, v_n)$ où $v_i \in D_i$.

Un des fondements du modèle relationnel est la théorie des ensembles et la notion de relation dans le modèle correspond strictement au concept mathématique dans cette théorie.

Une relation se représente sous forme de *table*, et on emploie le plus souvent ces deux termes comme des synonymes.

La définition d'une relation comme un ensemble (au sens mathématique) a quelques conséquences importantes :

1. *l'ordre des lignes n'a pas d'importance* car il n'y a pas d'ordre dans un ensemble ;
2. *on ne peut pas trouver deux fois la même ligne* car il n'y a pas de doublons dans un ensemble ;
3. *il n'y a pas de « case vide » dans la table*, donc toutes les valeurs de tous les attributs sont toujours connues ;

Dans la pratique les choses sont un peu différentes : nous y reviendrons.

Clé d'une relation

La clé d'une relation est le plus petit sous-ensemble des attributs qui permet d'identifier chaque ligne de manière unique. Comme on a vu que deux lignes sont toujours différentes, l'ensemble de tous les attributs est lui-même une clé mais on peut pratiquement toujours trouver un sous-ensemble qui satisfait la condition. Pour distinguer la clé, nous mettrons le (ou les) attribut(s) en gras.

Film (**titre**, année, genre)

Le choix de la clé est très important pour la qualité du schéma. Choisir d'identifier un film par son titre comme nous l'avons envisagé dans l'exemple précédent n'est pas un très bon choix : nous y reviendrons.

Tuples

Un *tuple* est une liste de n valeurs $(v_1, \ldots, v_n)$ où chaque valeur v_i est la valeur d'un attribut A_i de domaine D_i : $v_i \in D_i$. Exemple :

('Cyrano', 1992, 'Rappeneau')

Un tuple est donc simplement une ligne dans la représentation d'une relation sous forme de table. En théorie, on connaît les valeurs de tous les attributs du tuple.

Base de données

Une (instance de) base de données est un ensemble fini (d'instances) de relations. Le schéma de la base est l'ensemble des schémas des relations de cette base.

La création d'un schéma de base de données est simple une fois que l'on a déterminé toutes les relations qui constituent la base. En revanche le choix de ces relations est un problème difficile car il détermine en grande partie les caractéristiques, qualités de la base : performances, exactitude, exhaustivité, disponibilité des informations, etc. Un des aspects importants de la théorie des bases de données relationnelles consiste précisément à définir ce qu'est un « bon » schéma et propose des outils formels pour y parvenir.

En pratique, on procède d'une manière moins rigoureuse mais plus accessible, en concevant le schéma à l'aide d'un modèle de données « conceptuel », puis en transcrivant le shéma conceptuel obtenu en schéma relationnel. La technique la plus répandue consiste à partir d'un schéma Entité/Association. La section suivante donne les règles du processus de transformation, en s'appuyant sur l'exemple de la figure 4.2.

4.2 Passage d'un schéma E/A à un schéma relationnel

On passe donc d'un modèle disposant de deux structures (entités et associations) à un modèle disposant d'une seule structure (relations). Logiquement, entités et associations seront donc toutes deux transformées en relations. La subtilité réside en fait dans la nécesité de préserver les *liens* existant explicitement dans un schéma E/A et qui semblent manquer dans le modèle relationnel. Dans ce dernier cas on utilise en fait un mécanisme de référence par valeur basé sur les *clés* des relations.

Le choix de la clé d'une relation est un problème central dans la conception de schéma.

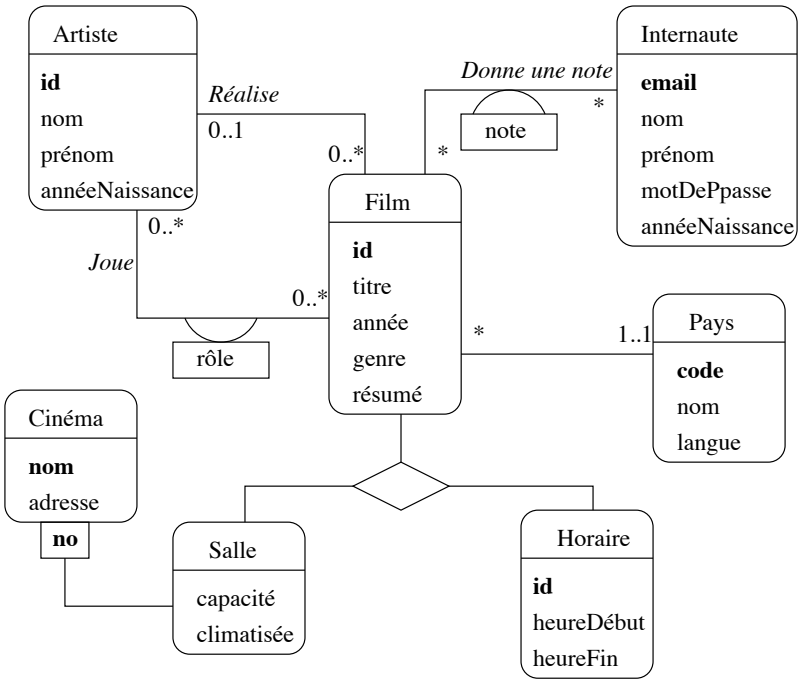


FIG. 4.2 – *Le schéma de la base de données Films*

4.2.1 Règles générales

Nous allons considérer dans un premier temps que la clé est définie, pour chaque type d'entité E, par un identifiant abstrait, **idE**.

Règles de passage : entités

Rappel : le schéma d'une relation est constitué du nom de la relation, suivi de la liste des attributs. Alors, pour chaque entité du schéma E/A:

1. On crée une relation de même nom que l'entité.
2. Chaque propriété de l'entité, y compris l'identifiant, devient un attribut de la relation.
3. Les attributs de l'identifiant constituent la clé de la relation.

Exemple 4.1 *À partir du schéma E/A Officiel des spectacles, à l'exception des entités concernant les cinémas, les salles et les horaires, on obtient les relations suivantes :*

- *Film (**idFilm**, titre, année, genre, résumé)*
- *Artiste (**idArtiste**, nom, prénom, anneéNaissance)*
- *Internaute (**email**, nom, prénom, région)*
- *Pays (**code**, nom, langue)*

On peut remarquer que l'on a perdu pour l'instant tout lien entre les relations.

Règles de passage : associations de un à plusieurs

Soit une association de un à plusieurs [1] entre A et B. Le passage au modèle logique suit les règles suivantes :

1. On crée les relations R_A et R_B correspondant respectivement aux entités A et B.
2. L'identifiant de B devient un attribut de R_A.

L'idée est qu'une occurence de A « référence » l'occurence de B qui lui est associée à l'aide d'une *clé étrangère*. Cette référence se fait de manière unique et suffisante à l'aide de l'identifiant.

Exemple 4.2 *Voici le schéma obtenu pour représenter l'association entre les types d'entité* Film, Artiste *et* Pays. *Les clés étrangères sont en italiques.*

- *Film (**idFilm**, titre, année, genre, résumé, idArtiste, codePays)*
- *Artiste (**idArtiste**, nom, prénom, annéeNaissance)*
- *Pays (**code**, nom, langue)*

Le rôle précis tenu par l'artiste dans l'association disparaît. L'artiste dans *Film* a un rôle de metteur en scène, mais il pourrait tout aussi bien s'agir du décorateur ou de l'accessoiriste. Rien n'empêche cependant de donner un nom plus explicite à l'attribut. Il n'est pas du tout obligatoire en fait que les attributs constituant une clé étrangères aient le même nom que ceux de le clé primaire auxquels ils se réfèrent. Voici le schéma de la table *Film*, dans lequel la clé étrangère pour le metteur en scène est nommée *idMES*.

Film (**idFilm**, titre, année, genre, résumé, *idMES*)

Les tables ci-dessous montrent un exemple de la représentation des associations entre *Film* et *Artiste* d'une part, *Film* et *Pays* d'autre part (on a omis le résumé du film). Noter que si on ne peut avoir qu'un artiste dont l'**id** est 2 dans la table *Artiste*, en revanche rien n'empêche cet artiste 2 de figurer plusieurs fois dans la colonne *idMES* de la table *Film*. On a donc bien l'équivalent de l'association un à plusieurs élaborée dans le schéma E/A.

idFilm	titre	année	genre	*idMES*	*codePays*
100	Alien	1979	Science Fiction	1	US
101	Vertigo	1958	Suspense	2	US
102	Psychose	1960	Suspense	2	US
103	Kagemusha	1980	Drame	3	JP
104	Volte-face	1997	Action	4	US
105	Van Gogh	1991	Drame	8	FR
106	Titanic	1997	Drame	6	US
107	Sacrifice	1986	Drame	7	FR

La table *Film*

idArtiste	nom	prénom	annéeNaiss
1	Scott	Ridley	1943
2	Hitchcock	Alfred	1899
3	Kurosawa	Akira	1910
4	Woo	John	1946
6	Cameron	James	1954
7	Tarkovski	Andrei	1932
8	Pialat	Maurice	1925

La table *Artiste*

code	nom	langue
US	Etats Unis	anglais
FR	France	français
JP	Japon	japonais

La table *Pays*

1. Il s'agit ici des cardinalités *maximales* qui déterminent pour l'essentiel le schéma relationnel obtenu.

Règles de passage : associations avec type entité faible

Une entité faible est toujours identifiée par rapport à une autre entité. C'est le cas par exemple de l'association en *Cinéma* et *Salle* (voir chapitre 3). Cette association est de type « un à plusieurs » car l'entité faible (une salle) est liée à une seule autre entité (un cinéma) alors que, en revanche, un cinéma peut être lié à plusieurs salles.

Le passage à un schéma relationnel est donc identique à celui d'une association 1-n classique. On utilise un mécanisme de clé étrangère pour référencer l'entité forte dans l'entité faible. La seule nuance est que la clé étrangère est une partie de l'identifiant de l'entité faible.

Exemple 4.3 *Voici le schéma obtenu pour représenter l'association entre les types d'entité* Cinéma *et* Salle. *On note que l'identifiant d'une salle est constitué de l'identifiant du cinéma (ici on a considéré que le nom du cinéma suffisiat à l'identifier), et d'un numéro complémentaire permettant de distinguer les salles au sein d'un même cinéma. La clé étrangère est donc une partie de la clé primaire.*

- *Cinéma (**nomCinéma**, numéro, rue, ville)*
- *Salle (**nomCinéma, no**, capacité)*

Règles de passage : associations binaires de plusieurs à plusieurs

Soit une association binaire n-m entre A et B.

1. On crée les relations R_A et R_B correspondant respectivement aux entités A et B.
2. On crée une relation R_{A-B} pour l'association.
3. La clé de R_A et la clé de R_B deviennent des attributs de R_{A-B}.
4. La clé de cette relation est la concaténation des clés des relations R_A et R_B.
5. Les propriétés de l'association deviennent des attributs de R_{A-B}.

Exemple 4.4 *Toujours à partir du schéma Officiel des spectacles, on obtient la table* Rôle *représentant l'association entre les films et les acteurs.*

- *Film (**idFilm**, titre, année, genre, résumé, idMES, codePays)*
- *Artiste (**idArtiste**, nom, prénom, annéeNaissance)*
- *Role (**idFilm, idActeur**, nomRôle)*

De même, on obtient une table Notation *pour représenter l'association entre un internaute et les films qu'il a notés.*

- *Film (**idFilm**, titre, année, genre, résumé, idMES, codePays)*
- *Internaute (**email**, nom, prénom, région)*
- *Notation (**email, idFilm**, note)*

Les tables suivantes montrent un exemple de représentation de *Rôle*. On peut constater le mécanisme de référence unique obtenu grâce aux clés des relations. Chaque rôle correspond à un unique acteur et à un unique film. De plus on ne peut pas trouver deux fois la même paire (`idFilm`, `idActeur`) dans cette table, ce qui n'aurait pas de sens. En revanche un même acteur peut figurer plusieurs fois (mais pas associé au même film), ainsi qu'un même film (mais pas associé au même acteur).

idFilm	titre	année	genre	*idMES*	*codePays*
20	Impitoyable	1992	Western	100	USA
21	Ennemi d'état	1998	Action	102	USA

La table *Film*

idArtiste	nom	prénom	annéeNaiss
100	Eastwood	Clint	1930
101	Hackman	Gene	1930
102	Scott	Tony	1930
103	Smith	Will	1968

La table *Artiste*

idFilm	***idActeur***	rôle
20	100	William Munny
20	101	Little Bill
21	101	Bril
21	103	Robert Dean

La table *Rôle*

On peut remarquer finalement que chaque partie de la clé de la table *Rôle* est elle-même une clé étrangère qui fait référence à une ligne dans une autre table :

1. l'attribut `idFilm` fait référence à une ligne de la table *Film* (un film) ;
2. l'attribut `idActeur` fait référence à une ligne de la table *Artiste* (un acteur) ;

Le même principe de référencement et d'identification des tables s'applique à la table *Notation* dont un exemple est donné ci-dessous. Il faut bien noter que, par choix de conception, on a interdit qu'un internaute puisse noter plusieurs fois le même film, de même qu'un acteur ne peut pas jouer plusieurs fois dans un même film. Ces contraintes ne constituent pas des limitations, mais des décisions prises au moment de la conception sur ce qui est autorisé, et sur ce qui ne l'est pas.

idFilm	titre	année	genre	*idMES*	*codePays*
100	Alien	1979	Science Fiction	1	USA
101	Vertigo	1958	Suspense	2	USA
102	Psychose	1960	Suspense	2	USA
103	Kagemusha	1980	Drame	3	JP
104	Volte-face	1997	Action	4	USA
105	Van Gogh	1991	Drame	8	FR
106	Titanic	1997	Drame	6	USA
107	Sacrifice	1986	Drame	7	FR

La table *Film*

email	nom	prénom	annéeNaiss
doe@void.com	Doe	John	1945
fogg@verne.fr	Fogg	Phileas	1965

La table *Internaute*

idFilm	***email***	note
100	fogg@verne.fr	4
104	fogg@verne.fr	3
100	doe@void.com	5
107	doe@void.com	2
106	fogg@verne.fr	5

La table *Notation*

Le processus de conception détaillé ci-dessus permet de décomposer toutes les informations d'une base de données en plusieurs tables dont chacune stocke un des ensembles d'entités gérés par l'application. Les liens sont définis par un mécanisme de référencement basé sur les clés primaires et les clés étrangères. Il est important de bien comprendre ce mécanisme pour maîtriser la construction de bases de données qui ne nécessiteront par de réorganisation – nécessairement douloureuse – par la suite.

Associations ternaires

Dans le cas d'associations impliquant plus de deux entités, on atteint une des limites du modèle Entité/Association en matière de spécification de contraintes. En première approche, on peut appliquer la règle énoncée précédemment pour les associations binaires et la généraliser. On obtiendrait alors, pour l'association *Séance* :

– Cinéma (**nomCinéma**, numéro, rue, ville)

idCinéma	**no**	capacité
Le Rex	1	200
Kino	1	130
Le Rex	2	180

La table *Salle*

idHoraire	heureDébut	heureFin
1	14	16
2	16	18

La table *Horaire*

idFilm	titre	année	genre	*idMES*	*codePays*
100	Alien	1979	Science Fiction	1	USA
101	Vertigo	1958	Suspense	2	USA
102	Psychose	1960	Suspense	2	USA
103	Kagemusha	1980	Drame	3	JP
104	Volte-face	1997	Action	4	USA
105	Van Gogh	1991	Drame	8	FR
106	Titanic	1997	Drame	6	USA
107	Sacrifice	1986	Drame	7	FR

La table *Film*

idFilm	***nomCinéma***	***noSalle***	***idHoraire***	tarif
103	Le Rex	2	1	23
103	Le Rex	2	2	56
100	Kino	1	2	45
102	Le Rex	2	1	50

La table *Séance*

FIG. 4.3 – *Représentation d'une association ternaire :* Séance

– Salle (***nomCinéma*****, no**, capacité)

– Film (**idFilm**, titre, année, genre, résumé, *idMES*, *codePays*)

– Horaire (**idHoraire**, heureDébut, heureFin)

– Séance (***idFilm***, ***nomCinéma***, ***noSalle***, ***idHoraire***, tarif)

Donc, la relation *Séance* a pour clé la concaténation des identifiants de chacune des entités composantes, ce qui donne une clé d'une taille assez importante. On autorise alors une base comme celle de la figure 4.3. On ne peut pas trouver deux fois le même triplet constituant la clé.

Maintenant on s'aperçoit que la même salle présente deux films différents au même horaire. Si on souhaite éviter cette situation, la clé devient **(nomCinéma, noSalle, idHoraire)**, et on ne respecte plus la règle de passage du schéma E/A vers le schéma relationnel.

En d'autres termes, en cas d'association entre plus de 2 entités, la clé de la table représentant l'association est un sous-ensemble de la concaténation des clés. Il faut se poser soigneusement la question de la (ou des) clé(s) au moment de la création de la table car elle(s) ne peu(ven)t plus être déduite(s) du schéma E/A. On parle parfois de *clé candidate*. Ces clés peuvent être spécifiées avec la clause `UNIQUE` du langage SQL2.

4.2.2 Retour sur le choix des identifiants

Il est préférable en général de choisir un identifiant « neutre » qui ne soit pas une propriété de l'entité. En effet :

1. Chaque valeur de l'identifiant doit caractériser de manière unique une occurence.
   Exemple : **titre** pour la relation *Film* ou **nom** pour la relation *Acteur* ne sont clairement pas des bons choix.

2. Si on utilise un *ensemble* de propriétés comme identifiant, la référence à une occurence est très lourde. Exemple : la clé de *Cinéma* pourraît être **(nom, rue, ville)**.

3. L'identifiant sert de référence externe *et ne doit donc jamais être modifiable* (il faudrait répercuter les mises à jour).

L'inconvénient de l'identifiant « neutre » est qu'il ne donne pas d'indication sur l'occurence qu'il réfère. Par exemple, quand on consulte la table *Séance*, on ne sait pas dire de quel film il s'agit sans aller rechercher la ligne de la table *Film* correspondant à l'identifiant du film.

En admettant – pour un sintant – que le titre identifie de manière unique un film, le schéma de la table *Film* devient :

Film (**titre**, année, genre, résumé, *idMES*, *codePays*)

La relation *Séance* a alors pour schéma :

Séance (***titreFilm***, ***nomCinéma***, ***noSalle***, ***idHoraire***, tarif)

ce qui permet d'obtenir le titre du film directement, comme le montre l'instance ci-dessous.

titreFilm	***nomCinéma***	***noSalle***	***idHoraire***	tarif
Kagemusha	Le Rex	2	1	23
Kagemusha	Le Rex	2	2	56
Alien	Kino	1	2	45
Psychose	Le Rex	2	1	50

La table *Séance*, avec le titre du film

Un problème du schéma ci-dessus est que la référence à une ligne de la table *Séance* devient compliquée, et donc peu performante. Il faut veiller à limiter le nombre de champs constituant une clé car l'expression des requêtes est plus lourde, et leur exécution peut être ralentie par la taille des index.

4.2.3 Dénormalisation du modèle logique

Le terme de *dénormalisation* s'applique à un non-respect des règles énoncées précédemment, avec deux objectifs principaux :

1. *Simplifier* le schéma relationnel en réduisant le nombre d'éléments qui le composent (fichiers ou segments ou records ou relation).

2. *Faciliter* l'accès aux données en introduisant un certain degré de redondance.

Ces techniques reviennent à introduire des anomalies dans le schéma. Il faut donc systématiquement comparer le gain attendu avec les risques courus !

Suppression de relations

On peut supprimer les entités qui portent peu d'attributs en les déplaçant vers une autre relation. Exemple : si le schéma de *Cinema* est simplement *Cinéma* (**nom**, adresse), on peut supprimer la relation et placer l'adresse dans *Salle*.

Salle (**nomCinéma**, **noSalle**, adresse)

L'adresse d'un cinéma est dupliquée autant de fois qu'il y a de salles. Cette option implique une perte de place due à la redondance, un effort de saisie supplémentaire, et des risques d'incohérences. Elle ne peut être valable que tant qu'il n'y a pas d'attrbuts à ajouter pour qualifier un cinéma. Quand ce sera le cas, il faudra finalement se décider à (1) créer la relation *Cinéma* et (2) supprimer les attributs mal placés dans *Salle*.

Autre exemple : il peut paraître inutile de créer *Horaire* pour gérer un couple d'heures. On peut alors placer l'horaire dans la relation *Séance*. Afin de permettre l'existence de plusieurs lignes avec le même film et la même salle, il faut introduire l'attribut **heureDébut** dans la clé. On obtient le schéma suivant.

Séance (***idFilm*, *nomCinema*, *noSalle*, heureDébut**, heureFin, tarif)

Cette variante présente peu d'inconvénients. On peut tout juste citer le fait qu'il y a duplication de certains horaires, et que la gestion contraintes (heure début < heure fin) doit être gérée pour plus de lignes.

En fait la création d'un type d'entité *Horaire* ou *Date*, même si elle se justifie en théorie, présente plus d'inconvénients que d'avantages. En pratique, on place toujours un attribut `horaire` ou `date` dans le schéma de la relation.

Introduction de redondance

En principe il faut éviter les redondances dans une base de données. Donc une information est représentée soit *explicitement* (elle figure une fois et une seule), soit *implicitement* (elle peut être déduite ou calculée).

L'accès à une information peut cependant être long et/ou compliqué, et justifier l'introduction de redondances. Par exemple :

– à partir de la table *Séance*, il faut consulter *Salle* puis *Cinéma* pour connaître l'adresse du cinéma. ;

– il faut faire un calcul pour obtenir le nombre de salles d'un cinéma.

En pratique on peut être amené à introduire (prudemment) des redondances. Les problèmes précédents pourraient ainsi se résoudre de la manière suivante :

– ajout de l'adresse du cinéma dans la table *Séance*.

  Séance (***idFilm*, *nomCinema*, *noSalle*, *idHoraire***, adresse)

  Le risque peut être considéré comme mineur car l'adresse d'un cinéma change rarement.

– ajout du nombre de salles dans la relation *Cinéma*.

  Cinéma (**nomCinema**, numéro, rue, ville, nbSalles).

  Il faut mettre à jour `nbSalles` quand une salle est ajoutée ou supprimée d'un cinéma. On peut considérer que cela arrive rarement, et que la redondance est donc sans grand danger.

– ajout du nombre de rôles tenus dans la table *Artiste*.

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

  Il faut mettre à jour `nbRôles` quand un artiste obtient un nouveau rôle. Cela arrive fréquemment, et le risque induit par la redondance est alors important.

L'introduction de redondances présente principalement le danger d'introduire des *incohérences* dans la base. On peut utiliser le mécanisme des *triggers* pour effectuer automatiquement la répercussion de la modification d'une donnée sur ses autres versions présentes dans la base.

4.3 Le langage de définition de données SQL2

Ce chapitre présente le *langage de définition de données* (LDD) qui permet de spécifier le schéma d'une base de données relationnelle. Ce langage correspond à une partie de la norme SQL (*structured query language*), l'autre partie étant relative à la *manipulation des données* (LMD).

La définition d'un schéma logique comprend essentiellement deux parties : d'une part la description des *tables* et de leur contenu, d'autre part les *contraintes* qui portent sur les données de la base. La spécification des contraintes est souvent placée au second plan bien qu'elle soit en fait très importante : elle permet d'assurer, *au niveau de la base* des contrôles sur l'intégrité des donnés qui s'imposent à toutes les applications accédant à cette base. Un dernier aspect de la définition d'un schéma, rapidement survolé ici, est la description de la représentation physique.

Il existe plusieurs versions de SQL. Le plus ancien standard date de 1989. Il a été révisé de manière importante en 1992 : la norme résultant de cette révision est SQL-92 ou SQL2. Une extension (SQL3) comprenant l'introduction de caractéristiques orientées-objet est en cours de discussion depuis très longtemps, et certains systèmes ont déjà anticipé en proposant de nouvelles fonctionnalités. Le matériel présenté dans ce cours est essentiellement SQL2, sauf pour quelques nouveautés explicitement indiquées.

4.3.1 Types SQL

La norme SQL ANSI propose un ensemble de types qui sont donnés dans le tableau 4.1. Ce tableau présente également la taille, en octets, des instances de chaque type, cette taille n'étant ici qu'à titre indicatif car elle peut varier selon les systèmes.

Type	**Description**	**Taille**
`INTEGER`	Type des entiers relatifs	4 octets
`SMALLINT`	Idem.	2 octets
`BIGINT`	Idem.	8 octets
`FLOAT`	Flottants simple précision	4 octets
`DOUBLE PRECISION`	Flottants double précision	8 octets
`REAL`	Synonyme de `FLOAT`	4 octets
`NUMERIC` (*M*, *D*)	Numérique avec précision fixe.	*M* octets
`DECIMAL` (*M*, *D*)	Idem.	*M* octets
`CHAR`(*M*)	Chaînes de longueur fixe	*M* octets
`VARCHAR`(*M*)	Chaînes de longueur variable	*L+1* avec $L \leqslant M$
`BIT VARYING`	Chaînes d'octets	Longueur de la chaîne.
`DATE`	Date (jour, mois, an)	env. 4 octets
`TIME`	Horaire (heure, minutes, secondes)	env. 4 octets
`DATETIME`	Date et heure	8 octets
`YEAR`	Année	2 octets

TAB. 4.1 – *Types SQL ANSI*

Types numériques exacts

La norme SQL ANSI distingue deux catégories d'attributs numériques : les *numériques exacts*, et les *numériques flottants*. Les types de la première catégorie (essentiellement `INTEGER` et `DECIMAL`) permettent de spécifier la précision souhaitée pour un attribut numérique, et donc de représenter une valeur exacte. Les numériques flottants correspondent aux types couramment utilisés en programmation (`FLOAT`, `DOUBLE`) et ne représentent une valeur qu'avec une précision limitée.

Le type `INTEGER` permet de stocker des entiers, sur 4 octets en général, mais la taille du stockage n'est pas spécifiée par la norme. Il existe deux variantes du type `INTEGER` : `SMALLINT` et `BIGINT`. Ces types différent par la taille utilisée pour le stockage : voir le tableau 4.1.

Le type DECIMAL (*M*, *D*) correspond à un numérique de taille maximale *M*, avec un nombre de décimales fixé à *D*. Le type NUMERIC est un synonyme pour DECIMAL. Ces types sont surtout utiles pour manipuler des valeurs dont la précision est connue, comme les valeurs monétaires. Afin de préserver cette précision, les instances de ces types sont stockées comme des chaînes de caractères.

Types numériques flottants

Ces types s'appuient sur la représentation des numériques flottants propre à la machine, en simple ou double précision. Leur utilisation est donc analogue à celle que l'on peut en faire dans un langage de programmation comme le C.

1. Le type FLOAT correspond aux flottants en simple précision.
2. Le type DOUBLE PRECISION correspond aux flottants en double précision. Le raccourci DOUBLE est accepté.
3. Le type REAL est un synonyme pour DOUBLE.

Caractères et chaînes de caractères

Les deux types principaux de la norme ANSI, disponibles dans la plupart des SGBD relationnels, sont CHAR et VARCHAR. Ces deux types permettent de stocker des chaînes de caractères d'une taille maximale fixée par le paramètre *M*. Les syntaxes sont identiques. Pour le premier, CHAR(*M*), et pour le second VARCHAR(*M*).

La différence essentielle entre les deux types est qu'une valeur CHAR a une taille fixée, et se trouve donc complétée avec des blancs si sa taille est inférieure à *M*. En revanche une valeur VARCHAR a une taille variable et est tronquée après le dernier caractère non blanc.

Quand on veut stocker des chaînes de caractères très longues (des textes, voire des livres), le type VARCHAR ne suffit plus. La norme SQL propose un type BIT VARYING qui correspond à de très longues chaînes de caractères. Souvent les systèmes proposent des variantes de ce type sous le nom BLOB (pour *Binary Long Object*) ou LONG.

Dates

Un attribut de type DATE stocke les informations jour, mois et année (sur 4 chiffres). La représentation interne n'est pas spécifiée par la norme. Tous les systèmes proposent de nombreuses opérations de conversion (non normalisées) qui permettent d'obtenir un format d'affichage quelconque.

Un attribut de type TIME stocke les informations « heure », « minute » et « seconde ». L'affichage se fait par défaut au format HH:MM:SS. Le type DATETIME permet de combiner une date et un horaire, l'affichage se faisant au format AAAA-MM-JJ HH:MM:SS.

4.3.2 Création des tables

La commande principale est CREATE TABLE. Voici la commande de création de la table *Internaute*.

```
CREATE TABLE Internaute (email VARCHAR (50) NOT NULL,
                         nom VARCHAR (20) NOT NULL,
                         prenom VARCHAR (20),
                         motDePasse VARCHAR (60) NOT NULL,
                         anneeNaiss DECIMAL (4))
```

La syntaxe se comprend aisément. La seule difficulté est de choisir correctement le type de chaque attribut. Le NOT NULL dans la création de table *Internaute* indique que l'attribut correspondant doit *toujours* avoir une valeur.

Il s'agit d'une différence importante entre la pratique et la « théorie » : on admet que certains attributs peuvent ne pas avoir de valeur, ce qui est très différent d'une chaîne vide ou de 0. Quand on parle de valeur `NULL` en SQL2, il s'agit en fait d'une absence de valeur. En conséquence :

1. on ne peut pas faire d'opération incluant un `NULL` ;
2. on ne peut pas faire de comparaison avec un `NULL`.

L'option `NOT NULL` oblige à toujours indiquer une valeur. L'option suivante permet ainsi de garantir que tout internaute a un mot de passe.

```
motDePasse  VARCHAR(60) NOT NULL
```

Le SGBD rejettera alors toute tentative d'insérer une ligne dans *Internaute* sans donner de mot de passe. Si les valeurs à `NULL` sont autorisées, il faudra en tenir compte quand on interroge la base. Cela peut compliquer les choses, voire donner des résultats surprenants : il est préférable de forcer les attributs important à avoir une valeur.

Une autre manière de forcer un attribut à toujours prendre une valeur est de spécifier une *valeur par défaut* avec l'option `DEFAULT`.

```
CREATE TABLE Cinéma (nom VARCHAR (50) NOT NULL,
                     adresse VARCHAR (50) DEFAULT 'Inconnue')
```

Quand on insérera une ligne dans la table *Cinéma* sans indiquer d'adresse, le système affectera automatiquement la valeur `'Inconnu'` à cet attribut. En général on utilise comme valeur par défaut une constante, sauf pour quelques variables fournies par le système (par exemple `SYSDATE` qui peut indiquer la date du jour).

4.3.3 Contraintes

La création d'une table telle qu'on l'a vue précédemment est extrêmement sommaire car elle n'indique que le contenu de la table sans spécifier les contraintes que doit respecter ce contenu. Or il y a *toujours* des contraintes et il est indispensable de les inclure dans le schéma pour assurer (dans la mesure du possible) l'intégrité de la base.

Voici les règles (ou *contraintes d'intégrité*) que l'on peut demander au système de garantir :

1. Un attribut doit toujours avoir une valeur. C'est la contrainte `NOT NULL` vue précédemment.
2. Un attribut (ou un ensemble d'attributs) constitue(nt) la clé de la relation.
3. Un attribut dans une table est liée à la clé primaire d'une autre table (*intégrité référentielle)*.
4. La valeur d'un attribut doit être unique au sein de la relation.
5. Enfin toute règle s'appliquant à la valeur d'un attribut (min et max par exemple).

Les contraintes sur les clés doivent être systématiquement spécifiées. La dernière (clause `CHECK`) s'appuie en grande partie sur la connaissance du langage d'interrogation de SQL et sera vue ultérieurement.

Clés d'une table

Une *clé* est un attribut (ou un ensemble d'attributs) qui identifie(nt) de manière unique un tuple d'une relation. Il peut y avoir plusieurs clés mais l'une d'entre elles doit être choisie comme *clé primaire*. Ce choix est important : la clé primaire est la clé utilisée pour référencer une ligne et une seule à partir d'autres tables. Il est donc assez délicat de la remettre en cause après coup. En revanche les clés secondaires peuvent être créées ou supprimées beaucoup plus facilement. La clé primaire est spécifiée avec l'option `PRIMARY KEY`.

```
CREATE TABLE Internaute (email VARCHAR (50) NOT NULL,
```

```
                        nom VARCHAR (20) NOT NULL,
                        prenom VARCHAR (20),
                        motDePasse VARCHAR (60) NOT NULL,
                        anneeNaiss INTEGER,
                        PRIMARY KEY (email))
```

Il devrait *toujours* y avoir une `PRIMARY KEY` dans une table pour ne pas risquer d'insérer involontairement deux lignes strictement identiques. Une clé peut être constituée de plusieurs attributs :

```
CREATE TABLE Notation (idFilm INTEGER NOT NULL,
                       email VARCHAR (50) NOT NULL,
                       note INTEGER DEFAULT 0,
                       PRIMARY KEY (titre, email))
```

Tous les attributs figurant dans une clé doivent être déclarés `NOT NULL`. Cela n'a pas vraiment de sens en effet d'identifier des lignes par des valeurs absentes.

On peut également spécifier que la valeur d'un attribut est unique pour l'ensemble de la colonne. Cela permet d'indiquer des *clés secondaires*. On peut par exemple indiquer que deux artistes ne peuvent avoir les mêmes nom et prénom avec l'option `UNIQUE`.

```
CREATE TABLE Artiste(id INTEGER NOT NULL,
                     nom VARCHAR (30) NOT NULL,
                     prenom VARCHAR (30) NOT NULL,
                     anneeNaiss INTEGER,
                     PRIMARY KEY (id),
                     UNIQUE (nom, prenom));
```

Il est facile de supprimer cette contrainte de clé secondaire par la suite. Ce serait beaucoup plus difficile si on avait utilisé la paire `(nom, prenom)` comme clé primaire puisqu'elle serait alors utilisée pour référencer un artiste dans d'autres tables.

Voici un autre exemple d'utilisation d'une clé secondaire : on indique ci-dessous qu'on ne peut pas trouver deux cinémas à la même adresse. Ce deuxième exemple montre que l'on peut placer une contrainte comme `UNIQUE` sur la ligne de l'attribut auquel elle se rapporte. Ce n'est bien entendu possible que quand cette contrainte ne concerne qu'un seul attribut.

```
CREATE TABLE Cinema
       (nom          VARCHAR (30) NOT NULL,
       adresse       VARCHAR(50) UNIQUE,
       PRIMARY KEY (nomCinema))
```

La clause `UNIQUE` ne s'applique pas aux valeurs `NULL` : il peut y avoir plusieurs cinémas d'adresse inconnue. En revanche le nom du cinéma est obligatoire (clause `NOT NULL`) et il est unique (clause `PRIMARY KEY`).

Clés étrangères

La norme SQL ANSI permet d'indiquer quelles sont les clés étrangères dans une table, autrement dit, quels sont les attributs qui font référence à une ligne dans une autre table. On peut spécifier les clés étrangères avec l'option `FOREIGN KEY`.

```
CREATE TABLE Film (idFilm INTEGER NOT NULL,
                   titre VARCHAR (50) NOT NULL,
                   annee INTEGER NOT NULL,
                   idMES INTEGER,
                   codePays INTEGER,
                   PRIMARY KEY (idFilm),
                   FOREIGN KEY (idMES) REFERENCES Artiste,
                   FOREIGN KEY (codePays) REFERENCES Pays);
```

La commande

```
FOREIGN KEY (idMES) REFERENCES Artiste
```

indique que `idMES` référence la clé primaire de la table *Artiste*. Le SGBD vérifiera alors, pour toute modification pouvant affecter le lien entre les deux tables, que la valeur de *idMES* correspond bien à une ligne de *Artiste*. Ces modifications sont :

1. l'insertion dans *Film* avec une valeur inconnue pour `idMES` ;
2. la destruction d'un artiste ;
3. la modification de `id` dans *Artiste* ou de `idMES` dans *Film*.

En d'autres termes le lien entre *Film* et *Artiste* est toujours valide. Cette contrainte est importante pour garantir qu'il n'y a pas de fausse référence dans la base, par exemple qu'un film ne fait pas référence à un artiste qui n'existe pas. Il est beaucoup plus confortable d'écrire une application par la suite quand on sait que les informations sont bien là où elles doivent être.

Il faut noter que l'attribut `idMES` n'est pas déclaré `NOT NULL`, ce qui signifie que l'on s'autorise à ne pas connaître le metteur en scène d'un film. Quand un attribut est à `NULL`, la contrainte d'intégrité référentielle ne s'applique pas.

Que se passe-t-il quand la violation d'une contrainte d'intégrité est détectée par le système ? Par défaut, la mise à jour est rejetée, mais il est possible de demander la répercussion de cette mise à jour de manière à ce que la contrainte soit respectée. Les événements que l'on peut répercuter sont la modification ou la destruction de la ligne référencée, et on les désigne par `ON UPDATE` et `ON DELETE` respectivement. La répercussion elle-même consiste soit à mettre la clé étrangère à `NULL` (option `SET NULL`), soit à appliquer la même opération aux lignes de l'entité composante (option `CASCADE`).

Voici comment on indique que la destruction d'un metteur en scène déclenche la mise à `NULL` de la clé étrangère `idMES` pour tous les films qu'il a réalisé.

```
CREATE TABLE Film (titre VARCHAR (50) NOT NULL,
                   annee INTEGER NOT NULL,
                   idMES INTEGER,
                   codePays INTEGER,
                   PRIMARY KEY (titre),
                   FOREIGN KEY (idMES) REFERENCES Artiste
                        ON DELETE SET NULL,
                   FOREIGN KEY (codePays) REFERENCES Pays);
```

Dans le cas d'une entité faible, on décide en général de détruire le *composant* quand on détruit le *composé*. Par exemple, quand on détruit un cinéma, on veut également détruire les salles ; quand on modifie la clé d'un cinéma, on veut répercuter la modification sur ses salles.

```
CREATE TABLE Salle (nomCinema VARCHAR (30) NOT NULL,
                    no INTEGER NOT NULL,
                    capacite INTEGER,
                    PRIMAR KEY (nomCinema, no),
                    FOREIGN KEY (nomCinema) REFERENCES Cinema
                    ON DELETE CASCADE
                    ON UPDATE CASCADE
                    )
```

Il est important de noter que *nomCinema* fait partie de la clé et ne peut donc pas être `NULL`. On ne pourrait donc pas spécifier ici `ON DELETE SET NULL`.

La spécification des actions `ON DELETE` et `ON UPDATE` simplifie considérablement la gestion de la base par la suite : on n'a plus par exemple à se soucier de détruire les salles quand on détruit un cinéma.

Énumération des valeurs possibles avec `CHECK`

La norme SQL ANSI comprend une option `CHECK (condition)` pour exprimer des contraintes portant soit sur un attribut, soit sur une ligne. La condition elle-même peut être toute expression suivant la clause `WHERE` dans une requête SQL. Les contraintes les plus courantes sont celles consistant à restreindre un attribut à un ensemble de valeurs, comme expliqué ci-dessous. On peut trouver des contraintes arbitrairement complexes, faisant référence à d'autres relations. Nous reviendrons sur cet aspect après avoir étudié le langage d'interrogation SQL.

Voici un exemple simple qui restreint les valeurs possibles des attributs `annee` et `genre` dans la table *Film*.

```
CREATE TABLE Film (titre VARCHAR (50) NOT NULL,
                   annee INTEGER
                      CHECK (annee BETWEEN 1890 AND 2000) NOT NULL,
                   genre VARCHAR (10)
                      CHECK (genre IN ('Histoire','Western','Drame')),
                   idMES INTEGER,
                   codePays INTEGER,
                   PRIMARY KEY (titre),
                   FOREIGN KEY (idMES) REFERENCES Artiste,
                   FOREIGN KEY (codePays) REFERENCES Pays);
```

Au moment d'une insertion dans la table *Film*, ou d'une modification de l'attribut `annee` ou `genre=`, le SGBD vérifie que la valeur insérée dans `genre` appartient à l'ensemble énuméré défini par la clause `CHECK`.

Une autre manière de définir, dans la base, l'ensemble des valeurs autorisées pour un attribut – en d'autres termes, une codification imposée – consiste à placer ces valeurs dans une table et la lier à l'attribut par une contrainte de clé étrangère. C'est ce que nous pouvons faire par exemple pour la table *Pays*.

```
CREATE TABLE Pays (code VARCHAR (4) DEFAULT 0 NOT NULL,
                   nom VARCHAR (30) NOT NULL,
                   langue VARCHAR (30) NOT NULL,
                   PRIMARY KEY (code))

INSERT INTO Pays VALUES (0, 'Inconnu', 'Inconnue');
INSERT INTO Pays VALUES (1, 'France', 'Français');
INSERT INTO Pays VALUES (2, 'USA', 'Anglais');
INSERT INTO Pays VALUES (3, 'Allemagne', 'Allemand');
INSERT INTO Pays VALUES (4, 'Angleterre', 'Anglais');
...
```

Si on ne fait pas de vérification automatique, soit avec `CHECK`, soit avec la commande `FOREIGN KEY`, il faut faire cette vérification dans l'application, ce qui est plus lourd à gérer.

4.3.4 Modification du schéma

La création d'un schéma n'est qu'une première étape dans la vie d'une base de données. On est toujours amené par la suite à créer de nouvelles tables, à ajouter des attributs ou à en modifier la définition. La forme générale de la commande permettant de modifier une table est :

```
ALTER TABLE nomTable ACTION description
```

où *`ACTION`* peut être principalement `ADD`, `MODIFY`, `DROP` ou `RENAME`, et *`description`* est la commande de modification associée à *`ACTION`*. La modification d'une table peut poser des problèmes si elle est incompatible avec le contenu existant. Par exemple passer un attribut à `NOT NULL` implique que cet attribut a déjà des valeurs pour toutes les lignes de la table.

Modification des attributs

Voici quelques exemples d'ajout et de modification d'attributs. On peut ajouter un attribut `region` à la table *Internaute* avec la commande :

```
ALTER TABLE Internaute ADD region VARCHAR(10);
```

S'il existe déjà des données dans la table, la valeur sera à `NULL` ou à la valeur par défaut. La taille de `region` étant certainement insuffisante, on peut l'agrandir avec `MODIFY`, et la déclarer `NOT NULL` par la même occasion :

```
ALTER TABLE Internaute MODIFY region VARCHAR(30) NOT NULL;
```

Il est également possible de diminuer la taille d'une colonne, avec le risque d'une perte d'information pour les données existantes. On peut même changer son type, pour passer par exemple de `VARCHAR` à `INTEGER`, avec un résultat imprévisible.

L'option `ALTER TABLE` permet d'ajouter une valeur par défaut.

```
ALTER TABLE Internaute ALTER region SET DEFAULT 'PACA';
```

Enfin on peut détruire un attribut avec `DROP`.

```
ALTER TABLE Internaute DROP region;
```

Création d'index

Pour compléter le schéma d'une table, on peut définir des *index*. Un index offre un chemin d'accès aux lignes d'une table qui est considérablement plus rapide que le balayage de cette table – du moins quand le nombre de lignes est très élevé. Les SGBDL créent systématiquement un index sur la clé primaire de chaque table. Il y a deux raisons à cela ;

1. l'index permet de vérifier rapidement, au moment d'une insertion, que la clé n'existe pas déjà ;
2. beaucoup de requêtes SQL, notamment celles qui impliquent plusieurs tables (*jointures*), se basent sur les clés des tables pour reconstruire les liens. L'index peut alors être utilisé pour améliorer les temps de réponse.

Un index est également créé pour chaque clause `UNIQUE` utilisée dans la création de la table. On peut de plus créer d'autres index, sur un ou plusieurs attributs, si l'application utilise des critères de recherche autres que les clés primaire ou secondaires.

La commande pour créer un index est la suivante :

```
CREATE [UNIQUE] INDEX nomIndex ON nomTable (attribut1 [, ...])
```

La clause `UNIQUE` indique qu'on ne peut pas trouver deux fois la même clé. La commande ci-dessous crée un index de nom *idxNom* sur les attributs `nom` et `prenom` de la table *Artiste*. Cet index a donc une fonction équivalente à la clause `UNIQUE` déjà utilisée dans la création de la table.

```
CREATE UNIQUE INDEX idxNom ON Artiste (nom, prenom);
```

On peut créer un index, cette fois non unique, sur l'attribut `genre` de la table *Film*.

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

Cet index permettra d'exécuter très rapidement des requêtes SQL ayant comme critère de recherche le genre d'un film.

```
SELECT *
FROM Film
WHERE genre = 'Western'
```

Cela dit il ne faut pas créer des index à tort et à travers, car ils ont un impact négatif sur les commandes d'insertion et de destruction. À chaque fois, il faut en effet mettre à jour tous les index portant sur la table, ce qui représente un coût certain.

4.4 Exercices

Exercice 4.1 *La relation de la figure 4.4 est-elle conforme à la définition ? Si non, citez les anomalies.*

titre	année	metteurEnScène	acteur
'Cyrano'	1992	'Rappeneau'	'Depardieu', 'Perez'
'Les oiseaux'	1963	'Hitchcock'	'Taylor'
'Titanic'		'Cameron'	'DiCaprio'
'Les oiseaux'	1963	'Hitchcock'	'Taylor'

FIG. 4.4 – *Une relation*

Exercice 4.2 *Donnez le schéma relationnel de la base de données « Centre médical » décrite par un schéma E/A dans le précédent TD. Pour chaque table, il faut indiquer précisément, à l'aide de la syntaxe vue en cours :*

- *La clé primaire.*
- *Les clés étrangères.*

Exercice 4.3 *Même exercice que précédemment, pour l'application "Discothèque".*

Exercice 4.4 *Même exercice, portant sur les schémas* `SOCIETE, DIRECTEUR`, `ORDINATEUR, UTILISATEUR`, `ORDINATEUR, DISQUE DUR` *que vous avez étudiés dans la séance consacrée au schéma E/A.*

Cette fois, il est demandé de spécifier, pour chaque clé étrangère, la stratégie en cas de mise-à-jour ou de destruction de la ligne référencée (clauses `ON UPDATE` *et* `ON DELETE` *vues en cours).*

Exercice 4.5 *Même exercice, pour le schéma "Cours". Donner les spécifications complètes (clés primaires et étrangères,* `NOT NULL`*, clauses* `UNIQUE`*, etc).*

Exercice 4.6 *Des éditeurs se réunissent pour créer une Base de Données sur leurs publications scientifiques. Dans de telles publications, plusieurs auteurs se regroupent pour écrire un livre en se répartissant les chapitres à rédiger. Après discussion, voici le schéma obtenu :*

1. *Livre (**titreLivre**, année, éditeur, chiffreAffaire)*
2. *Chapitre (**titreLivre, titreChapitre**, nbPages)*
3. *Auteur (**nom**, prénom, annéeNaissance)*
4. *Redaction (**nomAuteur, titreLivre, titreChapitre**)*

Les clés primaires sont en gras, mais les clés étrangères ne sont pas signalées.

1. *Donnez le schéma Entité/Association correspondant au schéma relationnel.*
2. *Donnez les ordres* `CREATE TABLE` *pour le schéma, en spécifiant soigneusement clés primaires et étrangères avec la syntaxe SQL2. Le type des données est secondaire: choisissez ce qui vous semble logique.*
3. *Sur quelles clés étrangères devrait-on mettre l'option* `ON DELETE CASCADE` *?*

Exercice 4.7 *On trouve dans un SGBD relationnel les relations ci-dessous. Les clés primaires sont en gras, les clés étrangères ne sont pas signalées.*

- *Immeuble (**nom**, adresse, nbEtages, annéeConstruction, nomGérant)*
- *Appart (**nomImm, noApp**, type, superficie, étage)*
- *Personne (**nom**, prenom, age, codeProfession)*
- *Occupant (**nomImm, noApp, nomOccupant**, annéeArrivée)*
- *Propriété (**nomImm, nomPropriétaire**, quotePart)*
- *TypeAppart (**code**, libellé)*
- *Profession (**code**, libellé)*

Identifier les clés étrangères dans chaque relation, et reconstruire le schéma E/A.

Chapitre 5

L'algèbre relationnelle

Sommaire

Le premier langage étudié dans ce cours est *l'algèbre relationnelle*. Il consiste en un ensemble d'opérations qui permettent de manipuler des *relations*, considérées comme des ensemble de tuples : on peut ainsi faire *l'union* ou la *différence* de deux relations, *sélectionner* une partie de la relation, effectuer des *produits cartésiens* ou des *projections*, etc.

Une propriété fondamentale de chaque opération est qu'elle prend une ou deux relations en entrée, et produit une relation en sortie. Cette propriété permet de *composer* des opérations : on peut appliquer une sélection au résultat d'un produit cartésien, puis une projection au résultat de la sélection et ainsi de suite. En fait on peut construire des *expressions algébriques* arbitrairement complexes qui permettent d'exprimer des manipulations sur un grand nombre de relations.

Une *requête* est une expression algébrique qui s'applique à un ensemble de relations (la base de données) et produit une relation finale (le résultat de la requête). On peut voir l'algèbre relationnelle comme un langage de programmation très simple qui permet d'exprimer des requêtes sur une base de données relationnelle.

Dans tout ce chapitre on va prendre l'exemple de la (petite) base de données d'un organisme de voyage. Cet organisme propose des séjours (sportifs, culturels, etc) se déroulant dans des stations de vacances. Chaque station propose un ensemble d'activtés (ski, voile, tourisme). Enfin on maintient une liste des clients (avec le solde de leur compte !) et des séjours auxquels ils ont participé avec leurs dates de début et de fin.

- Station (**nomStation**, capacité, lieu, région, tarif)
- Activite (***nomStation***, **libellé**, prix)
- Client (**id**, nom, prénom, ville, région, solde)
- Séjour (***idClient***, **station**, **début**, nbPlaces)

nomStation	capacité	lieu	région	tarif
Venusa	350	Guadeloupe	Antilles	1200
Farniente	200	Seychelles	Océan Indien	1500
Santalba	150	Martinique	Antilles	2000
Passac	400	Alpes	Europe	1000

La table *Station*

NomStation	Libellé	Prix
Venusa	Voile	150
Venusa	Plongée	120
Farniente	Plongée	130
Passac	Ski	200
Passac	Piscine	20
Santalba	Kayac	50

La table *Activité*

FIG. 5.1 – *Les stations et leurs activités*

id	nom	prénom	ville	région	solde
10	Fogg	Phileas	Londres	Europe	12465
20	Pascal	Blaise	Paris	Europe	6763
30	Kerouac	Jack	New York	Amérique	9812

La table *Client*

idClient	station	début	nbPlaces
10	Passac	1998-07-01	2
30	Santalba	1996-08-14	5
20	Santalba	1998-08-03	4
30	Passac	1998-08-15	3
30	Venusa	1998-08-03	3
20	Venusa	1998-08-03	6
30	Farniente	1999-06-24	5
10	Farniente	1998-09-05	3

La table *Séjour*

FIG. 5.2 – *Les clients et leurs séjours*

5.1 Les opérateurs de l'algèbre relationnelle

L'algèbre se compose d'un ensemble d'opérateurs, parmi lesquels 5 sont nécessaires et suffisants et permettent de définir les autres par composition. Ce sont :

1. La sélection, dénotée σ ;
2. La projection, dénotée π ;
3. Le produit cartésien, dénoté $\times$;
4. L'union, $\cup$;
5. La différence, $-$.

Les deux premiers sont des opérateurs *unaires* (ils prennent en entrée une seule relation) et les autres sont des opérateurs *binaires*. A partir de ces opérateurs il est possible d'en définir d'autres, et notamment la *jointure*, $\bowtie$, qui est la composition d'un produit cartésien et d'une sélection.

Ces opérateurs sont maintenant présentés tour à tour.

5.1.1 La sélection, σ

La sélection $\sigma_F(R)$ s'applique à une relation, R, et extrait de cette relation les tuples qui satisfont un critère de sélection, F. Ce critère peut être :

- La comparaison entre un attribut de la relation, A, et une constante a. Cette comparaison s'écrit $A\Theta a$, où Θ appartient à $\{=, <, >, \leqslant, \geqslant\}$.
- La comparaison entre deux attributs A_1 et A_2, qui s'écrit $A_1\Theta A_2$ avec les mêmes opérateurs de comparaison que précédemment.

Premier exemple : exprimer la requête qui donne toutes les stations aux Antilles.

$$\sigma_{region='Antilles'}(Station)$$

On obtient donc le résultat :

nomStation	capacité	lieu	région	tarif
Venusa	350	Guadeloupe	Antilles	1200
Santalba	150	Martinique	Antilles	2000

La sélection a pour effet de supprimer des lignes, mais chaque ligne garde l'ensemble de ses attributs.

5.1.2 La projection, π

La projection $\pi_{A_1,A_2,...,A_k}(R)$ s'applique à une relation R, et ne garde que les attributs $A_1, A_2, \ldots A_k$. Donc, contrairement à la sélection, on ne supprime pas des lignes mais des colonnes.

Exemple : donner le nom des stations, et leur région.

$$\pi_{nomStation,region}(Station)$$

On obtient le résultat suivant, après suppression des colonnes `capacité`, `lieu` et `tarif` :

nomStation	région
Venusa	Antilles
Farniente	Océan Indien
Santalba	Antilles
Passac	Europe

En principe le nombre de lignes dans le résultat est le même que dans la relation initiale. Il y a cependant une petite subtilité : comme le résultat est une relation, il ne peut pas y avoir deux lignes identiques (il n'y a pas deux fois le même élément dans un ensemble). Il peut arriver qu'après une projection, deux lignes qui étaient distinctes initialement se retrouvent identiques, justement parce ce que l'attribut qui permettait de les distinguer a été supprimé. Dans ce cas on ne conserve qu'une seule des deux (ou plus) lignes identiques.

Exemple : on souhaite connaître toutes les régions où il y a des stations. On exprime cette requête par :

$$\pi_{region}(Station)$$

et on obtient :

région
Antilles
Océan Indien
Europe

La ligne 'Antilles' était présente deux fois dans la relation *Station*, et n'apparaît plus qu'en un seul exemplaire dans le résultat.

5.1.3 Le produit cartésien, $\times$

Le premier opérateur binaire, et le plus important, est le produit cartésien, $\times$. Le produit cartésien entre deux relations R et S se note $R \times S$, et permet de créer une nouvelle relation où chaque tuple de R est associé à chaque tuple de S.

Voici deux relations R et S :

A	B
a	b
x	y

C	D
c	d
u	v
x	y

Et voici le résultat de $R \times S$:

A	B	C	D
a	b	c	d
a	b	u	v
a	b	x	y
x	y	c	d
x	y	u	v
x	y	x	y

Le nombre de lignes dans le résultat est exactement $|R| \times |S|$ ($|R|$ dénote le nombre de lignes dans la relation R).

En lui-même, le produit cartésien ne présente pas un grand intérêt puisqu'il associe aveuglément chaque ligne de R à chaque ligne de S. Il ne prend vraiment son sens qu'associé à l'opération de sélection, ce qui permet d'exprimer des *jointures*, opération fondamentale qui sera détaillée plus loin.

Conflits de noms d'attributs

Quand les schémas des relations R et S sont complêtement distincts, il n'y a pas d'ambiguité sur la provenance des colonnes dans le résultat. Par exemple on sait que les valeurs de la colonne A dans $R \times S$ viennent de la relation R. Il peut arriver (il arrive de fait très souvent) que les deux relations aient des attributs qui ont le même nom. On doit alors se donner les moyens de distinguer l'origine des colonnes dans la table résultat en donnant un nom distinct à chaque attribut.

Voici par exemple une table T qui a les mêmes noms d'attributs que R. Le schéma du résultat du produit cartésien $R \times T$ a pour schéma (A, B, A, B) et présente donc des ambiguités, avec les colonnes A et B en double.

A	B
m	n
o	p

La table T

La première solution pour lever l'ambiguité est de préfixer un attribut par le nom de la table d'où il provient. Le résultat de $R \times T$ devient alors :

R.A	R.B	T.A	T.B
a	b	m	n
a	b	n	p
x	y	m	n
x	y	n	p

Au lieu d'utiliser le nom de la relation en entier, on peut s'autoriser tout synonyme qui permet de lever l'ambiguité. Par exemple, dans le produit cartésien `Station × Activite`, on peut utiliser `S` comme préfixe pour les attributs venant de `Station`, et `A` pour ceux venant d'`Activite`. Le résultat peut alors être représenté comme sur la Fig. 5.3. On lève l'ambiguité sur les attributs `nomStation` qui apparaissent deux fois. Ce n'est pas nécessaire pour les autres attributs.

S.nomStation	cap.	lieu	région	tarif	A.nomStation	libelle	prix
Venusa	350	Guadeloupe	Antilles	1200	Venusa	Voile	150
Venusa	350	Guadeloupe	Antilles	1200	Venusa	Plongée	120
Venusa	350	Guadeloupe	Antilles	1200	Farniente	Plongée	130
Venusa	350	Guadeloupe	Antilles	1200	Passac	Ski	200
Venusa	350	Guadeloupe	Antilles	1200	Passac	Piscine	20
Venusa	350	Guadeloupe	Antilles	1200	Santalba	Kayac	50
Farniente	200	Seychelles	Océan Indien	1500	Venusa	Voile	150
Farniente	200	Seychelles	Océan Indien	1500	Venusa	Plongée	120
Farniente	200	Seychelles	Océan Indien	1500	Farniente	Plongée	130
Farniente	200	Seychelles	Océan Indien	1500	Passac	Ski	200
Farniente	200	Seychelles	Océan Indien	1500	Passac	Piscine	20
Farniente	200	Seychelles	Océan Indien	1500	Santalba	Kayac	50
Santalba	150	Martinique	Antilles	2000	Venusa	Voile	150
Santalba	150	Martinique	Antilles	2000	Venusa	Plongée	120
Santalba	150	Martinique	Antilles	2000	Farniente	Plongée	130
Santalba	150	Martinique	Antilles	2000	Passac	Ski	200
Santalba	150	Martinique	Antilles	2000	Passac	Piscine	20
Santalba	150	Martinique	Antilles	2000	Santalba	Kayac	50
Passac	400	Alpes	Europe	1000	Venusa	Voile	150
Passac	400	Alpes	Europe	1000	Venusa	Plongée	120
Passac	400	Alpes	Europe	1000	Farniente	Plongée	130
Passac	400	Alpes	Europe	1000	Passac	Ski	200
Passac	400	Alpes	Europe	1000	Passac	Piscine	20
Passac	400	Alpes	Europe	1000	Santalba	Kayac	50

FIG. 5.3 – *Produit cartésien* Station × Activité*, avec levée des ambiguïtés*

Renommage

Il existe une deuxième possibilité pour résoudre les conflits de noms : le *renommage*. Il s'agit d'un opérateur particulier, dénoté ρ, qui permet de renommer un ou plusieurs attributs d'une relation. L'expression $\rho_{A\to C,B\to D}(T)$ permet ainsi de renommer A en C et B en D dans la relation T. Le produit cartésien

$$R \times \rho_{A\to C,B\to D}(T)$$

ne présente alors plus d'ambiguités. Le renommage est une solution très générale, mais asez lourde à utiliser

Il est tout à fait possible de faire le produit cartésien d'une relation avec elle-même. Dans ce cas le renommage où l'utilisation d'un préfixe distinctif est impératif. Voici par exemple le résultat de $R \times R$, dans lequel on préfixe par $R1$ et $R2$ respectivement les attributs venant de chacune des opérandes.

R1.A	R1.B	R2.A	R2.B
a	b	a	b
a	b	x	y
x	y	a	b
x	y	x	y

5.1.4 L'union, ∪

Il existe deux autres opérateurs binaires, qui sont à la fois plus simples et moins fréquemment utilisés. Le premier est l'union. L'expression $R \cup S$ crée une relation comprenant tous les tuples existant dans l'une ou l'autre des relations R et S. Il existe une condition impérative : *les deux relations doivent avoir le même schéma*, c'est-à-dire même nombre d'attributs, mêmes noms et mêmes types.

L'union des relations $R(A,B)$ et $S(C,D)$ données en exemple ci-dessus est donc interdite (on ne saurait pas comment nommer les attributs dans le résultat). En revanche, en posant $S' = \rho_{C\to A, D\to B}(S)$, il devient possible de calculer $R \cup S'$, avec le résultat suivant :

A	B
a	b
x	y
c	d
u	v

Comme pour la projection, il faut penser à éviter les doublons. Donc le tuple `(x,y)` qui existe à la fois dans R et dans S' ne figure qu'une seule fois dans le résultat.

5.1.5 La différence, $-$

Comme l'union, la différence s'applique à deux relations qui ont le même schéma. L'expression $R - S$ a alors pour résultat tous les tuples de R qui ne sont pas dans S.

Voici la différence de R et S', les deux relations étant définies comme précédemment.

A	B
a	b

La différence est le seul opérateur qui permet d'exprimer des requêtes comportant une négation (on veut 'rejeter' quelque chose, on 'ne veut pas' des lignes ayant telle propriété). Il s'agit d'une fonctionnalité importante et difficile à manier : elle sera détaillée plus loin.

5.1.6 Jointure, $\bowtie$

Toutes les requêtes exprimables avec l'algèbre relationnelle peuvent se construire avec les 5 opérateurs présentés ci-dessus. En principe, on pourrait donc s'en contenter. En pratique, il existe d'autres opérations, très couramment utilisées, qui peuvent se contruire par composition des opérations de base. La plus importante est la jointure.

Afin de comprendre l'intérêt de cet opérateur, regardons à nouveau la Fig. 5.3 qui représente le produit cartésien `Station × Activite`. Le résultat est énorme et comprend manifestement un grand nombre de lignes qui ne nous intéressent pas. Cela ne présente pas beaucoup de sens de rapprocher des informations sur Santalba, aux Antilles et sur l'activité de ski à Passac.

Si, en revanche, on considère le produit cartésien comme un *résultat intermédiaire*, on voit qu'il devient maintenant possible, avec une simple sélection, de rapprocher les informations générales sur une station et la liste des activités de cette station.

La sélection est la suivante :

$$\sigma_{S.nomStation=A.nomStation}(Station \times Activite)$$

Et on obtient le résultat de la Fig. 5.4.

On a donc effectué une *composition* de deux opérations (un produit cartésien, une sélection) afin de rapprocher des informations réparties dans plusieurs tables, mais ayant des liens entre elles (toutes les informations dans un tuple du résultat sont relatives à une seule station). Cette opération est une *jointure*, que l'on peut directement, et simplement, noter :

$$Station \bowtie_{nomStation=nomStation} Activite$$

La jointure consiste donc à rapprocher les lignes de deux relations pour lesquelles les valeurs d'un (ou plusieurs) attributs sont identiques. De fait, dans 90% des cas, ces attributs communs sont (1) la clé primaire d'une des relations et (2) la clé étrangère dans l'autre relation. Dans l'exemple ci-dessus, c'est le cas pour `nomStation`. La jointure permet alors de *reconstruire* l'association entre `Station` et `Activite`.

S.nomStation	cap.	lieu	région	tarif	A.nomStation	libelle	prix
Venusa	350	Guadeloupe	Antilles	1200	Venusa	Voile	150
Venusa	350	Guadeloupe	Antilles	1200	Venusa	Plongée	120
Farniente	200	Seychelles	Océan Indien	1500	Farniente	Plongée	130
Santalba	150	Martinique	Antilles	2000	Santalba	Kayac	50
Passac	400	Alpes	Europe	1000	Passac	Ski	200
Passac	400	Alpes	Europe	1000	Passac	Piscine	20

FIG. 5.4 – *Une jointure* $\sigma_{S.nomStation=A.nomStation}(Station \times Activite)$*, composition de* $\times$ *et* σ

Une jointure $R \bowtie_F S$ peut simplement être définie comme étant équivalent à $\sigma_F(R \times S)$. Le critère de rapprochement, F, peut être n'importe quelle opération de comparaison liant un attribut de R à un attribut de S. En pratique, on emploie peu les $\neq$ ou '$<$' qui sont difficiles à interpréter, et on effectue des égalités.

Remarque : Si on n'exprime pas de critère de rapprochement, la jointure est équivalente à un produit cartésien.

Il faut être attentif aux ambiguités dans le nommage des attributs qui peut survenir dans la jointure au même titre que dans le produit cartésien. Les solutions à employer sont les mêmes : on préfixe par le nom de la relation ou par un synonyme clair, ou bien on renomme des attributs avant d'effectuer la jointure.

5.2 Expression de requêtes avec l'algèbre

Cette section est consacrée à l'expression de requêtes algébriques complexes impliquant plusieurs opérateurs. On utilise la *composition* des opérations, rendue possible par le fait que tout opérateur produit en sortie une relation sur laquelle on peut appliquer à nouveau des opérateurs.

5.2.1 Sélection généralisée

Regardons d'abord comment on peut généraliser les critères de sélection de l'opérateur σ. Jusqu'à présent on a vu comment sélectionner des lignes satisfaisant *un* critère de sélection, par exemple : 'les stations aux Antilles'. Maintenant supposons que l'on veuille retrouver les stations qui sont aux Antilles *et* dont la capacité est supérieure à 200. On peut exprimer cette requête par une composition :

$$\sigma_{capacite>200}(\sigma_{region='Antilles'}(Station))$$

Ce qui revient à pouvoir exprimer une sélection avec une *conjonction* de critères. La requête précédente est donc équivalente à celle ci-dessous, où le '$\wedge$' dénote le 'et' logique.

$$\sigma_{capacite>200 \wedge region='Antilles'}(Station)$$

Donc la composition de plusieurs sélections permet d'exprimer une conjonction de critères de recherche. De même la composition de la sélection et de l'union permet d'exprimer la *disjonction*. Voici la requête qui recherche les stations qui sont aux Antilles, *ou* dont la capacité est supérieure à 200.

$$\sigma_{capacite>200}(Station) \cup \sigma_{region='Antilles'}(Station)$$

Ce qui permet de s'autoriser la syntaxe suivante, où le '$\vee$' dénote le 'ou' logique.

$$\sigma_{capacite>200 \vee region='Antilles'}(Station)$$

Enfin la *différence* permet d'exprimer la *négation* et "d'éliminer" des lignes. Par exemple, voici la requête qui sélectionne les stations dont la capacité est supérieure à 200 mais qui ne sont *pas* aux Antilles.

$$\sigma_{capacite>200}(Station) - \sigma_{region='Antilles'}(Station)$$

Cette requête est équivalente à une sélection où on s'autorise l'opérateur '$\neq$' :

$$\sigma_{capacite>200 \wedge region\neq'Antilles'}(Station)$$

En résumé, les opérateurs d'union et de différence permettent de définir une sélection σ_F où le critère F est une expression booléenne quelconque. Attention cependant : si toute sélection avec un 'ou' peut s'exprimer par une union, l'inverse n'est pas vrai (exercice).

5.2.2 Requêtes conjonctives

Les requêtes dites *conjonctives* constituent l'essentiel des requêtes courantes. Intuitivement, il s'agit de toutes les recherches qui s'expriment avec des 'et', par opposition à celles qui impliquent des 'ou' ou des 'not'. Dans l'algèbre, ces requêtes sont toutes celles qui peuvent s'écrire avec seulement trois opérateurs : $\pi, \sigma, \times$ (et donc, indirectement, $\bowtie$).

Les plus simples sont celles où on n'utilise que π et σ. En voici quelques exemples.

1. Nom des stations aux Antilles : $\pi_{nomStation}(\sigma_{region='Antilles'}(Station))$
2. Nom des stations où l'on pratique la voile. $\pi_{nomStation}(\sigma_{libelle='Voile'}(Activite))$
3. Nom et prénom des clients européens $\pi_{nom,prenom}(\sigma_{region='Europe'}(Client))$

Des requêtes légèrement plus complexes - et extrêmement utiles - sont celles qui impliquent la jointure (le produit cartésien ne présente d'intérêt que quand il est associé à la sélection). On doit utiliser la jointure dès que les attributs nécessaires pour évaluer une requête sont réparties dans au moins deux tables. Ces "attributs nécessaires" peuvent être :

– Soit des attributs qui figurent dans le résultat ;

– Soit des attributs sur lesquels on exprime un critère de sélection.

Considérons par exemple la requête suivante : "Donner le nom et la région des stations où l'on pratique la voile". Une analyse très simple suffit pour constater que l'on a besoin des attributs `région` qui apparaît dans la relation `Station`, `libelle` qui apparaît dans `Activite`, et `nomStation` qui apparaît dans les deux relations.

Donc il faut faire une jointure, de manière à rapprocher les lignes de `Station` et de `Activité`. Il reste donc à déterminer le (ou les) attribut(s) sur lesquels se fait ce rapprochement. Ici, comme dans la plupart des cas, la jointure permet de "recalculer" l'association entre les relations `Station` et `Activité`. Elle s'effectue donc sur l'attribut `nomStation` qui permet de représenter cette association.

$$\pi_{nomStation,region}(Station \bowtie_{nomStation=nomStation} \sigma_{libelle='Voile'}(Activite))$$

En pratique, la grande majorité des opérations de jointure s'effectue sur des attributs qui sont clé primaire dans une relation, et clé secondaire dans l'autre. Il ne s'agit pas d'une règle absolue, mais elle résulte du fait que la jointure permet le plus souvent de reconstituer le lien entre des informations qui sont naturellement associées (comme une station et ses activités, ou une station et ses clients), mais qui ont été réparties dans plusieurs relations au moment de la modélisation logique de la base. Cette reconstitution s'appuie sur le mécanisme de clé étrangère qui a été étudié dans le chapitre consacré à la conception.

Voici quelques autres exemples qui illustrent cet état de fait :

– Nom des clients qui sont allés à Passac :

$$\pi_{nom}(Client \bowtie_{id=idClient} \sigma_{nomStation='Passac'}(Sejour))$$

– Quelles régions a visité le client 30 :

$$\pi_{region}(\sigma_{idClient=30}(Sejour) \bowtie_{station=nomStation} (Station))$$

– Nom des clients qui ont eu l'occasion de faire de la voile :

$$\pi_{nom}(Client \bowtie_{id=idClient} (Sejour \bowtie_{station=nomStation} \sigma_{libelle='Voile'}(Activite)))$$

La dernière requête comprend deux jointures, portant à chaque fois sur des clés primaires et/ou étrangères. Encore une fois ce sont les clés qui définissent les liens entre les relations, et elle servent donc naturellement de support à l'expression des requêtes.

Voici maintenant un exemple qui montre que cette règle n'est pas systématique. On veut exprimer la requête qui recherche les noms des clients qui sont partis en vacances dans leur région, ainsi que le nom de cette région.

Ici on a besoin des informations réparties dans les relations `Station`, `Sejour` et `Client`. Voici l'expression algébrique :

$$\pi_{nom,client.region}(Client \bowtie_{id=idClient \wedge region=region} (Sejour \bowtie_{station=nomStation} Station))$$

Les jointures avec la table `Sejour` se font sur les couples (clé primaire, clé étrangère), mais on a en plus un critère de rapprochement relatif à l'attribut `région` de `Client` et de `Station`. Noter que dans la projection finale, on utilise la notation `client.region` pour éviter toute ambiguité.

5.2.3 Requêtes avec $\cup$ et $-$

Pour finir, voici quelques exemples de requêtes impliquant les deux opérateurs $\cup$ et $-$. Leur utilisation est moins fréquente, mais elle peut s'avérer absolument nécessaire puisque ni l'un ni l'autre ne peuvent s'exprimer à l'aide des trois opérateurs "conjonctifs" étudiés précédemment. En particulier, la différence permet d'exprimer toutes les requêtes où figure une négation : on veut sélectionner des données qui *ne* satisfont *pas* telle propriété, ou tous les "untels" *sauf* les 'x' et les 'y', etc.

Illlustration concrète sur la base de données avec la requête suivante : quelles sont les stations qui *ne* proposent *pas* de voile ?

$$\pi_{nomStation}(Station) - \pi_{nomStation}(\sigma_{libelle='Voile'}(Activite))$$

Comme le suggère cet exemple, la démarche générale pour construire une requête du type 'Tous les O qui ne satisfont pas la propriété p" est la suivante :

1. Construire une première requête A qui sélectionne tous les O.
2. Construire une deuxième requête B qui sélectionne tous les O *qui satisfont* p.
3. Finalement, faire $A - B$.

Les requêtes A et B peuvent bien entendu être arbitrairement complexes et mettre en oeuvre des jointures, des sélections, etc. La seule contrainte est que le résultat de A et de B comprenne le même nombre d'attributs.

Voici quelques exemples complémentaires qui illustrent ce principe.

– Régions où il y a des clients, mais pas de station.

$$\pi_{region}(Client) - \pi_{region}(Station)$$

– Nom des stations qui n'ont pas reçu de client américain.

$$\pi_{nomStation}(Station) - \pi_{station}(Sejour \bowtie_{idClient=id} \sigma_{region='Amerique'}(Client))$$

– Id des clients qui ne sont pas allés aux Antilles.

$$\pi_{idClient}(Client) - \pi_{idClient}(\sigma_{region='Antilles'}(Station) \bowtie_{nomStation=station} Sejour)$$

La dernière requête construit l'ensemble des `idClient` pour les clients qui ne sont pas allés aux Antilles. Pour obtenir le nom de ces clients, il suffit d'ajouter une jointure (exercice).

Complément d'un ensemble

La différence peut être employée pour calculer le *complément* d'un ensemble. Prenons l'exemple suivant : on veut les ids des clients *et* les stations où ils ne sont pas allés. En d'autres termes, parmi toutes les associations Client/Station possibles, on veut justement celles qui *ne sont pas* représentées dans la base !

C'est un des rares cas où le produit cartésien seul est utile : il permet justement de constituer "toutes les associations possibles". Il reste ensuite à en soustraire celles qui sont dans la base avec l'opérateur $-$.

$$(\pi_{id}(Client) \times \pi_{nomStation}(Station)) - \pi_{idClient,station}(Sejour)$$

Quantification universelle

Enfin la différence est nécessaire pour les requêtes qui font appel à la quantification universelle : celles où l'on demande par exemple qu'une propriété soit *toujours* vraie. A priori, on ne voit pas pourquoi la différence peut être utile dans de tels cas. Cela résulte simplement de l'équivalence suivante : une propriété est vraie pour *tous* les éléments d'un ensemble si et seulement si *il n'existe pas* un élément de cet ensemble pour lequel la propriété est fausse.

En pratique, on se ramène toujours à la seconde forme pour exprimer des requêtes : donc on emploie toujours la négation et la quantification existentielle à la place de la quantification universelle.

Par exemple : quelles sont les stations dont *toutes* les activités ont un prix supérieur à 100 ? On l'exprime également par 'quelles sont stations pour lesquelles *il n'existe pas* d'activité avec un prix *inférieur* à 100'. Ce qui donne l'expression suivante :

$$\pi_{nomStation}(Station) - \pi_{nomStation}(\sigma_{prix<100}(Activite))$$

Pour finir, voici une des requêtes les plus complexes, la *division*. L'énoncé (en français) est simple, mais l'expression algébrique ne l'est pas du tout. L'exemple est le suivant : on veut les ids des clients qui sont allés dans *toutes* les stations.

Traduit avec négation et quantification existentielle, cela donne : les ids des clients tels *qu'il n'existe pas* de station où ils *ne soient pas* allés. On constate une double négation, ce qui donne l'expression algébrique suivante :

$$\pi_{id}(Client) - \pi_{id}((\pi_{id}(Client) \times \pi_{nomStation}(Station)) - \pi_{idClient,station}(Sejour))$$

On réutilise l'expression donnant les clients et les stations où ils ne sont pas allés (voir plus haut) : on obtient un ensemble B. Il reste à prendre tous les clients, sauf ceux qui sont dans B. Ce type de requête est rare (heureusement) mais illustre la capacité de l'algèbre à exprimer par de simples manipulations ensemblistes des opérations complexes.

5.3 Exercices

Exercice 5.1 *La figure 5.5 donne une instance de la base "Immeubles" (le schéma est légèrement simplifié).*

Pour chacune des requêtes suivantes, exprimez en français sa signification, et donnez son résultat.

1. $\pi_{nom,age}(Personne)$

2. $\pi_{nomImm}(Immeuble)$
3. $\pi_{nomImm,noApp}(\sigma_{superficie>150}(Appart))$
4. $\pi_{nomOccupant}(\sigma_{nomImm='Barabas' \vee anneeArrivee>1994}(Occupant))$
5. $\pi_{nomImm,noApp}(\sigma_{noApp=etage}(appart))$

6. $\pi_{nomGerant,superficie}(Immeuble \bowtie_{nomImm=nomImm} Appart)$
7. $\pi_{nomOccupant,anneeArrivee,superficie}(Appart \bowtie_{nomImm=nomImm \wedge noApp=noApp} Occupant)$
8. $\pi_{profession}(Immeuble \bowtie_{nomGerant=nom} Personne)$
9. $\pi_{nomGerant}(Immeuble \bowtie_{nomGerant=nomOccupant \wedge nomImm=nomImm} Occupant)$
10. $\pi_{superficie}(\sigma_{nomOccupant='Rachel'}(Occupant) \bowtie_{nomImm=nomImm \wedge noApp=noApp} Appart)$

11. $\pi_{nomImm,noApp}(Appart) - \pi_{nomImm,noApp}(Occupant)$
12. $\pi_{nomImm}(Immeuble) - \pi_{nomImm}(\sigma_{nomOccupant='Doug'}(Occupant))$.
13. $\pi_{nomImm}(\sigma_{nomOccupant\neq'Doug'}(Occupant))$.

Quelle est la différence entre les deux dernières requêtes ?

Exercice 5.2 *Exprimez les requêtes suivantes, en algèbre relationnelle. Pour chaque requête, donnez le résultat sur la base "Immeubles".*

1. *Nom des immeubles ayant strictement plus de 10 étages.*
2. *Nom des personnes ayant emménagé avant 1994.*
3. *Qui habite le Koudalou ?*
4. *Nom des informaticiens de plus de 25 ans.*
5. *Nom des immeubles ayant un appartement de plus de 150 m^2.*
6. *Qui gère l'appartement où habite Rachel ?*
7. *Dans quel immeuble habite un acteur ?*
8. *Qui habite un appartement de moins de 70 m^2 ?*
9. *Nom des personnes qui habitent au dernier étage de leur immeuble.*
10. *Qui a emménagé au moins 20 ans après la construction de son immeuble ?*
11. *Profession du gérant du Barabas ?*
12. *Couples de personnes ayant emménagé dans le même immeuble la même année.*
13. *Age et profession des occupants de l'immeuble géré par Ross ?*
14. *Qui habite, dans un immeuble de plus de 10 étages, un appartement de plus de 100 m^2 ?*
15. *Couples de personnes habitant, dans le même immeuble, un appartement de même superficie.*
16. *Qui n'habite pas un appartement géré par Ross ?*

nomImm	adresse	nbEtages	annéeConstruction	nomGérant
Koudalou	3 Rue Blanche	15	1975	Doug
Barabas	2 Allee Nikos	2	1973	Ross

La table *Immeuble*

nomImm	noApp	superficie	étage
Koudalou	1	150	14
Koudalou	34	50	15
Koudalou	51	200	2
Koudalou	52	50	5
Barabas	1	250	1
Barabas	2	250	2

La table *Appart*

nom	âge	profession
Ross	51	Informaticien
Alice	34	Cadre
Rachel	23	Stagiaire
William	52	Acteur
Doug	34	Rentier

La table *Personne*

nomImm	noApp	nomOcc	anneeArrivee
Koudalou	1	Rachel	1992
Barabas	1	Doug	1994
Barabas	2	Ross	1994
Koudalou	51	William	1996
Koudalou	34	Alice	1993

La table *Occupant*

FIG. 5.5 – *Les immeubles et leurs occupants*

17. *Qui n'habite pas un appartement qu'il gère lui-même ?*
18. *Quels sont les immeubles où personne n'a emménagé en 1996 ?*
19. *Quels sont les immeubles où tout le monde a emménagé en 1994 ?*

Exercice 5.3

1. *Montrez que la requête 4 dans l'exercice 5.1 peut s'exprimer avec l'union.*
2. *Inversement, donner une requête sur la base 'Immeuble' qui peut s'exprimer avec l'union, mais pas avec σ et $\vee$.*
3. *Exprimez la dernière requête de l'exercice 5.1 avec la différence, et sans $\neq$.*

Chapitre 6

Le langage SQL

Sommaire

Ce chapitre présente le langage SQL d'interrogation et de manipulation de données (insertion, mise-à-jour, destruction). La syntaxe est celle de la norme SQL2, implantée plus ou moins complètement dans la plupart des principaux SGBDR. Certaines fonctionnalités (*triggers* par exemple) sont en cours de normalisation dans SQL3, mais existent déjà dans quelques systèmes en raison de leur importance. Ils seront présentés brièvement.

SQL est un langage *déclaratif* qui permet d'interroger une base de données sans se soucier de la représentation interne (physique) des données, de leur localisation, des chemins d'accès ou des algorithmes nécessaires. A ce titre il s'adresse à une large communauté d'utilisateurs potentiels (pas seulement des informaticiens) et constitue un des atouts les plus spectaculaires (et le plus connu) des SGBDR. On peut l'utiliser de manière interactive, mais également en association avec des interfaces graphiques, des outils de *reporting* ou, très généralement, des langages de programmation.

Ce dernier aspect est très important en pratique car SQL ne permet pas de faire de la programmation au sens courant du terme et doit donc être associé avec un langage comme le C, le COBOL ou JAVA pour réaliser des traitements complexes accédant à une base de données. L'interface de SQL et du langage C sera présentée dans le chapitre 8.

6.1 Requêtes simples SQL

Dans tout ce chapitre on va prendre l'exemple de la (petite) base de données présentée dans le chapitre sur l'algèbre.

6.1.1 Sélections simples

Commençons par les requêtes les plus simples : la figure 5.1 montre une instance de la base pour les relations `Station` et `Activite`. Première requête : on souhaite extraire de la base le nom de toutes les stations se trouvant aux Antilles.

```
SELECT nomStation
FROM   Station
WHERE  region = 'Antilles'
```

Ce premier exemple montre la structure de base d'une requête SQL, avec les trois clauses `SELECT`, `FROM` et `WHERE`.

- `FROM` indique la (ou les) tables dans lesquelles on trouve les attributs utiles à la requête. Un attribut peut être 'utile' de deux manières (non exclusives) : (1) on souhaite afficher son contenu, (2) on souhaite qu'il ait une valeur particulière (une constante ou la valeur d'un autre attribut).
- `SELECT` indique la liste des attributs constituant le résultat.
- `WHERE` indique les conditions que doivent satisfaire les n-uplets de la base pour faire partie du résultat.

Dans l'exemple précédent, l'interprétation est simple : on parcourt les n-uplets de la relation `Station`. Pour chaque n-uplet, si l'attribut `region` a pour valeur 'Antilles', on place l'attribut `nomStation` dans le résultat [1]. On obtient donc le résultat :

nomStation
Venusa
Santalba

Le résultat d'un ordre SQL est toujours une relation (une table) dont les attributs sont ceux spécifiés dans la clause `SELECT`. On peut donc considérer en première approche ce résultat comme un 'découpage', horizontal et vertical, de la table indiquée dans le `FROM`, similaire à une utilisation combinée de la sélection et de la projection en algèbre relationnelle. En fait on dispose d'un peu plus de liberté que cela. On peut :

- Renommer les attributs
- Appliquer des fonctions aux valeurs de chaque tuple.
- Introduire des constantes.

Les fonctions applicables aux valeurs des attributs sont par exemple les opérations arithmétiques (+, *, ...) pour les attributs numériques ou des manipulations de chaîne de caractères (concaténation, sous-chaînes, mise en majuscule, ...). Il n'existe pas de norme mais la requête suivante devrait fonctionner sur tous les systèmes : on convertit le prix des activités en euros et on affiche le cours de l'euro avec chaque tuple.

```
SELECT libelle, prix / 6.56, 'Cours de l'euro = ', 6.56
FROM   Activite
WHERE  nomStation = 'Santalba'
```

1. Attention, ce n'est pas forcément ainsi que la requête est *exécutée* par le système.

Ce qui donne le résultat :

libelle	prix / 6.56	'Cours de l'euro ='	'6.56'
Kayac	7.62	'Cours de l'euro ='	6.56

Renommage

Les noms des attributs sont par défaut ceux indiqués dans la clause `SELECT`, même quand il y a des expressions complexes. Pour renommer les attributs, on utilise le mot-clé `AS`.

```
SELECT libelle, prix / 6.56 AS prixEnEuros,
          'Cours de l'euro = ', 6.56 AS cours
FROM   Activite
WHERE  nomStation = 'Santalba'
```

On obtient alors :

libelle	prixEnEuros	'Cours de l'euro ='	cours
Kayac	7.69	'Cours de l'euro ='	6.56

Remarque : Sur certains systèmes, le mot-clé `AS` est optionnel.

Doublons

L'introduction de fonctions permet d'aller au-delà de ce qui est possible en algèbre relationnelle. Il existe une autre différence, plus subtile : SQL permet l'existence de doublons dans les tables (il ne s'agit donc pas d'ensemble au sens strict du terme). La spécification de clés permet d'éviter les doublons dans les relations stockées, mais il peuvent apparaître dans le résultat d'une requête. Exemple :

```
SELECT libelle
FROM   Activite
```

donnera autant de lignes dans le résultat que dans la table `Activite`.

libelle
Voile
Plongee
Plongee
Ski
Piscine
Kayac

Pour éviter d'obtenir deux tuples identiques, on peut utiliser le mot-clé `DISTINCT`.

```
SELECT DISTINCT libelle
FROM   Activite
```

Attention : l'élimination des doublons peut être une opération coûteuse.

Tri du résultat

Il est possible de trier le résultat d'un requête avec la clause `ORDER BY` suivie de la liste des attributs servant de critère au tri. Exemple :

```
SELECT   *
FROM     Station
ORDER BY tarif, nomStation
```

trie, en ordre ascendant, les stations par leur tarif, puis, pour un même tarif, présente les stations selon l'ordre lexicographique. Pour trier en ordre descendant, on ajoute le mot-clé `DESC` après la liste des attributs.

Voici maintenant la plus simple des requêtes SQL : elle consiste à afficher l'intégralité d'une table. Pour avoir toutes les lignes on omet la clause `WHERE`, et pour avoir toutes les colonnes, on peut au choix lister tous les attributs ou utiliser le caractère '*' qui a la même signification.

```
SELECT *
FROM   Station
```

6.1.2 La clause `WHERE`

Dans la clause `WHERE`, on spécifie une condition booléenne portant sur les attributs des relations du `FROM`. On utilise pour cela de manière standard le `AND`, le `OR`, le `NOT` et les parenthèses pour changer l'ordre de priorité des opérateurs booléens. Par exemple :

```
SELECT nomStation, libelle
FROM   Activite
WHERE  nomStation  = 'Santalba'
AND    (prix > 50 AND prix < 120)
```

Les opérateurs de comparaison sont ceux du Pascal : $<$, $<=$, $>$, $>$, $=$, et $<>$ pour exprimer la différence ($!=$ est également possible). Pour obtenir une recherche par intervalle, on peut également utiliser le mot-clé `BETWEEN`. La requête précédente est équivalente à :

```
SELECT nomStation, libelle
FROM   Activite
WHERE  nomStation = 'Santalba'
AND    prix BETWEEN 50 AND 120
```

Chaînes de caractères

Les comparaisons de chaînes de caractères soulèvent quelques problèmes délicats.

1. Il faut être attentif aux différences entre chaînes de longueur fixe et chaînes de longueur variable. Les premières sont complétées par des blancs (' ') et pas les secondes.

2. Si SQL ne distingue pas majuscules et minuscules pour les mot-clés, il n'en va pas de même pour les valeurs. Donc 'SANTALBA' est différent de 'Santalba'.

SQL fournit des options pour les recherches par motif (*pattern matching*) à l'aide de la clause `LIKE`. Le caractère '_' désigne n'importe quel caractère, et le '%' n'importe quelle chaîne de caractères. Par exemple, voici la requête cherchant toutes les stations dont le nom termine par un 'a'.

```
SELECT nomStation
FROM   Station
WHERE  nomStation LIKE '%a'
```

Quelles sont les stations dont le nom commence par un 'V' et comprend exactement 6 caractères ?

```
SELECT nomStation
FROM   Station
WHERE  nomStation LIKE 'V_____'
```

Dates

Une autre différence avec l'algèbre est la possibilité de manipuler des dates. En fait tous les systèmes proposaient bien avant la normalisation leur propre format de date, et la norme préconisée par SQL2 n'est de ce fait pas suivie par tous.

Une date est spécifiée en SQL2 par le mot-clé `DATE` suivi d'une chaîne de caractères au format 'aaaa-mm-jj', par exemple `DATE '1998-01-01'`. Les zéros sont nécessaires afin que le mois et le quantième comprennent systématiquement deux chiffres.

On peut effectuer des sélections sur les dates à l'aide des comparateurs usuels. Voici par exemple la requête 'ID des clients qui ont commencé un séjour en juillet 1998'.

```
SELECT idClient
FROM   Sejour
WHERE  debut BETWEEN DATE '1998-07-01' AND  DATE '1998-07-31'
```

Les systèmes proposent de plus des fonctions permettant de calculer des écarts de dates, d'ajouter des mois ou des années à des dates, etc.

6.1.3 Valeurs nulles

Autre spécificité de SQL par rapport à l'algèbre : on admet que la valeur de certains attributs soit inconnue, et on parle alors de *valeur nulle*, désignée par le mot-clé `NULL`. Il est très important de comprendre que la 'valeur nulle' n'est en fait pas une valeur mais une absence de valeur, et que l'on ne peut donc lui appliquer aucune des opérations ou comparaisons usuelles.

- Toute opération appliquée à `NULL` donne pour résultat `NULL`.
- Toute comparaison avec `NULL` donne un résultat qui n'est ni vrai, ni faux mais une troisième valeur booléenne, `UNKNOWN`.

Les valeurs booléennes `TRUE`, `FALSE` et `UNKNOWN` sont définies de la manière suivante : `TRUE` vaut 1, `FALSE` 0 et `UNKNOWN` 1/2. Les connecteurs logiques donnent alors les résultats suivants :

1. x `AND` $y = min(x, y)$
2. x `OR` $y = max(x, y)$
3. `NOT` $x = 1 - x$

Les conditions exprimées dans une clause `WHERE` sont évaluées pour chaque tuple, et ne sont conservés dans le résultat que les tuples pour lesquels cette évaluation donne `TRUE`. La présence d'une valeur nulle dans une comparaison a donc souvent (mais pas toujours !) le même effet que si cette comparaison échoue et renvoie `FALSE`.

Voici une instance de la table `SEJOUR` avec des informations manquantes.

SEJOUR			
idClient	station	début	nbPlaces
10	Passac	1998-07-01	2
20	Santalba	1998-08-03	
30	Passac		3

La présence de `NULL` peut avoir des effets surprenants. Par exemple la requête suivante

```
SELECT station
FROM   Sejour
WHERE  nbPlaces <= 10 OR nbPlaces >= 10
```

devrait en principe ramener toutes les stations de la table. En fait 'Santalba' ne figurera pas dans le résultat car `nbPlaces` est à `NULL`.

Autre piège : `NULL` est un mot-clé, pas une constante. Donc une comparaison comme `nbPlaces = NULL` est incorrecte. Le prédicat pour tester l'absence de valeur dans une colonne est x `IS NULL` (et son inverse `IS NOT NULL`). La requête suivante sélectionne tous les séjours pour lesquels on connaît le nombre de places.

```
SELECT *
FROM   Sejour
WHERE  nbPlaces IS NOT NULL
```

La présence de `NULL` est une source de problèmes : dans la mesure du possible il faut l'éviter en spécifiant la contrainte `NOT NULL` ou en donnant une valeur par défaut.

6.2 Requêtes sur plusieurs tables

Les requêtes SQL décrites dans cette section permettent de manipuler simultanément plusieurs tables et d'exprimer les opérations binaires de l'algèbre relationnelle : jointure, produit cartésien, union, intersection, différence.

6.2.1 Jointures

La jointure est une des opérations les plus utiles (et donc une des plus courantes) puisqu'elle permet d'exprimer des requêtes portant sur des données réparties dans plusieurs tables. La syntaxe pour exprimer des jointures avec SQL est une extension directe de celle étudiée précédemment dans le cas des sélections simples : on donne simplement la liste des tables concernées dans la clause `FROM`, et on exprime les critères de rapprochement entre ces tables dans la clause `WHERE`.

Prenons l'exemple de la requête suivante : *donner le nom des clients avec le nom des stations où ils ont séjourné*. Le nom du client est dans la table `Client`, l'information sur le lien client/station dans la table `Sejour`. Deux tuples de ces tables peuvent être joints s'ils concernent le même client, ce qui peut s'exprimer à l'aide de l'identifiant du client. On obtient la requête :

```
SELECT nom, station
FROM   Client, Sejour
WHERE  id = idClient
```

On peut remarquer qu'il n'y a pas dans ce cas d'ambiguité sur les noms des attributs : `nom` et `id` viennent de la table `Client`, tandis que `station` et `idClient` viennent de la table `Sejour`. Il peut arriver (il arrive de fait fréquemment) qu'un même nom d'attribut soit partagé par plusieurs tables impliquées dans une jointure. Dans ce cas on résout l'ambiguité en préfixant l'attribut par le nom de la table.

Exemple : afficher le nom d'une station, son tarif hebdomadaire, ses activités et leurs prix.

```
SELECT nomStation, tarif, libelle, prix
FROM   Station, Activite
WHERE  Station.nomStation = Activite.nomStation
```

Comme il peut être fastidieux de répéter intégralement le nom d'une table, on peut lui associer un synonyme et utiliser ce synonyme en tant que préfixe. La requête précédente devient par exemple : [2]

```
SELECT S.nomStation, tarif, libelle, prix
FROM   Station S, Activite A
WHERE  S.nomStation = A.nomStation
```

2. Au lieu de `Station S`, la norme SQL2 préconise `Station AS S`, mais le `AS` est parfois inconnu ou optionnel.

Bien entendu, on peut effectuer des jointures sur un nombre quelconque de tables, et les combiner avec des sélections. Voici par exemple la requête qui affiche le nom des clients habitant Paris, les stations où ils ont séjourné avec la date, enfin le tarif hebdomadaire pour chaque station.

```
SELECT nom, station, debut, tarif
FROM   Client, Sejour, Station
WHERE  ville = 'Paris'
AND    id = idClient
AND    station = nomStation
```

Il n'y a pas d'ambiguité sur les noms d'attributs donc il est inutile en l'occurence d'employer des synonymes. Il existe en revanche une situation où l'utilisation des synonymes est indispensable : celle ou l'on souhaite effectuer une jointure d'une relation avec elle-même.

Considérons la requête suivante : *Donner les couples de stations situées dans la même région*. Ici toutes les informations nécessaires sont dans la seule table `Station`, mais on construit *un tuple* dans le résultat avec *deux tuples* partageant la même valeur pour l'attribut `région`.

Tout se passe comme s'il on devait faire la jointure entre deux versions distinctes de la table `Station`. Techniquement, on résout le problème en SQL en utilisant deux synonymes distincts.

```
SELECT s1.nomStation, s2.nomStation
FROM   Station s1, Station s2
WHERE  s1.region = s2.region
```

On peut imaginer que `s1` et `s2` sont deux 'curseurs' qui parcourent indépendamment la table `Station` et permettent de constituer des couples de tuples auxquels on applique la condition de jointure.

Interprétation d'une jointure

L'interprétation d'une jointure est une généralisation de l'interprétation d'un ordre SQL portant sur une seule table. Intuitivement, on parcourt tous les tuples définis par la clause `FROM`, et on leur applique la condition exprimée dans le `WHERE`. Finalement on ne garde que les attributs spécifiés dans la clause `SELECT`.

Quels sont les tuples définis par le `FROM` ? Dans le cas d'une seule table, il n'y a pas de difficulté. Quand il y a plusieurs tables, on peut donner (au moins) deux définitions équivalentes :

1. **Boucles imbriquées**. On considère chaque synonyme de table (ou par défaut chaque nom de table) comme une *variable tuple*. Maintenant on construit des boucles imbriquées, chaque boucle correspondant à une des tables du `FROM` et permettant à la variable correspondante d'itérer sur le contenu de la table.

   A l'intérieur de l'ensemble des boucles, on applique la clause `WHERE`.

2. **Produit cartésien**. On construit le produit cartésien des tables du `FROM`, en préfixant chaque attribut par le nom ou le synonyme de sa table pour éviter les ambiguités.

   On est alors ramené à la situation où il y a une seule table (le résultat du produit cartésien) et on interprête l'ordre SQL comme dans le cas des requêtes simples.

La première interprétation est proche de ce que l'on obtiendrait si on devait programmer une requête avec un langage comme le C ou Pascal, la deuxième s'inspire de l'algèbre relationnelle.

6.2.2 Union, intersection et différence

L'expression de ces trois opérations ensemblistes en SQL est très proche de l'algèbre relationnelle. On construit deux requêtes dont les résultats ont même arité (même nombre de colonnes et mêmes types d'attributs), et on les relie par un des mot-clé `UNION`, `INTERSECT` ou `EXCEPT`.

Trois exemples suffiront pour illustrer ces opérations.

1. Donnez tous les noms de région dans la base.

```
SELECT region FROM Station
 UNION
SELECT region FROM Client
```

2. Donnez les régions où l'on trouve à la fois des clients et des stations.

```
SELECT region FROM Station
 INTERSECT
SELECT region FROM Client
```

3. Quelles sont les régions où l'on trouve des stations mais pas des clients ?

```
SELECT region FROM Station
 EXCEPT
SELECT region FROM Client
```

La norme SQL2 spécifie que les doublons doivent être éliminés du résultat lors des trois opérations ensemblistes. Le coût de l'élimination de doublons n'étant pas négligeable, il se peut cependant que certains systèmes fassent un choix différent.

L'union ne peut être exprimée autrement qu'avec `UNION`. En revanche `INTERSECT` peut être exprimée avec une jointure, et la différence s'obtient, souvent de manière plus aisée, à l'aide des requêtes imbriquées.

6.3 Requêtes imbriquées

Jusqu'à présent les conditions exprimées dans le `WHERE` consistaient en comparaisons de valeurs scalaires. Il est possible également avec SQL d'exprimer des conditions sur des *relations*. Pour l'essentiel, ces conditions consistent en l'existence d'au moins un tuple dans la relation testée, ou en l'appartenance d'un tuple particulier à la relation.

La relation testée est construite par une requête `SELECT ... FROM ... WHERE` que l'on appelle sous-requête ou *requête imbriquée* puisqu'elle apparaît dans la clause `WHERE`

6.3.1 Conditions portant sur des relations

Une première utilisation des sous-requêtes est d'offrir une alternative syntaxique à l'expression de jointures. Les jointures concernées sont celles pour lesquelles le résultat est constitué avec des attributs provenant d'une seule des deux tables, l'autre ne servant que pour exprimer des conditions.

Prenons l'exemple suivant : on veut les noms des stations où ont séjourné des clients parisiens. On peut obtenir le résultat avec une jointure classique.

```
SELECT station
FROM   Sejour, Client
WHERE  id = idClient
AND    ville = 'Paris'
```

Ici, le résultat, `station`, provient de la table `SEJOUR`. D'où l'idée de séparer la requête en deux parties : (1) on constitue avec une sous-requête les `id`s des clients parisiens, puis (2) on utilise ce résultat dans la requête principale.

```
SELECT  station
```

```
FROM    Sejour
WHERE   idClient IN (SELECT id FROM Client
                     WHERE  ville = 'Paris')
```

Le mot-clé `IN` exprime clairement la condition d'appartenance de `idClient` à la relation formée par la requête imbriquée. On peut remplacer le `IN` par un simple '=' *si on est sûr que la sous-requête ramène un et un seul tuple*. Par exemple :

```
SELECT  nom, prenom
FROM    Client
WHERE   region = (SELECT region FROM Station
                      WHERE  nomStation = 'Santalba')
```

est (partiellement) correct car la recherche dans la sous-requête s'effectue par la clé. En revanche il se peut qu'aucun tuple ne soit ramené, ce qui génère une erreur.

Voici les conditions que l'on peut exprimer sur une relation R construite avec une requête imbriquée.

1. `EXISTS R`. Renvoie `TRUE` si R n'est pas vide, `FALSE` sinon.
2. t `IN R` où est un tuple dont le type est celui de R. `TRUE` si t appartient à R, `FALSE` sinon.
3. v *cmp* `ANY R`, où *cmp* est un comparateur SQL ($<, <, =$, etc.). Renvoie `TRUE` si la comparaison avec *au moins un* des tuples de la relation unaire R renvoie `TRUE`.
4. v *cmp* `ALL R`, où *cmp* est un comparateur SQL ($<, <, =$, etc.). Renvoie `TRUE` si la comparaison avec *tous* les tuples de la relation unaire R renvoie `TRUE`.

De plus, toutes ces expressions peuvent être préfixées par `NOT` pour obtenir la négation. Voici quelques exemples.

– *Où (station, lieu) ne peut-on pas faire du ski ?*

```
SELECT nomStation, lieu
FROM   Station
WHERE  nomStation NOT IN (SELECT nomStation FROM Activite
                          WHERE  libelle = 'Ski')
```

– *Quelle station pratique le tarif le plus élevé ?*

```
SELECT nomStation
FROM   Station
WHERE  tarif >= ALL (SELECT tarif FROM Station)
```

– *Dans quelle station pratique-t-on une activité au même prix qu'à Santalba ?*

```
SELECT nomStation, libelle
FROM   Activite
WHERE  prix IN (SELECT prix FROM Activite
                WHERE  nomStation = 'Santalba')
```

Ces requêtes peuvent s'exprimer sans imbrication (exercice), parfois de manière moins élégante ou moins concise. La différence, en particulier, s'exprime facilement avec `NOT IN` ou `NOT EXISTS`.

6.3.2 Sous-requêtes correllées

Les exemples de requêtes imbriquées donnés précédemment pouvait être évalués indépendamment de la requête principale, ce qui permet au système (s'il le juge nécessaire) d'exécuter la requête en deux phases. Il peut arriver que la sous-requête soit basée sur une ou plusieurs valeurs issues des relations de la requête principale. On parle alors de *requêtes correllées*.

Exemple : *quels sont les clients (nom, prénom) qui ont séjourné à Santalba*.

```
SELECT nom, prenom
FROM   Client
WHERE  EXISTS (SELECT 'x' FROM Sejour
               WHERE  station = 'Santalba'
               AND    idClient = id)
```

Le `id` dans la requête imbriquée n'appartient pas à la table `Sejour` mais à la table `Client` référencée dans le `FROM` de la requête principale.

Remarque : on peut employer un `NOT IN` à la place du `NOT EXISTS` (exercice), de même que l'on peut toujours employer `EXISTS` à la place de `IN`. Voici une des requêtes précédentes où l'on a appliqué cette transformation, en utilisant de plus des synonymes.
Dans quelle station pratique-t-on une activité au même prix qu'à Santalba ?

```
SELECT nomStation
FROM   Activite A1
WHERE  EXISTS (SELECT 'x'FROM Activite A2
               WHERE  nomStation = 'Santalba'
               AND    A1.libelle = A2.libelle
               AND    A1.prix = A2.prix)
```

Cette requête est elle-même équivalente à une jointure sans requête imbriquée.

6.4 Agrégration

Toutes les requêtes vues jusqu'à présent pouvaient être interprétées comme une suite d'opérations effectuées *tuple à tuple*. De même le résultat était toujours constitué de valeurs issues de tuples individuels. Les fonctionnalités *d'agrégation* de SQL permettent d'exprimer des conditions *sur des groupes de tuples*, et de constituer le résultat par *agrégation de valeurs* au sein de chaque groupe.

La syntaxe SQL fournit donc :

1. Le moyen de partitioner une relation en *groupes* selon certains critères.
2. Le moyen d'exprimer des conditions sur ces groupes.
3. Des fonctions d'agrégation.

Il existe un groupe par défaut : c'est la relation toute entière. Sans même définir de groupe, on peut utiliser les fonctions d'agrégation.

6.4.1 Fonctions d'agrégation

Ces fonctions s'appliquent à une colonne, en général de type numérique. Ce sont :

1. `COUNT` qui compte le nombre de valeurs *non nulles*.
2. `MAX` et `MIN`.
3. `AVG` qui calcule la moyenne des valeurs de la colonne.

4. `SUM` qui effectue le cumul.

Exemple d'utilisation de ces fonctions :

```
SELECT COUNT(nomStation), AVG(tarif), MIN(tarif), MAX(tarif)
FROM   Station
```

Remarque importante : on ne peut pas utiliser simultanément dans la clause `SELECT` des fonctions d'agrégation et des noms d'attributs (sauf dans le cas d'un `GROUP BY`, voir plus loin). La requête suivante est incorrecte (pourquoi ?).

```
SELECT nomStation, AVG(tarif)
FROM   Station
```

A condition de respecter cette règle, on peut utiliser les ordres SQL les plus complexes. Exemple : *Combien de places a réservé Mr Kerouac pour l'ensemble des séjours ?*.

```
SELECT SUM (nbPlaces)
FROM   Client, Sejour
WHERE  nom = 'Kerouac'
AND    id = idClient
```

6.4.2 La clause `GROUP BY`

Dans les requêtes précédentes, on appliquait la fonction d'agrégation à l'ensemble du résultat d'une requête (donc éventuellement à l'ensemble de la table elle-même). Une fonctionnalité complémentaire consiste à *partitioner* ce résultat en groupes, et à appliquer la ou les fonction(s) à chaque groupe.

On construit les groupes en associant les tuples partageant la même valeur pour une ou plusieurs colonnes.

Exemple : afficher les régions avec le nombre de stations.

```
SELECT   region, COUNT(nomStation)
FROM     Station
GROUP BY region
```

Donc ici on constitue un groupe pour chaque région. Puis on affiche ce groupe sous la forme d'un tuple, dans lequel les attributs peuvent être de deux types :

1. les attributs *dont la valeur est constante pour l'ensemble du groupe*, soit précisément les attributs du `GROUP BY`. Exemple ici l'attribut `region` ;
2. le résultat d'une fonction d'agrégation appliquée au groupe : ici `COUNT(nomStation)`.

Bien entendu il est possible d'exprimer des ordres SQL complexes et d'appliquer un `GROUP BY` au résultat. De plus, il n'est pas nécessaire de faire figurer tous les attributs du `GROUP BY` dans la clause `SELECT`.

Exemple : on souhaite consulter le nombre de places reservées, *par client*.

```
SELECT   nom, SUM (nbPlaces)
FROM     Client, Sejour
WHERE    id = idClient
GROUP BY id, nom
```

L'interprétation est simple : (1) on exécute d'abord la requête `SELECT ... FROM ... WHERE`, puis (2) on prend le résultat et on le partitione, enfin (3) on calcule le résultat des fonctions.

A l'issue de l'étape (2), on peut imaginer une relation qui n'est pas en première forme normale : on y trouverait des tuples avec les attributs du `GROUP BY` sous forme de valeur atomique, puis des attributs de type *ensemble* (donc interdits dans le modèle relationnel). C'est pour se ramener en 1FN que l'on doit appliquer des fonctions d'agrégation à ces ensembles.

Exercice : pourquoi grouper sur `id, nom` ? Quels sont les autres choix possibles et leurs inconvénients ?

6.4.3 La clause `HAVING`

Finalement, on peut faire porter des conditions sur les groupes avec la clause `HAVING`. La clause `WHERE` ne peut exprimer des conditions que sur les tuples pris un à un.

Exemple : on souhaite consulter le nombre de places reservées, par client, *pour les clients ayant réservé plus de 10 places*.

```
SELECT   nom, SUM (nbPlaces)
FROM     Client, Sejour
WHERE    id = idClient
GROUP BY nom
HAVING   SUM(nbPlaces) >= 10
```

On voit que la condition porte ici sur une propriété de l'ensemble des tuples du groupe, et pas de chaque tuple pris individuellement. La clause `HAVING` est donc toujours exprimée sur le résultat de fonctions d'agrégation.

6.5 Mises-à-jour

Les commandes de mise-à-jour (insertion, destruction, modification) sont considérablement plus simples que les requêtes.

6.5.1 Insertion

L'insertion s'effectue avec la commande `INSERT` dont la syntaxe est la suivante :

```
INSERT INTO R( A1, A2, ... An) VALUES (v1, v2, ... vn)
```

`R` est le nom d'une relation, et les `A1, ... An` sont les noms des attributs dans lequels on souhaite placer une valeur. *Les autres attributs seront donc à* `NULL` *(ou à la valeur par défaut)*. Tous les attributs spécifiés `NOT NULL` (et sans valeur par défaut) doivent donc figurer dans une clause `INSERT`.

Les `v1, ... vn` sont les valeurs des attributs. Exemple de l'insertion d'un tuple dans la table `Client`.

```
INSERT INTO Client (id, nom, prenom)
VALUES (40, 'Moriarty', 'Dean')
```

Donc, à l'issue de cette insertion, les attributs `ville` et `region` seront à `NULL`.

Il est également possible d'insérer dans une table le résultat d'une requête. Dans ce cas la partie `VALUES ...` est remplacée par la requête elle-même. Exemple : on a créé une table `Sites (lieu, region)` et on souhaite y copier les couples `(lieu, region)` déjà existant dans la table `Station`.

```
INSERT INTO Sites (lieu, region)
  SELECT lieu, region FROM Station
```

Bien entendu le nombre d'attributs et le type de ces derniers doivent être cohérents.

6.5.2 Destruction

La destruction s'effectue avec la clause `DELETE` dont la syntaxe est :

```
DELETE FROM R
WHERE condition
```

R est bien entendu la table, et condition est toute condition valide pour une clause WHERE. En d'autres termes, si on effectue, avant la destruction, la requête

```
SELECT * FROM R
WHERE condition
```

on obtient l'ensemble des lignes qui seront détruites par DELETE. Procéder de cette manière est un des moyens de s'assurer que l'on va bien détruire ce que l'on souhaite....

Exemple : destruction de tous les clients dont le nom commence par 'M'.

```
DELETE FROM Client
WHERE nom LIKE 'M%'
```

6.5.3 Modification

La modification s'effectue avec la clause UPDATE. La syntaxe est proche de celle du DELETE :

```
UPDATE R SET A1=v1, A2=v2, ... An=vn
WHERE condition
```

R est la relation, les Ai sont les attributs, les vi les nouvelles valeurs et condition est toute condition valide pour la clause WHERE. Exemple : augmenter le prix des activités de la station Passac de 10%.

```
UPDATE Activite
SET  prix = prix * 1.1
WHERE nomStation = 'Passac'
```

Une remarque importante : toutes les mises-à-jour ne deviennent définitives qu'à l'issue d'une validation par commit. Entretemps elles peuvent être annulées par rollback. Voir le cours sur la concurrence d'accès.

6.6 Exercices

Exercice 6.1 *Reprendre les expressions algébriques du premier exercice du chapitre algèbre, et les exprimer en SQL.*

Exercice 6.2 *Donnez l'expression SQL des requêtes suivantes, ainsi que le résultat obtenu avec la base du chapitre "Le langage SQL".*

1. *Nom des stations ayant strictement plus de 200 places.*
2. *Noms des clients dont le nom commence par 'P' ou dont le solde est supérieur à 10 000.*
3. *Quelles sont les régions dont l'intitulé comprend (au moins) deux mots ?*
4. *Nom des stations qui proposent de la plongée.*
5. *Nom des clients qui sont allés à Santalba.*
6. *Donnez les couples de clients qui habitent dans la même région. Attention : un couple doit apparaître une seule fois.*
7. *Nom des régions qu'a visité Mr Pascal.*
8. *Nom des stations visitées par des européens.*
9. *Qui n'est pas allé dans la station Farniente ?*
10. *Quelles stations ne proposent pas de la plongée ?*

11. *Combien de séjours ont eu lieu à Passac ?*
12. *Donner, pour chaque station, le nombre de séjours qui s'y sont déroulés.*
13. *Donner les stations où se sont déroulés au moins 3 séjours.*
14. (♣)*Les clients qui sont allés dans* toutes *les stations.*

Exercice 6.3 (Valeurs nulles) *On considère la table suivante :*

STATION				
NomStation	*Capacité*	*Lieu*	*Région*	*Tarif*
Gratuite	*80*	*Guadeloupe*	*Antilles*	
NullePart	*150*			*2000*

1. *Donnez les résultats des requêtes suivantes (rappel : le* `'||'` *est la concaténation de chaînes de caractères.).*

   (a)
   ```
   SELECT nomStation FROM Station
   WHERE tarif >200
   ```

   (b)
   ```
   SELECT tarif * 3 FROM Station
   WHERE nomStation LIKE '%l%' AND lieu LIKE '%'
   ```

   (c)
   ```
   SELECT ' Lieu = ' || lieu FROM Station
   WHERE capacite >= 100 OR tarif >= 1000
   ```

   (d)
   ```
   SELECT ' Lieu = ' || lieu FROM Station
   WHERE NOT (capacite <100 AND tarif <1000)
   ```

2. *Les deux dernières requêtes sont-elles équivalentes (i.e. donnent-elles le même résultat quel que soit le contenu de la table) ?*
3. *Supposons que l'on ait conservé une logique bivaluée (avec* `TRUE` *et* `FALSE`*) et adopté la règle suivante : toute comparaison avec un* `NULL` *donne* `FALSE`*. Obtient-on des résultats équivalents ? Cette règle est-elle correcte ?*
4. *Même question, en supposant que toute comparaison avec* `NULL` *donne* `TRUE`*.*

Exercice 6.4 *On reprend la requête constituant la liste des stations avec leurs activités, légèrement modifiée.*

```
SELECT S.nomStation, tarif, libelle, prix
FROM   Station S, Activites A, Sejour
WHERE  S.nomStation = A.nomStation
```

1. *La table* `Sejour` *est-elle nécessaire dans le* `FROM` *?*
2. *Qu'obtient-on dans les trois cas suivants : (1) la table* `Sejour` *contient 1 tuple, (2) la table* `Sejour` *contient 100 000 tuples, (3) la table* `Sejour` *est vide.*
3. *Soit trois tables* `R`, `S` *et* `T` *ayant chacune un seul attribut* `A`*. On veut calculer* `R ∩ (S ∪ T)`*.*

   (a) *La requête suivante est-elle correcte ? Expliquez pourquoi.*
   ```
   SELECT R.A FROM R, S, T WHERE R.A=S.A OR R.A=T.A
   ```

   (b) *Donnez la bonne requête.*

Chapitre 7

Schémas relationnels

Sommaire

Ce chapitre présente l'environnement d'un utilisateur travaillant avec un SGBD relationnel. On se place donc dans la situation d'un informaticien connecté à une machine sur laquelle tourne un SGBD gérant une base de données.

Le principal élément d'un tel environnement est le *schéma* consistant principalement en un ensemble de tables relatives à une même application. Il peut y avoir plusieurs schémas dans une même base : par exemple on peut très bien faire coexister le schéma « Officiel des spectacles » et le schéma « Agence de voyage ». En toute rigueur il faudrait donc distinguer « l'instance de schéma » de la « base de données » qui est un sur-ensemble. Comme on se situe en général dans un et un seul schéma, on peut utiliser les deux termes de manière équivalente.

Le schéma est le principal concept étudié dans ce chapitre. Outre les tables, un schéma peut comprendre des *vues*, des *contraintes* de différents types, des *triggers* (« reflexes ») qui sont des procédures déclenchées par certains évènements, enfin des spécifications de stockage et/ou d'organisation physique (comme les index) qui ne sont pas traitées ici.

Dans tout ce chapitre, on basera les exemples sur le schéma « Officiel » qui est rappelé ci-dessous.

- Cinéma (**nomCinéma**, numéro, rue, ville)
- Salle (**nomCinéma, no**, capacité, climatisée)
- Horaire (**idHoraire**, heureDébut, heureFin)
- Séance (**idFilm**, **nomCinéma**, **noSalle**, **idHoraire**, tarif)
- Film (**idFilm**, titre, année, genre, résumé, idMES)
- Artiste (**id**, nom, prénom, annéeNaissance)
- Rôle (**idActeur**, **idFilm**, nomRôle)

7.1 Schémas

Cette section décrit la notion de schéma au sens SQL2, ainsi que le système de droits d'accès à un schéma destiné à garantir la sécurité de la base.

7.1.1 Définition d'un schéma

Un schéma est l'ensemble des déclarations décrivant une base de données *au niveau logique* : tables, vues, domaines, etc. On crée un schéma en lui donnant un nom puis en donnant les listes des commandes (de type `CREATE` en général) créant les éléments du schéma. Voici par exemple le squelette de la commande de création du schéma « Agence de voyage ».

```
CREATE SCHEMA agence_de_voyage
CREATE TABLE Station (...
CREATE TABLE Client (...
...
CREATE VIEW ....
...
CREATE ASSERTION ...
...
CREATE TRIGGER ...
...
```

Deux schémas différents sont indépendants : on peut créer deux tables ayant le même nom. Bien entendu on peut modifier un schéma en ajoutant ou en supprimant des éléments. La modification a lieu dans le *schéma courant* que l'on peut modifier avec la commande `SET SCHEMA`. Par exemple, avant de modifier la table `FILM`, on exécute :

```
SET SCHEMA officiel_des_spectacles
```

Quand on souhaite accéder à une table qui n'est pas dans le schéma courant (par exemple dans un ordre SQL), il faut préfixer le nom de la table par le nom du schéma.

```
SELECT * FROM officiel_des_spectacles.film
```

La norme SQL2 définit deux niveaux supérieurs au schéma : les *catalogues* et les *groupes* (*cluster*). Ils correspondent respectivement aux niveaux d'unicité de nom de schéma, et d'accessibilité à une table dans une requête. Donc un utilisateur ne peut spécifier dans sa requête que les tables du *cluster* courant.

7.1.2 Utilisateurs

L'accès à une base de données est restreint, pour des raisons évidentes de sécurité, à des *utilisateurs* connus du SGBD et identifiés par un nom et un mot de passe. Chaque utilisateur se voit attribuer certains droits sur les schémas et les tables de chaque schéma.

La connexion se fait soit dans le cadre d'un programme, soit interactivement par une commande du type :

```
CONNECT utilisateur
```

Suivie de l'entrée du mot de passe demandée par le système. Une session est alors ouverte pour laquelle le SGBD connaît l'ID de l'utilisateur courant. Les droits de cet utilisateur sont alors les suivants :

1. Tous les droits sur les éléments du schéma comme les tables ou les vues des schémas que l'utilisateur a lui-même créé. Ces droits concernent aussi bien la manipulation des données que la modification ou la destruction des éléments du schéma.

2. Les droits sur les éléments d'un schéma dont on n'est pas propriétaire sont accordés par le propriétaire du schéma. Par défaut, on n'a aucun droit.

En tant que propriétaire d'un schéma, on peut donc accorder des droits à d'autres utilisateurs sur ce schéma ou sur des éléments de ce schéma. SQL2 définit 6 types de droits. Les quatre premiers portent sur le contenu d'une table, et se comprennent aisément.

1. Insertion (`INSERT`).
2. Modification (`UPDATE`).
3. Recherche (`SELECT`).
4. Destruction (`DELETE`).

Il existe deux autres droits :

1. `REFERENCES` donne le droit à un utilisateur non propriétaire du schéma de faire référence à une table dans une contrainte d'intégrité.
2. `USAGE` permet à un utilisateur non propriétaire du schéma d'utiliser une définition (autre qu'une table ou une assertion) du schéma.

Les droits sont accordés par la commande `GRANT` dont voici la syntaxe :

```
GRANT <privilege>
ON    <element du schema>
TO    <utilisateur>
[WITH GRANT OPTION]
```

Bien entendu, pour accorder un privilège, il faut en avoir le droit, soit parce que l'on est propriétaire de l'élément du schéma, soit parce que le droit a été accordé par la commande `WITH GRANT OPTION`.

Voici la commande permettant à Marc de consuter les films.

```
GRANT SELECT ON Film TO Marc;
```

On peut désigner tous les utilisateurs avec le mot-clé `PUBLIC`, et tous les privilèges avec l'expression `ALL PRIVILEGES`.

```
GRANT ALL PRIVILEGES ON Film TO PUBLIC
```

On supprime un droit avec la commande `REVOKE` dont la syntaxe est semblable à celle de `GRANT`.

```
REVOKE SELECT ON Film FROM Marc
```

7.2 Contraintes et assertions

Nous revenons maintenant de manière plus approfondie sur les contraintes portant sur le contenu des tables. Il est très important de spécifier les contraintes dans le schéma, afin que le SGBD puisse les prendre en charge. Cela évite d'effectuer des contrôles de manière répétitive (et souvent lacunaire) dans les applications qui accèdent à la base.

Les contraintes (essentielles) portant sur les clés simples, primaires et étrangères ont déjà été vues dans le chapitre 4. Les contraintes décrites ici sont moins spécifiques et portent sur les valeurs des attributs ou plus globalement sur le contenu d'une ou plusieurs relations.

La commande

```
CHECK (condition)
```

permet d'exprimer toute contrainte portant soit sur un attribut, soit sur un tuple. La condition elle-même peut être toute expression suivant la clause `WHERE` dans une requête SQL. Les contraintes les plus courantes sont celles consistant à restreindre un attribut à un ensemble de valeurs, mais on peut trouver des contraintes arbitrairement complexes, faisant référence à d'autres relations. Dans ce cas, on doit obligatoirement utiliser des sous-requêtes.

Exemple simple : on restreint les valeurs possibles des attributs `capacité` et `climatisée` dans la table *Salle*.

```
CREATE TABLE Salle
       (nomCinema  VARCHAR (30) NOT NULLL,
       no          INTEGER,
       capacite    INTEGER CHECK (capacite < 300),
       climatisee  CHAR(1) CHECK (climatisee IN ('O','N')),
       PRIMARY KEY (nomCinema, no),
       FOREIGN KEY nomCinema REFERENCES Cinema)
```

Il s'agit de contraintes portant sur des valeurs d'attributs. A chaque insertion d'un tuple, ou mise-à-jour de ce tuple affectant un des attributs contraints, le contrôle sera effectué. La règle est que la condition ne doit pas s'évaluer à `FALSE` (donc la valeur `UNKNOWN` est acceptée).

Voici un autre exemple illustrant l'utilisation d'une sous-requête : on souhaite remplacer la contrainte `FOREIGN KEY` par une clause `CHECK`.

```
CREATE TABLE Salle
       (nomCinema  VARCHAR (30) NOT NULLL,
       no          INTEGER,
       capacite    INTEGER CHECK (capacite < 300),
       climatisee  CHAR(1) CHECK (climatisee IN ('O','N')),
       PRIMARY KEY (nomCinema, no),
       CHECK (nomCinéma IN (SELECT nom FROM Cinema)))
```

Il s'agit d'une illustration simple d'une clause `CHECK` avec sous-requête, mais elle est incorrecte pour garantir l'intégrité référentielle. Pourquoi ? (penser aux événements qui déclenchent respectivement les contrôles des clauses `CHECK` et `FOREIGN KEY`).

Au lieu d'associer une contrainte à un attribut particulier, on peut la définir globalement. Dans ce cas la contrainte peut faire référence à n'importe quel attribut de la table et est testée tuple à tuple.

Exemple : toute salle de plus de 300 places doit être climatisée :

```
CREATE TABLE Salle
       (nomCinema  VARCHAR (30) NOT NULLL,
       no          INTEGER,
       capacite    INTEGER,
       climatisee  CHAR(1),
       PRIMARY KEY (nomCinema, no),
       FOREIGN KEY nomCinema REFERENCES Cinema,
       CHECK (capacité<300 OR Climatisé = 'O'))
```

L'utilisation des sous-requêtes n'est pas recommandée, à cause du problème souligné précédemment : la contrainte peut être satisfaite au moment de l'insertion du tuple, et ne plus l'être après.

Exemple : la grande salle du Rex doit rester la plus grande !

```
CREATE TABLE Salle
       (nomCinema  VARCHAR (30) NOT NULLL,
       no          INTEGER,
```

```
capacite    INTEGER,
climatisee  CHAR(1),
PRIMARY KEY (nomCinema, no),
FOREIGN KEY nomCinema REFERENCES Cinema,
CHECK (capacité < (SELECT Max(capacite) FROM Salle
            WHERE nomCinema = 'Rex')))
```

Problème : si on diminue la taille de la salle du Rex, la contrainte peut ne plus être respectée. Il est donc préférable de ne pas utiliser la clause `CHECK` pour des contraintes impliquant d'autres tables.

Il est possible, et recommandé, de donner un nom aux contraintes avec la clause `CONSTRAINT`.

```
CONSTRAINT clim CHECK (capacite < 300 OR climatisee = 'O')
```

Cela facilite la compréhension des messages, et permet de modifier ou de détruire une contrainte :

```
ALTER TABLE Salle DROP CONSTRAINT clim
```

7.3 Vues

Cette section est consacrée à l'une des fonctionnalités les plus remarquables des SGBD relationnels : les vues.

Comme nous l'avons vu dans la partie consacrée à SQL, une requête produit toujours une relation. Cela suggère la possibilité d'ajouter au schéma des tables 'virtuelles' qui ne sont rien d'autres que le résultat de requêtes stockées. De telles tables sont nommées des *vues* dans la terminologie relationnelle. On peut interroger des vues comme des tables stockées.

Une vue n'induit aucun stockage puisqu'elle n'existe pas physiquement, et permet d'obtenir une représentation différente des tables sur lesquelles elle est basée.

7.3.1 Création et interrogation d'une vue

Une vue est tout point comparable à une table : en particulier on peut l'interroger par SQL. La grande différence est qu'une vue est le résultat d'une requête, avec la caractéristique essentielle que ce résultat est réévalué à chaque fois que l'on accède à la vue. En d'autre termes une vue est *dynamique* : elle donne une représentation fidèle de la base au moment de l'évaluation de la requête.

Une vue est essentiellemnt une requête à laquelle on a donné un nom. La syntaxe de création d'une vue est très simple :

```
CREATE VIEW <nom-vue>
AS          <requête>
[WITH CHECK OPTION]
```

Exemple : on peut créer une vue qui ne « contient » que les cinémas parisiens :

```
CREATE VIEW ParisCinemas
AS     SELECT * FROM Cinema WHERE ville = 'Paris'
```

On peut aussi en profiter pour restreindre la vision des cinémas parisiens à leur nom et à leur nombre de salles.

```
CREATE VIEW SimpleParisCinemas
AS     SELECT nom, COUNT(*) AS nbSalles
       FROM Cinema c, Salle s
       WHERE ville = 'Paris'
       AND c.nom = s.nomCinema
       GROUP BY c.nom
```

Enfin un des intérêts des vues est de donner une représentation « dénormalisée » de la base, en regroupant des informations par des jointures. Par exemple on peut créer une vue *Casting* donnant explicitement les titres des films, leur année et les noms et prénoms des acteurs.

```
CREATE VIEW Casting (film, annee, acteur, prenom) AS
SELECT titre, annee, nom, prenom
FROM   Film f, Role r, Artiste a
WHERE  f.idFilm = r.idFilm
AND    r.idActeur = a.idArtiste
```

Remarque : on a donné explicitement des noms d'attributs au lieu d'utiliser les attributs de la clause `SELECT`.

Maintenant, on peut utiliser les vues et les tables dans des requêtes SQL. Par exemple la requête « Quels acteurs ont tourné un film en 1997 » s'exprime par :

```
SELECT acteur, prenom
FROM   Casting
WHERE  annee = 1997
```

On peut ensuite donner des droits en lecture sur cette vue pour que cette information limitée soit disponible à tous.

```
GRANT SELECT ON Casting TO PUBLIC
```

7.3.2 Mise à jour d'une vue

L'idée de modifier une vue peut sembler étrange puisqu'une vue n'a pas de contenu. En fait il s'agit bien entendu de modifier la table qui sert de support à la vue. Il existe de sévères restrictions sur les droits d'insérer ou de mettre-à-jour des tables à travers les vues. Un exemple suffit pour comprendre le problème. Imaginons que l'on souhaite insérer une ligne dans la vue *Casting*.

```
INSERT INTO CASTING (film, annee, acteur, prenom)
VALUES    ('Titanic', 1998, 'DiCaprio', 'Leonardo');
```

Cet ordre s'adresse à une vue issue de trois tables. Il n'y a clairement pas assez d'information pour alimenter ces tables de manière cohérente, et l'insertion n'est pas possible (de même que toute mise à jour). De telles vues sont dites *non modifiables*.

Les règles définissant les vues modifiables sont très strictes.

1. La vue doit être basée sur une seule table.
2. Toute colonne non référencée dans la vue doit pouvoir être mise à `NULL` ou disposer d'une valeur par défaut.
3. On ne peut pas mettre-à-jour un attribut qui résulte d'un calcul ou d'une opération.

Il est donc tout à fait possible d'insérer, modifier ou détruire la table `Film` au travers de la vue *ParisCinema*.

```
INSERT INTO ParisCinema
VALUES (1876, 'Breteuil', 12, 'Cite', 'Lyon')
```

En revanche, en admettant que la ville est définie à `NOT NULL`, il n'est pas possible d'insérer dans *SimpleParisCinemas*.

L'insertion précédente illustre une petite subtilité : on peut insérer dans une vue, sans être en mesure de voir la ligne insérée au travers de la vue par la suite ! Afin d'éviter ce genre d'incoéherence, SQL2 propose

l'option `WITH CHECK OPTION` qui permet de garantir que toute ligne insérée dans la vue satisfait les critères de sélection de la vue.

```
CREATE VIEW ParisCinemas
AS     SELECT * FROM Cinema
       WHERE ville = 'Paris'
WITH CHECK OPTION
```

L'insertion donnée en exemple ci-dessus devient impossible. Enfin on détruit une vue avec la syntaxe courante SQL :

```
DROP VIEW ParisCinemas
```

7.4 *Triggers*

Le mécanisme de *triggers* (que l'on peut traduire par 'déclencheur' ou 'réflexe') implanté dans de nombreux SGBD n'est pas évoqué dans la norme SQL2 mais constitue un des points de discussion de la norme SQL3. Un *trigger* est une procédure qui est déclenchée par des évènements de mise-à-jour *spécifiés par l'utilisateur* et ne s'exécute que quand une condition est satisfaite.

On peut considérer les *triggers* comme une extension du système de contraintes proposé par la clause `CHECK` : à la différence de cette dernière, l'évènement déclencheur est explictement indiqué, et l'action n'est pas limitée à la simple alternative acceptation/rejet. Les possibilités offertes sont très intéressantes. Citons :

1. La possibilité d'éviter les risques d'incohérence dus à la présence de redondance.
2. L'enregistrement automatique de certains évèvenements (*auditing*).
3. La spcécification de contraintes liées à l'évolution des données (exemple : le prix d'une séance ne peut qu'augmenter).
4. Toute règle complexe liée à l'environnement d'exécution (restrictions sur les horaires, les utilisateurs, etc).

7.4.1 Principes des *triggers*

Le modèle d'exécution des *triggers* est basé sur la séquence *Evénement-Condition-Action* (ECA) que l'on peut décrire ainsi :

1. un *trigger* est déclenché par un *évènement*, spécifié par le programmeur, qui est en général une insertion, destruction ou modification sur une table ;
2. la première action d'un *trigger* est de tester une *condition* : si cette condition ne s'évalue pas à `TRUE`, l'exécution s'arrête ;
3. enfin *l'action* proprement dite peut consister en toute opération sur la base de données : les SGBD fournissent un langage impératif permettant de créer de véritables procédures.

Une caractéristique importante de cette procédure (action) est de pouvoir manipuler simultanément les valeurs ancienne et nouvelle de la donnée modifiée, ce qui permet de faire des tests sur l'évolution de la base.

Parmi les autres caractéristiques importantes, citons les deux suivantes. Tout d'abord un *trigger* peut être exécuté au choix une fois pour un seul ordre SQL, ou à chaque ligne concernée par cet ordre. Ensuite l'action déclenchée peut intervenir *avant* l'évènement, ou *après*.

L'utilisation des *triggers* permet de rendre une base de données *dynamique* : une opération sur la base peut en déclencher d'autres, qui elles-mêmes peuvent entraîner en cascade d'autres réflexes. Ce mécanisme n'est pas sans danger à cause des risques de boucle infinie.

7.4.2 Syntaxe

Voici tout d'abord un exemple de *trigger* qui maintient la capacité d'un cinéma à chaque mise-à-jour sur la table *Salle*.[1]

```
CREATE TRIGGER CumulCapacite
AFTER UPDATE ON Salle
FOR EACH ROW
WHEN (new.capacite != old.capacite)
BEGIN
  UPDATE Cinema
  SET    capacite = capacite - :old.capacite  + :new.capacite
  WHERE  nom = :new.nomCinema;
END;
```

Pour garantir la validité du cumul, il faudrait créer des *triggers* sur les événements `UPDATE` et `INSERT`. Une solution plus concise (mais plus coûteuse) est de recalculer systématiquement le cumul : dans ce cas on peut utiliser un *trigger* qui se déclenche globalement pour la requête :

```
CREATE TRIGGER CumulCapaciteGlobal
AFTER UPDATE OR INSERT OR DELETE ON Salle
BEGIN
  UPDATE Cinema C
  SET    capacite = (SELECT SUM (capacite)
                     FROM   Salle S
                     WHERE  C.nom = S.nomCinema);
END;
```

La syntaxe de création d'un *trigger* est la suivante :

```
CREATE trigger <nom-trigger>
<quand> <événements> ON <table>
[FOR EACH ROW [WHEN <condition>]]
BEGIN
 <action>
END;
/
```

Les choix possibles sont :

- `quand` peut être `BEFORE` ou `AFTER`.
- `événements` spécifie `DELETE`, `UPDATE` ou `INSERT` séparés par des `OR`.
- `FOR EACH ROW` est optionnel. En son absence le *trigger* est déclenché une fois pour toute requête modifiant la table, *et ce sans condition*.

  Sinon `condition` est toute condition booléenne SQL. De plus on peut rérférencer les anciennes et nouvelles valeurs du tuple courant avec la syntaxe `new.attribut` et `old.attribut` respectivement.
- `action` est une procédure qui peut être implantée, sous Oracle, avec le langage PL/SQL. Elle peut contenir des ordres SQL *mais pas de mise-à-jour de la table courante*.

  Les anciennes et nouvelles valeurs du tuple courant sont référencées par `:new.attr` et `:old.attr`.

  Il est possible de modifier `new` et `old`. Par exemple `:new.prix=500;` forcera l'attribut `prix` à 500 dans un `BEFORE` *trigger*.

Remarque : la disponibilité de `new` et `old` dépend du contexte. Par exemple `new` est à `NULL` dans un *trigger* déclenché par `DELETE`.

1. Tous les exemples qui suivent utilisent la syntaxe des *triggers* Oracle, très proche de celle de SQL3.

7.5 Exercices

Exercice 7.1 *On reprend le schéma suivant décrivant une bibliothèque avec des livres et leurs auteurs.*

1. *Livre (**titreLivre**, année, éditeur, chiffreAffaire)*
2. *Chapitre (**titreLivre, titreChapitre**, nbPages)*
3. *Auteur (**nom**, prénom, annéeNaissance)*
4. *Redaction (**nomAuteur, titreLivre, titreChapitre**)*

Indiquez comment exprimer les contraintes suivantes sur le schéma relationnel, avec la syntaxe SQL2.

1. *L'année de parution d'un livre est toujours connue.*
2. *Le nombre de pages d'un chapitre est supérieur à 0.*
3. *Un éditeur doit faire partie de la liste {Seuil, Gallimard, Grasset}*

Exercice 7.2 *On veut créer un ensemble de vues sur la base « Agence de voyages » qui ne donne que les informations sur les stations aux Antilles. Il existe un utilisateur,* `lambda`*, qui ne doit pouvoir accéder qu'à cet ensemble de vues, et en lecture uniquement.*

1. *Définir la vue* StationAntilles *qui est identique à* Station*, sauf qu'elle ne montre que les stations aux Antilles.*
2. *Définir la vue* ActivitéAntilles *donnant les activités proposées dans les stations de* StationAntilles
3. *Définir la vue* SejourAntilles *avec les attributs nomClient, prénomClient, ville, station, début. Cette vue donne un résumé des séjours effectués dans les stations aux Antilles.*
4. *Dans quelles vues peut-on insérer ? Sur quelles vues est-il utile de mettre la clause* `WITH CHECK OTPION` *?*
5. *Donner les ordres* `GRANT` *qui ne donnent à l'utilisateur* `lambda` *que la possibilité de voir les informations sur les vues 'Antilles'.*

Exercice 7.3 *Une vue sur une table R qui ne comprend pas la clé primaire de R est-elle modifiable ?*

Exercice 7.4 *Les cinémas changent régulièrement les films qui passent dans leurs salles. On souhaite garder l'historique de ces changements, afin de connaître la succesion des films proposés dans chaque salle.*

Voici l'ordre de création de la table Séance.

```
CREATE TABLE Seance  (idFilm       INTEGER NOT NULL,
                      nomCinema    VARCHAR (30) NOT NULL,
                      noSalle      INTEGER NOT NULL,
                      idHoraire    INTEGER NOT NULL,
                      PRIMARY KEY (idFilm, nomCinema, noSalle, idHoraire),
                      FOREIGN KEY (nomCinema, noSalle) REFERENCES Salle,
                      FOREIGN KEY (idFilm) REFERENCES Film,
                      FOREIGN KEY (idHoraire) REFERENCES Horaire);
```

Chaque fois que l'on modifie le code du film dans une ligne de cette table, il faut insérer dans une table AuditSeance *l'identifiant de la séance, le code de l'ancien film et le code du nouveau film.*

1. *Donnez l'ordre de création de la table* AuditSéance.

2. *Donnez le* trigger *qui alimente la table* AuditSéance *en fonction des mises à jour sur la table* Séance.

Exercice 7.5 *Supposons qu'un système propose un mécanisme de triggers, mais pas de contrainte d'intégrité référentielle. On veut allors implanter ces contraintes avec des triggers.*

Donner les triggers qui implantent la contrainte d'intégrité référentielle entre Artiste *et* Film. *Pour simplifier, on suppose qu'il n'y a jamais de valeur nulle.*

Chapitre 8

Programmation avec SQL

Sommaire

Ce chapitre présente l'intégration de SQL et d'un langage de programmation classique. L'utilisation conjointe de deux langages résulte de l'insuffisance de SQL en matière de traitement de données : on ne sait pas faire de boucle, de tests, etc. On utilise donc SQL pour extraire les données de la base, et le C (ou tout autre langage) pour manipuler ces données. La difficulté principale consiste à transcrire des données stockées selon des types SQL en données manipulées par le langage de programmation

La présentation reste volontairement très succincte : il s'agit d'illustrer le mécanisme de cohabitation entre les deux langages. L'objectif est donc simplement de montrer comment on se connecte, comment on extrait des données de la base, et de donner quelques indications et conseils sur la gestion des erreurs, la structuration du code et les erreurs à éviter.

8.1 Interfaçage avec le langage C

Le contenu consiste essentiellement à détailler un programme réel qui peut s'exécuter sous le SGBD Oracle. L'interface SQL/C est normalisée dans SQL2, et l'exemple que l'on va trouver ensuite est très proche de cette norme.

8.1.1 Un exemple complet

Pour commencer, voici un exemple complet. Il suppose qu'il existe dans la base une table `Film` dont voici le schéma (voir `film.sql`) :

```
CREATE TABLE film  (ID_film        NUMBER(10) NOT NULL,
                    Titre          VARCHAR (30),
                    Annee          NUMBER(4),
                    ID_Realisateur NUMBER(10));
```

Voici un programme qui se connecte et recherche le film d'id 1. Bien entendu les numéros en fin de ligne sont destinés aux commentaires.

```
#include <stdio.h>

EXEC SQL INCLUDE sqlca;                              (1)

typedef char asc31[31];                              (2)

int main (int argc, char* argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;                    (3)

  EXEC SQL TYPE asc31 IS STRING(31) REFERENCE;       (2')

  int        ora_id_film, ora_id_mes, ora_annee;
  char  user_id = '/';
  asc31        ora_titre;                            (2'')

  short vi1, vi2, vi3;                               (4)

  EXEC SQL END DECLARE SECTION;                      (3')
  EXEC SQL WHENEVER SQLERROR goto sqlerror;          (5)

  EXEC SQL CONNECT :user_id;                         (6)

  ora_id_film = 1;                                   (7)
  ora_id_mes = 0; ora_annee = 0;
  strcpy (ora_titre,"");

  EXEC SQL SELECT titre, annee, id_realisateur       (8)
  INTO  :ora_titre:vi1, :ora_annee:vi2,
        :ora_id_mes:vi3
  FROM  film
  WHERE id_film = 1;

  printf ("Titre : %s Annee : %d Id mes %d  \n",
   ora_titre, ora_annee, ora_id_mes);

sqlerror: if (sqlca.sqlcode != 0)                    (9)
        fprintf (stderr,"Erreur  NO %d : %s\n",
          sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
}
```

Il reste à précompiler ce programme (le SGBD remplace alors tous les `EXEC SQL` par des appels à ses propres fonctions C), à compiler le `.c` résultant de la précompilation, et à faire l'édition de lien avec les librairies pertinentes. Voici les commentaires pour chaque partie du programme ci-dessus.

1. cette ligne est spécifique à Oracle aui communique avec le programme *via* la structure `sqlca` : il faut inclure le fichier `sqlca.h` **avant** la précompilation, ce qui se fait avec la commande `EXEC SQL INCLUDE`.

2. Le principal problème en PRO*C est la conversion des `VARCHAR` de SQL en chaînes de caractères C contenant le fameux `\0`. Le plus simple est de définir explicitement l'équivalence entre un type

manipulé par le programme et le type SQL correspondant. Cela se fait en deux étapes :

(a) On fait un `typedef` pour définir le type du programme : ici le type `asc31` est synonyme d'une chaîne de 31 caractères C.

(b) On utilise (ligne 2') la commande `EXEC SQL TYPE` pour définir l'équivalence avec le type SQL.

Maintenant, le SGBD gérera convenablement la conversion des `VARCHAR` vers une variable C (2") en ajoutant le `\0` après le dernier caractère non-blanc.

3. Le transfert entre la base de données et le C se fait par l'intermédiaire de "variables hôtes" qui doivent être déclarées dans un bloc spécifique (3 et 3').

4. Il n'y a pas, en C, l'équivalent de la "valeur nulle" (qui correspond en fait à **l'absence** de valeur). Pour savoir si une valeur ramenée dans un ordre SQL est nulle, on doit donc utiliser une variable spécifique, dite **indicatrice**. Cette variable est toujours de type `short`

5. Dans le cas où une erreur survient au moment de l'exécution d'un ordre SQL, il faut indiquer le comportement à adopter. Ici on se déroute sur une étiquette `sqlerror`.

6. Connexion à la base : indispensable avant tout autre ordre. Ici on utilise la connexion automatique Oracle.

7. Initialisation des variables de communication. **Cette initialisation est indispensable**.

8. Exemple d'un ordre `SELECT`. Pour chaque attribut sélectionné, on indique dans la clause `INTO` la variable réceptrice **suivi de la variable indicatrice**. Attention le SGBD génère une erreur si on lit une valeur nulle sans utiliser de variable indicatrice.

9. Gestion des erreurs : le champ `sqlcode` de la structure `sqlca` et mis à jour après chaque ordre SQL. Quand il vaut 0, c'est qu'on n'a pas rencontré d'erreur. La valeur 1403 (spécifique Oracle) indique qu'aucune ligne n'a été trouvée. Toute valeur négative indique un erreur, auquel cas le message se trouve dans `sqlca.sqlerrm.sqlerrmc`.

A peu près l'essentiel de ce qui est suffisant pour écrire un programme C/SQL se trouve dans le code précédent. La principale fonctionnalité non évoquée ci-dessus est l'emploi de **curseurs** pour parcourir un ensemble de n-uplets. SQL manipule des ensembles, notion qui n'existe pas en C : il faut donc parcourir l'ensemble ramené par l'ordre SQL et traiter les tuples un à un. Voici la partie du code qui change si on souhaite parcourir l'ensemble des films.

```
/* Comme précédemment, jusqu'a EXEC SQL WHENEVER SQLERROR ... */

EXEC SQL DECLARE CFILM CURSOR FOR
SELECT id_film, titre, annee, id_realisateur
FROM film;

EXEC SQL CONNECT :user_id;

ora_id_film = 0; ora_id_mes = 0;
ora_annee = 0; strcpy (ora_titre,"");

EXEC SQL OPEN CFILM;

EXEC SQL FETCH CFILM INTO :ora_id_film:vi1, :ora_titre:vi2,
                          :ora_annee:vi3, :ora_id_mes:vi4;

while (sqlca.sqlcode != ORA_NOTFOUND)
```

```
  {
    printf ("Film no %d. Titre : %s Annee : %d Id mes %d  \n",
          ora_id_film, ora_titre, ora_annee, ora_id_mes);

    EXEC SQL FETCH CFILM INTO :ora_id_film:vi1, :ora_titre:vi2,
                              :ora_annee:vi3, :ora_id_mes:vi4;
  }
  EXEC SQL CLOSE CFILM;

  /* Comme avant ... */
```

On déclare un curseur dès qu'un ordre SQL ramène potentiellement plusieurs n-uplets. Ensuite chaque appel à la clause `FETCH` accède à un n-uplet, jusqu'à ce que `sqlca.sqlcode` soit égal à 1403 (ici on a déclaré une constante `ORA_NOTFOUND`).

Comme d'habitude, il est recommandé d'organiser le code avec des fonctions. D'une manière générale, il paraît préférable de bien séparer le code gérant les accès à la base du code implantant l'application proprement dite. Quelques suggestions sont données dans la section suivante.

8.1.2 Développement en C/SQL

La recherche d'information dans une table est une bonne occasion d'isoler une partie bien spécifique du code en créant une fonction chargée d'accéder à cette table, de vérifier la validité des données extraites de la base, d'effectuer les conversions nécessaires, etc. De plus une telle fonction a toutes les chances d'être utile à beaucoup de monde.

Les deux cas les plus courants d'accès à une table sont les suivants.

1. **Recherche** d'un n-uplet avec la clé.
2. **Boucle** sur les n-uplets d'une table en fonction d'un intervalle de valeurs pour la clé.

Du point de vue de la structuration du code, voici les stratégies qui me semblent les plus recommandables pour chaque cas.

Recherche avec la clé

1. On définit une structure correspondant au sous-ensemble des attributs de la table que l'on souhaite charger dans le programme.
2. On définit une fonction de lecture (par exemple `LireFilm`) qui prend en entrée un pointeur sur une structure et renvoie un booléen. Au moment de l'appel à la fonction, on doit avoir initialisé le champ correspondant à la clé.
3. Dans la fonction, on exécute l'ordre SQL, on effectue les contrôles nécessaires, on place les données dans les champs de la structure. On renvoie `TRUE` si on a trouvé quelque chose, `FALSE` sinon.

Voici par exemple le squelette de la fonction `LireFilm`.

```
boolean LireFilm (Film *film)
{
  /* Declarations */
  ....
  /* Initialisations */
  ...
  ora_id_film = film->id_film;
  ...
```

```
  /* Ordre SELECT  */
  EXEC SQL SELECT ...

  /* Test          */
  if (sqlca.sqlcode == ORA_NOTFOUND) return FALSE;
  else ...

  /* Controles divers et placement dans la structure */
  ...
  film->id_film = ora_id_film;
  ...
  return TRUE;
}
```

Et voici comment on appelle la fonction.

```
  Film film;
  ...
  film.id_film = 34;
  if (LireFilm (&film))
    ... /* On a trouve le n-uplet  */
  else
    ... /* On n'a rien trouve */
```

Donc la fonction appelante ne voit rien de l'interface SQL et peut se consacrer uniquement à la manipulation des données.

Recherche par intervalle

On peut suivre à peu près les mêmes principes, à ceci près qu'il faut :

1. Passer en paramètre les critères de recherche.
2. Gérer l'ouverture et la fermeture du curseur.

Pour le deuxième point on peut procéder comme suit. On place dans la fonction une variable statique initialisée à 0. Au premier appel, cette variable est nulle, et on doit ouvrir le curseur et changer la valeur à 1 avant de faire le premier `FETCH`. Aux appels suivants la valeur est 1 et on peut faire simplement des `FETCH`. Quand on a atteint le dernier n-uplet, on ferme le curseur et on remet la variable à 0. Voici le squelette de la fonction `BoucleFilms` qui effectue une recherche sur un intervalle de clés.

```
boolean BoucleFilms (Film *film, int cle_min, int cle_max)
{
 /* Declarations des variables et du curseur, initialisations ... */
 ...
 static debut = 0;
 ...

 /* Test d'ouverture du curseur      */
 if (debut == 0)
 {
   EXEC SQL OPEN ...
   debut = 1;
 }

 /* Dans tous les cas on fait le FETCH     */
```

```
 EXEC SQL FETCH ...

 if (sqlca.sqlcode == ORA_NOTFOUND)
 {
   EXEC SQL CLOSE ...
   debut = 0;
   return FALSE;
 }
 else
 {
   /* Faire les contrôles et placer les données dans film */
   ...
   return TRUE
 }
}
```

Voici comment on utilise cette fonction.

```
Film film;
int cle_min, cle_max;
...
while (BoucleFilms (&film, cle_min, cle_max))
{
  ....
}
```

Notez qu'avec l'utilisation combinée des fonctions et des structures, non seulement on clarifie beaucoup le code, mais on rend très facile l'ajout d'une nouvelle donnée. Il suffit de modifier la structure et l'implantation de la fonction de lecture. Tout le reste est inchangé.

8.1.3 Autres commandes SQL

Voici, à titre de complément, les principales fonctionnalités dáccès à une base de données et leur expression en C/SQL.

Validation et annulation

1. Validation : `EXEC SQL COMMIT WORK;`
2. Anulation : `EXEC SQL ROLLBACK WORK;`

Si on ne fait pas de `COMMIT` explicite, Oracle effectue un `ROLLBACK` à la fin du programme.

`UPDATE, DELETE, INSERT`

On utilise ces commandes selon une syntaxe tout à fait semblable à celle du `SELECT`. Voici des exemples.

```
/* Les variables ora_... sont déclarées comme précédemment */

EXEC SQL INSERT INTO film (id_film, titre, annee, id_mes)
VALUES   (:ora_id_film, :ora_titre, :ora_annee, :ora_mes);
...
EXEC SQL DELETE FROM film
WHERE id_film = :ora_id_film;
...
EXEC SQL UPDATE film SET annee = :ora_annee, id_mes=:ora_id_mes
WHERE  id_film = :ora_id_film;
```

Valeurs nulles

On teste les valeurs nulles avec les variables indicatrices (voir ci-dessus). Une valeur de -1 après l'exécution d'un `SELECT` indique que la valeur extraite de la base est nulle (spécifique Oracle).

```
#define ORA_NULL -1
  ...
  EXEC SQL SELECT ... INTO :ora_id_mes:vi ...
  ...
  if (vi == ORA_NULL)
    /* L'identifiant du metteur en scene est inconnu */
  ...
```

8.2 L'interface Java/JDBC

JDBC (acronyme qui signifie probablement "Java Database Connectivity" par analogie avec ODBC), est un ensemble de classes Java qui permet de se connecter à une base de données distante sur le réseau, et d'interroger cette base afin d'en extraire des données. JDBC offre quelques différences notables par rapport à une solution CGI ou à une interface web propriétaire et spécifique à un système particulier :

- JDBC offre une intégration très étroite du client et des modules chargés de l'accès à la base. Cela permet de limiter le trafic réseau.
- JDBC est complètement indépendant de tout SGBD : la même application peut être utilisée pour accéder à une base ORACLE, SYBASE, MySQL, etc. Conséquences : pas besoin d'apprendre une nouvelle API quand on change de système, et réutilisation totale du code.
- Enfin, JDBC est relativement simple, beaucoup plus simple par exemple que l'interface C+SQL proposée par les SGBD relationnels.

Cette présentation ne couvre pas tous les aspects de JDBC. Il existe un livre, très correct, qui donne une présentation presque exhaustive :

George Reese, *Database programming with JDBC and Java*, O'Reilly.

Une traduction en français est disponible. Dans la suite de ce texte vous trouverez une description des principes de JDBC, et une introduction à ses fonctionnalités, essentiellement basée sur des exemples simples utilisant un accès à une base MySQL ou ORACLE. Le code est disponible sur le site `http://sikkim.cnam.fr/or`

8.2.1 Principes de JDBC

L'utilisation de JDBC se fait dans le cadre de code Java. Ce peut être un programme classique (voir l'exemple ci-dessous) ou une applet destinée à être transférée sur un site client via le Web. Le client peut alors interroger une ou plusieurs bases distantes au travers du réseau : ce dernier aspect est le plus original et le plus intéressant.

La figure 8.1 montre les couches logicielles utilisées lors d'une connexion à distance à des bases de données. L'applet comprend du code java standard, et une partie basée sur les classes JDBC qui permet d'effectuer des requêtes SQL. Le dialogue avec la base distante se fait par l'intermédiaire d'une *connexion*, qui elle-même utilise les services d'un *driver*. Le driver communique avec un *serveur* sur la machine hébergeant la base de données.

Connexion

Quand une requête doit être exécutée, elle le fait par l'intermédiaire d'une *connexion*. Une connexion est un objet Java de la classe `Connection` qui est chargé de dialoguer avec une base de données. Dans

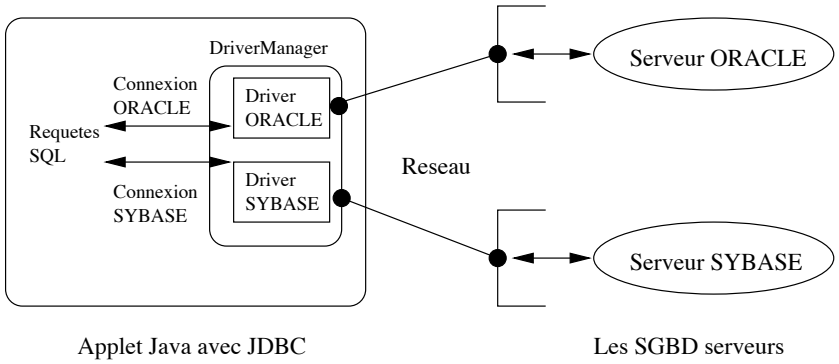


FIG. 8.1 – *Mise en œuvre des drivers JDBC*

le cas où on souhaite accéder à plusieurs bases de données, comme montré sur la figure 8.1, il faut autant d'objets `Connection`.

Une connexion correspond à la notion de transaction : on effectue des requêtes ou des mises-à-jour que l'on valide ou annule ensuite. On peut donc ouvrir plusieurs connexions sur une même base si on souhaite gérer plusieurs transactions simultanément.

Drivers

Quand on veut établir une connexion avec une base distante, on doit passer par l'intermédiaire d'un *driver*. Le driver est la partie de JDBC qui est spécifique à un SGBD particulier comme ORACLE ou SYBASE. Le driver ORACLE sait comment dialoguer avec un serveur ORACLE, mais est incapable d'échanger des données avec un serveur SYBASE. Pour accéder à un SGBD particulier, il suffit d'instancier un object de la classe `Driver` propre à ce SGBD.

Ce n'est pas en contradiction avec l'indépendance du code Java. Tous les drivers ont la même interface et s'utilisent de la même façon. On peut, de manière totalement dynamique (par exemple au moment de l'exécution de l'applet) choisir la base à laquelle on veut accéder, et instancier le driver correspondant.

Il existe plusieurs types de drivers. Le choix dépend de l'utilisation de JDBC. En local, pour une application, ou en distribué, pour une *applet*. Dans ce dernier cas on utilise un driver de type 3 ou 4 qui ne nécessite pas l'installation d'un logiciel spécifique sur le client. Le driver utilisé dans les exemples ci-dessous et le driver « *thin* » MySQL, de type 4, qui communique directement avec le serveur MySQL par des sockets.

Serveur

Dernier élément de cette architecture : le SGBD doit gérer un serveur sur la machine hôte, qui reçoit, interprète et exécute les demandes du driver. Il existe plusieurs solutions possibles, qui dépendent du type de driver utilisé. Ce qui importe, du point de vue de l'utilisateur d'une applet JDBC, c'est de connaître le nom de la machine hôte, et le numéro du port sur lequel le serveur est en écoute. Pour MySQL, le port est en général le 3306, pour ORACLE le 1521.

Dans ce qui suit, nous prendrons l'exemple de la machine `cartier.cnam.fr` hébergeant une base de données MySQL dont le nom est `Films`. Comme son nom l'indique, cette base contient des données diverses et variées sur des films, des metteurs en scène, des acteurs, etc. Le serveur est en écoute sur le port 3306.

Important : quand on utilise une applet, les règles de sécurité Java limitent les possibilités d'ouverture de socket pour dialoguer avec d'autres machines. La règle, par défaut, est de n'autoriser un dialogue par socket qu'avec la machine qui héberge le serveur `httpd`. Cela signifie ce serveur et le serveur MySQL ou ORACLE doivent être situés sur le même hôte.

Au lieu d'utiliser un navigateur, on peut toujours tester une applet avec le programme `appletviewer` qui ne passe pas par le réseau et n'est donc pas soumis aux règles de sécurité Java.

8.2.2 Le plus simple des programmes JDBC

Voici un premier programme JDBC (ce n'est pas une applet !). Il se connecte à la base, recherche l'ensemble des films, et affiche les titres à l'écran.

```
// D'abord on importe le package JDBC
import java.sql.*;

class films
{
  public static void main (String args [])
       throws SQLException
  {
    Connection conn ;

    // Chargement du driver de MySQL
    DriverManager.registerDriver(new org.gjt.mm.mysql.Driver());

    // Connection à la base
   try {
    conn = DriverManager.getConnection
         ("jdbc:mysql://localhost/Films",
   "visiteurFilms", "mdpVisiteur");

    // Exécution de la requête qui ramène les titres des films
    Statement stmt = conn.createStatement ();

    ResultSet rset = stmt.executeQuery ("select titre from Film");

    // Affichage du résultat
    while (rset.next ())
System.out.println (rset.getString (1));
   }
   catch (SQLException e)
   {
     System.out.println ("Problème quelque part !!!");
     System.out.println (e.getMessage());
   }
 }
}
```

Le programme commence par importer le package `java.sql.*` qui correspond à l'ensemble des classes JDBC[1].

La première instruction consiste à instancier le driver MySQL, et à l'enregistrer dans le `DriverManager`. Ce dernier est alors prêt à utiliser ce driver si on demande une connexion à une base MySQL.

C'est justement ce que fait l'instruction suivante : on instancie un objet de la classe `Connection` en lui passant en paramètres :

– Une URL contenant les coordonnées de la base. Ici on indique le driver MySQL dont le nom est

1. Attention : ce package n'existe en standard qu'avec la version 1.1 du JDK. Pour la version 1.0, les classes JDBC font partie du package comprenant le driver, fourni par chaque SGBD.

`org.gjt.mm.mysql.Driver()`, le nom de la machine (ici `localhost`), et le nom de la base (`Films`). Le format de l'URL dépend de chaque driver.

– Le nom et le mot de passe de l'utilisateur qui se connecte à la base. Ici, il s'agit d'un compte qui peut seulement effectuer des lectures dans la base.

Il est important de noter que l'instanciation du driver et la connexion (ainsi que toutes les requêtes qui suivent) peuvent échouer pour quantité de raisons. Dans ce cas une exception de type `SQLException` est levée. Il est indispensable de placer les instructions dans des blocs `try` et de gérer les exceptions.

Il ne reste plus qu'à effectuer une requête pour tester la connexion. Une requête (au sens large : interrogation ou mise à jour) correspond à un objet de la classe `Statement`. Cet objet doit avoir été créé par un objet `Connection`, ce qui le rattache automatiquement à l'une des transactions en cours.

La méthode `executeQuery`, comme son nom l'indique, exécute une requête (d'interrogation) placée dans une chaîne de caractères. Le résultat est placé dans un objet `ResultSet` qui, comme son nom l'indique encore une fois, contient l'ensemble des lignes du résultat.

Un objet `ResultSet` correspond à peu près à la notion de *curseur* employée systématiquement dans les interfaces entre un langage de programmation et SQL. Un curseur permet de récupérer les lignes du résultat à la demande, une par une. Ici on appelle simplement la méthode booléenne `next` qui renvoie `true` tant que le résultat n'a pas été parcouru entièrement. Chaque appel à `next` positionne le curseur sur une nouvelle ligne.

Finalement, la classe `ResultSet` propose un ensemble de méthodes `get***` qui prennent un numéro d'attribut en entrée et renvoient la valeur de cet attribut. Toute erreur de type ou d'indice renvoie une `SQLException`.

Il est facile de se convaincre, à la lecture de ce petit programme, de la simplicité de JDBC. L'utilisation de quelques classes bien conçues permet de s'affranchir de tous les détails techniques fastidieux que l'on trouve, par exemple, dans les protocoles d'échange C/SQL.

8.2.3 Exemple d'une applet avec JDBC

Voici maintenant un exemple complet d'une applet JDBC. Le but de cette applet est de permettre la saisie du titre d'un film et de l'horaire souhaité, et la partie JDBC se charge de rechercher dans la base les films qui satisfont les critères de sélection.

Description de l'applet

L'applet s'appelle `JdbcFilms` et se trouve dans le fichier `JdbcFilms.java`. On inclut la demande d'exécution dans une page HTML, dont voici le contenu :

```
<html>
<head>
<title>Illustration d'une Applet JDBC</title>
</head>
<body>

Cette page contient un exemple d'une applet permettant
de se connecter à MySQL (ou ORACLE) par l'intermédiaire
d'un driver JDBC, et d'interroger une base de données.

<p>

Le code source de cette applet est dans
<a href="JdbcFilms.java">JdbcFilms.java</a>.
Attention au nom du driver, ainsi qu'à la chaîne de connexion.
<P>
```

```
Saisissez un titre de film, complet, ou partiel en plaçant
le caractère '%'. Indiquez également un intervalle d'années.

<hr>
<CENTER><applet code="JdbcFilms" width=700 height=200>
</applet>
</CENTER>
<hr>
</BODY>
</HTML>
```

Le code Java/JDBC

Voici le code complet de l'applet. La majeure partie correspond à la création de l'interface graphique. Le code JDBC lui même est très réduit.

```
// Import du package JDBC.
import java.sql.*;

// Import des packages de base
import java.awt.*;
import java.io.*;
import java.util.*;

public class JdbcFilms extends java.applet.Applet
{
  // La requete

  String requete;
  int   nb_lignes;

  // Les boutons pour exécuter ou effacer
  Button bouton_exe, bouton_eff;

  // Les champs éditables
  TextField titre_f, anMax, anMin;

  // La fenêtre où on affiche le résultat
  TextArea fenetre_res;

  // La connexion à la base de données
  Connection conn = null;

  ResultSet rset;

  // L'initialisation de l'applet crée l'interface graphique
  public void init ()
  {
    this.setLayout (new BorderLayout ());
    Panel p = new Panel ();
    p.setLayout (new FlowLayout (FlowLayout.LEFT));
    bouton_exe = new Button ("Exécuter la requête");
    p.add (bouton_exe);
    bouton_eff = new Button ("Effacer tout");
    p.add (bouton_eff);
    this.add ("North", p);
```

```
  Panel p2 = new Panel();
  Label titre_l = new Label ("Titre du film: ", Label.LEFT);
  p2.add (titre_l);
  titre_f = new TextField ("%", 20);
  p2.add (titre_f);
  Label periode_l = new Label ("Période. De", Label.LEFT);
  p2.add (periode_l);
  anMin = new TextField ("1900", 4);
  anMax = new TextField ("2100", 4);
  p2.add(anMin);
  Label anmax_l = new Label ("à");
  p2.add (anmax_l);
  p2.add(anMax);
  this.add ("South", p2);

  fenetre_res = new TextArea (30, 700);
  this.add ("Center", fenetre_res);

  // Chargement du driver, et création d'une connexion
  try
  {
     DriverManager.registerDriver(new org.gjt.mm.mysql.Driver());
     //DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
     //conn = DriverManager.getConnection
     //  ("jdbc:oracle:thin:@celsius.cnam.fr:1521:CNAMTP",
     //    "scott", "tiger");
  }
  catch (SQLException e)
  {
    fenetre_res.appendText (e.getMessage () + "\n");
  }
}

/** Méthode déclenchée quand on appuie sur le bouton */
public boolean action (Event ev, Object arg)
{
  if (ev.target == bouton_exe)
  {
    try
    {
      // Création de la requête
      conn = DriverManager.getConnection
                          ("jdbc:mysql://cartier.cnam.fr/Films",
 "visiteurFilms", "mdpVisiteur");
      Statement stmt = conn.createStatement ();

      requete = "select  titre, annee, codePays, prenom,  nom"
                + " from   Film f, Artiste a "
         + " where titre LIKE '" + titre_f.getText()
       + "' and  f.idMES = a.id "
       + " and annee between " + anMin.getText()
       + " and " + anMax.getText();

      // Execution de la requête

      nb_lignes =0;
      rset = stmt.executeQuery (requete);
```

```
        // Affichage du résultat
        while (rset.next ())
        {
          fenetre_res.appendText (
            "Film: " +  rset.getString (1)
            + ", " + rset.getString (2)
            + ", " + rset.getString (3)
            + " Réalisé par  " + rset.getString (4)
            + " " +  rset.getString (5) + "\n");
          nb_lignes++;
        }

        if (nb_lignes == 0)
         fenetre_res.appendText (
             "Rien trouvé pour les films " + titre_f.getText()
            + " et la période " + anMin.getText()
            + "/" + anMax.getText());
      }
      catch (Exception e)
      {
        // Caramba: pb quelque part
        fenetre_res.appendText ("Erreur rencontrée !\n");
        fenetre_res.appendText (e.getMessage () + "\n");
      }
      return true;
    }
    else if (ev.target == bouton_eff)
    {
      fenetre_res.setText (" ");
      titre_f.setText("%");
      anMin.setText ("1900");
      anMax.setText ("2100");

      return true;
    }
    else   return false;
  }
}
```

Deuxième partie

Aspects systèmes

Chapitre 9

Techniques de stockage

Sommaire

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

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

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

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

9.1 Stockage de données

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

9.1.1 Supports

D'une manière générale, plus une mémoire est rapide, plus elle est chère et – conséquence directe – plus sa capacité est réduite. Les différentes mémoires utilisées par un ordinateur constituent donc une hiérarchie (figure 9.1), allant de la mémoire la plus petite mais la plus efficace à la mémoire la plus volumineuse mais la plus lente.

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

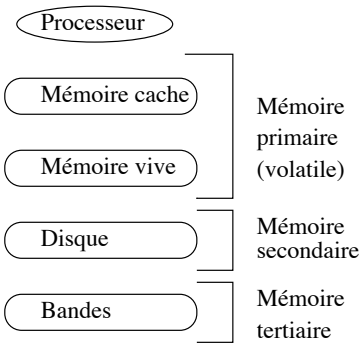


FIG. 9.1 – *Hiérarchie des mémoires*

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

La technologie évoluant rapidement, il est délicat de donner des valeurs précises pour la taille et les temps d'accès des différentes mémoires. Le tableau 9.1 propose quelques ordres de grandeur. On peut retenir qu'en 2001, un ordinateur est équipé de quelques centaines de mégaoctets de mémoire vive (typiquement 256 Mo à l'heure où ces lignes sont écrites) et que les disques stockent quelques gigaoctets (typiquement 6 à 10 Go). En contrepartie, le temps d'accès à une information en mémoire vive est de

Type mémoire	Taille (en Mo)	Temps d'accès (secondes)
Mémoire *cache*	Env. 1 Mo	$\approx 10^{-8}$ (10 nanosec.)
Mémoire principale	$O(10^2)$ Mo	$\approx 10^{-8} - 10^{-7}$ (10-100 nanosec.)
Mémoire secondaire (disque)	$O(10^{12})$ (Gygaoctets)	$\approx 10^{-2}$ (10 millisec.)
Mémoire tertiaire (bande/CD)	$O(10^{15})$ (Téraoctets)	≈ 1 seconde

TAB. 9.1 – *Caractéristiques des différentes mémoires*

l'ordre de 10 nanosecondes (10^{-8}) tandis que le temps d'accès sur un disque est de l'ordre de 10 millisecondes (10^{-2}), ce qui représente un ratio approximatif de 1 000 000 entre les performances respectives de ces deux supports ! Il est clair dans ces conditions que le système doit tout faire pour limiter les accès au disque.

9.1.2 Fonctionnement d'un disque

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

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

Dispositif

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

Un disque est entraîné dans un mouvement de rotation régulier par un axe. Une *tête de lecture* (deux si le disque est double-face) vient se positionner sur une des pistes du disque et y lit ou écrit les données. Le nombre minimal d'octets lus par une tête de lecture est physiquement défini par la taille d'un secteur (en général 512K). 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 512K (un secteur), 1024K (deux secteurs) ou 4096K (huit secteurs).

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

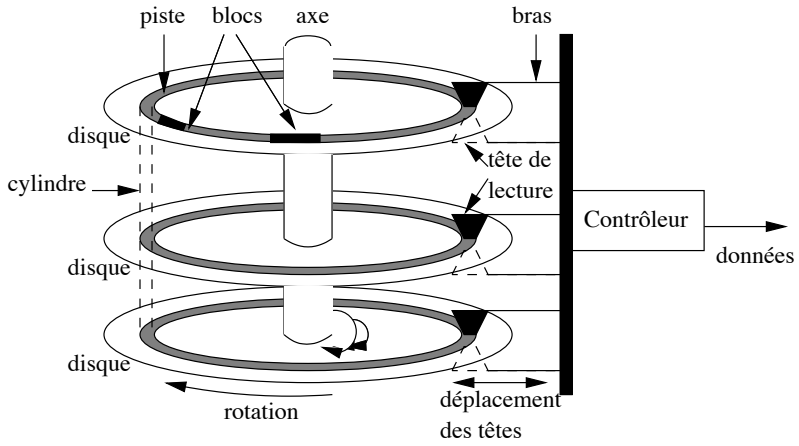


FIG. 9.2 – *Un disque magnétique*

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 4096 octets pour accéder à un entier de 4 octets, les 4094 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. Cette motivation est à la base du mécanisme de *regroupement* qui fonde, notamment, les structures d'index et de hachage.

La tête de lecture n'est pas entraînée dans le mouvement de rotation. Elle se déplace dans un plan fixe qui lui permet de se rapprocher ou de s'éloigner de l'axe de rotation des disques, et d'accéder à une des pistes. Pour limiter le coût de l'ensemble de ce dispositif et augmenter la capacité de stockage, les disques sont empilés et partagent le même axe de rotation (voir figure 9.2). 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.

Accès aux données

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

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

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

– *positionnement de la tête de lecture* sur la piste contenant le bloc ;

– *rotation du disque* pour attendre que le bloc passe sous la tête de lecture (rappelons que les têtes sont fixe, c'est le disque qui tourne) ;

– *transfert du bloc*.

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

Le tableau 9.2 donne les spécifications d'un disque en 2001, telles qu'on peut les trouver sur le site de n'importe quel constructeur (ici *Seagate*, *www.seagate.com*). Les chiffres donnent un ordre de grandeur pour les performances d'un disque, étant bien entendu que les disques destinés aux serveurs sont beaucoup plus performants que ceux destinés aux ordinateurs personnels. Le modèle donné en exemple dans le tableau 9.2 appartient au mileu de gamme.

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

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

TAB. 9.2 – *Spécifications du disque Cheetah 18LP (source www.seagate.com)*

Le nombre de secteurs par piste n'est pas constant, car les pistes situées près de l'axe sont bien entendu beaucoup plus petite que elles situées près du bord du disque. On ne peut, à partir des spécifications, que calculer le nombre moyen de secteurs par piste, qui est égal à $2\,963\,906/9\,772 = 303$. On peut donc estimer qu'une piste stocke en moyenne $303 \times 512 = 155\,292$ octets. Ce chiffre donne le nombre d'octets qui peuvent être lus sans délai de latence ni délai de positionnement.

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

Ce temps de transfert peut être considéré comme négligeable dans le cas d'un bloc unique, comme le montre le raisonnement qui suit, basé sur les performances du tableau 9.2. Le disque effectue 10000 rotations par minute, ce qui correspond à 166,66 rotations par seconde, soit une rotation toutes les 0,006 secondes (6 ms). C'est le temps requis pour lire une piste entièrement. Cela donne également le temps moyen de latence de 3 ms.

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

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

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

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

9.1.3 Optimisations

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

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

Regroupement

Prenons un exemple simple pour se persuader de l'importance d'un bon regroupement des données sur le disque : le SGBD doit lire 5 chaînes de caractères de 1000 octets chacune. Pour une taille de bloc égale à 4096 octets, deux blocs peuvent suffire. La figure 9.3 montre deux organisations sur le disque. Dans la première chaque chaîne est placée dans un bloc différent, et les blocs sont répartis aléatoirement sur les pistes du disque. Dans la seconde organisation, les chaînes sont rassemblés dans deux blocs qui sont consécutifs sur une même piste du disque.

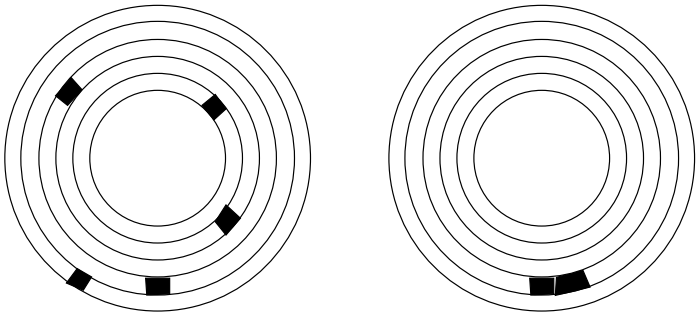


FIG. 9.3 – *Mauvaise et bonne organisation sur un disque*

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

Les performances obtenues sont dans un rapport de 1 à 5, le tempsminimal 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 :

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

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

Pour que le SGBD puisse effectuer ces optimisations, il doit se voir confier, à la création de la base, un espace important sur le disque dont il sera le seul à gérer l'organisation. Si le SGBD se contentait de demander au système d'exploitation de la place disque quand il en a besoin, le stockage physique obtenu serait extrêmement fragmenté.

Séquencement

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

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

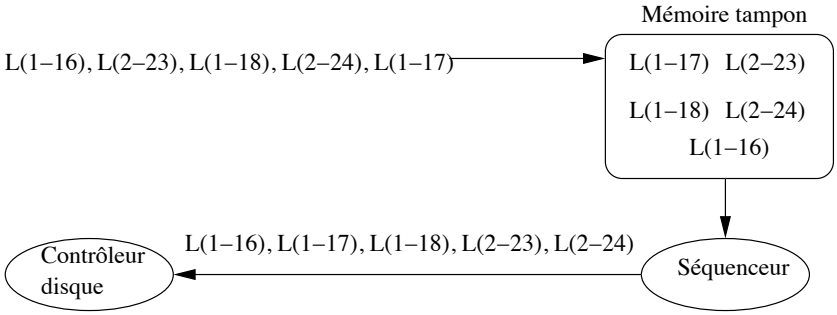


FIG. 9.4 – *Séquencement des entrées/sorties*

Le système d'exploitation, ou le SGBD, peuvent réduire cet inconvénient en conservant temporairement les demandes d'entrées/sorties dans une zone tampon (*cache*) et en réorganisant (*séquencement*) l'ordre des accès. La figure 9.4 montre le fonctionnement d'un séquenceur. Un ensemble d'ordres de lectures est reçu, *L(1-16)* désignant par exemple la demande de lecture du bloc 16 sur la piste 1. On peut supposer sur cet exemple que deux utilisateurs effectuent séparément des demandes d'entrée/sortie qui s'imbriquent quand elles sont transmises vers le contrôleur.

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

Une technique systématique 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 consomateurs 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 attende un temps respectable avant de voir sa requête satisfaite.

Mémoire tampon

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

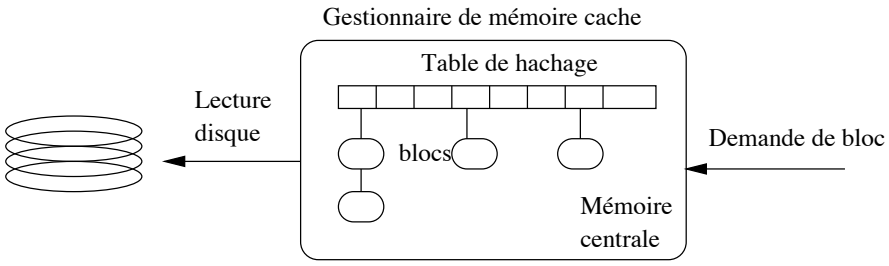


FIG. 9.5 – *Mémoire cache*

L'idée est donc simplement de maintenir en mémoire principale une copie aussi large que possible du disque, même si une grande partie des blocs mis ainsi dans un buffer n'est pas directement utile. Une part importante du paramétrage et de l'administration d'une base de données consiste à spécifier quelle est la part de la mémoire disponible qui peut être attribuée en permanence au SGBD. Plus cette mémoire est importante, et plus il sera possible d'y conserver une partie significative de la base, avec des gains importants en terme de performance.

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

9.1.4 Technologie RAID

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

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

La technologie RAID (pour *Redundant Array of Independent Disks*) a pour objectif principal de limiter les conséquences des pannes en répartissant les données sur un grand nombre de disques, et de manière à

s'assurer que la défaillance de l'un des disques n'entraîne ni perte de données, ni même l'indisponibilité du système.

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

Duplication (RAID 1)

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

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

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

Répartition et parité (RAID 4)

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

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

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

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

Exemple 9.1 *Supposons qu'il y ait 3 disques de données, et que le contenu du premier octet de chaque disque soit le suivant :*

D1: 11110000
D2: 10101010
D3: 00110011

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

DC: 01101001

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

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

D1: 11110000
D3: 00110011
DC: 01101001

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

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

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

Exemple 9.2 Reprenons l'exemple précédent, avec trois disques D1, D2, D3, et le disque de contrôle DC.

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

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

avant : 11110000
après : 10011000

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

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

□

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

Répartition des données de parité (RAID 5)

Dans le RAID 4, le disque de contrôle a tendance a devenir le goulot d'étranglement du système puisque qu'il doit supporter n fois plus d'écritures que les autres. Le RAID 5 propose de remédier à ce problème

en se basant sur une remarque simple : si c'est le disque de contrôle lui-même qui tombe en panne, il est possible de le reconstituer en fonction des autres disques. En d'autres termes, pour la recontruction après une panne, la distinction disque de contrôle/disque de données n'est pas pertinente.

D'où l'idée de ne pas dédier un disque aux données de parité, mais de répartir les blocs de parité sur les $n+1$ disques. La seule modification à apporter par rapport au RAID 5 est de savoir, quand on modifie un bloc sur un disque D_1, quel est le disque D_2 qui contient les données de parité pour ce bloc. Il est possible par exemple de décider que pour le bloc i, c'est le disque $i mod n$ qui stocke le bloc de parité.

Défaillances simultanées (RAID 6)

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

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

9.2 Fichiers

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

La gestion de fichier est un aspect commun aux systèmes d'exploitation et aux SGBD. En théorie le SGBD pourrait s'appuyer sur les fonctionnalités du système d'exploitation qui l'hébèrge, mais cette solution soulève quelques inconvénients

1. il est délicat, en terme d'implantation, de dépendre de modules qui peuvent varier d'un système à l'autre ;
2. les éditeurs de SGBD peuvent ne pas se satisfaire des techniques d'accès aux données proposées par le système ;
3. enfin les systèmes gèrent en général une mémoire tampon qui peut être génante pour les SGBD, notamment pour tout ce qui concerne la concurrence d'accès (rappelons qu'un `commit` doit garantir l'écriture des données sur le disque).

Sauf exception (par exemple MySQL qui a choisi le parti-pris d'une simplicité maximale), les SGBD ont donc leur propre module de gestion de fichiers et de mémoire cache. Leurs principes généraux sont décrits dans ce qui suit.

9.2.1 Enregistrements

Pour le système d'exploitation, un fichier est une suite d'octets répartis sur un ou plusieurs blocs. Les fichiers gérés par un SGBD sont un peu plus structurés. Ils sont constitués *d'enregistrements* (*records* en anglais) qui représentent physiquement les « entités » du SGBD. Selon le modèle logique du SGBD, ces entités peuvent être des n-uplets dans une relation, ou des objets. Nous nous limiterons au premier cas dans ce qui suit.

Un n-uplet dans une table relationnelle est constitué d'une liste d'attributs, chacun ayant un type. À ce n-uplet correspond un enregistrement, constitué de *champs* (*field* en anglais). Chaque type d'attribut détermine la taillle du champ nécessaire pour stocker une instance du type. Le tableau 9.3 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.

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

TAB. 9.3 – *Types SQL et tailles (en octets)*

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

Champs de tailles fixe et variable

Comme l'indique le tableau 9.3, les types de la norme SQL peuvent être divisés en deux catégories : ceux qui peuvent être représentés par un champ une taille fixe, et ceux qui sont représentés par un champ de taille variable.

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

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

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

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

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

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

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

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

Le type `BIT VARYING` peut être représenté comme un `VARCHAR`, mais comme linformation stockée ne contient pas que des caractères codés en ASCII, on ne peut pas utiliser de caractère de terminaison puisqu'on ne saurait par le distinguer des caractères de la valeur stockée. On préfixe donc le champ par la taille utile, sur 2, 4 ou 8 octets selon la taille maximale 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 9.3 Prenons l'exemple d'une table *Film* avec les attributs `id` de type `INTEGER`, `titre` de type `VARCHAR(50)` et `annee` de type `INTEGER`. Regardons la représentation de l'enregistrement (123, 'Vertigo', NULL) (donc l'année est inconnue).

L'identifiant est stocké sur 4 octets, et le titre sur 8 octets, dont un pour la longueur. L'en-tête de l'enregistrement contient un pointeur vers le schéma de la table, sa longueur totale (soit 4 + 8), et un masque de bits 110 indiquant que le troisième champ est à `NULL`. La figure 9.6 montre cet enregistrement : notez qu'en lisant l'en-tête, on sait calculer l'adresse de l'enregistrement suivant.

□

9.2.2 Blocs

Le stockage des enregistrements dans un fichier doit tenir compte du découpage en blocs de ce fichier. En général il est possible de placer plusieurs enregistrements dans un bloc, et on veut éviter qu'un enregistrement chevauche deux blocs. Le nombre maximal d'enregistrements de taille E pour un bloc de taille B est donné par $\lfloor B/R \rfloor$ où la notation $\lfloor x \rfloor$ désigne le plus grand entier inférieur à x.

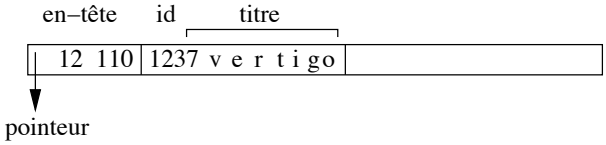


FIG. 9.6 – *Exemple d'un enregistrement avec en-tête*

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 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.

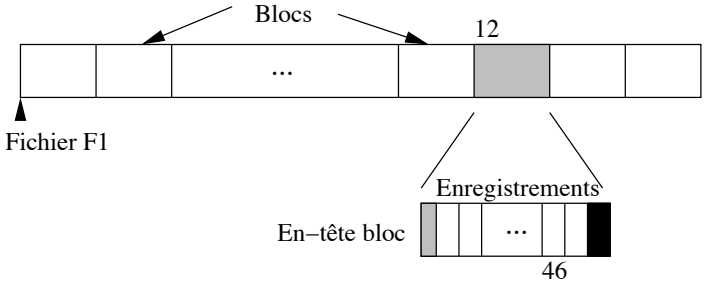


FIG. 9.7 – *Fichier avec blocs et enregistrements*

Le transfert en mémoire de l'enregistrement 563 de ce fichier est simplement effectué en déterminant dans quel bloc il se trouve (soit $\lfloor 563/47 \rfloor + 1 = 12$), en chargeant le douzième bloc en mémoire centrale et en prenant dans ce bloc l'enregistrement. Le premier enregistrement du bloc 12 a le numéro $11 \times 47 + 1 = 517$, et le dernier enregistrement le numéro $12 \times 47 = 564$. L'enregistrement 563 est donc l'avant-dernier du bloc, avec pour numéro interne le 46 (voir figure 9.7).

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

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

Pour permettre le déplacement des enregistrements on peut combiner une *adresse logique* qui identifie un enregistrement indépendamment de sa localisation. Une table de correspondance permet de gérer l'association entre l'adresse physique et l'adresse logique (voir figure 9.8). 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.

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

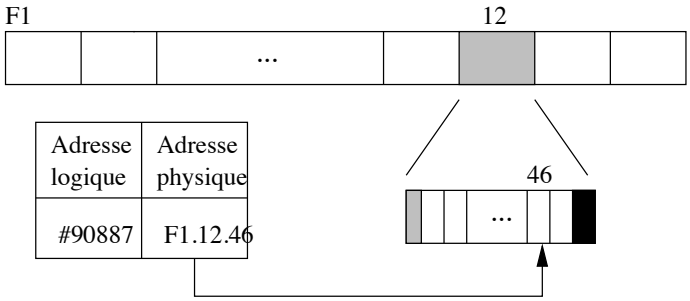


FIG. 9.8 – *Adressage avec indirection*

revanche 46 est une identification logique de l'enregistrement, gérée au sein du bloc. La figure 9.9 montre cet adressage à deux niveaux : dans le bloc F1.12, l'enregistrement 46 correspond à un emplacement au sein du bloc, tandis que l'enregistrement 57 a été déplacé dans un autre bloc.

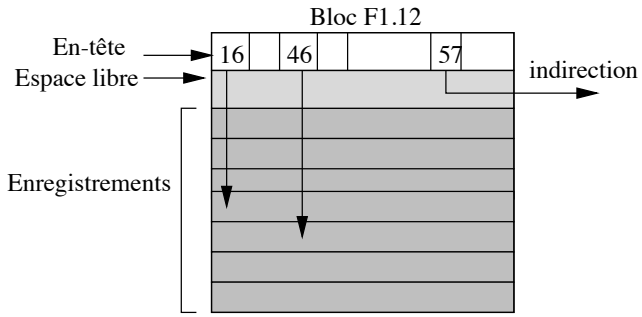


FIG. 9.9 – *Combinaison adresse logique/adresse physique*

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

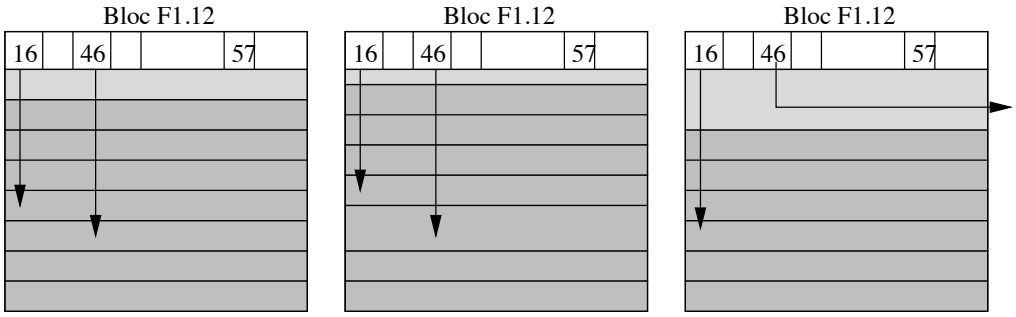
Ce mode d'identification offre beaucoup d'avantages, et est utilisé par ORACLE par exemple. Il permet de réorganiser souplement l'espace interne à un bloc.

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 puise déterminer le début de l'enregistrement suivant.

Il peut arriver que l'enregistrement soit mis à jour, soit pour compléter la valeur d'un attribut, soit pour donner une valeur à un attribut qui était initialement à `NULL`. Dans un tel cas il est possible que la place initialement réservée soit insuffisante pour contenir les nouvelles informations qui doivent être stockées dans un autre emplacement du même fichier. Il faut alors créer un *chaînage* entre l'enregistrement initial et les parties complémentaires qui ont dû être créées. Considérons par exemple le scénario suivant, illustré dans la figure 9.10 :

1. on insère dans la table *Film* un film « Marnie », sans résumé ; l'enregistrement correspondant est stocké dans le bloc F1.12, et prend le numéro 46 ;



(a) Situation initiale

(b) Après agrandissement de l'enregistrement 46

(c) Déplacement de l'enregistrement 46

FIG. 9.10 – *Mises à jour d'un enregistrement de taille variable*

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

3. on s'aperçoit alors que le titre exact est « Pas de printemps pour Marnie », ce qui peut se corriger avec un ordre `UPDATE` : si l'espace libre restant dans le bloc est suffisant, il suffit d'effectuer une réorganisation interne pendant que le bloc est en mémoire centrale, réorganisation qui a un coût nul en terme d'entrées/sorties ;

4. enfin on met à nouveau l'enregistrement à jour pour stocker le résumé qui était resté à `NULL` : cette fois il ne reste plus assez de place libre dans le bloc, et l'enregistrement doit être déplacé dans un autre bloc, tout en gardant la même adresse.

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

Les enregistrements de taille variable sont un peu plus compliqués à gérer que ceux de taille fixe. Les programmes 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.

9.2.3 Organisation d'un fichier

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

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

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

– **Temps**. Une bonne organisation doit favoriser les opérations sur un fichier. En pratique, en 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 p 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 :

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

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

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

Recherche

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

La complexité des critères de sélection n'influe pas sur le coût de la recherche dans un fichier séquentiel. Dans tous les cas on doit partir du début du fichier, lire un par un tous les enregistrements en mémoire centrale, et effectuer à ce moment là le test sur les critères de sélection. Ce test s'effectuant en mémoire centrale, sa complexité peut être considérée comme négligeable par rapport au temps de chargement de tous les blocs du fichier.

Quand on ne sait par 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'arréter 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 :

1. prendre le bloc au milieu du fichier ;
2. si on y trouve *Scream* la recherche est terminée ;
3. sinon, soit les films contenus dans le bloc 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 ;
4. 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.

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

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

Mises à jour

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

La première est une liste doublement chaînée des blocs libres (voir figure 9.11). Quand de l'espace se libère dans un bloc plein, on l'insère à la fin de la liste chaînée. Inversement, quand un bloc libre devient plein, on le supprime de la liste. Dans l'exemple de la figure 9.11, 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.

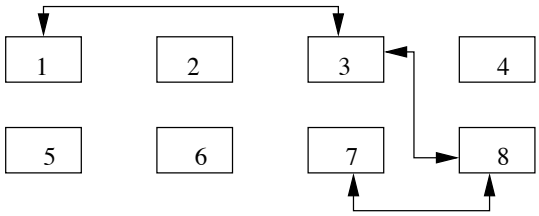


FIG. 9.11 – *Gestion des blocs libres avec liste chaînée*

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

La seconde solution repose sur une structure séparée des blocs du fichier. Il s'agit d'un répertoire qui donne, pour chaque page, un indicateur O/N indiquant s'il reste de l'espace, et un champ donnant le nombre d'octets (figure 9.12). Pour trouver un bloc avec une quantité d'espace libre donnée, il suffit de parcourir ce répertoire.

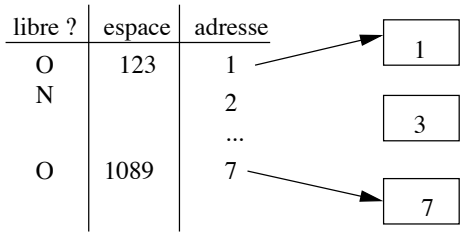


FIG. 9.12 – *Gestion des blocs libres avec répertoire*

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

9.3 Oracle

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

Un système Oracle (une *instance* dans la documentation) stocke les données dans un ou plusieurs *fichiers*. Ces fichiers sont entièrement attribués au SGBD. Ils sont divisés en *blocs* dont la taille – pa-

ramétrable – 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 paramétres, et les fichiers auxquels il s'applique, portent le nom de *tablespace*.

Nous revenons maintenant en détail sur ces concepts.

9.3.1 Fichiers et blocs

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

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

Les blocs ORACLE

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

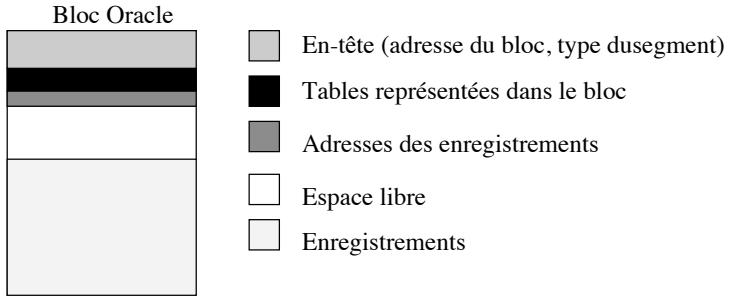


FIG. 9.13 – *Structure d'un bloc Oracle*

La structure d'un bloc est identique quel que soit le type d'information qui y est stocké. Elle est constituée des cinq parties suivantes (voir figure 9.13) :

– *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 de l'application, ou plus précisément des caractéristiques des données stockées dans une table particulière. Une petite valeur pour `PCTFREE` permet aux insertions de remplir plus complètement le bloc, et peut donc mieux exploiter l'espace disque. Ce choix peut être valable pour des données qui sont rarement modifiées. En contrepartie une valeur plus importante de `PCTFREE` va occuper plus de blocs pour les mêmes données, mais offre plus de flexibilité pour des mises à jour fréquentes.

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

Dans ce premier scénario, on accepte d'avoir beaucoup d'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 tableau 9.3, page 118. Par exemple les données de type `CHAR(`n`)` sont stockées dans un tableau d'octets de longueur $n+1$. Le premier octet indique la taille de la chaîne, qui doit donc être comprise entre 1 et 255. Les n octets suivants stockent les caractères de la chaînes, complétés par des blancs si la longueur de cette dernière est inférieure à la taille maximale. Pour les données de type `VARCHAR(`n`)` en revanche, seuls les octets utiles pour la représentation de la chaîne sont stockés. C'est un cas où une mise à jour élargissant la chaîne entraîne une réorganisation du bloc.

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

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

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

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

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

Extensions et segments

Une extension est 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 tailles importante favorisent de bonnes performances, mais il existe des contreparties :

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

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

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

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

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

9.3.2 Les *tablespaces*

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

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

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

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

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

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

Voici un exemple de création de *tablespace*.

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

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

1. la taille de la première extension allouée à une table (ou un index) ;
2. la taille de la prochaine extension, si l'espace alloué à la table doit être agrandi ;

3. le nombre maximal d'extensions, ici 20 ;

4. enfin chaque nouvelle extension est 20% plus grande que la précédente.

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

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

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

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

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

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

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

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

   ```
   ALTER TABLESPACE TB1 OFFLINE;
   ```

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

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

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

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

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

Au moment de la création d'une base, on doit donner la taille et l'emplacement d'un premier fichier qui sera affecté au *tablespace* `SYSTEM`. À chaque création d'un nouveau *tablespace* par la suite, il faudra créer un fichier qui servira d'espace de stockage initial pour les données qui doivent y être stockées. Il faut bien noter qu'un fichier n'appartient qu'à un seul *tablespace*, et que, dès le moment où ce fichier est créé, son contenu est 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 peu 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;

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

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

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

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

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

```
SELECT tablespace_name, file_id,
```

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

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

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

9.3.3 Création des tables

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

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

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

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

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

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

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

Chapitre 10

Indexation

Sommaire

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 répércussion de cette mise à jour sur tous les index de la table. Un choix judicieux des index, ni trop ni trop peu, est donc un des facteurs essentiels de la performance d'un système.

Ce chapitre présente les structures d'index les plus classiques utilisées dans les systèmes relationnels. Après 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

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

TAB. 10.1 – *La table à indexer*

de *balanced* en anglais, et signifie que l'arbre est équilibré : tous les chemins partant de la racine vers une feuille ont la même longueur. L'arbre B est utilisé dans tous les SGBD relationnels.

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

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

Exemple 10.1 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 10.1. Nous ne donnons que les deux attributs `titre` et `année` qui seront utilisés pour l'indexation. □

Exemple 10.2 Le deuxième exemple est destiné à montrer, avec des ordres de grandeur réalistes, l'amélioration obtenue par des structures d'index, et les caractéristiques, en espace et en temps, de ces structures. Nous supposerons que la table contient 1 000 000 films, la taille de chaque enregistrement étant de 120 octets. Pour une taille de bloc de 4 096 octets, on aura donc au mieux 34 enregistrements par bloc. Il faut donc 29 412 blocs ($\lceil 1000000/34 \rceil$) occupant un peu plus de 120 Mo (120 471 552 octets, le surplus étant imputable à l'espace perdu dans chaque bloc). C'est sur ce fichier de 120 octets que nous allons construire nos index. □

10.1 Indexation de fichiers

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

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

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

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

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

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

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

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

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

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

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

10.1.1 Index non-dense

Nous commençons par considérer le cas d'un fichier trié sur la clé primaire (il n'y a donc qu'un seul enregistrement pour une valeur de clé). Dans ce cas restreint il est possible, comme nous l'avons vu dans le chapitre 9, 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 occupant plusieurs dizaines, voire centaines de mégaoctets. La première structure que nous étudions permet d'effectuer des recherches sur un fichier triés, même si ce fichier est fragmenté.

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

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

La figure 10.1 montre un index non-dense sur le fichier des 16 films, la clé étant le titre du film. On suppose que chaque bloc du fichier de données contient 4 enregistrements, ce qui donne un minimum de 4 blocs. Il suffit alors de quatre paires `[titre, Adr]` pour indexer le fichier. Les titres utilisés sont ceux des premiers enregistrements de chaque bloc, soit respectivement *Annie Hall*, *Greystoke*, *Metropolis* et *Smoke*.

Si on désigne par $\{c_1, c_2, \ldots c_n\}$ 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 c_i telle que $c_i \leqslant c < c_{i+1}$. Supposons que l'on recherche le film *Shining*. En consultant l'index on constate que ce titre est compris

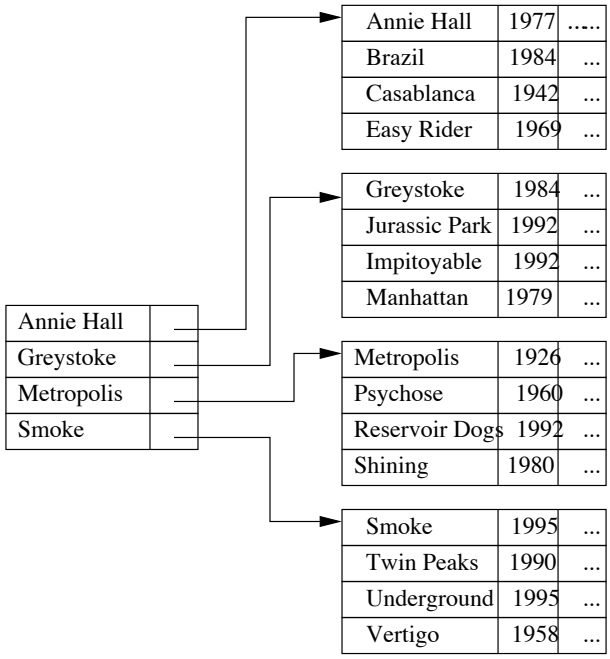


FIG. 10.1 – *Un index non dense*

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

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

Exemple 10.3 Considérons l'exemple 10.2 de notre fichier contenant un million de films (voir page 134). Il est constitué de 29 142 blocs. Supposons qu'un titre de films occupe 20 octets en moyenne, et l'adresse d'un bloc 8 octets. La taille de l'index est donc $29142 * (20 + 8) = 815976$ octets, à comparer aux 120 Mo du fichier de données. □

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

Dans le cas d'une recherche par intervalle, l'agorithme est très semblable : on recherche dans l'index l'adresse de l'enregistrement correspondant à a 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 10.4 Supposons que l'on recherche tous les films dont le titre commence par une lettre entre 'J' et 'P'. On procède comme suit :

1. on recherche dans l'index la plus grande valeur strictement inférieure à 'J' : pour l'index de la figure 10.1 c'est *Greystoke* ;
2. on accède au bloc du fichier de données, et on y trouve le premier enregistrement avec un titre commençant par 'J', soit *Jurassic Park* ;

3. on parcourt la suite du fichier jusqu'à trouver *Reservoir Dogs* qui est au-delà de l'intervalle de recherche, : tous les enregistrements trouvés durant ce parcours constituent le résultat de la requête.

□

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

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

10.1.2 Index dense

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

La figure 10.2 montre le même fichier contenant seize films, trié sur le titre, et indexé maintenant sur l'année de parution des films. On constate d'une part que toutes les années du fichier de données sont reportées dans l'index, ce qui accroît considérablement la taille de ce dernier, et d'autre part que la même année apparaît autant qu'il y a de films parus cette année là.

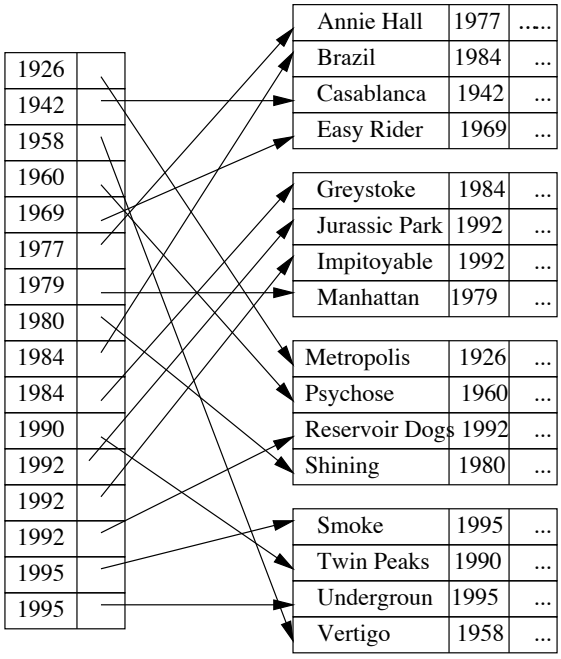


FIG. 10.2 – *Un index dense*

Exemple 10.5 Considérons l'exemple 10.2 de notre fichier contenant un million de films (voir page 134). Il faut créer une entrée d'index pour chaque film. Une année occupe 4 octets, et l'adresse d'un bloc 8 octets.

La taille de l'index est donc $1000000 * (4 + 8) = 12\,000\,000$ octets, soit seulement dix fois moins que les 120 Mo du fichier de données. □

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

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

La recherche par clé ou par préfixe avec un index dense est similaire à celle déjà présentée pour un index non-dense. Si la clé n'est pas unique (cas des années de parution des films), il faut prendre garde à lire dans l'index *toutes* les entrées correspondant au critère de recherche. Par exemple, pour rechercher tous les films parus en 1992 dans l'index de la figure 10.2, on trouve d'abord la première occurence de 1992, pointant sur *Jurasic Park*, puis on lit en séquence les entrées suivantes dans l'index pour accéder successivement à *Impitoyable* puis *Reservoir Dogs*. La recherche s'arrête quand on trouve l'entrée 1995 : l'index étant trié, aucun film paru en 1992 ne peut être trouvé en continuant.

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

Exemple 10.6 Voici l'algorithme qui recherche tous les films parus dans l'intervalle $[1950, 1979]$.

1. on recherche dans l'index la première valeur comprise dans l'intervalle : pour l'index de la figure 10.2 c'est `1958` ;
2. on accède au bloc du fichier de données pour y prendre l'enregistrement *Vertigo* : notez que cet enregistrement est placé dans le dernier bloc du fichier ;
3. on parcourt la suite de l'index, en accédant à chaque fois à l'enregistrement correspondant dans le fichier de données, jusqu'à trouver une année supérieure à 1979 : on trouve successivement *Psychose* (troisème bloc), *Easy Rider*, *Annie Hall* (premier bloc) et *Manhattan* (deuxième bloc).

□

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

10.1.3 Index multi-niveaux

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

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

On peut alors créer un deuxième niveau d'index, comme illustré sur la figure 10.3. On a supposé, pour la clarté de l'illustration, qu'un bloc de l'index de premier niveau ne contient que quatre entrées [`date, Adr`]. Il faut donc quatre blocs (marqués par des traits gras) pour cet index.

L'index de second niveau est construit sur les valeurs de clés des premiers enregistrements des quatre blocs. Il suffit donc d'un bloc pour ce second niveau. S'il y avait deux blocs (par exemple parce que les

blocs ne sont pas complètement pleins) on pourrait envisager de créer un troisième niveau, avec un seul bloc contenant deux entrées pointant vers les deux blocs du second niveau, etc.

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

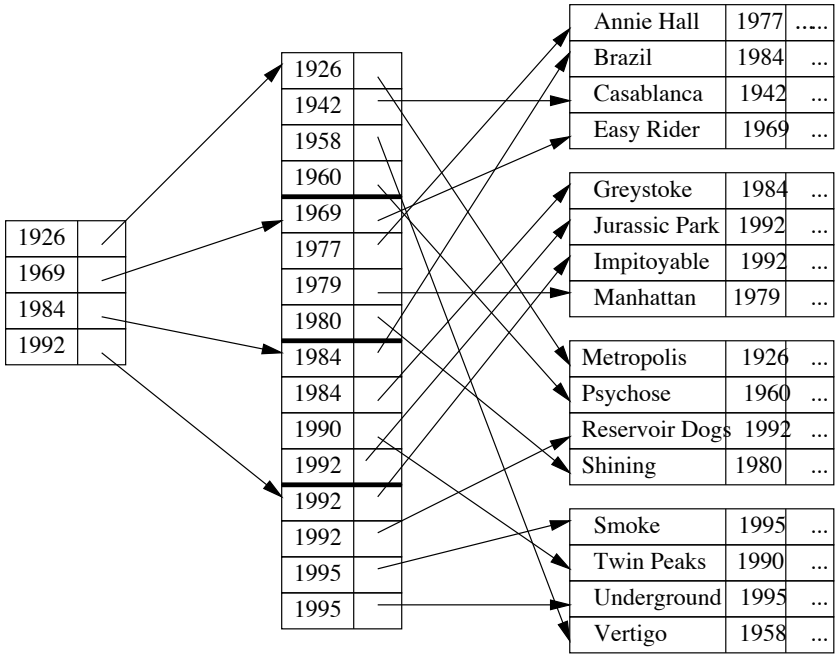


FIG. 10.3 – *Index multi-niveaux*

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

Exemple 10.7 On recherche le ou les films parus en 1990. Partant du second niveau d'index, on sélectionne la troisième entrée correspondant à la clé 1984. L'entrée suivante a en effet pour valeur 1992, et ne pointe donc que sur des films antérieurs à cette date.

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

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

10.2 L'arbre-B

L'arbre-B est une structure d'index qui offre un excellent compromis pour les opérations de recheche par clé et par intervalle, ainsi que pour les mises à jour. Ces qualités expliquent qu'il soit systématiquement utilisé par tous les SGBD, notamment pour indexer la clé primaire des tables relationnelles. Dans ce qui

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

10.2.1 Présentation intuitive

La figure 10.4 montre un arbre-B+ indexant notre collection de 16 films. L'index est organisé en blocs de taille égale, ce qui ajoute une souplesse supplémentaire à l'organisation en niveaux étudiées précédemment. En pratique un bloc peut contenir un assez grand nombre de titres de films, mais pour la clarté de l'illustration nous supposerons que l'on peut stocker au plus 4 titres dans un bloc. Notons qu'au sein d'un même bloc, les titres sont triés par ordre lexicographique.

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

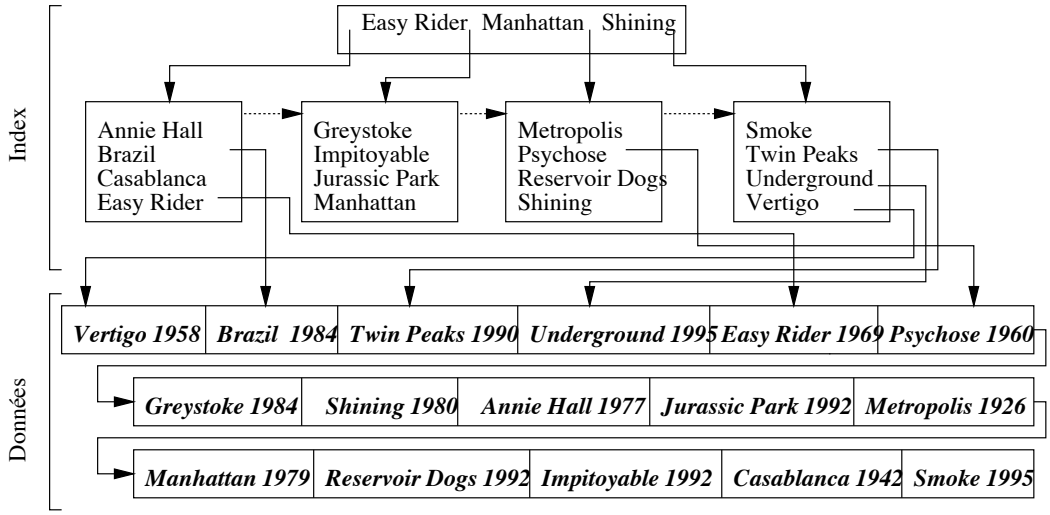


FIG. 10.4 – *Le fichier des films, avec un index unique sur le titre.*

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

2. dans le bloc situé entre *Easy Rider* et *Manhattan*, on ne trouve que des titres strictement supérieurs à *Easy Rider* et inférieurs ou égaux à *Manhattan* ;

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

Le dernier niveau (le second dans notre exemple) est celui des feuilles de l'arbre. Il constitue un index *dense* alors que les niveaux supérieurs sont non-denses. À ce niveau on associe à chaque titre l'adresse du film dans le fichier des données. Étant donné cette adresse, on peut accéder directement au film sans avoir besoin d'effectuer un parcours séquentiel sur le fichier de données. Dans la figure 10.4, nous ne montrons que quelques-uns de ces chaînages *(index, données)*.

Il existe un deuxième chaînage dans un arbre-B+ : les feuilles sont liées entre elles en suivant l'ordre lexicographique des valeurs qu'elles contiennent. Ce second chaînage – représenté en pointillés dans la figure 10.4 – permet de répondre aux recherches par intervalle.

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

La figure 10.5 montre un index construit sur l'année de parution des films. Cet index est naturellement non-unique puisque plusieurs films paraissent la même année. On constate par exemple que la valeur 1995 est associée à deux films, *Smoke* et *Underground*. La valeur 1995 est associé à trois films (non illustré sur la figure), etc. Ce deuxième index permet d'optimiser des requêtes utilisant l'année comme critère de recherche.

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

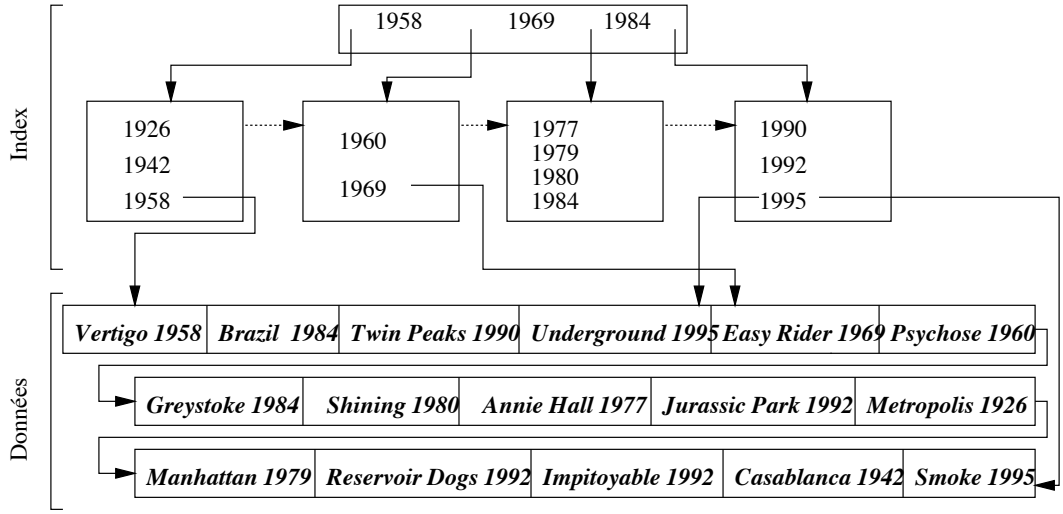


FIG. 10.5 – *Le fichier des films, avec un index unique.*

10.2.2 Recherches avec un arbre-B+

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

Recherche par clé

Prenons l'exemple suivant :

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

En l'absence d'index, la seule solution est de parcourir le fichier. Dans l'exemple de la figure 10.4, cela implique de lire inutilement 13 films avant de trouver *Impitoyable* qui est en quatorzième position. L'index permet de trouver l'enregistrement beaucoup plus rapidement.

– on lit la racine de l'arbre : *Impitoyable* étant situé dans l'ordre lexicographique entre *Easy Rider* et *Manhattan*, on doit suivre le chaînage situé entre ces deux titres ;

– on lit le bloc feuille dans lequel on trouve le titre *Impitoyable* associé à l'adresse de l'enregistrement dans le fichier des données ;

– il reste à lire l'enregistrement.

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

Exemple 10.8 Reprenons l'exemple 10.2 de notre fichier contenant un million de films. Une entrée d'index occupe 28 octets. On place donc $\lfloor \frac{4096}{28} \rfloor = 146$ entrées (au maximum) dans un bloc. Il faut $\lceil \frac{1000000}{146} \rceil = 6\,850$ blocs pour le premier niveau de l'arbre-B+.

Le deuxième niveau comprend 650 entrées, une pour chaque bloc du premier niveau. Il faut donc $\lfloor \frac{650}{146} \rfloor = 5$ blocs. Finalement, un troisième niveau, constitué d'un bloc avec 5 entrées suffit pour compléter l'arbre-B+. □

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

Le gain est d'autant plus spectaculaire que le nombre d'enregistrements est élevé. Voici une petite extrapolation montrant le nombre de films indexés en fonction du nombre de niveaux dans l'arbre [1].

1. avec un niveau d'index (la racine seulement) on peut donc référencer 146 films ;
2. avec deux niveaux on indexe 146 blocs de 146 films chacun, soit $146^2 = 21\,316$ films ;
3. avec trois niveaux on indexe $146^3 = 3\,112\,136$ films ;
4. enfin avec quatre niveaux on index plus de 450 millions de films.

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

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

Recherche par intervalle

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

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

On peut utiliser l'index sur les clés pour répondre à cette requête. Tout d'abord on fait une recherche par clé pour l'année 1960. On accède alors à la seconde feuille (voir figure 10.5) dans laquelle on trouve la valeur 1960 associée à l'adresse du film correspondant (*Psychose*) dans le fichier des données.

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

On parcourt ensuite les feuilles en suivant le chaînage indiqué en pointillés dans la figure 10.5. On accède ainsi successivement aux valeurs 1969, 1977 (dans la troisième feuille) puis 1979. Arrivé à ce point, on sait que toutes les valeurs suivantes seront supérieures à 1979 et qu'il n'existe donc pas de film paru en 1975 dans la base de données. Toutes les adresses des films constituant le résultat de la requête ont été récupérées : il reste à lire les enregistrements dans le fichier des données.

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

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

Recherche avec un préfixe de la clé

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

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

On veut donc tous les films dont le titre commence par 'M'. Cela revient à faire une recherche par intervalle sur toutes les valeurs comprises, selon l'ordre lexicographique, entre le 'M' (compris) et le 'N' (exclus). Avec l'index, l'opération consiste à effectuer une recherche par clé avec la lettre 'M', qui mène à la seconde feuille (figure 10.4) dans laquelle on trouve le film *Manhattan*. En suivant le chaînage des feuilles on trouve le film *Metropolis*, puis *Psychose* qui indique que la recherche est terminée.

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

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

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

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

10.3 Hachage

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

10.3.1 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'élement doit être placé. En mémoire principale cet espace de stockage est en général une liste chaînée, et en mémoire secondaire un ou plusieurs blocs sur le disque.

Exemple d'une table de hachage

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

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

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

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

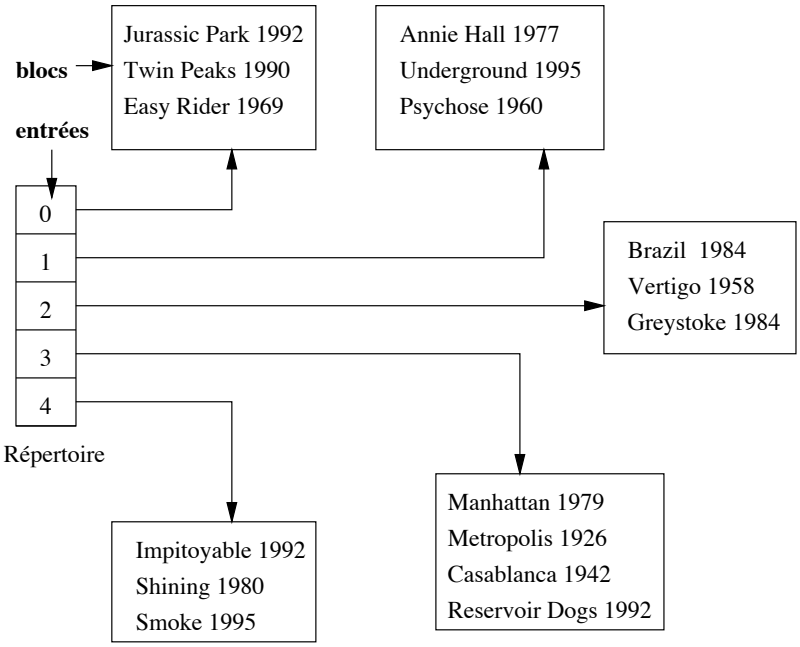


FIG. 10.6 – *Exemple d'une table de hachage*

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

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 figure 10.6 montre la table de hachage obtenue avec cette fonction. Tous les films commençant par a, f, k, p, u et z sont affectés au bloc 1 ce qui donne, pour notre ensemble de films, *Annie Hall*, *Psychose* et *Underground*. les lettres b, g, l, q et v sont affectées au bloc 2 et ainsi de suite. Notez que la lettre e a pour rang 5 et se trouve donc affectée au bloc 0.

La figure 10.6 présente, outre les cinq blocs stockant des films, un répertoire à cinq *entrées* permettant d'associer une valeur entre 0 et 4 à l'adresse d'un bloc sur le disque. Ce répertoire fournit une indirection entre l'identification « logique » du bloc et son emplacement physique, selon un mécanisme déjà rencontré dans la partie du chapitre 9 consacrée aux techniques d'adressage de blocs (voir section 9.2.2, page 119). On peut raisonnablement supposer que sa taille est faible et qu'il peut donc résider en mémoire principale.

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

Recherche dans une table de hachage

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

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

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

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

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

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

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

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

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

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

Mises à jour

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

Cette situation est illustrée dans la figure 10.7. On a inséré un nouveau film, *Citizen Kane*. La valeur donnée par la fonction de hachage est 3, rang de la lettre c dans l'aplhabet, mais le bloc 3 est déjà plein.

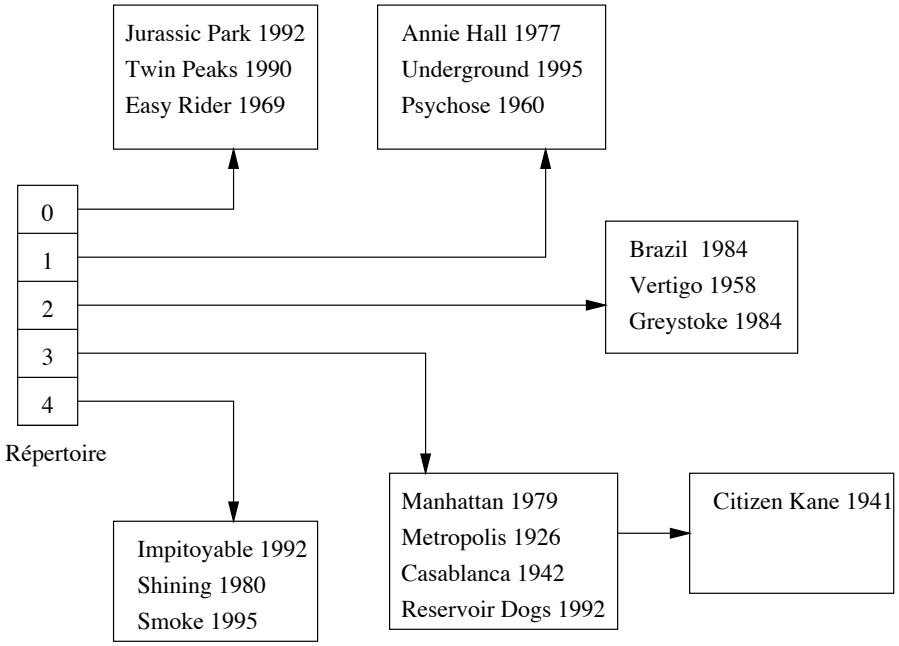


FIG. 10.7 – *Table de hachage avec page de débordement*

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 bloc au bloc 3 et y stocker le nouveau film. À une entrée dans la table, correspondant à l'adresse logique 3, sont associés maintenant deux blocs physiques, avec une dégradation potentielle des performances puisqu'il faudra, lors d'une recherche, suivre le chaînage et inspecter tous les enregistrements pour lesquels la fonction de hachage donne la valeur 3.

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

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

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

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

TAB. 10.2 – *Valeurs du hachage extensible pour les titres*

– **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. 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+.

Nous présentons dans ce qui suit 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.

10.3.2 Hachage extensible

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

Maintenant nous supposons donnée une fonction de hachage h dont le résultat est toujours un entier sur 4 octets, soit 32 bits. La table 10.2 donne les 8 premiers bits des valeurs obtenues par application de cette fonction aux titres de nos films. Comme il n'y a que deux entrées, nous nous intéresons seulement au premier de ces 32 bits, qui peut valoir 0 ou 1. La figure 10.8 montre l'insertion des cinq premiers films de notre liste, et leur affectation à l'un des deux blocs. Le film *Vertigo* par exemple a pour valeur de hachage 01110010 qui commence par 0, et se trouve donc affecté à la première entrée.

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

On va alors doubler la taille du répertoire pour la faire passer à quatre entrées, avec pour valeurs respectives 00, 01, 10, 11, soit les 2^2 combinaisons possibles de 0 et de 1 sur deux bits. Ce doublement de taille du répertoire entraine la réorganisation suivante (figure 10.9) :

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

2. les films de l'ancienne entrée 1 n'ont pas de raison d'être répartis dans deux blocs puisqu'il n'y a pas eu de débordement pour cette valeur : *on va donc associer le même bloc aux deux entrées 10 et 11*.

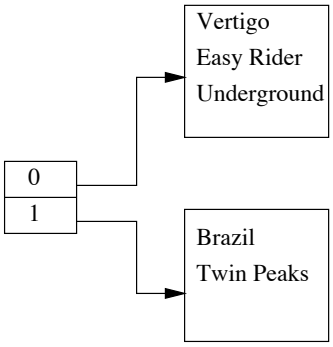


FIG. 10.8 – *Hachage extensible avec 2 entrées*

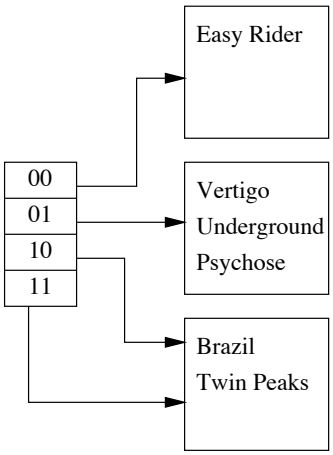


FIG. 10.9 – *Doublement du répertoire dans le hachage extensible*

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

On va donc allouer un nouveau bloc à la structure, et l'associer à l'entrée 11, l'ancien bloc restant associé à la seule entrée 10. Les films sont répartis dans les deux blocs, *Brazil* et *Greystoke* avec l'entrée 10, *Twin Peaks* et *Shining* avec l'entrée 11 (figure 10.10.

10.4 Les index *bitmap*

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

– si le bit est à 1, l'attribut A a pour valeur v dans la ligne l ;

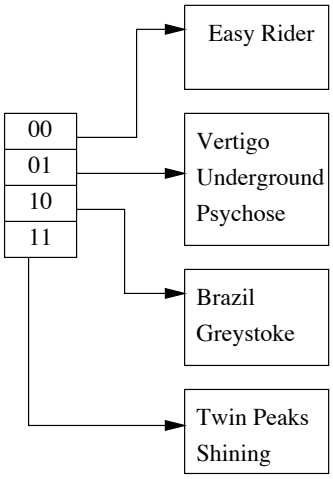


FIG. 10.10 – *Jeu de pointeurs pour éviter de doubler le répertoire*

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

TAB. 10.3 – *Les films et leur genre*

– sinon le bit est à 0.

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

Exemple 10.9 Reprenons l'exemple de nos 16 films, et créons un index sur le genre (voir le table 10.3). L'utilisation d'un arbre-B ne donnera pas de bons résultats car l'attribut est trop peu sélectif (autrement dit une partie importante de la table peut être sélectionnée quand on effectue une recherche par valeur).

En revanche un index *bitmap*est tout à fait approprié puisque le genre fait partie d'un ensemble énuméré avec relativement peu de valeurs. Voici par exemple le *bitmap* pour les valeurs « Drame », « Science-

Fiction » et « Comédie ». Chaque colonne correspond à l'un des films.

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

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

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

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

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

10.5 Indexation dans Oracle

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

10.5.1 Arbres B+

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

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

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

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

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

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

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

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

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

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

10.5.2 Arbres B

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

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

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

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

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

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

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

10.5.3 Indexation de documents

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

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

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

10.5.4 Tables de hachage

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

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

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

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

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

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

1. *la clé de hachage* est spécifiée dans l'exemple comme étant un `id` de type `INTEGER` :
2. *le nombre d'entrées* dans la table est spécifié par `HASHKEYS` ;
3. *la taille estimée pour chaque entrée* est donnée par `SIZE`.

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

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

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

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

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

10.5.5 Index bitmap

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

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

Chapitre 11

Introduction à la concurrence d'accès

Sommaire

Les bases de données sont le plus souvent accessibles à plusieurs utilisateurs qui peuvent rechercher, créer, modifier ou détruire les informations contenues dans la base. Ces accès simultanés à des informations partagées soulèvent de nombreux problèmes de cohérence : le système doit pouvoir gérer le cas de deux utilisateurs accédant simulatément à une même information en vue d'effectuer des mises-à-jour. Plus généralement, on doit savoir contrôler les accès concurrents de plusieurs programmes complexes effectuant de nombreuses lectures/écritures.

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

11.1 Préliminaires

Commençons dès maintenant par un exemple illustrant le problème. Supposons que l'application *Officiel des spectacles* propose une réservation des places pour une séance. Voici le programme de réservation (simplifié) :

Programme RESERVATION
Entrée : Une séance s
Le nombre de places souhaité $NbPlaces$
Le client c
debut
Lire la séance s
si (nombre de places libres $>$ $NbPlaces$)
Lire le compte du spectateur c

Débiter le compte du client
Soustraire $NbPlaces$ au nombre de places vides
Ecrire la séance s
Ecrire le compte du client c
finsi
fin

Il est important de noter dès maintenant que, du point de vue du contrôle de concurrence, des instructions comme les tests, les boucles ou les calculs n'ont pas vraiment d'importance. Ce qui compte, ce sont les accès aux données. Ces accès sont de deux types

1. Les **lectures**, que l'on notera à partir de maintenant par *r*.
2. Les **écritures** que l'on notera *w*.

De plus, la nature des informations manipulées est indifférente : les règles pour le contrôle de la concurrence sont les mêmes pour des films, des comptes en banques, des stocks industriels, etc. Tout ceci mène à représenter un programme de manière simplifiée comme une suite de lectures et d'écritures opérant sur des données désignées abstraitement par des variables (généralement *x, y, z, ...*).

Le programme $RESERVATION$ se représente donc simplement par la séquence

r[s] r[c] w[c] w[s]

11.1.1 Exécutions concurrentes : sérialisabilité

On va maintenant s'intéresser aux **exécutions** d'un programme dans un contexte multi-utilisateur. Il pourra donc y avoir plusieurs exécutions simultanées du même programme : pour les distinguer, on emploie simplement un indice : on parlera du programme P_1, du programme P_2.

Exemple 11.1 *Voici un exemple de deux exécutions concurrentes du programme* $RESERVATION$ P_1 *et* P_2. *Chaque programme veut réserver des places dans la même séance, pour deux clients distincts* c_1 *et* c_2.

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

Donc on effectue d'abord les lectures pour P_1, puis les lectures pour P_2 enfin les écritures pour P_2 et P_1 dans cet ordre. Imaginons maintenant que l'on se trouve dans la situation suivante :

1. Il reste 50 places libres pour la séance s.
2. P_1 veut réserver 5 places pour la séance s.
3. P_2 veut réserver 2 places pour la séance s.

Voici le déroulement imbriqué des deux exécutions $P_1(s, 5, c_1)$ et $P_2(s, 10, c_2)$, 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.

1. P_1 lit s et c_1. Nb places vides : 50.
2. P_2 lit s et c_2. Nb places vides : 50.
3. P_2 écrit s avec nb places = $50 - 2 = 48$.
4. P_2 écrit le nouveau compte de c_2.
5. P_1 écrit s avec nb places = $50 - 5 = 45$.
6. P_1 écrit le nouveau compte de c_1.

A la fin de l'exécution, il y a 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 P_1 et P_2 : P_2 lit et modifie une information que P_1 a déjà lue en vue de la modifier.

Ce genre d'anomalie est évidemment fortement indésirable. Une solution brutale est d'exécuter **en série** les programmes : on bloque un programme tant que le précédent n'a pas fini de s'exécuter.

Exemple 11.2 *Exécution en série de P_1 et P_2 :*

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

On est assuré dans ce cas qu'il n'y a pas de problème : P_2 lit une donnée écrite par P_1 qui a fini de s'exécuter et ne créera donc plus d'interférence.

Cela étant cette solution de "concurrence zéro" n'est pas viable : on ne peut se permettre de bloquer tous les utilisateurs sauf un, en attente d'un programme qui peut durer extrèmement longtemps. Heureusement l'exécution en série est une contrainte trop forte, comme le montre l'exemple suivant.

Exemple 11.3 *Exécution imbriquée de P_1 et P_2.*

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

Suivons pas à pas l'exécution :

1. P_1 lit s et cs_1. Nb places vides : 50.
2. P_1 écrit s avec nb places = $50 - 5 = 45$.
3. P_2 lit s. Nb places vides : 45.
4. P_2 lit cs_2.
5. P_2 écrit s avec nb places = $45 - 2 = 43$.
6. P_1 écrit le nouveau compte du client c_1.
7. P_2 écrit le nouveaux 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. On parle d'exécutions **sérialisables** pour indiquer qu'elles sont équivalentes à des exécutions en série. Les techniques qui permettent d'obtenir de telles exécutions relèvent de la **sérialisabilité**.

11.1.2 Transaction

Le fait de garantir une imbrication correcte des exécutions de programmes concurrents 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 : il arrive que l'on doive annuler les opérations d'entrées sorties effectuées par un programme. On peut envisager deux cas :

1. Un problème matériel ou logiciel entaîne l'interruption de l'exécution.
2. Le programme choisit lui-même d'annuler ce qu'il a fait.

Imaginons que le programme de réservation soit interrompu après avoir exécuté les E/S suivantes :

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

La situation obtenue n'est pas satisfaisante : on a diminué le nombre de places libres, sans débiter le compte du client. Il y a là une incohérence regrettable que l'on ne peut corriger que d'une seule manière : en annulant les opérations effectuées.

Dans le cas ci-dessus, c'est simple : on annule $w_1(s)$. Mais la question plus générale, c'est : jusqu'où doit-on annuler quand un programme a déjà effectué des centaines, voire des milliers d'opérations ? Rappelons que l'objectif, c'est de ramener la base dans un état **cohérent**. Or le système lui-même ne peut pas déterminer cette cohérence : tout SGBD digne de ce nom doit donc offrir à un utilisateur ou à un programme d'application la possibilité de spécifier les suites d'instructions qui forment un tout, que l'on doit valider ensemble ou annuler ensemble (on parle **d'atomicité**). En pratique, on dispose des deux instructions suivantes :

1. **La validation** (*commit* en anglais). Elle consiste à rendre les mises-à-jour permanentes.

2. **L'annulation** (*rollback* en anglais). Elle annule les mises-à-jour effectuées.

Ces instructions permettent de définir la notion de **transaction** : une transaction est l'ensemble des instructions séparant un *commit* ou un *rollback* du *commit* ou du *rollback* suivant. On adopte alors les règles suivantes :

1. Quand une transaction est validée (par *commit*), toutes les opérations sont validées ensemble, **et on ne peut plus en annuler aucune**. En d'autres termes les mises-à-jour deviennent définitives.

2. Quand une transaction est annulée par *rollback* ou par une panne, on annule toutes les opérations depuis le dernier *commit* ou *rollback*, ou depuis le premier ordre SQL s'il n'y a ni *commit* ni *rollback*.

Il est de la responsabilité du programmeur de définir ses transactions de manière à garantir que la base est dans un état cohérent au début et à la fin de la transaction, même si on passe inévitablement par des états incohérents dans le courant de la transaction. Ces états incohérents transitoires seront annulés par le système en cas de panne.

Exemple 11.4 *Les deux premières transactions ne sont pas correctes, la troisième l'est (C signifie $Commit$, R $Rollback$).*

1. $r(s)r(c)w(s)Cw(c)$

2. $r(s)r(c)w(s)Rw(c)C$

3. $r(s)r(c)w(s)w(c)C$

Du point de vue de l'exécution concurrente, cela soulève de nouveaux problèmes qui sont illustrés ci-dessous.

11.1.3 Exécutions concurrentes : recouvrabilité

Revenons maintenant à la situation où plusieurs utilisateurs accèdent simultanément aux mêmes données et considérons l'impact de la possibilité qu'a chaque utilisateur de valider ou d'annuler ses transactions. A nouveau plusieurs problèmes peuvent survenir. Le premier est lié à la **recouvrabilité** des transactions et illustré par l'exemple suivant :

Exemple 11.5 (Exécutions recouvrables).

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

Conséquence sur l'exemple : le nombre de places disponibles a été diminué par T_1 et repris par T_2, avant que T_1 n'annule ses réservations. Le nombre de sièges réservé sera plus grand que le nombre effectif de clients ayant validé leur réservation.

Le problème ici vient du fait que la transaction T_1 est annulée **après** que la transaction T_2 ait lu une information mise-à-jour par T_1, manipulé cette information et effectué des MAJ pour son propre compte, puis validé. On parle de "lectures sales" (*dirty read* en anglais) pour désigner l'accès par une transaction à des MAJ non encore validées d'une autre transaction. Ici le problème est de plus agravé par le fait que T_1 annule la MAJ qui a fait l'objet d'une "lecture sale".

Pour annuler proprement T_1, il faudrait annuler en plus les MAJ de T_2, ce qui n'est pas possible puisque un *commit* a été fait par cette dernière : on parle alors d'exécution **non recouvrable**.

Une exécution non-recouvrable est à éviter absolument puisqu'elle introduit un conflit insoluble entre les *rollback* effectués par une transaction et les *commit* d'une autre. On pourrait penser à interdire à une transaction T_2 ayant effectué des *dirty read* d'une transaction T_1 de valider avant T_1. On accepterait alors la situation suivante :

Exemple 11.6 (Annulations en cascade).

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

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. Donc il faut tout simplement interdire les *dirty read*.

Celà laisse encore une dernière possibilité d'anomalie qui fait intervenir la nécessité d'annuler les écritures effectuées par une transaction. Imaginons qu'une transaction T ait modifé la valeur d'une donnée d, puis qu'un *rollback* intervienne. Dans ce cas il est nécessaire de restaurer la valeur qu'avait d **avant** le début de la transaction : on parle **d'image avant** pour cette valeur. Outre le problème de connaître cette image avant, cela soulève des problèmes de concurrence illustré ci-dessous.

Exemple 11.7 (Exécution non stricte)

$$r_1(s)r_1(c_1)r_2(s)w_1(s)w_1(c_1)r_2(c_2)w_2(s)C_1w_2(c_2)R_2$$

Ici il n'y a pas de *dirty read*, mais une "écriture sale" (*dirty write*). En effet, T_1 a validé après que T_2 ait écrit dans s. Donc la validation de T_1 enregistre la mise-à-jour de T_2 alors que celle-ci s'apprête à annuler ses mises-à-jour par la suite.

Au moment où T_2 va annuler, le gestionnaire de transaction doit remettre la valeur du tuple s connue au début de la transaction : ce qui revient à annuler la mise-à-jour de T_1. Autre exemple.

Exemple 11.8 *Cette fois c'est T_1 qui annule et T_2 qui valide :*

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

Que se passe-t-il au moment de l'annulation de T_1 ? On devrait restaurer l'image avant connue de T_1, mais cela revient à annuler la mise-à-jour de T_2.

En résumé, on distingue trois niveaux de recouvrabilité selon le degré de contrainte imposé au système :

1. **Recouvrabilité** : on ne doit pas avoir à annuler une transaction déjà validée.
2. **Annulation en cascade** : un *rollback* sur une transaction ne doit pas entraîner l'annulation d'autres transactions.
3. **Recouvrabilité stricte** : deux transactions ne peuvent pas accéder simultanément en écriture à la même donnée, sous peine de ne pouvoir utiliser la technique de récupération de l'image avant.

On peut avoir des transactions sérialisables et non recouvrables et réciproquement. Le niveau maximal de cohérence offert par un SGBD assure la sérialisabilité des transactions et la recouvrabilité stricte. Cela définit un ensemble de "bonnes" propriétés pour une transaction qui est souvent désigné par l'acronyme ACID pour :

– **A**tomicité. Une transaction est l'unité de validation.

– **C**ohérence. Une transaction est un ensemble de mises-à-jour entre deux états cohérents de la base.

– **I**solation. Les lectures ou écritures effectuées par une transaction doivent être invisibles des autres transactions.

– **D**urabilité : une transaction validée ne peut plus être annulée.

Il reste maintenant à étudier les techniques mises en oeuvre pour assurer ces bonnes propriétés.

11.2 Contrôle de concurrence

Le **contrôle de concurrence** est la partie de l'activité d'un SGBD qui consiste à ordonner l'exécution des transactions de manière à éviter les anomalies présentées précédemment.

Il existe deux types de méthodes pour contrôler la concurrence :

1. **Contrôle continu** : au vérifie au fur et à mesure de l'exécution des opérations que le critère de sérialisabilité est bien respecté. Ces méthodes sont dites **pessimistes** : elles reposent sur l'idée que les conflits sont fréquents et qu'il faut les traiter le plus tôt possible.

2. **Contrôle par certification** : cette fois on se contente de vérifier la sérialisabilité quand la transaction s'achève. Il s'agit d'une approche dite **optimiste** : on considère que les conflits sont rares et que l'on peut accepter de réexécuter les quelques transactions qui posent problème.

La première approche est la plus fréquente. Le mécanisme adopté est celui du **verrouillage**. L'idée est simple : on bloque l'accès à une donnée dès qu'elle est lue ou écrite par une transaction ("pose de verrou") et on libère cet accès quand la transaction termine par *commit* ou *rollback* ("libération du verrou"). Reprenons l'exemple 11.1, et supposons que tout accès en lecture ou en écriture pose un verrou bloquant les autres transactions. Clairement les transactions T_1 et T_2 s'exécuteront en série et les anomalies disparaîtront.

Le bloquage systématique des transactions est cependant une contrainte trop forte, comme le montre l'exemple 11.3. L'exécution est correcte, mais le verrouillage total bloquerait pourtant sans nécessité la transaction T_2.

On doit donc trouver une solution plus souple. On peut en fait considérer deux critères : le degré de restriction, et la granularité du verrouillage (i.e. le niveau de la **ressource** à laquelle il s'applique : tuple, page, table, etc). Il existe essentiellement deux degrés de restriction :

1. Le verrou **partagé** est typiquement utilisé pour permettre à plusieurs transactions concurrentes de **lire** la même ressource.

2. Le verrou **exclusif** réserve la ressource en écriture à la transaction qui a posé le verrou.

Ces verrous sont posés de manière automatique par le SGBD en fonction des opérations effectuées par les transactions/utilisateurs. Mais il est également possible de demander explicitement le verrouillage de certaines ressources.

Il est important d'être conscient d'une part de la politique de verrouillage pratiquée par un SGBD, d'autre part de l'impact du verrouillage explicite d'une ressource. Le verrouillage influence considérablement les performances d'une BD soumises à un haut régime transactionnel.

11.2.1 Verrouillage à deux phases

Le verrouillage à deux phases est le protocole le plus répandu pour assurer des exécutions concurrentes correctes. On utilise des verrous en lecture qui seront notés *rl* (comme *read lock*) dans ce qui suit, et des verrous en écritures 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*).

Le principe de base est de surveiller les **conflits** entre deux transactions sur la même ressource.

Definition 11.1 *Deux verrous $pl_i[x]$ et $ql_j[y]$ sont* **en conflit** *si $x = y$, $i \neq j$ et pl ou ql est un verrou en écriture.* □

Le respect du protocole est assuré par un module dit *scheduler* qui reçoit les opérations émises par les transactions et les traite selon l'algorithme suivant :

1. Le *scheduler* reçoit $p_i[x]$ et consulte le verrou déjà posé sur x, $ql_j[x]$, s'il existe.
   - si $pl_i[x]$ est en conflit avec $ql_j[x]$, $p_i[x]$ est retardée et la transaction T_i est mise en attente.
   - sinon, T_i obtient le verrou $pl_i[x]$ et l'opération $p_i[x]$ est exécutée.
2. Un verrou pour $p_i[x]$ n'est **jamais** relâché avant la confirmation de l'exécution par un autre module, le gestionnaire de données GD.
3. Dès que T_i relâche un verrou, elle ne peut plus en obtenir d'autre.

Le terme "verrouillage à deux phases" s'explique par le processus détaillé ci-dessus : il y a d'abord **accumulation** de verrou pour une transaction T, puis **libération** des verrous.

Theorem 1 *Toute exécution obtenue par un verrouillage à deux phases est sérialisable.*

De plus on obtient une exécution stricte en ne relachant les verrous qu'au moment du *commit* ou du *rollback*. Les transactions obtenues satisfont les propriétés ACID.

Il est assez facile de voir que ce 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). A ce moment là il n'y a plus de conflit possible puisque T_1 ne demandera plus de verrou (règle 3).

La règle 2 a principalement pour but de s'assurer de la synchronisation entre la **demande d'exécution** d'une opération, et **l'exécution effective** de cette opération. Rien ne garantit en effet que les exécutions vont se faire dans l'ordre dans lequel elles ont été demandées.

Pour illustrer l'intérêt de ces règles, on peut prendre l'exemple suivant :

Exemple 11.9 *(Non-respect de la règle de relâchement des verrous). Soit les deux transactions suivantes :*

1. T_1: $r_1[x] \rightarrow w_1[y] \rightarrow c_1$
2. T_2: $w_2[x] \rightarrow w_2[y] \rightarrow c_2$

et l'ordre d'exécution suivant : $r_1[x]\ w_2[x]\ w_2[y]\ c_2\ w_1[y]\ c_1$. *Supposons que l'exécution avec pose et relâchement de verrous soit la suivante :*

$$rl_1[x]\ \underline{r_1[x]}\ ru_1[x]\ wl_2[x]\ \underline{w_2[x]}\ wl_2[y]\ \underline{w_2[y]}\ wu_2[x]\ wu_2[y]\ \underline{c_2}\ wl_1[y]\ \underline{w_1[y]}\ wu_1[y]\ \underline{c_1}$$

On a violé la règle 3 : T_1 a relaché le verrou sur x puis en a repris un sur y. Un "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 ($r_1[x] < w_2[x]$ et $w_2[y] < w_1[y]$).

Reprenons le même exemple, avec un verrouillage à deux phases :

Exemple 11.10 *(exécution correcte)*

$$rl_1[x]\underline{r_1[x]}wl_2[x]\underline{T_2\ en\ attente}wl_1[y]\underline{w_1[y]}c_1ru_1[x]wu_1[y]wl_2[x]\underline{w_2[x]}wl_2[y]\underline{w_2[y]}c_2wu_2[x]wu_2[y]$$

Le verrouillage à deux phases (avec des variantes) est utilisé dans tous les SGBD. Il permet en effet une certaine imbrication des opérations. Notons cependant qu'il est un peu trop strict dans certains cas : voici l'exemple d'une exécution sérialisable

Exemple 11.11 *Une exécution sérialisable impossible par verrouillage*

$$r_1[x]\ w_2[x]\ c_2\ w_3[y]\ c_3\ r_1[y]\ w_1[z]\ c_1$$

T_j relâche $rl[x]$ pour $w_k[x]$, mais a besoin ensuite de $rl[y]$ pour $r_j[y]$.

Le principal défaut du verrouillage à deux phases est d'autoriser des **interblocages** : deux transactions concurrentes demandent chacune un verrou sur une ressource détenue par l'autre. Voici l'exemple type :

Exemple 11.12 *Les deux transactions sont les suivantes :*

- $T_1 : r_1[x] \rightarrow w_1[y] \rightarrow c_1$
- $T_2 : r_2[y] \rightarrow w_2[x] \rightarrow c_2$

Considérons maintenant que T_1 et T_2 s'exécutent en concurrence dans le cadre d'un verrouillage à deux phases :

$$rl_1[x]r_1[x]rl_2[y]r_2[y]wl_1[y]\underline{T_1 \text{ en attente}}wl_2[x]\underline{T_2 \text{ en attente}}$$

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 et ré-éxécutée. Autre solution : tester le temps d'attente et annuler les transactions qui dépassent le temps limite.

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.

A titre d'exercice, on peut reprendre le programme de réservation donné initialement, mais dans une version légèrement différente :

Programme RESERVATION2
Entrée : Une séance s
Le nombre de places souhaité $NbPlaces$
Le client c
debut
Lire la séance s
si (nombre de places libres $>$ $NbPlaces$)
Lire le compte du spectateur c
Débiter le compte du client
Soustraire $NbPlaces$ au nombre de places vides
Ecrire le compte du client c
Ecrire la séance s
finsi
fin

Exercice : donner une exécution concurrente de RESERVATION et de RESERVATION2 qui aboutisse à un interblocage.

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.

Autre solution pour fluidifier les transactions : ne pas poser de verrou en lecture (mode `READ COMMITTED` en SQL2). Cette solution affecte l'isolation entre deux transactions.

Exemple 11.13 *Toujours avec le programme $RESERVATION$. Chaque programme veut réserver des places dans la même séance, pour deux clients distincts c_1 et c_2.*

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_1(s)w_2(s)w_2(c_2)w_1(c_1)C_1C_2$$

Sans verrou en lecture : on bloque mois T_2, mais la transaction n'est plus sérialisable.

11.2.2 Contrôle par estampillage

C'est une méthode beaucoup plus simple qui consiste à fixer, *a priori*, l'ordre de sérialisabilité des transactions soumises au *scheduler*. Pour cela, on affecte à chaque transaction une *estampille*. Chaque valeur d'estampille est unique et les valeurs sont croissantes : on garantit ainsi un ordre total entre les transactions.

Quand un conflit survient entre op_i et op_j, on vérifie simplement que l'ordre du conflit est compatible avec l'ordre des transactions (i.e. $i < j \iff op_i \rightarrow op_j$). Sinon on rejette la transaction qui veut effectuer l'opération conflictuelle.

Sur l'exemple suivant :

$$r_1(s)w_2[x]r_3[x]r_2[x]w_1[x]$$

en admettant que l'estampille est le numéro de la transaction, on rejette la transation T_1 au moment de $w_1[x]$.

Méthode peu utilisée car elle entraîne des abandons inutiles de transactions, et le risque de *privation* : une transaction est toujours rejettée.

11.3 Gestion des transactions en SQL

Il est possible en SQL de choisir explicitement le niveau de protection que l'on souhaite obtenir contre les incohérences résultant de la concurrence d'accès. Le comportement par défaut est d'assurer sérialisabilité et recouvrabilité stricte, mais ce mode a l'inconvénient de ralentir le débit transactionnel pour des applications qui n'ont peut-être pas besoin de contrôles aussi stricts.

La première option disponible 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 SQL est :

```
SET TRANSACTION READ ONLY;
```

Il devient alors interdit d'effectuer des ordres UPDATE, INSERT ou DELETE jusqu'au prochain *commit* ou *rollback* : le système rejette ces instructions. Le double avantage de cette option est (i) d'être sûr de ne jamais être bloqué, et (ii) d'augmenter le débit transactionnel global.

Une conséquence insidieuse est que deux lectures successives avec le même ordre SELECT peuvent donner des résultats différents : entre temps une autre transaction a pu mettre à jour les données.

L'option par défaut est qu'une transaction peut lire et écrire. On peut spécifier ce mode explicitement par :

```
SET TRANSACTION READ WRITE;
```

Qu'en est-il maintenant des "bonnes" propriétés des exécutions concurrentes ? La norme SQL2 spécifie que ces exécutions doivent être sérialisables : il s'agit là du mode par défaut. Un verrouilage strict doit alors être assuré par le SGBD. Il peut arriver que certaines applications ne demandent pas une sécurité aussi stricte et soient pénalisées par le surcoût en temps induit par la gestion du verrouillage. SQL2 propose des options moins fortes, explicitées par la commande

```
SET TRANSACTION ISOLATION LEVEL option
```

Voici la liste des options, sachant que tous les sytèmes ne les proposent pas intégralement.

1. `READ UNCOMMITTED`. C'est le mode qui offre le moins d'isolation : on autorise les lectures "sales", i.e. les lectures de tuples écrits par d'autres transactions mais non encore validées.

2. `READ COMMITED`. On ne peut lire que les tuples validés, mais il peut arriver que deux lectures successives donnent des résultats différents.

   En d'autres termes, un lecteur ne pose pas de verrou sur la donnée lue, ce qui évite de bloquer les écrivains. C'est le mode par défaut dans ORACLE par exemple.

3. `REPEATABLE READ`. Le nom semble indiquer que l'on corrige le défaut de l'exemple précédent. En fait ce mode garantit qu'un tuple lu au début de la transaction sera toujours visible ensuite, mais des tuples peuvent apparaître s'ils ont étés insérés par d'autres transactions (on parle de "tuples fantômes").

4. `SERIALIZABLE`. Le défaut. Ce mode garantit les bonnes propriétés (sérialisabilité et recouvrabilité) des transactions telles que présentées précédemment, mais de plus on garantit que plusieurs lectures avec le même ordre SQL donneront le même résultat, même si des insertions ou mises-à-jour validées ont eu lieu entretemps.

   Tout se passe alors comme si on travaillait sur une "image" de la base de données prise au début de la transaction.

Signalons enfin que certains systèmes permettent de poser explicitement des verrous. C'est le cas de ORACLE qui propose par exemple des commandes telles que :

```
LOCK TABLE ... IN EXCLUSIVE MODE
```

11.4 Exercices

Exercice 11.1 *On considère maintenant le problème (délicat) de la réservation des places pour un match. Pour cela on ajoute les tables* **Match** (Match, NomStade, PlacesPrises) *et* **Client** (NoClient, Solde). *Les opérateurs disposent du petit programme suivant pour réserver des places*[1] *:*

```
Places (Client C, Match M, Nb-Places N)
begin
  Lire le match M                       // Donne le nbre de places prises
  Lire le stade S                       // Donne le nbre total de places

  if ( (S.places - M.PlacesPrises) > N) // Il reste assez de places
  begin
    Lire le client C
    M.PlacesPrises += N; C.solde   -= N * 300;
    Ecrire le match M
    Ecrire le client C
  end
  commit
end
```

Les organisateurs s'aperçoivent rapidement qu'il n'y a pas assez de place dans les stades en entreprennent des travaux d'agrandissement. On a le deuxième programme :

```
Augmenter (Stade S, Places P)
begin
  Lire S
  S.Places += P
  Ecrire S
  commit
end
```

1. *On lance simultanément deux exécutions du programme* Augmenter (SF, 1000) *pour augmenter la capacité du Stade de France.*

   (a) *Donnez un exemple d'une 'histoire' (imbrication des lectures/écritures) non sérialisable.*

   (b) *Donnez un exemple d'une histoire non recouvrable (en plaçant des ordres* `commit`*).*

1. On suppose que chaque place vaut 300F.

2. *On a maintenant une exécution concurrente de* Places (C, M, 1500) *et de* Augmenter (SF, 1000), M *étant un match qui se déroule au Stade de France. Voici le début de l'exécution*[2] *:*

$$\mathbf{H} = r_P[M]r_P[SF]r_A[SF]w_A[SF]c_A \ldots$$

   (a) *Donner l'histoires complète en supposant qu'il y a 2000 places libres au début de l'exécution. Donnez également le nombre de places libres dans le stade à la fin de l'exécution.*

   (b) *L'histoire obtenue est-elle sérialisable ?*

   (c) *Appliquer un verrouillage à deux phases (les verrous sont relâchés au moment du* `commit`.

   (d) *Conclusion ? Y-avait-il un risque d'anomalie ? Que dire du comportememt du verrouillage à deux phases ?*

3. *Soit l'histoire suivante, représentant deux exécutions concurrentes du programme* Places *:* T_1 = Places (C, M, 100) *et* T_2 = Places (C, M, 200) *pour le même match et le même client.*

$$\mathbf{H} = r_1[S]r_2[S]r_1[M]r_1[C]w_1[M]r_2[M]r_2[C]w_2[M]w_1[C]c_1w_2[C]c_2$$

   (a) *Montrer que* **H** *n'est pas sérialisable.*

   (b) *On suppose qu'il y a 1000 places prises dans* $Match$ *au début de l'exécution. Combien y en a-t-il à la fin de l'exécution de* **H** *? Combien de places* C *a-t-il réellement payées ? A qui profite l'erreur ?*

   (c) *Appliquer un verrouillage à deux phases sur* **H** *et donner l'exécution* **H'** *résultante. Les verrous sont relâchés après le* commit.

2. l'indice P désigne $Places$, A $Augmenter$.

Chapitre 12

Travaux pratiques

Sommaire

Ce petit document explique l'essentiel de ce qu'il faut savoir pour les TP Oracle. Pour tout ce qui n'est pas essentiel, l'enseignant est là pour vous aider !

Commencez par lire en entier la section **Connexion au système** avant de toucher à la machine. Cela peut vous éviter des ennuis. Ensuite, essayez de vous connecter en procédant lentement la première fois. Une fois que vous êtes connectés et aue vous avez effectué votre premier ordre SQL, le plus dur est fait !

Dans ce qui suit, la partie **Les commandes utiles** vous explique les commandes UNIX de base : situer son répertoire, éditer un fichier, obtenir la liste de ses fichiers, etc. La partie **Comment effectuer des requêtes SQL** donne quelques recommandations pour utiliser de manière optimale votre environnement de travail.

Les subsections suivantes traitent du coeur du sujet : la base SQL et les exercices proposés.

Bon courage !

12.1 Environnement

Prenez la peine de lire ceci jusqu'à la fin sans toucher à rien.

12.1.1 Connexion au système

Sauf imprévu, vous êtes face à un terminal connecté au réseau du CNAM. Vous disposez d'un compte sur la machine du réseau qui s'appelle `celsius`. Pour accéder à `celsius`, on procède comme suit :

1. Avec le bouton droite de la souris, on clique sur le fond de l'écran.

2. Un menu aparaît, proposant entre autres le choix `telnet` : il faut activer ce choix.

3. On demande le nom de votre serveur : il faut taper `celsius`.

Il peut y avoir quelques variantes en fonction de votre terminal, mais en expérimentant un peu, vous devez vous en tirer. Si tout s'est bien passé, votre terminal communique avec `celsius` et vous demande votre nom (`username`), puis votre mot de passe.

1. Votre `username` est votre nom propre, **limité aux 6 premières lettres**, plus le caractère _ ("souligné") , plus la première lettre de votre prénom.
   Exemple : l'auditeur Michel Platini a pour username `platin_m`.

2. Votre mot de passe figure sur votre carte CNAM. **ATTENTION : il s'agit du numéro en haut à gauche**.

Une fois connecté, vous devez initialiser votre environnement ORACLE. Cela se fait avec la commande suivante :

```
source ~rigaux/env_oracle
```

Vous avez maintenant accès à ORACLE en tapant la commande suivante :

```
sqlplus /
```

Attention : il y a un blanc ' ' après les `sqlplus`. SQLPLUS est l'utilitaire d'ORACLE qui permet de soumettre directement des commandes SQL. Vous devriez normalement obtenir l'affichage des messages suivants :

```
SQL*Plus: Release 3.2.2.0.0 - Production on Mon Nov 24 12:16:03 1997

Copyright (c) Oracle Corporation 1979, 1994.  All rights reserved.

Connected to:
Oracle7 Server Release 8.0.1.0.0 - Production Release
With the distributed option
PL/SQL Release 2.2.3.0.0 - Production

SQL>
```

Il ne reste plus qu'à effectuer votre première requête (attention au point-virgule à la fin de la commande) :

```
SQL> select titre from film;
```

Et voilà ! Si quelque chose cloche et que vous ne comprenez pas pourquoi (après y avoir réfléchi ...) demandez à l'enseignant.

Un bon truc pour finir : la touche `CTRL-C` interrompt une commande.

12.1.2 Les commandes utiles

La machine `celsius` fonctionne sous le système d'exploitation UNIX. Pas besoin d'être un expert pour les TP SQL. Voici juste quelques commandes qui peuvent s'avérer utiles.

```
% ls -l               // Liste des fichiers du repertoire courant
% pwd                 // Nom du repertoire courant
% cd rep              // Se positionne dans le repertoire 'rep'
% cd                  // Ramene au repertoire de depart
% cp source cible     // Copie le fichier source vers le fichier cible
% mv source cible     // Renomme  le fichier 'source' en fichier 'cible'
% nedit &             // Lance l'editeur de texte nedit en tache de fond
% netscape &          // Lance netscape en tache de fond
```

Il est conseillé de lancer `nedit` et `netscpape`. Le premier permet d'éditer très facilement des fichiers pour y saisir des requêtes (ou autres) ; le deuxième donne accès au WEB et donc à beaucoup d'informations, y compris le corrigé du TP. Le fait de lancer un processus en tâche de fond avec l'option '&' siginifie qu'il s'exécute sans bloquer votre terminal.

12.1.3 Utilisation de SQLPLUS

Vous allez utiliser SQLPLUS pour exécuter des commandes SQL de type DDL (création de tables) et DML (recherche, mises-à-jour, ...). On peut entrer directement les commandes en les tapant sous SQLPLUS, mais en cas de faute de frappe ou d'erreur, il est difficile de corriger le texte. Il est donc fortement recommandé de procéder de la manière suivante :

1. Avec `nedit`, tapez votre commande, et sauvegardez-la dans un fichier (par exemple, entrez `select titre from film;` dans `nedit` et enregistrez ce texte dans le fichier `req0.sql`).
2. **Sous SQLPLUS** : demandez l'exécution du fichier `req0.sql` comme suit :

```
SQL> @req0
```

3. Si quelque chose cloche, corrigez avec `nedit`, **enregistrez le fichier**, et ré-exécutez-le.

Créez ainsi un fichier pour chaque commande : celà vous permettra de ne rien perdre et d'avoir un minimum de frappe clavier.

Sous SQLPLUS, vous avez quelques commandes utiles dont voici une brève liste.

```
SQL> desc tab                    // Donne le schema de la table 'tab'.
SQL> select table_name
        from user_tables;        // Liste des tables que vous avez creees.
SQL> list                        // Affiche le dernier ordre execute
SQL> spool fic                   // Copie l'affichage a l'ecran dans 'fic.lst'
SQL> spool off                   // Stoppe la copie dans 'fic.lst'
SQL> exit;                       // Sortir de SQLPLUS
```

Les commandes `DROP TABLE` permettent d'exécuter le fichier de création plusieurs fois de suite, en détruisant d'abord les tables éventuellement crées lors des exécutions précédentes. Il faut détruire les tables dans l'ordre inverse de création, afin de ne pas violer les contraintes de `FOREIGN KEY`.

Au départ, il n'y a aucune table dans votre propre espace ORACLE, mais vous avez accès au schéma et à la base de données "Officiel des spectacles", vue et revue en cours, qui est partagée (en lecture) par tout le monde. Voici les commandes qui ont étées utilisées pour la création du schéma de cette base (**attention** : les tables existent déjà, ne retapez pas les commandes de création ci-dessous) :

```
DROP TABLE seance;
DROP TABLE salle;
DROP TABLE cinema;
DROP TABLE role;
DROP TABLE film;
DROP TABLE artiste;

CREATE TABLE Artiste  (Nom VARCHAR2 (20) NOT NULL,
                       Prenom VARCHAR2 (15),
                       Annee_naissance NUMBER(4) ,
                       PRIMARY KEY (Nom));

CREATE TABLE film  (ID_film         NUMBER(10) NOT NULL,
                    Titre           VARCHAR2(30),
```

```
                    Annee           NUMBER(4),
                    Nom_Realisateur VARCHAR2(20),
                    PRIMARY KEY (ID_film),
                    FOREIGN KEY (Nom_realisateur) REFERENCES Artiste);

CREATE TABLE Role  (Nom_role    VARCHAR2(20) NOT NULL,
                    ID_film     NUMBER (10) NOT NULL,
                    Nom_acteur  VARCHAR2 (20) NOT NULL,
                    PRIMARY KEY (ID_film, nom_acteur),
                    FOREIGN KEY (ID_film) REFERENCES Film
                    ON DELETE CASCADE,
                    FOREIGN KEY (Nom_acteur) REFERENCES Artiste
                    ON DELETE CASCADE);

CREATE TABLE cinema  (Nom_cinema     VARCHAR2 (10) NOT NULL,
                      Arrondissement NUMBER (2),
                      Adresse        VARCHAR2 (30),
                      PRIMARY KEY (Nom_cinema));

CREATE TABLE salle  (Nom_cinema   VARCHAR2(10) NOT NULL,
                     No_salle     NUMBER(2) NOT NULL,
                     Climatise    CHAR(1),
                     Capacite     NUMBER(4),
                     PRIMARY KEY (Nom_cinema, No_salle),
                     FOREIGN KEY (Nom_cinema) REFERENCES cinema
                     ON DELETE CASCADE);

CREATE TABLE seance  (Nom_cinema   VARCHAR2(10) NOT NULL,
                      No_salle     NUMBER(2) NOT NULL,
                      No_seance    NUMBER(2) NOT NULL,
                      Heure_debut  NUMBER (4,2),
                      Heure_fin    NUMBER (4,2),
                      ID_film      NUMBER(10)  NOT NULL,
                      PRIMARY KEY (Nom_cinema, No_salle, No_seance),
                      FOREIGN KEY (Nom_cinema, No_salle) REFERENCES salle
                      ON DELETE CASCADE);
```

Vous pouvez remarquer que des hypothèses simplificatrices ont été faites sur les clés de certaines tables : on admet par exemple que le nom du cinéma est la clé pour un cinéma. Ce n'est certainement pas tout à fait correct (cf. le cours sur le modèle relationnel), mais cela permet de simplifier les requêtes.
Remarques importantes :

- **TOUTES LES COMMANDES SQL DOIVENT SE TERMINER PAR UN ';'**. Si vous oubliez le ';', une ligne '2' vous est proposée. Dans ce cas tapez un ';' pour finir la commande.
- Les chaînes de caractères s'écrivent avec une simple quote : 'Vertigo' et pas "Vertigo".
- Les majuscules et les minuscules sont interprétés différemment. Par exemple 'Vertigo' est considéré comme différent de 'vertigo' ou 'VERTIGO'. Pensez-y en faisant des sélections ! Un moyen d'éviter les probèmes est d'utiliser la fonctions `UPPER` qui met tout en majuscule. Par exemple :

  ```
  select * from film where UPPER(titre) = 'VERTIGO';
  ```

  Sinon, vous pouvez vous baser sur la convention suivante : toutes les chaînes de caractères commencent par une majuscule suivie de minuscules.

– Oracle n'implante que partiellement la norme SQL2. Voici quelques différences importantes :

1. La clause `ON UPDATE ..` n'existe pas.
2. La notion de schéma se confond avec celle d'utilisateur : toutes les tables (et les vues, contraintes, triggers, etc) crées par un même utilisateur constituent un schéma.
3. On ne peut pas utiliser de sous-requêtes dans une clause `CHECK`.

Maintenant, à vous de jouer !

12.2 Requêtes SQL

Quand vous vous connectez à la base, vous avez automatiquement accès à la base "Officiel des spectacles" dont le schéma a été présenté précédemment. Cette base contient un petit jeu de données plus ou moins réaliste. A vous de jouer : il faut concevoir, saisir et exécuter les ordres SQL correspondant aux requêtes suivantes.

12.2.1 Sélections simples

1. Les titres de films triés.
2. Nom et année de naissance des artistes nés avant 1950.
3. Les cinémas du 12ème arrondissement.
4. Les artistes dont le nom commence par 'H' (commande LIKE).
5. Quels sont les acteurs dont on ignore la date de naissance ? (Attention : cela signifie que la valeur n'existe pas).
6. Combien de fois Bruce Willis a-t-il joué le role de McLane ?

12.2.2 Jointures

1. Qui a joué Tarzan (nom et prénom) ?
2. Nom des acteurs de Vertigo.
3. Films dont le réalisateur est Tim Burton, et un des acteurs avec Jonnhy Depp.
4. Quels films peut-on voir au Rex, et à quelle heure ?
5. Titre des films dans lesquels a joué Woody Allen. Donner aussi le rôle.
6. Quel metteur en scène a tourné dans ses propres films ? Donner le nom, le rôle et le titre des films.
7. Quel metteur en scène a tourné en tant qu'acteur ? Donner le nom, le rôle et le titre des films où le metteur en scène a joué.
8. Où peut-on voir Shining ? (Nom et adresse du cinéma, horaire).

9. Dans quels films le metteur-en-scène a-t-il le même prénom que l'un des interprètes ? (titre, nom du metteur-en-scne, nom de l'inteprète). Le metteur-en-scène et l'interprète ne doivent pas être la même personne.

10. Où peut-on voir un film avec Clint Eastwood ? (Nom et adresse du cinéma, horaire).

11. Quel film peut-on voir dans le 12e arrondissement, dans une salle climatisée ? (Nom du cinéma, No de la salle, horaire, titre du film).

12. Liste des cinémas (Adresse, Arrondissement) ayant une salle de plus de 150 places et passant un film avec Bruce Willis.

13. Liste des cinémas (Nom, Adresse) dont TOUTES les salles ont plus de 100 places.

12.2.3 Négation

1. Quels acteurs n'ont jamais mis en scène de film ?

2. Les cinémas (nom, adresse) qui ne passent pas un film de Tarantino.

12.2.4 Fonctions de groupe

1. Total des places dans les salles du Rex.

2. Année du film le plus ancien et du film le plus récent.

3. Total des places offertes par cinéma.

4. Nom et prénom des réalisateurs, et nombre de films qu'ils ont tournés.

5. (♣)Nom des cinémas ayant plus de 1 salle climatisée.

6. (♣)Les artistes (nom, prénom) ayant joué au moins dans trois films depuis 1985, dont au moins un passe a l'affiche a Paris (donner ausssi le nombre de films).

12.3 Concurrence d'accès

On va maintenant étudier le comportement concret d'un SGBD en cas d'accès concurrents à la même ressource. Pour celà on va simuler l'exécution concurrente de programmes à l'aide du petit ensemble de lectures/écritures sur la base "Agence de voyage" créé dans le premier exercice : modification du solde de certains clients et sélection des clients. Créez 4 fichiers, nommés `SEL.sql, MAJpas.sql, MAJfog.sql et MAJker.sql` et entrez-y les commandes suivantes :

1. `SEL.sql`

```
PROMPT 'Affichage des clients =>';
SELECT * FROM client;
```

2. `MAJpas.sql`

```
PROMPT 'Augmentation du client Pascal =>';
UPDATE client  SET solde = solde + 1000
WHERE  client = 'Pascal';
```

3. `MAJfog.sql`

```
PROMPT 'Augmentation du client Fogg =>';
UPDATE client SET solde = solde + 1000
WHERE  client = 'Fogg';
```

4. `MAJker.sql`

```
PROMPT 'Augmentation du client Kerouac =>';
UPDATE client SET solde = solde + 1000
WHERE  client = 'Kerouac';
```

Ensuite, ouvrez deux fenêtres et lancez SQLPLUS dans chacune : chaque session est considérée par ORACLE comme un utilisateur, et on a donc 2 utilisateurs, nommés 1 et 2, en situation de concurrence. Dans tout ce qui suit, on note $INSTR_i$ l'exécution de l'instruction $INSTR$ par l'utilisateur i. Par exemple $MAJker_1$ corespond à l'exécution du fichier `MAJker` dans la première fenêtre par la commande `@MAJker`. On note de même ROL_i et COM_i l'exécution des commandes `rollback;` et `commit;` dans la fenêtre i.

Questions Executez les séquences d'instruction décrites ci-dessous. Dans chacun des cas, expliquez ce qui se passe.

1. Première expérience : l'utilisateur 1 effectue des mises-à-jour, tandis que l'utilisateur 2 ne fait que des sélections. Que constate-t-on ?

$$SEL_1, SEL_2, MAJker_1, SEL_2, MAJpas, SEL_2, ROL_1, SEL_2.$$

2. idem, mais avec des *commit*.

$$SEL_1, SEL_2, MAJker_1, SEL_2, COM_1, SEL_2, MAJker_1, SEl_2, COM_1, SEL_2.$$

3. Maintenant les deux utilisateurs effectuent des MAJ simultanées.

$$SEL_1, SEL_2, MAJker_1, MAJpas_2, SEl_1, SEL_2, MAJfog_1, MAJfog_2, SEL_1, COM_1, COM_2.$$

   Un bloquage est apparu. Pourquoi ?

4. Idem, avec un ordre des opérations qui diffère d'un utilisateur à l'autre.

$$SEL_1, SEL_2, MAJker_1, MAJpas_2, SEl_1, SEL_2, MAJpas_1, MAJker_2 ROL_1 ROL_2.$$

   Que constate-t-on ?

5. (♣)En fait, ORACLE pratique une verrouilage à deux phases assez libéral, qui ne garantit pas la sérialisabilité : aucun verrrou n'est placé sur une ligne lors d'une lecture. L'utilisateur peut verrouiller explicitement en ajoutant la clause `FOR UPDATE`. Exemple :

```
SELECT * FROM client FOR UPDATE;
```

Chaque ligne sélectionnée est alors verrouillée. Refaites les expériences précédentes avec un verrouillage explicite des lignes que vous voulez modifier.

6. (♣)Expérimentez les exécutions écédentes en spécifiant le mode suivant :

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

12.4 Normalisation d'un schéma relationnel

On va maintenant créer des tables, y insérer des informations et effectuer des requêtes sur le schéma obtenu. L'application visée est le système d'information d'un zoo, et on suppose que l'on se trouve dans la situation suivante : une personne peu avertie (elle n'a pas suivi les enseignements du CNAM !) a créé en tout et pour tout une seule table dans laquelle on trouve toutes les informations. Voici le schéma de cette table.

```
CREATE TABLE Zoo (Animal          Number(4),
                  Nom             VARCHAR2 (20),
                  Annee_naissance NUMBER(4),
                  Espece          VARCHAR2(10),
                  Gardien         VARCHAR2 (20),
                  Prenom          VARCHAR2 (10),
                  Salaire         NUMBER (10,2),
                  Classe          VARCHAR2 (10),
                  Origine         VARCHAR2 (10),
                  Emplacement     NUMBER(4),
                  Surface         NUMBER (3),
                  Type_empl       NUMBER(2),
                  Libelle_empl    VARCHAR2 (20));
```

Chaque ligne corrrespond à un animal auquel on attribue un nom propre, une année de naissance et une espèce (Ours, Lion, Boa, etc.). Cet animal est pris en charge par un gardien (avec prénom et salaire) et occupe un emplacement dans le zoo (numéro d'emplacement, surface, type_emplacement et libellé_emplacement : savane, désert, forêt, etc.). Enfin chaque espèce appartient à une classe (les mammifères, poissons, reptiles, batraciens ou oiseaux) et on considère pour simplifier qu'elle provient d'une origine unique (Afrique, Europe, etc.).

Vous pouvez consulter avec SQL le contenu de cette table `Zoo` qui vous est acessible en lecture. On constate à l'oeil nu de nombreuses redondances et anomalies (les voyez-vous ?). Commencez par copier la table chez vous avec la commande suivante :

```
CREATE TABLE zoo AS
SELECT * FROM zoo;
```

Le schéma n'est évidemment pas correct : on n'y trouve même pas de contraintes (`NOT NULL`) ou de définition de clé primaire. Le premier travail est donc de définir un bon schéma relationnel. Pour cela, on vous donne les spécifications suivantes, sous forme de dépendances fonctionnelles :

- Animal → Nom, Année_naissance, Espèce, Emplacement.
- Nom, Espèce → Animal.
- Espèce → Origine, Classe.
- Gardien → Prénom, Salaire.
- Emplacement → Surface, Type_emplacement, Gardien.

– Type_emplacement → Libellé_emplacement.

Questions

1. Le contenu actuel de la table est-il conforme à ces spécifications ? Indication : pour chaque dépendance fonctionnelle $A \rightarrow B$, où B est un attribut, exécutez l'ordre SQL

   ```
   SELECT A, count(DISTINCT B)
   FROM zoo
   GROUP BY A;
   ```

   Si la table respecte la DF, `count(*)` doit valoir `...` (à votre avis ?). Vous pouvez donc chercher les anomalies en ajoutant un `HAVING COUNT (DISTINCT B) >...` à la requête ci-dessus.

2. Effectuez les corrections nécessaires avec des ordres `UPDATE`. Quand il y a une anomalie ou une incohérence entre deux lignes, on considère que la première est la bonne. IMPORTANT : on valide une mise-à-jour avec la commande `commit;`.

3. Montrer que `Animal` et `Nom, Espèce` sont des clés de la table `Zoo` ?

4. (♣)Montrer que ce sont les seules clés.

5. Est-elle en troisième forme normale (donnez un argument formel) ? Y-a-t-il des redondances/anomalies prévisibles ? Les retrouvez-vous dans la table `Zoo` ?

6. Trouvez un schéma en troisième forme normale, créez les ordres `CREATE TABLE` correspondant (avec des contraintes `PRIMARY KEY`, `FOREIGN KEY NOT NULL`, `UNIQUE`) et exécutez-les. ATTENTION : une table à laquelle on fait référence dans un `FOREIGN KEY` doit avoir été créée avant. Il faut donc faire attention à l'ordre de création des tables.

7. Une fois le schéma créé, insérez les données dans les tables du 'bon' schéma en les copiant à partir de la table `Zoo`. Pour copier des données d'une table `A` vers une table `B(B1,B2, ... Bn)` , SQL fournit une commande qui est un mélange de `SELECT` et de `INSERT`.

   ```
   INSERT INTO TABLE B (B1, B2, ... Bn)
   SELECT DISTINCT A1, A2, ... An
   FROM A
   WHERE ...;
   ```

   En fait l'ordre `SELECT` peut accéder à plusieurs tables (jointures, différences). Contrainte importante : il doit y avoir autant de `Ai` que de `Bi`, et pour les mêmes types. Exemple :

   ```
   INSERT INTO ESPECE (espece, classe, origine)
   SELECT DISTINCT espece, classe, origine FROM zoo;
   ```

Vous devez maintenant avoir un bon schéma et un mauvais (la table `Zoo` toute seule). Sur ces deux schémas, exprimez les requêtes SQL qui suivent.

1. Quels sont les Ours du zoo ?
2. Quels animaux s'appellent Martin ?
3. Quels animaux habitent dans la jungle ?
4. De quels animaux s'occupe le gardien Dupond ?

5. (♣)Sur quel(s) emplacement(s) y-a-il des animaux de classes diffrentes (no, surface et libellé du type de l'emplacement).
6. Somme des salaires des gardiens.
7. ...

Considérons les anomalies qui existaient initialement : sont-elles encore possibles dans le 'bon' schéma. D'un autre côté, y-a-t-il des requêtes que l'on peut exprimer sur `Zoo` et pas sur le nouveau schéma ? (autrement dit : a-t-on perdu de l'information ?)

Conclusion : qu'a-t-on gagné, qu'a-t-on perdu ?

12.5 (♣)Optimisation

ORACLE fournit sous SQLPLUS un outil, EXPLAIN, qui donne une description du plan d'exécution choisi par le système pour une requête quelconque. EXPLAIN est très simple à utiliser. Il fonctionne de la manière suivante :

1. Tout d'abord on crée une table, `plan_table`, qui est destinée à contenir toutes les informations relatives à un plan d'exécution. La table doit être créée avec le fichier de commandes `plan_table.sql` [1].
2. Ensuite on exécute une requête en demandant le stockage des explications relatives à cette requête. Exemple :

```
EXPLAIN PLAN
       SET statement_id = 'cin0'
       FOR SELECT titre, heure_debut
           FROM   seance s, film f
           WHERE  s.id_film = f.id_film
           AND    f.titre='Vertigo';
```

   La clause '`statement_id = 'cin0''` attribue un identifiant au plan d'exécution de cette requête dans la table `plan_table`. Bien entendu chaque requête stockée dans `plan_table` doit avoir un identifiant spécifique.
3. Pour connaître le plan d'exécution, on interroge la table `plan_table`. L'information est un peu difficile à interpréter : le plus simple est de faire tourner le fichier `explain.sql` (à récupérer au même endroit que précédemment). Quand on exécute ce fichier, il demande (2 fois) le nom de la requête à expliquer. Dans le cas de l'exemple ci-dessus, on répondrait deux fois `'cin0'`. On obtient l'affichage suivant qui présente de manière relativement claire le plan d'exécution (cf. le cours).

```
Plan d'execution
----------------------------------------------------------------------------
0 SELECT STATEMENT
  1 NESTED LOOPS
    2 TABLE ACCESS FULL SEANCE
    3 TABLE ACCESS BY ROWID FILM
      4 INDEX UNIQUE SCAN SYS_C004709
```

   Ici, le plan d'exécution est le suivant : on parcourt en séquence la table SEANCE (ligne 2) ; pour chaque séance, on accède à la table FILM par l'index [2] (ligne 4), puis pour chaque `ROWID` provenant de l'index, on accède à la table elle-même (ligne 3). Le tout est effectué dans une boucle imbriquée (ligne 1).

1. Vous pouvez récupérer ce fichier en le copiant avec la commande suivante : `cp /users/ensinf/rigaux/PUBLIC/utlxplan.sql .`
2. Cet index a été automatiquement créé en association avec la commande `PRIMARY KEY` lors de la création de la table.

Questions

1. Reprendre les requêtes définies au début du TP sur la base de données 'Officiel des Spectacles', et expliquer le plan d'exécution donné par ORACLE.
2. Supprimer quelques index, et regarder le changement dans les plans d'exécutions. NB : vous pouvez obtenir la liste des index existant sur vos tables avec la commande :

```
SELECT table_name, index_name FROM user_indexes;
```