Bases de données

Concepts et programmation

Antoine Cornuéjols

AgroParisTech, Spécialité Informatique (2009-2010)

Version du 19 octobre 2009

Table des matières

Chapitre 1

Concepts fondamentaux

Sommaire

1. Introduction

1.1 Un vieux problème : la bibliothèque de Leibniz

1.2 Données, bases de données et SGBD

Une *donnée* est une information quelconque, comme, par exemple, « `voici une personne, elle s'appelle Antoine` ». C'est aussi une relation entre des informations. Par exemple : « `Antoine 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 la caractéristique essentielle que l'on cherche à les mémoriser de manière permanente.

Définition 1.1 (Base de données)

Une base de données *est un gros ensemble d'informations* structurées *mémorisées sur un* support permanent *qui peut être* partagée *par plusieurs applications et qui est* interrrogeable par le contenu.

L'utilisation de fichiers classiques pourrait sembler pouvoir apporter une solution à ce problème. Mais l'utilisation directe de gros fichiers soulève de gros problèmes :

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

Il est donc nécessaire d'avoir 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'interfaces nécessaires à l'accès aux données. Ce logiciel (le SGDB) est très complexe.

Définition 1.2 (Système de Gestion de Bases de Données (SGBD))

Un Système de Gestion de Bases de Données (SGBD) *est un logiciel de haut niveau permettant aux utilisateurs de* structurer*, d'*insérer*, de* modifier*, de* rechercher *de manière efficace des données spécifiques, au sein d'une grande quantité d'informations, stockées sur mémoires secondaires* partagée *de manière* transparente *par* plusieurs utilisateurs.

Plus précisément, les **systèmes de gestion de bases de données** (SGBD) sont des programmes permettant à l'utilsateur de créer et de gérer des bases de données. les SGBD sont des *logiciels à usage général* qui assurent les processus de *définition*, de *construction*, de *manipulation* et de *partage* des bases de données par et entre les différents utilisateurs et applications.

La **définition** d'une base de données implique une spécification des types de données, des structures et des contraintes. La **construction** d'une base de données consiste à enregistrer les données proprement dites sur un support de stockage contrôlé par le SGBD. La **manipulation** de la base consiste notamment à l'interroger afin d'en extraire des informations, à l'actualiser en fonction des modifications du microcosme considéré et à générer des rapports. Le **partage** d'une base de données permet à différents utilsateurs et applications d'y accéder simultanément.

Les SGBD assurent d'autres fonctions importantes, notamment la *protection* de la base de données et son /em entretien à long terme. La **protection** implique à la fois la *protection du système* contre les pannes logicielles et matérielles et la *protection sécuritaire* conre les accès illicites ou malveillants. Une grande base de données peut être utilisées de nombreuses années. Le SGBD doit donc être capable d'**entretenir** et de faire évoluer ses propres structures dans la durée.

La complexité d'un SGBD tient essentiellement à la diversité des techniques mises en œuvre, à la multiplicité des composants intervenant dans son architecture, et aux différents types d'utilisateurs (adminis-

trateurs, programmeurs, utilisateurs non informaticiens, ...) qui sont confrontés, à différents niveaux, au système.

Ainsi, dans ce cours, seront abordés :
- *Les modèles de données* : entité-relation (mais pas les modèles en réseaux, hiérarchiques, relationnels, orientés objets, sémantiques).
- Les *langage de requête*, incluant leurs fondements théoriques et les langages comme SQL.
- Les *techniques de stockage*.
- L'*organisation des fichiers* : index, arbre-B, hachage, ...
- L'*architecture* : centralisée, distribuée, 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 panne*.

Ces fonctions sont remplies par trois niveaux, théoriquement distincts, d'un SGBD :

1. Le ***niveau physique*** : 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).
2. Le ***niveau logique*** : Définition de la structure des données : Langage de Description des Données (LDD) ; Consultation et Mise à Jour des données : Langage de Requête (LR) et Langage de Manipulation de Données (LMD) ; Gestion de la confidentialité (sécurité) ; Maintien de l'intégrité.
3. Le ***niveau externe*** : Vues ; Environnement de programmation (intégration avec un langage de programmation) ; Interfaces conviviales et Langage de 4ème Génération (L4G) ; Outils d'aides (e.g. conception de schémas) ; Outils de saisie, d'impression d'états.

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

Nous avons introduit un certain nombre de termes et de sigles. Nous allons progressivement en apprendre le sens.

1.3 Un exemple : BD sur l'INA-PG

- Qu'est-ce qu'on veut faire ?
- Quelles données sont utiles ?
- Quels types de requêtes ?
- Quelles mises à jour ?
  - Quels acteurs ?
  - Concurrence
  - Redondance et incohérence
- Quelle sécurité ?

1.3.1 Questions essentielles

- Comment concevoir une BD ?
- Comment optimiser sa conception ?
- Comment optimiser son fonctionnement ?
- Comment assurer la concurrence ?
- Comment assurer la sécurité ?

2. Quelles compétences pour utiliser un SGBD ?

L'utilisation d'un SGBD suppose de comprendre et 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* : recherches, mises-à-jour, etc.
3. *Partager les données* entre plusieurs utilisateurs (mécanismes de *transaction*).
4. *Optimiser les performances* par le réglage de l'organisation physique des données. cet aspect relève plutôt de l'administrateur de données.

En reprenant ces différents points.

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). Dans un SGBD, il existe plusieurs modèles plus ou moins abstraits des mêmes objets. Par exemple :
- Le *modèle conceptuel* : la description du système d'information
- Le *modèle logique* : réglant l'interface avec le SGBD
- Le *modèle physique* : les fichiers.

Ces différents modèles correspondent aux niveaux de 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 décrivant :
- l'analyse du monde réel telle que pertinente pour l'utilisation
- 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 les 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 : outre les structures et les opérations, un modèle logique doit permetre 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 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 (4à le SGBD se charge de transcrire les commandes du LMD en instructions appliquées à la représentation physique.

Cette approche offre de très grands avantages. Tout d'abord elle ouvre l'utilisation des SGBD à des 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 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'ambiguïtés. Ensuite, le langage ne devrait pas demander d'expertise technique (syntaxe compliquée, structure 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.

2.3 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.

2.4 Concurrence d'accès

Plusieurs utilisateurs doivent pouvoir accéder en même temps aux mêmes données. Le SGBD doit savoir :
- Gérer les conflits si ces utilisateurs font simultanément des mises-à-jour.
- Offrir un mécanisme de retour arrière si on décide d'annuler des modifications en cours.
- Donner une image cohérente des données sur l'un fait des requêtes et un autre des mises-à-jour.

Le but est d'éviter des blocages tout en empêchant des modifications anarchiques.

3. Caractéristiques de l'approche base de données

- Abstraction des données : requêtes en termes presque en langage naturel
- Notion de vues

3.1 Architecture des SGBD et indépendance des données

Architecture en trois schémas

3.2 Persistance des données

3.3 Langage de requête

3.4 Partage des données

3.5 Fiabilité des données

3.6 Sécurité des données

4. Le contexte

4.1 Pourquoi une Base de Données ?

- Intégration de données
  - Moins de duplications
- Partage de données
- Fiabilité de données
  - Transactions, Reprises sur pannes, Tolérance de pannes
- Sécurité de données
- Langages assertionnels de requêtes
  - SQL, QBE
- Interfaces conviviales
  - 4-GL & Web

4.2 Où trouve-ton des BD

4.2.1 Types de BDs

- BDs personnelles
  - MsAccess etc.
  - 10 KO – 100 KO
- BDs professionnelles typiques 100 KO – 100 GO
- BDs professionnelles très grandes
  - Very Large Databases (VLDB)
  - $>$ 100 GO

4.3 Applications récentes et à venir

4.4 Avantages liés à l'utilisation d'un SGBD

4.5 Les acteurs en jeu dans l'usage de SGBD

4.6 Historique des SGBD et types de SGBD

4.6.1 Historique

Moteur : évolution des modèles de données, elle-même motivée par l'évolution des besoins.

- 1ère génération 1950 – 65
  - SGF, SGF généralisés avec les langages booléens de manip.
- 2ème génération 1965 - 70
  - SGBD navigationnel
  - Hierarchique (IMS), Réseau (Codasyl), Pseudo-relationnel
- 3ème génération 1969 - ...

  - SGBD relationnel (DB2, Oracle, Informix, MsAcess...
- SGBD OO 1990 - 1999
  - En pratique : une impasse (O2, Objectstore, Objectivity..)
- SGBD relationnel – objet (RO) 1993 - ...
  - Évolution probable de tout SGBD relationnel

Il faut donc distinguer entre *modèles conceptuels*, /em a priori indépendants des SGBD, et /em modèles logiques, caractéristiques d'une génération de SGBD. Ces modèles logiques sont eux-mêmes dépendants d'organisations physiques (placement et méthodes d'accès) comme, par exemple, les graphes en arbres ou en réseaux, seules structures offertes par la première génération de SGBD, celle des modèles navigationnels, hiérarchique et réseau.

Ces SGBD obligeaient à "voir" - c'est à dire à traduire - une base de données comme un ensemble d'enregistrements, reliés les uns aux autres par des ensembles de pointeurs. Cette organisation fige les relations existant entre les différents enregistrements de la base et impose un type de manipulation, la navigation, qui consiste à suivre, d'enregistrement en enregistrement, les chaînes de pointeurs. En fait, il serait préférable de parler de modèles d'accès (physiques) plutôt que de modèles de données (logiques) pour ces anciens SGBD. En effet leur organisation physique des données imposait la méthode d'accès et, au delà, la nature des langages de manipulation.

4.6.2 Le modèle hiérarchique

Longtemps considéré comme le seul modèle permettant aux SGBD d'atteindre les performances exigées en production, le modèle hiérarchique possède effectivement la capacité de traiter rapidement des informations organisées sous la forme d'une hiérarchie stricte mais, par contre, interdit tout autre type de représentation, limitant ainsi considérablement la puissance d'expression du modèle. Le SGBD le plus connu dans cette catégorie est IMS, produit ancien de IBM, très répandu dans les applications de production.

4.6.3 Le modèle réseau

Ce modèle a été introduit en 1961 par BACHMAN. Il propose une utilisation de structures de listes afin de relier sémantiquement des entités. Il permet ainsi une meilleure représentation de la réalité que le modèle hierarchique. Un des intérêts du modèle fut l'existence d'une proposition de normalisation émise par le groupe DBTG (Data Base Task Group) du Comité CODASYL (COnference on DAta SYstem Language). On parle ainsi souvent de modèle CODASYL pour les systèmes réseaux qui suivent ces recommandations. Deux niveaux de recommandations ont été émis : l'un, en 1971, conseillait la séparation d'un niveau de schéma externe et d'un niveau de schéma interne/conceptuel ; le second, en 1978, préconisait les trois niveaux de schéma -interne, conceptuel, externe-. Seul le premier niveau de recommandations a été réellement suivi par les produits. En 1978, il était déjà tard pour voir les produits prendre en compte ces recommandations : l'heure du relationnel avait déjà sonné pour les investissements à long terme ; quasiment aucun SGBD réseau réellement nouveau n'a été conçu depuis cette date. Mais à l'heure où les SGBD relationnels sont à leur tour défiés par les nouveaux SGBD orientés objets, il est encore intéressant de comprendre ce que les SGBD réseau avaient apporté de plus novateur : l'expression des requêtes sur les données dans une logique de chemin (c.à.d. de navigation). Parmi les SGBD construits selon ce modèle, encore largement utilisés, on peut citer IDS2 chez BULL, IDMS de CULLINET, TOTAL de CINCOM, SOCRATE.

5. Objectifs du cours

5.0.4 Que doit-on savoir pour utiliser un SGBD

5.0.5 Définition du schéma de données

5.0.6 Opérations sur les données

5.0.7 Optimisation

5.0.8 Concurrence d'accès

5.0.9 Fonctionnement d'un SGBD

5.0.10 Plan du cours

Chapitre 2

Modèle abstrait des bases de données

Sommaire

1. Introduction : modélisation des données

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. En revanche, il admet certaines ambiguïtés.

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.

1.1 Généralités sur la conception d'une Base de Données

La réalisation d'une Base de Données implique trois grandes étapes (voir figure **??**) :

1. La définition d'un cahier des charges
2. La modélisation
3. L'implantation physique dans un système informatique.

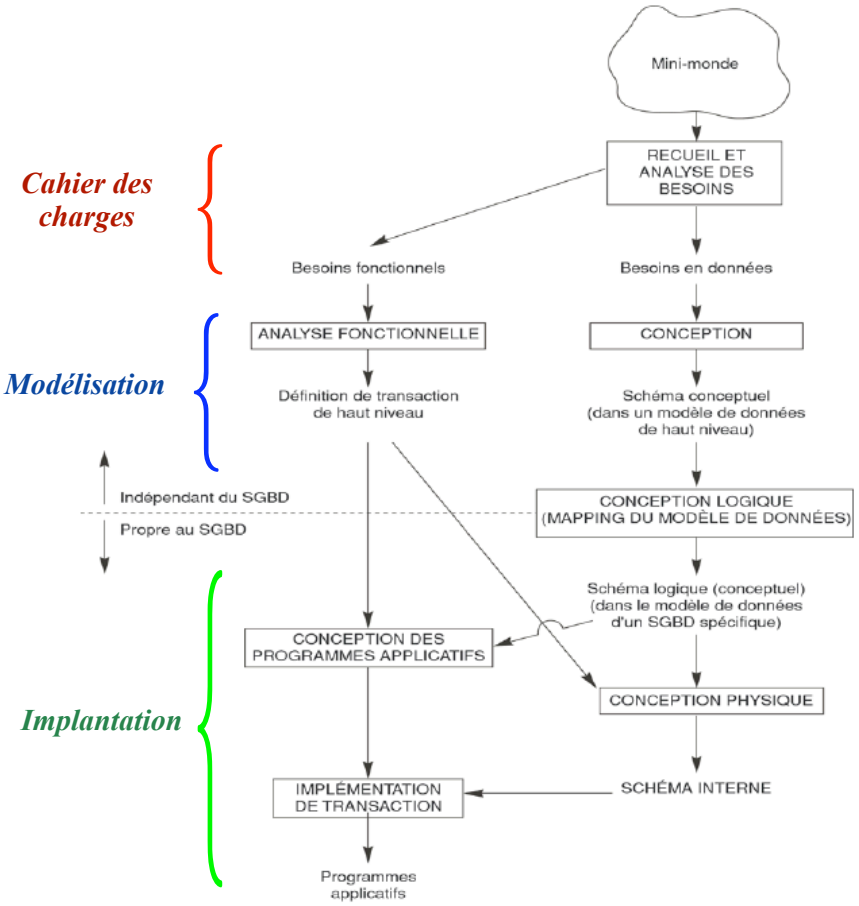

FIG. 2.1: *Grandes étapes de la conception d'une base de données.*

Nous allons nous intéresser ici à la deuxième étape.

Définition 2.1 (Modèle des données)

Un modèle des données *(« data model ») est une description formelle et structurée des données et de leurs relations dans un système d'information.*

Pour obtenir un modèle des données, il faut trois grandes phases :

1. Modélisation / analyse des données
2. Construction d'un modèle entité-association
3. Conversion en un schéma de base de données relationnelles.

Nous nous concentrons dans la suite sur la deuxième phases.

La méthode de conception 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.

1.2 Problèmes d'une conception sans précautions

Les redondances (duplications) présentes dans la table faite sans précaution peut conduire à plusieurs types de problèmes :
- Erreurs possibles lors de l'**insertion**
- Erreurs possibles lors de la **mise-à-jour**

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	Taskovski	Andrei	1932

FIG. 2.2: *Une table de films.*

  - Si l'on modifie l'année de naissance d'Hitchcock pour Vertigo ...
- Si **destruction** d'un film, on risque de supprimer le metteur en scène

1.3 Introduction à une 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.

EXEMPLE **Amélioration de la base de films**

A FAIRE

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

2. Le modèle entité-association (E/A)

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.

EXEMPLE **Schéma E/A pour la base de films**

A FAIRE

Voir figure 2.3.

Titre	Année	idMES
Alien	1979	1
Vertigo	1958	2
Psychose	1960	3
Kagemusha	1980	4
Volte-face	1997	5
Pulp Fiction	1995	6
Titanic	1997	7
Sacrifice	1986	8

Id	nomMES	PrénomMES	AnnéeNaiss
1	Scott	Ridley	1943
2	Hitchcock	Alfred	1899
3	Hitchcock	Alfred	1899
4	Kurosawa	Akira	1910
5	Woo	John	1946
6	Tarantino	Quentin	1963
7	Cameron	James	1954
8	Taskovski	Andrei	1932

FIG. 2.3: *Table des films et table des réalisateurs.*

2.1 Présentation informelle

Une **entité** est un objet concret ou abstrait
- ayant une existence propre
- fonction des besoins de modélisation

Exemple : `Assuré, Film, Contrat`

Une **association** est un lien sémantique entre entités
- fonction des besoins de modélisation

Exemple : `Réaliser` entre `réalisateur` et `film`.

Une **propriété** est une donnée élémentaire :
- ayant un sens
- pouvant être utilisée de manière autonome
  Ex : `NomAssuré, AnnéeSouscription, NbContrats`
- Servent à décrire les entités et les associations
- = *Attributs* ou *colonnes*
- prennent des valeurs appelées occurrences de la propriété

La modélisation conceptuelle est totalement *indépendante de tout choix d'implantation.* C'est la partie la plus stable d'une application. Elle vise à se concentrer sur l'essentiel : *Que veut-on stocker dans la base ?*

Le modèle E/A a été conçu en 1976 et est à la base de la plupart des méthodes de conception. La syntaxe est souvent celle d'UML En France, on utilise parfois la syntaxe MERISE (quasi équivalente)

Les types d'entités

Un type d'entité est composé de :
- son *nom*
- la *liste de ses attributs* avec, —optionnellement— le domaine dans lequel l'attribut prend ses valeurs : les entiers, les chaînes de caractères :
- l'indication du (ou des) attribut(s) premettant d'identifier l'entité : ils constituent la *clé.*

Une entité e est une *instance* de son type E.

Un ensemble d'entités $\{e_1, e_2, ..., e_n\}$ instances d'un même type E est une *extension* de E.

Définition 2.2 (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.*

EXEMPLE **Les clés**

Soit le type `Internaute` avec les attributs :
- `email`
- `nom`
- `prénom`

- `région`

`email` peut être une clé naturelle car c'est un attrinut unique et discriminant pour chaque internaute.
nom ne peut pas être une clé car plusieurs internautes peuvent avoir le même nom.
`(nom, prénom)` est une clé possible, mais peut poser des problèmes de performance et complique les manipulations par SQL.

2.2 Le modèle

2.2.1 Entités, attributs et identifiants

2.2.2 Associations binaires

2.2.3 Entités faibles

2.2.4 Associations généralisées

2.3 Avantages et inconvénients du modèle E/A

Chapitre 3

Langages de bases de données

Sommaire

1. Le modèle relationnel et ses règles

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 relationnel.

1.1 Un peu d'histoire

Le modèle relationnel a été introduit pour la première fois par Ted Codd[1] du centre de recherche d'IBM en 1970 dans un papier désormais classique XXX, et attira immédiatement un intérêt considérable en raison de sa simplicité et de ses fondations mathématiques. Le modèle utilise le concept de *relation mathématique* qui est associée à une table de valeurs comme primitive, et ses bases théoriques reposent sur la théorie des ensembles et sur la logique du premier ordre.

Au début des années 70, le modèle relationnel fait son apparition [Codd 70]. La recherche se passionne : impossible de nier les progrès apportés concernant la représentation et la manipulation des données par les systèmes. Dix ans passent, les spécialistes déchantent : ce top-model engendre en définitive des systèmes commerciaux bien moins performants que leurs concurrents fondés sur les modèles réseau ou hiérarchique. Deux ans plus tard et voilà que les produits relationnels peuvent prétendre relayer les "vieux" systèmes. Leurs apports sont fondamentaux : les nouvelles fonctionnalités permettent un confort d'utilisation sans précédent. Les systèmes commerciaux s'emparent des concepts de ce nouveau modèle. Celui-ci, désormais, s'impose.

Mais quels sont ces concepts, leurs avantages, leurs limites ? Certains sont déjà bien répandus. Ce sont les notions de base que nous détaillerons dans la première partie : domaine, relation, attribut ; puis la manipulation ensembliste des relations par les opérateurs de l'algèbre relationnelle (deuxième partie) sur lesquels sont construits des langages non procéduraux comme SQL (chapitre 4 XXX) ; l'importante base théorique du modèle enfin fournit des méthodes pour la conception des bases de données (chapitre 5 XXX). D'autres concepts sont toutefois moins connus qui constituent pourtant un progrès essentiel : les vues relationnelles permettent à chaque utilisateur de personnaliser sa vision des données et les contraintes d'intégrité complètent la description de l'information (chapitre 6 XXX). Les sections suivantes présentent chacun des ressorts qui font du modèle relationnel la référence obligée en matière de gestion de bases de données.

1.2 Les notions de base

Définition 3.1 (Domaine)

Un domaine *D est un ensemble de valeurs (distinctes)*

Exemple 1

booléen = {0, 1} ;

l'ensemble des entiers ;

{3, 5.7, 124} ;

('Jean', 'Paul', 'Louis', 'Arthur') ;

Définition 3.2 (Produit cartésien)

Le produit cartésien *d'un ensemble de domaines D_1, D_2,..., D_n, noté $D_1 \times D_2 \times \ldots \times D_n$, est l'ensemble de n-uplets (ou tuples) $< v_1, v_2, \ldots, v_n >$ tels que v_i appartient à D_i.*

[1] Codd introduisit aussi l'algèbre relationnelle et posa les fondations théoriques pour le modèle relationnel dans une série de papiers (Codd71, Codd72, Codd72a, Codd74). En 1979, il reçut le Turing award, la plus grande récompense de l'ACM (Association for Computing Machinery), pour son travail sur le modèle relationnel.

Exemple 2

Le produit cartésien de $D_1 = \{0, 1\}$ et de $D_2 = \{\texttt{rouge, vert, bleu}\}$ est :

rouge	0
rouge	1
vert	0
vert	1
bleu	0
bleu	1

Définition 3.3 (Relations)

Une relation *(ou encore* instance de relation*) est un sous-ensemble du produit cartésien d'une liste de domaines.*

Exemple 3

À partir de D_1 et de D_2, on construit la relation :

A_1	A_2
rouge	0
vert	1
bleu	0
bleu	1

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

- *L'ordre des lignes n'a pas d'importance* car il n'y a pas d'ordre dans un ensemble ;
- *On ne peut pas trouver deux fois la même ligne* car il n'y a pas de doublon dans un ensemble ;
- *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, cependant, les choses parfois un peu différentes.

Définition 3.4 (n-uplet ou tuples)

Un n-uplet *(élément) correspond à une ligne d'une relation.*

Une autre façon de considérer une relation à N domaines D_1, D_2,... D_N est de représenter un espace à N dimensions. Dans cet espace, chaque domaine correspond à l'une des dimension et chaque n-uplet ou tuple correspond à un point de l'espace.

Définition 3.5 (Extension)

L'ensemble des n-uplets est une extension *possible de R.*

Définition 3.6 (Attributs)

Un attribut *est le nom donné à une colonne d'un tableau représentant une relation. Un attribut est toujours associé à un domaine. Le nom d'un attribut peut apparaître dans plusieurs schémas de relations.*

Définition 3.7 (Schéma de relation)

Le schéma d'une relation *est composé de son nom suivi du nom de ses attributs et de leurs domaine :* $R(A_1 : D_1, A_2 : D_2, \ldots, A_N : D_N)$*. Lorsque le choix des domaines est évident, on simplifie l'écriture de la façon suivante :* $R(A_1, A_2, \ldots, N)$.

Définition 3.8 (Clé d'une relation)

L clé d'une relation *est le plus petit sous-ensemble des attributs qui permet d'identifier chaque ligne de manière unique.*

Définition 3.9 (Base de données)

Une base de données relationnelle *est constituée par l'ensemble des tuples de toutes les relations définies dans le schéma de la base.*

2. Bases théoriques des langages relationnels : l'algèbre relationnelle

2.1 Une manipulation ensembliste des données

Nous avons étudié au chapitre précédent les premiers modèles de SGBD. Que ce soit le segment dans le modèle hiérarchique ou l'article dans le modèle réseau, la manipulation des informations s'effectue enregistrement par enregistrement.

Dans un système relationnel, les informations ne sont pas forcément repérées individuellement ; on sait appliquer le même traitement à un ensemble d'enregistrements caractérisés, non par la liste des identifiants individuels, mais par le critère que vérifie chacun des enregistrements qui composent l'ensemble. On dit que la manipulation est ensembliste.

En particulier, pour rechercher des tuples, il suffit de préciser un critère de sélection ; le système déterminera l'ensemble des tuples satisfaisant ce critère et rendra un résultat. Les tuples de ce résultat, extraits de relations de la base, constituent eux-même une relation qu'il sera ainsi aisé de conserver, si nécessaire, pour le plus grand confort de l'utilisateur.

Par exemple, pour connaître les dates et durées des bains pris par Paul Coule ainsi que les noms des plages, on obtient le résultat suivant, qui est une relation RESULTAT à trois attributs NOMP, DATE, DUREE :

Typiquement, on peut classer les requêtes en deux grandes catégories : la mise à jour de tuples dans une relation et la recherche de tuples vérifiant une certaine condition. Eventuellement, ces deux types de requêtes peuvent être combinés.

La manipulation ensembliste est très utile en mise à jour. Elle permet, par exemple, de modifier directement la qualité de tous les nageurs ayant pris un bain sur la plage de Binic le 14 juillet 1989, sans avoir à déterminer préalablement la liste des nageurs qui présentent cette caractéristique. Cette puissance d'expression explique largement le succès du relationnel.

Les insertions/suppressions sont réalisées à l'aide de relations temporaires internes et d'opérateurs ensemblistes d'union et de différence (détails au paragraphe suivant). La combinaison de ces opérateurs et des opérateurs relationnels de sélection sur des critères de recherche facilite sensiblement les opérations de modification. C'est le cas en particulier des modifications calculées, c'est-à-dire des modifications portant sur un ensemble de tuples résultant eux-mêmes d'une sélection.

Exemple 4

- insertion ou suppression de Paul Brasse, mauvais nageur, qui a pris un bain de 2 minutes le 14/07/1989 sur la plage de sable très polluée de Binic, en Bretagne.
- recherche des noms des nageurs ayant pris des bains de plus d'une minute en Bretagne.
- suppression de tous les nageurs ayant pris, en février, des bains de plus de 2 minutes en Bretagne (hydrocution ?).

Pour manipuler ces relations, nous avons besoin d'un langage adapté dont la particularité est de savoir manipuler aisément ces tableaux de données. Ce langage constitue l'algèbre relationnelle.

2.2 Les opérateurs de l'algèbre relationnelle

L'algèbre relationnelle est le langage interne d'un SGBD relationnel. Ce langage consiste en un ensemble d'opérations qui permettent de manipuler des *relations*, considérées comme des ensembles 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. On parle de **langage assertionnel** car ils permettent de définir les données que l'on souhaite sans dire comment y accéder.

Cette algèbre se compose d'opérateurs de manipulation des relations. Ces opérateurs sont regroupés en deux familles : les opérateurs ensemblistes et les opérateurs relationnels. Chacune de ces familles contient quatre opérateurs.

2.2.1 Les opérateurs ensemblistes

L'union : $R \cup S$. L'expression : $R \cup S$, où R et S représentent deux relations de *même schéma*, crée une relation comprenant tous les tuples existant dans l'une ou l'autre des relations R et S. Cet opérateur permet de réaliser l'insertion de nouveaux tuples dans une relation.

Exemple 5

Soit la relation `Club-de-Sport` :

`Club-de-Sport`	E#	Nom	Rue	Ville de domicile
	E1	Meier	Rue Faucigny	Fribourg
	E7	Humbert	Route des Alpes	Bulle
	E19	Savoy	Avenue de la gare	Romont
	...			

Et la relation `Club-de-Photo` :

`Club-de-Photo`	E#	Membre	Rue	Ville
	E4	Brodart	Rue du tilleul	Fribourg
	E7	Humbert	Route des Alpes	Bulle
	...			

L'union `Club-de-Sport` $\cup$ `Club-de-Photo` est :

`Club-de-Sport` $\cup$ `Club-de-Photo`	E#	Nom	Rue	Ville de domicile
	E1	Meier	Rue Faucigny	Fribourg
	E7	Humbert	Route des Alpes	Bulle
	E19	Savoy	Avenue de la gare	Romont
	E4	Brodart	Rue du tilleul	Fribourg

L'intersection : $R \cap S$. L'expression : $R \cap S$, où R et S représentent deux relations de *même schéma*, crée une relation comprenant tous les tuples existant à la fois dans l'une et dans l'autre des relations R et S.

Exemple 6

L'intersection `Club-de-Sport` $\cap$ `Club-de-Photo` est :

Club-de-Sport ∩ Club-de-Photo	E#	Nom	Rue	Ville de domicile
	E7	Humbert	Route des Alpes	Bulle

La différence : $R \setminus S$. L'expression : $R \setminus S$, où R et S représentent deux relations de *même schéma*, crée une relation comprenant tous les tuples existant dans R et pas dans S. Cet opérateur permet de réaliser la suppression de tuples dans une relation.

Exemple 7

La différence Club-de-Sport \ Club-de-Photo est :

Club-de-Sport \ Club-de-Photo	E#	Nom	Rue	Ville de domicile
	E1	Meier	Rue Faucigny	Fribourg
	E19	Savoy	Avenue de la gare	Romont

On peut noter que l'intersection est une opération qui peut être dérivée de la différence entre deux tables : $R \cap S = R \setminus (R \setminus S)$.

Le produit cartésien : $R \times S$. Si R et S sont des relations de schémas respectifs (attR,1, attR,2,...,attR,N) et (attS,1, attS,2,..., attS,M) contenant respectivement $|R|$ et $|S|$ tuples, alors la relation $T = R \times S$ résultant du produit cartésien des deux relations a pour schéma (attR,1, attR,2,...,attR,N, attS,1, attS,2,..., attS,M) et contient $|R| * |S|$ tuples obtenus par concaténation des $|R|$ tuples de R et des $|S|$ tuples de S.

Exemple 8

Le produit cartésien (Club-de-Sport \ Club-de-Photo) × Club-de-Photo est :

E#	Nom	Rue	Ville de domicile	E#	Membre	Rue	Ville
E1	Meier	Rue Faucigny	Fribourg	E4	Brodart	Rue du tilleul	Fribourg
E7	Humbert	Route des Alpes	Bulle	E7	Humbert	Route des Alpes	Bulle
E19	Savoy	Avenue de la gare	Romont	E4	Brodart	Rue du tilleul	Fribourg
E4	Brodart	Rue du tilleul	Fribourg	E7	Humbert	Route des Alpes	Bulle

2.2.2 Les opérateurs relationnels

La sélection : $\sigma_F(R)$. La sélection $\sigma_F(R)$, où F exprime un prédicat sur une relation R, permet de ne conserver dans la relation R que les tuples dont les attributs vérifient une condition spécifiée par F.

La formule F contient un nombre déterminé d'attributs ou de constantes liés par des opérateurs de comparaison tels que $>$, $<$ ou $=$, ou par des opérateurs logiques AND, OR et NOT.

Exemple 9

Soit la relation Employé :

Employé	E#	Nom	Rue	Ville	Affectation
	E19	Savoy	Avenue de la gare	Romont	D6
	E1	Meier	Rue Faucigny	Fribourg	D3
	E7	Humbert	Route des Alpes	Bulle	D5
	E4	Brodart	Rue du tilleul	Fribourg	D6

La sélection $\sigma_{Ville=Fribourg}$(Employé) est :

Employé	E#	Nom	Rue	Ville	Affectation
	E1	Meier	Rue Faucigny	Fribourg	D3
	E4	Brodart	Rue du tilleul	Fribourg	D6

La sélection $\sigma_{Affectation=D6}$(Employé) est :

Employé	E#	Nom	Rue	Ville	Affectation
	E19	Savoy	Avenue de la gare	Romont	D6
	E4	Brodart	Rue du tilleul	Fribourg	D6

La sélection $\sigma_{Ville=Fribourg \texttt{AND} Affectation=D6}(\texttt{Employé})$ est :

Employé	E#	Nom	Rue	Ville	Affectation
	E4	Brodart	Rue du tilleul	Fribourg	D6

La projection : $\pi_M(R)$. La projection $\pi_M(R)$ produit, à partir d'une table R, une sous-table dont les attributs sont définis dans M.

Exemple 10

Soit la relation `Employé` :

Employé	E#	Nom	Rue	Ville	Affectation
	E19	Savoy	Avenue de la gare	Romont	D6
	E1	Meier	Rue Faucigny	Fribourg	D3
	E7	Humbert	Route des Alpes	Bulle	D5
	E4	Brodart	Rue du tilleul	Fribourg	D6

La projection $\pi_{Ville}(\texttt{Employé})$ est :

Ville
Romont
Fribourg
Bulle

La projection $\pi_{Affectation,Nom}(\texttt{Employé})$ est :

Affectation	Nom
D6	Savoy
D3	Meier
D5	Humbert
D6	Brodard

La jointure : $R \bowtie_P S$. La jointure $R \bowtie_P S$ de deux tables R et S d'après le prédicat P est une *combinaison de tous les tuples de R avec ceux de S qui satisfont le prédicat de jointure P*. Le prédicat de jointure contient un attribut de la table R et un attribut de S. Ces deux attributs sont liés par des opérateurs de comparaison, $<$, $>$ ou $=$, définissant ainsi le critère de combinaison des tables R et S.

Si le prédicat de jointure contient l'opérateur de comparaison $=$, on parle d'une *équi-jointure* (« *equi-join* », en anglais).

Exemple 11

Soit la relation `Employé` :

Employé	E#	Nom	Rue	Ville	Affectation
	E19	Savoy	Avenue de la gare	Romont	D6
	E1	Meier	Rue Faucigny	Fribourg	D3
	E7	Humbert	Route des Alpes	Bulle	D5
	E4	Brodart	Rue du tilleul	Fribourg	D6

Et soit la relation `Département` :

D#	Description
D3	Informatique
D5	Personnel
D6	Finances

Alors `Employé` $\bowtie_{Affectation=D\#}$ `Département` :

E#	Nom	Rue	Ville	Affectation	D#	Description
E19	Savoy	Avenue de la gare	Romont	D6	D6	Finances
E1	Meier	Rue Faucigny	Fribourg	D3	D3	Informatique
E7	Humbert	Route des Alpes	Bulle	D5	D5	Personnel
E4	Brodart	Rue du tilleul	Fribourg	D6	D6	Finances

La division : $R(Z) \div S(X)$. Z et X sont respectivement les ensembles d'attributs associés aux relations R et S, et $X \subseteq Z$. Soit $Y = Z \setminus X$ (et donc $Z = X \cup Y$) ; c'est-à-dire soit Y l'ensemble des attributs de R qui ne sont pas dans S. Le résultat de la division est une relation $T(Y)$ contenant l'ensemble des tuples t tels que t_R apparaît dans R avec $t_R[Y] = t$ et tel que $t_R[X] = t_S$ pour tous les tuples t_S dans S.

En d'autres termes, pour qu'un tuple t apparaisse dans le résultat T de la division, les valeurs dans t doivent apparaître en combinaison avec *tous* les tuples de S.

Remarque : le produit cartésien $T \times S$ doit être contenu dans la table R.

Exemple 12

Soit `R` la table d'attribution des projets aux employés :

R	E#	Proj#
	E1	P1
	E1	P2
	E1	P4
	E2	P1
	E2	P2
	E4	P2
	E4	P4

Et soit `S` la table de combinaison de projets :

S	Proj#
	P2
	P4

On trouve tous les employés participants aux projets P2 et P4 par la division `R'` = `R` ÷ `S` :

R'	E#
	E1
	E4

Il est possible d'exprimer l'opérateur de division en fonction des opérateurs de projection, de différence et du produit cartésien. Pour cette raison, il figure parmi les opérateurs substituables de l'algèbre relationnelle qui comprennent aussi les opérateurs d'intersection et de jointure.

En résumé, cinq de ces huit opérateurs forment les **opérateurs de base** (ce sont l'*union*, la *différence*, le *produit cartésien*, la *restriction* et la *projection*) tandis que les trois autres, appelés **opérateurs dérivés**, s'obtiennent plus ou moins facilement par combinaison des opérateurs de base :

- $R \cap S = R \setminus (R \setminus S) = (R \cup S) \setminus ((R \setminus S) \cup (S \setminus R))$
- $R \bowtie_P S = \sigma_P(R \times S)$
- $R \div S = \pi_Y(R) \setminus \pi_Y((S \times \pi_Y(R) \setminus R)$

Les cinq opérateurs de base permettent de répondre à toutes les questions que l'on peut poser avec la logique du premier ordre (c'est-à-dire sans les fonctions) : on dit que l'algèbre relationnelle est *complète*.

En réalité, nous n'utiliserons dans nos requêtes que les opérateurs les plus maniables : ce sont l'*union* et la *différence* pour l'insertion et la suppression de tuples dans la base et la restriction, la *projection* et la *jointure* pour la recherche sélective de tuples.

Les opérateurs de l'algèbre relationnelle ne présentent pas seulement un intérêt sur le plan théorique. Leur portée pratique est aussi importante. par exemple, nous en aurons besoin pour optimiser les requêtes au niveau du langage des systèmes de bases de données relationnelles (voir section 2). En outre, ils trouvent leur application dans la conception des ordinateurs de base de données : les opérateurs de l'algèbre relationnelle et leurs formes dérivées n'y sont pas mis en œuvre sous forme logicielle, mais implantées directement dans des composants matériels de l'ordinateur.

Exercices

D'abord les exercices simples de [Buche] pp.87-88, puis 91-94

Exemple 13

On suppose que l'on a le schéma de bases de données suivant :

Emprunt(Personne, Livre, DateEmprunt, DateRetourPrevue, DateRetourEffective)

Retard(Personne), Livre, DateEmprunt, PénélitéRetard)

Écrire en algèbre relationnelle les requêtes suivantes :

1. Quelles sont les personnes ayant emprunté tous les livres ?

Réponse : $\pi_{Personne,Livre}(Emprunt) \div \pi_{Livre}(Emprunt)$

2. Quelles sont les personnes ayant toujours rendu en retard les livres qu'elles ont empruntés ?

Réponse : $\pi_{Personne}(Emprunt) \setminus \pi_{Personne}[\pi_{Personne,Livre,DateEmprunt}(Emprunt) \setminus \pi_{Personne,Livre,DateEmprunt}(Retard)]$

Exemple 14

On suppose que l'on a le schéma de bases de données suivant :

Location(Personne, Voiture, DateDébut, DateFin)

Accident(Personne, Voiture, Date)

Écrire en algèbre relationnelle les requêtes suivantes :

1. Quelles sont les personnes ayant moué toutes les voitures ?

Réponse : $\pi_{Personne,Voiture}(Location) \div \pi_{Voiture}(Location)$

2. Quelles sont les personnes ayant toujours eu des accidents avec les voitures qu'elles ont louées ?

Réponse : $\pi_{Personne}(Location) \setminus \pi_{Personne}[\pi_{Personne,Voiture}(Location) \setminus \pi_{Personne,Voiture}(Accident)]$

Poly Rigaux, pp.62-64

Ensuite les exos (corrigés) de [Elmasri et Navathe], pp.230-232 (xerox pp.204-205)

Puis l'exo7.18 de [Elmasri et Navathe], pp.235 (xerox pp.204-205)

2.3 Expressions de requêtes avec l'algèbre

2.3.1 Sélection généralisée

2.3.2 Requêtes conjonctives

2.3.3 Requêtes avec $\cup$ et $-$

2.4 Conclusions

3. Bien concevoir une base de données

Nous étudions dans cette section comment bien concevoir une BD. Pour ce faire, à partir d'un même ensemble de connaissances ayant trait aux plages, nous proposons deux choix d'organisation des informations sous forme relationnelle. Nous étudions les qualités et les défauts de ces différents choix avant de présenter les règles de "bonne" conception d'une BD relationnelle.

Premier choix : BAINS (NN, NOM, PRENOM, QUALITE, DATE, DUREE, NP, NOMP, TYPE, REGION, POLLUTION)

Deuxième choix :
NAGEUR (NN, NOM, PRENOM, QUALITE)
PLAGE (NP, NOMP, TYPE, REGION, POLLUTION)
BAIGNADE (NN, NP, DATE, DUREE)

NN et NP sont des numéros permettant de distinguer respectivement les nageurs et les plages. NN est l'équivalent du numéro de sécurité sociale.

Nous constatons sur cet exemple l'existence de plusieurs façons de structurer un même ensemble d'infor-

mations. Si nous privilégions instinctivement l'une des deux solutions proposées, c'est qu'elle correspond davantage à notre perception du monde réel, dans laquelle nous distinguons naturellement certaines entités : les personnes, les plages, etc.

L'étape de conception est primordiale pour le bon fonctionnement d'un SGBD. Elle fait partie des quelques facteurs qui peuvent entraîner des incohérences dans les réponses et une diminution inacceptable des performances du système ; c'est pourquoi il est indispensable d'y attacher une attention toute particulière.

Exemple 15

Soit `R` la relation des nageurs :

`Nageurs`	N#	Nom	Prénom	Qualité
	100	Dupont	Jean	Mauvais
	103	Marin	Pierre	Excellente
	114	Palmier	Jean	Bonne

Soit `S` la table des plages :

`Plages`	NP#	NomP	Type	Région	Pollution
	110	Trégastel	Sable	Bretagne	Absente
	119	Nice	Galets	Côte d'Azur	Importante
	107	Oléron	Sable	Atlantique	Moyenne
	118	Binik	Sable	Bretagne	Importante

Et soit `S` la table des baignades :

`Baignades`	N#	NP#	Date	Durée
	103	118	14/07/89	2
	103	118	15/07/89	10
	114	119	12/07/89	120

Une solution "instinctive" n'est pas suffisante pour concevoir le schéma d'une base importante. Il est donc nécessaire d'isoler les critères de décision et de formaliser des méthodes de conception des bases de données. Tel est l'objet de cette section.

Les problèmes les plus courants rencontrés dans des bases de données mal conçues peuvent être regroupés selon les critères suivants :

Redondance des données. Certains choix de conception entraînent une répétition des données lors de leur insertion dans la base. Cette redondance est souvent la cause d'anomalies provenant de la complexité des insertions.

C'est, par exemple, le cas de la première organisation proposée : dès qu'une personne prend un nouveau bain, on doit non seulement répéter son numéro qui, par hypothèse, suffit à le déterminer, mais aussi toutes les informations liées à ce numéro (son nom, son prénom, sa qualité). Au contraire, dans le deuxième choix, seul le numéro indispensable à la distinction d'un nageur est répété. La situation est identique pour les plages.

Incohérence en modification. La redondance de l'information entraîne également des risques en cas de modification d'une donnée répétée en différents endroits : on oublie fréquemment de modifier toutes ses occurrences (en général par simple ignorance des différentes places où figure l'information).

Par exemple, dans la première organisation proposée, si une personne change de nom (cas fréquent lors de mariages), il faut changer ce nom dans tous les tuples où apparaissent ses coordonnées. Dans la deuxième organisation, un seul tuple est modifié.

Anomalie d'insertion. Une mauvaise conception peut parfois empêcher l'insertion d'un tuple, faute de connaître la valeur de tous les attributs de la relation. Pour remédier à ce problème, certains SGBD implantent une valeur non typée qui signifie que la valeur d'un attribut d'un tuple est inconnue ou indéterminée. Cette valeur (appelée usuellement NULL) indique réellement une valeur inconnue et non une chaîne de caractères vide ou un entier égal à zéro (analogie avec un pointeur égal à NIL en Pascal).

Dans le premier schéma proposé, insérer une nouvelle plage où personne ne s'est jamais baigné est aussi impossible.

Anomalie de suppression. Enfin, une mauvaise conception peut entraîner, lors de la suppression d'une information, la suppression d'autres informations, sémantiquement distinctes, mais regroupées au sein d'un même schéma. C'est ce qui se produit dans notre premier exemple, la suppression d'une plage entraîne automatiquement la suppression de tous les nageurs ne s'étant baignés que sur cette plage.

De nombreux travaux ont permis de mettre au point une théorie de conception d'une base de données relationnelle : la théorie de la normalisation, que nous allons maintenant développer.

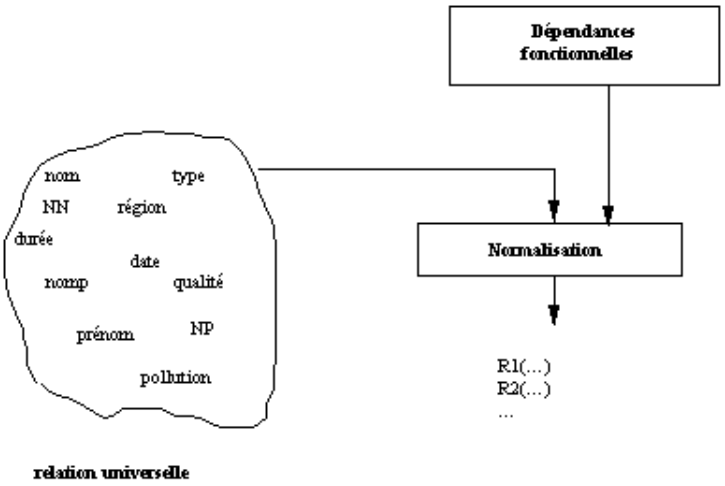

FIG. 3.1: La normalisation d'une base de données relationnelle.

3.1 Les dépendances fonctionnelles

Cette théorie est basée sur les « *dépendances fonctionnelles* » (DF). Les dépendances fonctionnelles traduisent des contraintes sur les données (par exemple, on décide que deux individus différents peuvent avoir même nom et prénom mais jamais le même numéro NN). Ces contraintes sont représentatives d'une perception de la réalité et imposent des limites à la base.

Les dépendances fonctionnelles et des propriétés particulières, définissent une suite de *formes normales* (FN). Elles permettent de décomposer l'ensemble des informations en diverses relations. Chaque nouvelle forme normale marque une étape supplémentaire de progression vers des relations présentant de moins en moins de redondance. Ces étapes traduisent une compréhension de plus en plus fine de la réalité.

Chacune de ces formes normales peut être obtenue au moyen d'algorithmes de décomposition. Le point de départ de ces algorithmes est la *relation universelle*, c'est-à-dire la relation qui regroupe toutes les informations à stocker (dans notre exemple, le premier schéma représente cette relation universelle) ; le but est d'obtenir, en sortie, une représentation canonique des données présentant un minimum de redondance à l'intérieur de chaque relation et un maximum d'indépendance entre les différentes relations .

3.1.1 Les dépendances fonctionnelles

Définition 3.10 (Dépendance fonctionnelle)

On dit qu'un attribut B dépend fonctionnellement d'un attribut A si, étant donné une valeur de A, il lui correspond une unique valeur de B (quel que soit l'instant t considéré).

On note cette dépendance par : $A \rightarrow B$.

Plus généralement, soit $R(A_1, A_2, \ldots, A_n)$ un schéma de relation, et X et Y deux sous-ensembles de $\{A_1, A_2, \ldots, A_n\}$, on dit que Y dépend fonctionnellement de X ssi à toute valeur de X correspond une valeur unique de Y.

Exemple 16

La dépendance fonctionnelle NN → NOM signifie qu'à un numéro est associé un nom unique (c'est, par exemple, le cas du numéro de sécurité sociale). Remarquons qu'une dépendance fonctionnelle n'est généralement pas symétrique, c'est-à-dire que NN → NOM n'interdit pas que deux personnes distinctes (correspondant à deux NN différents) portent le même nom.

Une dépendance fonctionnelle est une propriété définie sur tous les tuples d'une relation et pas seulement sur un tuple particulier. Elle traduit une certaine perception de la réalité (par exemple, deux personnes distinctes peuvent porter le même nom). Elle correspond à une contrainte qui doit être vérifiée en permanence.

Les dépendances fonctionnelles sont parties intégrantes du schéma d'une BD. Elles doivent donc théoriquement être déclarées par l'administrateur et contrôlées par le SGBD.

Exemple 17

Nous définissons les propriétés vérifiées par notre base de baignades. Deux personnes distinctes peuvent, par exemple, porter le même nom, le même prénom, et avoir la même qualité de nage. Deux numéros de nageur différent les distinguent l'une de l'autre. Les dépendances fonctionnelles que nous venons de décrire sont donc NN → NOM, NN → PRENOM, NN → QUALITE Nous pouvons supposer également que deux plages distinctes ont toujours deux numéros différent ; ce qui implique : NP → NOMP, NP → REGION, que la pollution et le type sont propres à une plage : NP → POLLUTION, NP → TYPE, et que deux plages d'une même région ne peuvent porter le même nom : (NOMP, REGION) → NP

3.1.2 Propriétés des dépendances fonctionnelles

Les dépendances fonctionnelles obéissent à certaines propriétés connues sous le nom d'*axiomes d'Armstrong.*

Réflexivité : $Y \subseteq X \quad \Rightarrow \quad X \rightarrow Y$

Augmentation : $X \rightarrow Y \quad \Rightarrow \quad XZ \rightarrow YZ$

Transitivité : $X \rightarrow Y$ et $Y \rightarrow Z \quad \Rightarrow \quad X \rightarrow Z$

D'autres propriétés se déduisent de ces axiomes :

Union : $X \rightarrow Y$ et $X \rightarrow Z \quad \Rightarrow \quad X \rightarrow YZ$

Pseudo-transitivité : $X \rightarrow Y$ et $YW \rightarrow Z \quad \Rightarrow \quad XW \rightarrow Z$

Décomposition : $X \rightarrow Y$ et $Z \subseteq Y \quad \Rightarrow \quad X \rightarrow Z$

L'intérêt de ces axiomes et des propriétés déduites est de pouvoir construire, à partir d'un premier ensemble de dépendances fonctionnelles, l'ensemble de toutes les dépendances fonctionnelles qu'elles génèrent. L'ensemble des dépendances de départ est alors appelé *dépendances fonctionnelles élémentaires.*

3.1.3 Dépendance fonctionnelle totale

Dans le cas des clés composées, nous devons compléter la notion de dépendance fonctionnelle par celle de la dépendance fonctionnelle totale.

Définition 3.11 (Dépendance fonctionnelle totale)

Une dépendance fonctionnelle $X \rightarrow A$ *est* totale *si*

- *A n'est pas inclus dans X ;*
- *il n'existe pas X' inclus dans X tel que X' → A.*

La dépendance fonctionnelle totale exprime donc le fait que la totalité de la clé composée détermine de manière unique les attributs non clés.

Cette notion de dépendance fonctionnelle totale est primordiale car elle permet de construire une sorte de famille génératrice minimale (appelée *couverture minimale*) des dépendances fonctionnelles utiles pour structurer la base.

Exemple 18

Deux plages d'une même région ne peuvent pas porter le même nom (contrairement à deux plages de régions différentes) ; le degré de pollution d'une plage dépend exclusivement de la plage et non de la région. On a alors (NOMP, REGION) → POLLUTION, mais on n'a aucunement : NOMP → POLLUTION ni REGION → POLLUTION donc (NOMP, REGION) → POLLUTION est une dépendance fonctionnelle élémentaire.

3.1.4 Graphe des dépendances fonctionnelles

C'est une représentation graphique permettant de visualiser aisément toutes les dépendances fonctionnelles et d'isoler les principales (i.e. les DF élémentaires).

Exemple 19

Toutes les dépendances fonctionnelles citées précédemment peuvent être représentées comme sur la figure 3.2.

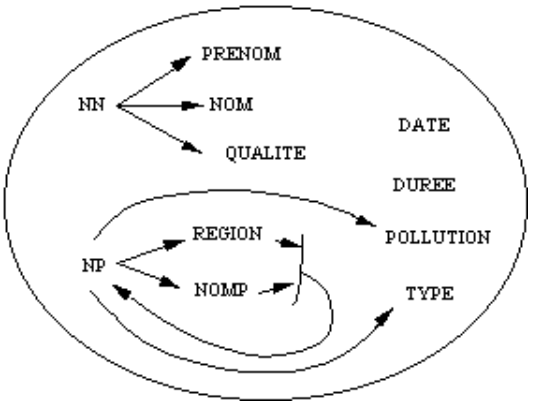

FIG. 3.2: Graphe de dépendances fonctionnelles.

3.1.5 Fermeture transitive

Définition 3.12 (Fermeture transitive)

La fermeture transitive F^+ *d'un ensemble F de dépendances fonctionnelles est ce même ensemble enrichi de toutes les dépendances fonctionnelles déduites par transitivité.*

On dira alors que F est *équivalente* à F' ssi $F^+ = (F')^+$

Exemple 20

De l'exemple précédent , on déduit par transitivité deux nouvelles dépendances fonctionnelles : (NOMP, REGION) → TYPE et (NOMP, REGION) → POLLUTION qui enrichissent le graphe comme sur la figure 3.3.

3.1.6 Couverture minimale

Définition 3.13 (Couverture minimale)

La couverture minimale*, notée $Min(F)$ d'un ensemble F de dépendances fonctionnelles est un sous*

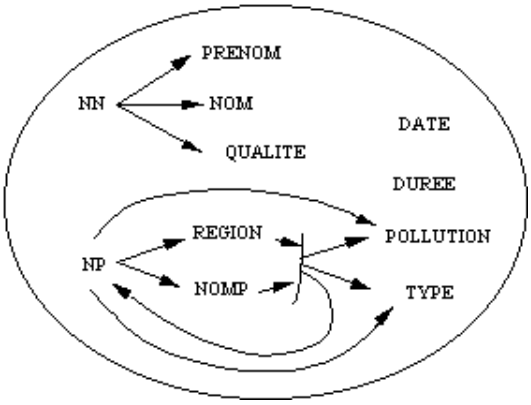

FIG. 3.3: Graphe de dépendances fonctionnelles enrichi.

ensemble minimum de dépendances fonctionnelles élémentaires permettant de générer toutes les autres.

On a les propriétés suivantes :
- Minimalité : Aucune DF de $Min(F)$ n'est redondante, i.e. $(Min(F) - f)^+ \neq (Min(F))^+$.
- Exhaustivité : $(Min(F))^+ = F^+$

Exemple 21

L'ensemble des dépendances fonctionnelles suivant : (NN → NOM, NN → PRENOM, NN → QUALITE, NP → NOMP, NP → REGION, (NOMP, REGION) → POLLUTION, (NOMP, REGION) → NP) est une couverture minimale de l'ensemble des dépendances fonctionnelles.

Théorème 3.1

Tout ensemble de dépendances fonctionnelles admet une couverture minimale, en général non unique.

Exemple 22

L'ensemble des dépendances fonctionnelles suivant : (NN → NOM, NN → PRENOM, NN → QUALITE, NP → NOMP, NP → REGION, NP → POLLUTION, (NOMP, REGION) → NP) est une autre couverture minimale de l'ensemble des dépendances fonctionnelles.

La recherche de la couverture minimale d'un ensemble de dépendances fonctionnelles est un élément essentiel du processus de normalisation, dont le but est de décomposer une relation en plusieurs relations plus petites.

3.1.7 Clé d'une relation

Définition 3.14 (Clé d'une relation)

Soit $R(A_1, A_2, ..., A_N)$ un schéma de relation, soit F+ l'ensemble des dépendances fonctionnelles associées à R, soit X un sous-ensemble de $A_1, A_2, \ldots, A_N$, on dit que X est une clé de R *si et seulement si*
- $X \rightarrow A_1, A_2, \ldots, A_N$;
- *il n'existe pas de sous ensemble Y inclus dans X tel que* $Y \rightarrow A_1, A_2, ..., A_N$.

Une clé d'une relation est un ensemble minimum d'attributs qui détermine tous les autres.

Exemple 23

NN est clé de la relation PERSONNE (NN, NOM, PRENOM, QUALITE) ;
(NOMP, REGION) est clé de la relation (NP, NOMP, REGION, POLLUTION, TYPE).

Plusieurs clés peuvent être candidates pour une même relation.

Exemple 24

NP et (NOMP, REGION) sont deux clés candidates à la relation (NP, NOMP, REGION, POLLUTION, TYPE).

Dans l'écriture des schémas de relations, on indique les clés en soulignant les attributs constitutifs, ou en les mettant en gras.

Exemple 25

PLAGE (**NP**, NOMP, REGION, POLLUTION, TYPE)

3.2 Décomposition en formes normales

L'étude des dépendances fonctionnelles et de certaines de leurs propriétés permet d'aborder la sémantique des schémas de relation. On suppose qu'un ensemble de dépendances fonctionnelles et de clés primaires est donné pour chaque relation. Ces informations utilisées avec des conditions pour des formes dites normales permettent de prescrire une procédure de normalisation.

C'est Codd ([**?**]) qui, le premier, a proposé de soumettre un schéma de relation à une séquence de tests pour certifier qu'il vérifie une certaine forme normale. Initialement, Codd proposa trois formes normales : les première, deuxième et troisième formes normales (1NF, 2NF et 3NF). Une définition plus forte de la 3NF, appelée forme normale de Boyce-Codd (BCNF), fut proposée plus tard par Boyce et Codd. Toutes ces formes normales sont fondées sur l'exploitation de dépendances fonctionnelles entre des attributs d'une relation. Une quatrième et une cinquième formes normales (4NF et 5NF) furent ensuite définies, fondées respectivement sur le concept de dépendances multivaluées et sur celui de dépendance de jointure.

Le processus de normalisation peut ainsi être vu comme comme une méthode d'analyse des relations afin (1) d'en *réduire la redondance* et (2) de *minimiser les anomalies d'insertion, de suppression et de mise à jour*.

Les formes normales imposent des critères de plus en plus restrictifs aux tables normalisées au fur et à mesure du passage d'une forme normale à la suivante, comme l'évoque la figure 3.4.

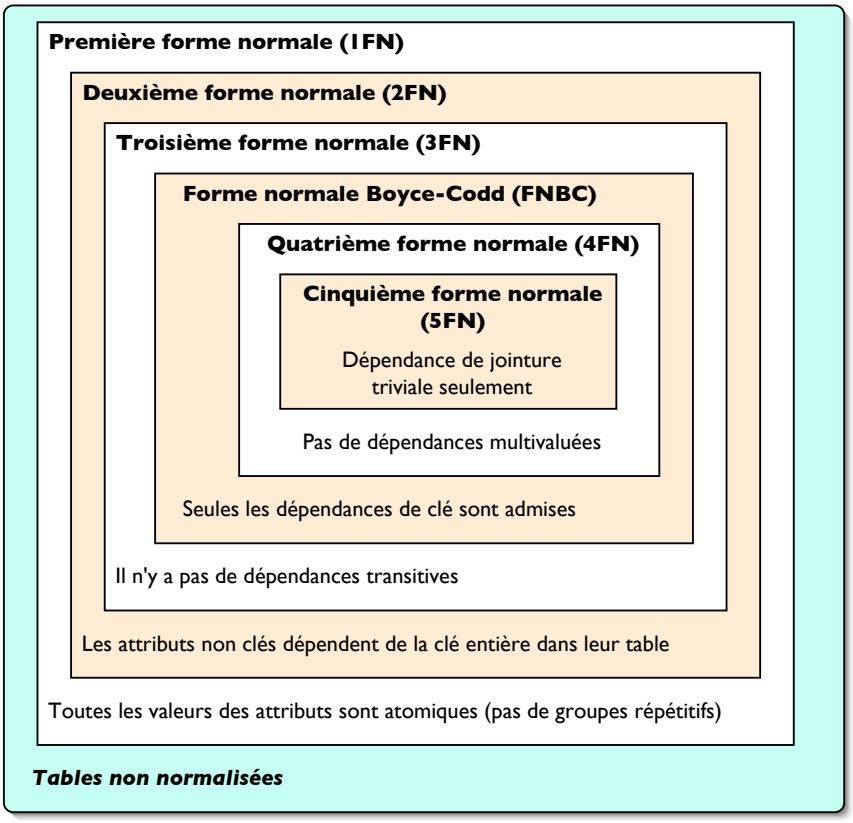

FIG. 3.4: Vue d'ensemble des formes normales et de leurs propriétés.

Les formes normales n'ont pas toutes la même importance, et, dans la pratique, on se limite généralement aux trois premières formes normales, car les dépendances multivaluées et les dépendances de jointure se présentent moins fréquemment.

3.2.1 Décomposition des relations

La théorie de la normalisation repose sur un principe de décomposition des relations.

Définition 3.15 (Décomposition d'une relation)

La décomposition d'un schéma de relation $R(A_1, A_2, ..., A_N)$ *est son remplacement par une collection de schémas de relations* $(R_1, R_2, ..., R_i)$ *telle que :* $SCHEMA(R) = SCHEMA(R_1) \cup SCHEMA(R_2) \cup \ldots \cup SCHEMA(R_i)$

Définition 3.16 (Décomposition sans perte)

Une décomposition d'une relation R *en* N *relations* $R_1, R_2, \ldots, R_N$ *est sans perte si et seulement si, pour toute extension de* R*, on a :* $R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_N$

Théorème 3.2 (Décomposition sans perte)

Une décomposition en deux relations est sans perte si l'attribut de jointure de la recomposition est clé d'une au moins des deux relations.

Définition 3.17 (Décomposition préservant les dépendances fonctionnelles)

Une décomposition $(R_1, R_2, \ldots, R_N)$ *de* R *préserve les dépendances fonctionnelles si la fermeture des dépendances fonctionnelles de* R *est la même que celle de l'union des dépendances fonctionnelles des relations* $R_1, R_2, \ldots, R_N$*.*

3.2.2 Première forme normale (1FN)

Définition 3.18 (Première forme normale)

Une table satisfait à la première forme normale si les domaines de ses attributs sont constitués de valeurs atomiques.

3.2.3 Deuxième forme normale (2FN)

Définition 3.19 (Deuxième forme normale)

Une table satisfait à la deuxième forme normale si elle est en première forme normale et s'il existe une dépendance fonctionnelle totale *reliant la clé à chaque attribut non clé.*

3.2.4 Troisième forme normale (3FN)

Définition 3.20 (Troisième forme normale)

*Une table satisfait à la troisième forme normale si elle est en deuxième forme normale et qu'*aucun attribut non clé ne dépend d'une clé quelconque par transitivité.

3.2.5 La forme normale de Boyce-Codd (FNBC) : les dépendances multivaluées

Les deuxième et troisièmes formes normales nous ont permis d'éliminer les redondances parmi les attributs non clés. Cependant, la détection des informations redondantes ne doit pas se limiter aux attributs non clés, car les clés composées peuvent aussi être redondantes.

3.2.6 Quatrième forme normale (4FN)

La présence des dépendances multivaluées dans une table entraîne des redondances et des anomalies. Pour les supprimer, il faut considérer une autre forme normale, dite quatrième forme normale.

Définition 3.21 (Quatrième forme normale)

Une table en quatrième forme normale ne doit pas contenir en même temps deux dépendances multivaluées par paire d'attributs.

3.2.7 Cinquième forme normale (5FN)

Il n'est pas du tout évident de réussir à décomposer une table en sous-tables sans perte d'information. C'est pourquoi des critères ont été définis afin de garantir une décomposition de tables sans perte d'information. Plus précisément, les informations originelles doivent se retrouver dans les nouvelles tables obtenues par décomposition (concept de dépendance de jointure).

La *cinquième forme normale* (5FN), appelée aussi *forme normale de projection-jointure*, indique les circonstances dans lesquelles une table peut être décomposée et, le cas échéant, reconstruite, sans problème. La décomposition s'effectue à l'aide de l'opérateur de projection, et la reconstruction à l'aide de l'opérateur de jointure.

3.3 Les contraintes d'intégrité structurelles

Les contraintes d'intégrité structurelles établissent des règles qui doivent s'appliquer à un schéma de base de données pour assurer sa cohérence. Dans le contexte des bases de données relationnelles, les contraintes d'intégrité structurelles sont classées en trois catégories :

- *Contrainte d'unicité* : chaque table possède une clé d'identification (un attribut ou une combinaison d'attributs) qui sert à différencier les tuples dans la table de manière unique ;
- *Contrainte de domaine* : un attribut dans une table ne peut prendre que des valeurs appartenant à un domaine de valeurs prédéfinies ;
- *Contrainte d'intégrité référentielle* : chaque valeur d'une clé étrangère doit exister comme valeur de la clé d'identification correspondante dans la table référencée.

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

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 et 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 (règles de normalisation).

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 » (e.g. le modèle Entité / Association), puis en transcrivant le schéma conceptuel obtenu en schéma relationnel.

3.4.1 Règles générales

3.4.2 Retour sur le choix des identifiants

3.4.3 Dénormalisation du modèle logique

4. Le langage relationnel SQL

4.1 Les langages relationnels complets

Les langages relationnels complets sont des langages qui permettent au moins d'exploiter les possibilités de traitement découlant de l'algèbre relationnelle, ou d'appliquer le calcul dit relationnel[2] qui est aussi

[2] Qui est fondé sur la logique des prédicats.

puissant que l'algèbre relationnelle.

Définition 3.22 (Langage relationnel complet)

Un langage d'interrogation de bases de données est relationnel complet *au sens de l'algèbre relationnelle lorsqu'il permet au moins l'emploi des trois opérateurs ensemblistes : union, différence et produit cartésien, et des deux opérateurs relationnels : la projection et la sélection.*

Cependant, pour qu'ils soient utiles en pratique, une extension des langages relationnels complets s'impose en y incluant les fonctionnalités suivantes :

- Le langage doit permettre de créer des tables, d'effectuer des opérations d'insertion, de modification et de suppression.
- Le langage doit inclure des fonctions d'agrégation, permettant de calculer, par exemple, la somme, le maximum, le minimum ou la moyenne des valeurs dans la colonne d'une table.
- Le langage doit permettre de formater et de présenter les tables d'après différents critères, tels que la séquence de tri ou les ruptures de séquences par groupes.
- Les langages de base de données relationnelles doivent absolument comporter des éléments pour gérer les autorisations d'accès et pour assurer la protection des bases de données.
- Les langages de bases de données relationnelles doivent prendre en considération l'environnement multiutilisateur et disposer de commandes pour garantir la sécurité des données.
- Il est avantageux de disposer d'un langage de base de données relationnelles qui supporte des expressions ou des calculs arithmétiques.

4.2 Le langage SQL : historique

SQL (Structured Query Language) est l'interface de communication avec les SGBD relationnels.

Hier norme de fait, SQL est devenu une norme de droit quand l'International Organization for Standardization l'a pris en compte en 86 [ISO 86] (SQL1 : normalisation ISO/CEI 9075 en 1989 ; SQL2 : normalisation ISO/CEI 9075 NS739 en 1991 ; SQL3 : normalisation ISO/CEI 9075 : 1999 ; SQL2003 : normalisation ISO/CEI 9075 : 2003). SQL est arrivé à maturité. Sa version normalisée par l'ISO fige ses différents aspects : définition des concepts, du vocabulaire, du langage de définition de données (déclaration du schéma, des vues, du contrôle des autorisations) et du langage de manipulation sont désormais stabilisés. Les procédures d'intégration des appels SQL dans un langage hôte (de type Cobol, Pascal, C, Fortran...), regroupées sous le nom de "Embedded SQL", souvent noté ESQL, sont également définies. Presque tous les constructeurs proposent le langage SQL. Plus de 70 produits du marché le fournissent, sur des machines qui vont du PC au plus importants mainframes. Celui-ci étend même sa percée vers des produits micros qui, sans devenir de véritables SGBD relationnels complets, intègrent déjà la partie de SQL qui concerne la manipulation des données.

SQL est un facteur important de promotion du modèle relationnel. Il apporte une vision unifiée et une large communauté de concepts et de vocabulaire. Reposant sur une base théorique solide (la logique des prédicats), il concerne à la fois les administrateurs, les programmeurs d'applications, et les utilisateurs finals pour la définition et la manipulation des données.

La norme SQL comporte trois parties distinctes concernant :

- la **manipulation de l'information** ; le (DML : Data Manipulation Language) concerne l'ensemble des requêtes de recherche et de mise à jour des informations : selection, insertion, modification et suppression. Il est essentiellement destiné à l'utilisateur final ;
- la **définition des structures de données** ; le DDL (Data Definition Language) contient l'ensemble des commandes de création, de suppression, de modification des relations. Elle intègre la créations, la suppressions de chemins d'accès (index) et les commandes concernant la gestion des droits et des autorisations. Elle est largement destinée à l'administrateur ;
- l'**accès aux informations stockées dans le SGBD depuis un langage de programmation** ; l'Embedded SQL définit l'ensemble des interfaces, l'utilisation des ordres SQL dans un langage de programmation, ainsi que l'ensemble des codes d'erreur renvoyés par le SGBD au programme d'application défini. L'Embedded SQL est destiné au programmeur d'applications.

SQL est un langage non procédural : on exprime ce que l'on veut obtenir, non comment l'obtenir.

SQL permet :

- de définir le schéma de la base de données (DDL)
- de charger les tables relationnelles (DML)
- de manipuler les données stockées dans la base (DML)
- de gérer la base de données (DDL : sécurité, organisation physique).

La liste des avantages et inconvénients de la norme SQL telle qu'elle se présente aujourd'hui peut être rapidement dressée : à son actif, citons sa simplicité, sa souplesse et ses fonctionnalités étendues ; à son passif, un manque de rigueur (il autorise - et encourage - certaines formulations procédurales), une utilisation lourde (mot clés, pas d'utilisation de moyens modernes d'interface homme/machine), le fait qu'il constitue, comme toute norme, un frein à la créativité des concepteurs de nouveaux systèmes. Rappelons enfin qu'il s'agit d'un langage inventé par IBM, argument stratégique délicat à placer dans les qualités ou les défauts du langage.

La normalisation de SQL, dont la première version a été achevée en 86, conduit également à se poser la question "à qui peut servir la normalisation ?". En effet, si l'on examine les différentes parties de la norme, on constate que l'administrateur doit disposer d'outils moins primitifs et plus spécifiques pour gérer convenablement son SGBD ; l'utilisateur final a bien du mal à utiliser directement SQL en interactif : il utilise des interfaces spécifiques (menus, écrans...) plus conviviales ; enfin le programmeur d'applications est aujourd'hui largement sollicité par les Langages de quatrième Génération (L4G) qui, sans être normalisés, sont plus pratiques et procurent les mêmes services de base que du Embedded SQL. Certaines mauvaises langues en concluent donc que la normalisation ne sert qu'aux normalisateurs... L'apport de la normalisation du SQL est cependant indiscutable pour les utilisateurs désirant garantir la portabilité de leurs applications d'un système SQL sur un autre. C'est l'interface standard de tout véritable SGBD relationnel. Il précise, d'autre part, un niveau d'interface précis lorsqu'on envisage des systèmes répartis et hétérogènes.

Les éléments du langage SQL donnés dans la suite se réfèrent principalement à la norme SQL2, et parfois à celle de SQL2003 (par exemple pour les *triggers*).

Verbes du DML (Data Manipulation Language) :

`select`	: Rechercher une donnée
`insert`	: Insérer une ligne (ou plusieurs) dans une table
`update`	: modifier une ligne (ou plusieurs) dans une table
`delete`	: supprimer une ligne (ou plusieurs) dans une table

Verbes du DDL (Data Description Language) :

`create`	: Création d'un objet
`alter`	: Modification d'un objet
`drop`	: Suppression d'un objet
`grant`	: Accorder une autorisation sur un objet
`revoke`	: Supprimer une autorisation sur un objet

4.3 Le DML : Data Manipulation Language en SQL

Les requêtes en SQL se font à l'aide d'une seule construction, le `SELECT`, dont la structure est la suivante :

```
SELECT      nom_table.nom_colonne*
FROM        nom_table*
[WHERE      conditions_de_sélection_sur_lignes*]
[GROUP      by nom_colonne_de_regroupement*]
[HAVING     conditions_de_sélection_sur_groupe*]
[ORDER BY   nom_colonne_tri*] ;
```

Notes : *= plusieurs occurrences possibles, [] = optionnel

Exemple 26

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

4.3.1 Requêtes simples SQL

Exemple 27

Requête 1 : chercher la date de naissance et l'adresse des employés dont le nom est 'John B. Smith'.

```
SELECT   bdate, address
FROM     EMPLOYEE
WHERE    fname = 'John' AND minint = 'B' AND lname = 'Smith';
```

Cette requête n'implique que la relation `EMPLOYEE` qui apparaît dans la clause `SELECT`. La requête sélectionne les tuples de la table `EMPLOYEE` qui satisfont aux conditions spécifiées dans la clause `WHERE`, puis *projette* le résultat sur les attributs `bdate` et `address` apparaissant dans la clause `SELECT`. Cette requête est similaire à l'expression suivante en algèble relationnelle (sauf que les doublons peuvent exister en SQL contrairement à ce qui se passe en algèbre relationnelle) :

$$\pi_{\texttt{bdate,address}}(\sigma_{\texttt{fname='John' AND minint='B' AND lname='Smith'}}(\texttt{EMPLOYEE}))$$

Ainsi, une requête SQL simple avec un seul nom de relation dans la clause `FROM` est similaire à une paire `select-project` d'opérations en algèbre relationnelle. La clause `SELECT` en SQL spécifie les *attributs de projection*, et la clause `WHERE` spécifie les *conditions de sélection*. La seule différence étant que l'on peut obtenir des tuples dupliqués en SQL parce que la contrainte ensembliste n'est pas assurée.

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

```
SELECT   DISTINCT  libelle
FROM     Activité
```

Remarque

L'élimination des doublons peut être une opération coûteuse.

Une sélection est une restriction suivie d'une projection. Une restriction est une combinaison booléenne (`or, and, not`) de conditions élémentaires portant sur les colonnes d'une table. Les prédicats de restriction permettent la comparaison d'une valeur portée par une colonne à une valeur constante. Ils peuvent s'exprimer de différentes manières :

- à l'aide des opérateurs $=, <>, <, >, <=, >=$ (cf Q1, Q2)
- à l'aide du prédicat d'intervalle `BETWEEN` qui teste si la valeur de l'attribut est compris entre deux valeurs constantes. L'attribut doit porter une valeur numérique ou de type date (cf Q3).
- à l'aide du prédicat de comparaison de texte `LIKE` qui teste si un attribut de type chaîne contient une ou plusieurs sous-chaînes. Le caractère souligné _ remplace un caractère quelconque. Le caractère % remplace une séquence de caractères (cf Q4 et Q5).
- à l'aide du prédicat de test de nullité (attribut non renseigné) `NULL` (cf Q7).
- à l'aide du prédicat d'appartenance `IN` qui teste si la valeur de l'attribut appartient à une liste de valeurs (cf Q6).

Exemple 28

Voici quelques exemples simples de requêtes SQL.

- (Q1) `SELECT titre, genre FROM Livre WHERE auteur = 'Dumas';`
  On recherche dans la table LIVRE le titre et le genre des ouvrages écrits par DUMAS

- (Q2) `SELECT * FROM Livre WHERE auteur <> 'Dumas' AND année > 1835;`
  On recherche les ouvrages non écrits par Dumas et postérieurs à 1835

- (Q3) `SELECT auteur, lieu FROM Ecrivain WHERE né_en BETWEEN 1802 AND 1850;`
  On recherche le nom des auteurs nés entre 1802 et 1850 (ainsi que leur lieu de naissance).

- (Q4) `SELECT * FROM Ecrivain WHERE auteur LIKE 'B %';`

On filtre les auteurs dont le nom commence par un B

- (Q5) `SELECT * FROM Ecrivain WHERE auteur LIKE '_A %' ;`
  On filtre les auteurs dont le nom contient un A en deuxième position

- (Q6) `SELECT auteur, titre, année FROM Livre WHERE année IN (1839, 1866, 1857) ;`
  On sélectionne les ouvrages sortis en 1839, 1866 ou 1857.

- (Q7) `SELECT auteur, titre, année FROM Livre WHERE année NOT NULL ;`
  On sélectionne les ouvrages pour lesquels on dispose de l'année de parution.

Tri du résultat. Il est possible de trier le résultat d'une requête à l'aide de la clause `ORDER BY` suivie de la liste des attributs servant de critère au tri.

Exemple 29

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

trie par 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.

Fonctions sur les chaînes de caractères.

- `LENGTH` renvoie le nombre de caractères d'une colonne

- `SUBSTR` extrait une sous-chaîne dans une chaîne de caractères :
  `SUBSTR(NOM, 1, 15)` extrait les 15 premiers caractères de la chaîne `NOM`.
  `SUBSTR(NOM, LENGTH(NOM)-2, 3)` extrait les 3 derniers caractères de la chaîne `NOM`.

- `LIKE` permet de rechercher un profil-type dans une chaîne
  `NOM LIKE '%DE%'` recherche tous les noms contenant la sous-chaîne `'DE'`

- `REPLACE` permet de remplacer une chaîne par une autre chaîne dans une colonne
  `UPDATE Ecrivain SET NOM=REPLACE(nom, 'Dupont', 'Dupond') ;`
  remplace dans la colonne `nom` de la table `Ecrivain` les occurrences de la chaîne « `Dupont` » par la chaîne « `Dupond` ».

- `LTRIM` et `RTRIM` permettent de supprimer les blancs parasites en début et en fin de chaîne.
  `UPDATE Ecrivain SET nom=LTRIM(RTRIM(nom)) ;`
  `UPDATE Ecrivain SET nom=LTRIM(nom, '1234567890') ;`
  supprime toutes les occurrences de ces caractères en début de chaîne.

- `LPAD` et `RPAD` permettent de formatter une chaîne en justifiant à droite ou à gauche. La chaîne est complétée par des blancs, par une chaîne constante ou le contenu d'une colonne.
  `LPAD(nom, 20)` justifie à droite sur 20 colonnes en complétant par des blancs
  `LPAD(nom, LENGTH(nom) + 5, 'nom : ')` justifie à droite sur la longueur du nom + 5 caractères en complétant par la chaîne 'Nom : ' `'Dupont'` → `'Nom : Dupont'`
  `LPAD(NOM, LENGTH(NOM) + LENGTH(PRENOM) + 2, PRENOM || ', ')` justifie à droite sur la longueur des 2 colonnes + 2 caractères `'Dupont'`, `'Jules'` → `'Jules, Dupont'`

- `LOWER, UPPER, INITCAP` permettent respectivement de mettre une chaîne en minuscules, majuscules et le premier caractère de chaque mot en majuscule

- DECODE permet de définir le transcodage d'une valeur numérique en chaîne de caractères.
  SELECT NO_JOUR, DECODE(NO_JOUR, 1, 'LUNDI', 2, 'MARDI', 3, 'MERCREDI', 4, 'JEUDI', 5, 'VENDREDI', 6, 'SAMEDI', 7, 'DIMANCHE') FROM MA_TABLE ;
  → 6 SAMEDI si MA_TABLE ne contient qu'une seule ligne avec 6 dans la colonne NO_JOUR.

- ASCII renvoie le code Ascii du premier caractère de la chaîne passée en paramètre.

Fonctions sur les dates.

- SYSDATE retourne la date et l'heure courante. SYSDATE ne peut pas être utilisé dans une contrainte CHECK.
  INSERT INTO Commande VALUES ('COM01', SYSDATE) ;

- TO_CHAR permet de convertir une date en chaîne de caractères.
  SELECT TO_CHAR(SYSDATE, 'HH24, MI') FROM DUAL ;

- TO_DATE permet de convertir une chaîne de caractères en date.
  INSERT INTO Planning VALUES (TO_DATE('15 :30', 'HH24, MI')) ;

Elément	Signification
D	jour de la semaine (1-7)
DAY	nom du jour
DD	jour du mois (1-31)
HH	heure du jour (1-12)
HH24	heure du jour (1-24)
MI	minute (0-59)
MM	mois (1-12)
MONTH	nom du mois
SS	seconde (0-59)
YYYY	année sur 4 chiffres
YY	année (2 derniers chiffres)

FIG. 3.5: Quelques éléments de formats de date.

Fonctions de conversion entre types.

- TO_CHAR permet de convertir des valeurs numériques en chaîne de caractères.
  SELECT TO_CHAR(SALAIRE) FROM EMPLOYE ;

  SELECT TO_CHAR(SALAIRE, '999999.99') FROM EMPLOYE ;
  55 → 55.00
  456 → 456.00

  SELECT TO_CHAR(SURFACE, '9.9999EEEE') FROM POLYGONE ;
  123 → 1.2300E+02
  12345 → 1.2345E+04

- `TO_NUMBER` permet de convertir des chaînes de caractères en valeurs numériques. Par exemple, si `ANNEE_DEB` et `ANNEE_FIN` sont des chaînes :
  `SELECT SUM(1000* TO_NUMBER(ANNEE_FIN) - TO_NUMBER(ANNEE_DEB)) FROM EMPLOYE_RECOMPENSE ;`

4.3.2 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 et différence.

Expression des jointures. La syntaxe pour exprimer des jointures avec SQL est une extension directe de celle étudiée précédemment dans le cas des sélections simples : on donne 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 `Séjour`. 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, Séjour
WHERE   id = idClient ;
```

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

Exemple 30

Afficher le nom d'une station, son tarif hebdomadaire ; ses activités et leurs prix.

```
SELECT  nomStation, tarif, libellé, prix
FROM    Station, Activité
WHERE   Station.nomStation = Activité.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 :

```
SELECT  nomStation, tarif, libellé, prix
FROM    Station S, Activité A
WHERE   S.nomStation = A.nomStation ;
```

Il existe des situations dans lesquelles l'utilisation des synonymes est indispensable : celles où l'on souhaite effectuer une jointure d'une relation avec elle-même.

Exemple 31

Soit la requête : *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 si l'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.région = s2.région ;
```

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

Voici d'autres exemples de jointures :

Exemple 32

Une jointure sans qualification (sans clause `WHERE`) est le produit cartésien (cf Q1). Dans une jointure avec qualification, le produit cartésien est restreint par une combinaison de prédicats de comparaison (cf Q2 à Q5).

Enfin, dans une jointure, il est possible de privilégier une table (cf Q6).

Voici les schémas de relation servant de support pour les exemples de requêtes :

```
CREATE TABLE LIVRE
   (AUTEUR CHAR (8),
    TITRE CHAR (24),
    ANNEE SMALLINT,
    GENRE CHAR (8),
    PRIX DECIMAL (5,2),
  CONSTRAINT LIVRE_PK
  PRIMARY KEY(AUTEUR, TITRE)) ;

CREATE TABLE ECRIVAIN
   (AUTEUR CHAR (8)
   CONSTRAINT ECRIVAIN_PK
   PRIMARY KEY,
    NE-EN SMALLINT,
    LIEU CHAR (20),
    RAYON SMALLINT ) ;

CREATE TABLE RAYON
   (RAYON SMALLINT
   CONSTRAINT RAYON_PK
  PRIMARY KEY,
   SALLE SMALLINT ) ;
```

Exemples de requêtes :

```
(Q1) SELECT A.auteur, B.lieu
     FROM   Livre A, Ecrivain B ;
```

forme le produit cartésien des tables `Livre` et `Ecrivain` sur les attributs `auteur` de `Livre` et `lieu` de `Ecrivain`. On peut associer un nom abrégé (alias) aux noms de table pour alléger l'écriture des requêtes.

```
(Q2) SELECT A.AUTEUR, A.TITRE, B.RAYON
      FROM  LIVRE A, ECRIVAIN B
      WHERE A.AUTEUR = B.AUTEUR ;
```

Equi-jointure : liste le titre, l'auteur et le rayon de rangement des ouvrages de la bibliothèque.

Soit la table `OUVRAGE` définie comme :

```
CREATE TABLE OUVRAGE
   (AUTEUR CHAR (8),
    TITRE CHAR (24),
    NE-EN SMALLINT,
    SALLE SMALLINT,
    RAYON SMALLINT ) ;

(Q3) SELECT X.AUTEUR, X.TITRE, Y.NE_EN, Z.SALLE, Z.RAYON
     FROM   LIVRE X, ECRIVAIN Y, RAYON Z
     WHERE  X.AUTEUR = Y.AUTEUR AND Y.RAYON = Z.RAYON ;
```

reconstitue la table `OUVRAGE` à partir des tables `LIVRE, ECRIVAIN` et `RAYON`.

```
(Q4) SELECT A.AUTEUR, A.TITRE, A.ANNEE, A.GENRE, A.PRIX, B.NE_EN, B.LIEU, B.RAYON
     FROM   LIVRE A, ECRIVAIN B
     WHERE  A.AUTEUR = B.AUTEUR ;
```

réalise une jointure naturelle.

```
(Q5) SELECT A.AUTEUR, A.NE_EN, B.AUTEUR
     FROM   ECRIVAIN A, ECRIVAIN B
     WHERE  A.NE_EN = B.NE_EN AND A.AUTEUR > B.AUTEUR ;
```

Une jointure d'une table sur elle-même : liste le nom, la date de naissance des écrivains nés la même année.

```
(Q6) SELECT A.RAYON, B.AUTEUR
     FROM   RAYON A, ECRIVAIN B
     WHERE  A.RAYON = B.RAYON (+) ;
```

réalise une jointure dans laquelle on privilégie la table `Rayon`. Cette jointure affiche la liste de tous les rayons de la bibliothèque et les noms d'auteurs qui y sont rangés si les rayons ne sont pas vides.

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 (mêm nombre de colonnes et mêmes types d'attributs), et on les relie par un des mots-clés `UNION, INTERSECTION` ou `EXCEPT`.

Exemple 33 (Union)

Donner tous les noms de régions dans la base.

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

Exemple 34 (Intersection)

Donner tous les noms de régions dans la base.

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

Exemple 35 (Différence)

Donner tous les noms de régions dans la base.

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

4.3.3 Requêtes imbriquées

On parle de requête imbriquée lorsqu'une requête est faite dans la clause `WHERE` d'une autre requête.

La 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 les attributs provenant d'une seule des deux tables*, l'autre ne servant que pour exprimer des conditions.

Exemple 36

On veut les noms des stations où ont séjourné des clients parisiens. On peut obtenir ce résultat avec la 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    id = 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.

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 t est un tuple dont le type est celui de R. `TRUE` si t appartient à R, `FALSE` sinon.
3. *vcmp* `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. *vcmp* `ALL R`, Où *cmp* est un comparateur SQL ($<, >, =$, etc.). Renvoie `TRUE` si la comparaison avec *tous* des 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 :

Exemple 37

1. *Où (station, lieu) ne peut-on pas faire de ski ?*
```
SELECT   nomStation, lieu
FROM     Station
WHERE    nomStation NOT IN   (SELECT   nomStation FROM Activite
                              WHERE    libelle = 'ski');
```
2. *Quelle station pratique le tarif le plus élevé ?*
```
SELECT   nomStation
FROM     Station
WHERE    tarif >= ALL   (SELECT   tarif FROM Station);
```
3. *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')
```

Sous-requêtes corrélées .

Les exemples de requêtes imbriquées donnés précédemment pouvaient être évalués indépendamment de la requête principale. Cela pourrait permettre 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 corrélées*.

Exemple 38

Quels sont les clients (nom, prénom) qui ont séjournés 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.

4.3.4 Agrégation

Toutes les requêtes vues jusqu'à présent pouvaient être considérées comme une suite d'opérations effectuées *tuple* à *tuple*. De même, le résultat était une collection 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 les résultat par *agrégation de valeurs* au sein de chaque groupe.

La syntaxe SQL fournit donc :

1. Le moyen de partitionner 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 : le relation toute entière. Sans même définir de groupe donc, on peut utiliser les fonctions d'agrégation.

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 39

```
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). Ainsi, la requête suivante est incorrecte :

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

On peut faire des requêtes complexes.

Exemple 40

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

Plus généralement, les fonctions de calcul permettent de réaliser un calcul sur un ensemble de valeurs, résultat d'une question avec ou sans partitionnement.

- utilisées dans un `SELECT` sans `GROUP BY`, le résultat ne contient qu'une ligne de valeurs,
- utilisées dans un `SELECT` avec `GROUP BY`, le résultat contient une ligne de valeurs par partition.

Les fonctions implantées sont :

- `AVG, STDDEV, VARIANCE` : calculent respectivement la moyenne, l'écart-type et la variance des valeurs d'une collection de nombres figurant dans une colonne
  ```
  SELECT AVG (PRIX)
  FROM LIVRE ;
  ```
  donne le prix moyen des ouvrages de la table `LIVRE`

- `SUM` calcule la somme des valeurs d'une collection de nombres
  ```
  SELECT SUM (PRIX)
  FROM LIVRE
  WHERE GENRE = 'ROMAN' ;
  ```
  donne la somme des prix des romans de la table `LIVRE`

- `MIN` et `MAX` permettent d'obtenir la valeur minimum (resp. maximum) d'une collection de valeurs
  ```
  SELECT MAX (PRIX), GENRE
  FROM LIVRE
  GROUP BY GENRE ;
  ```
  donne le prix maximum des ouvrages de la table `LIVRE` pour chaque genre

- `COUNT` permet de dénombrer une collection de lignes

  ```
  SELECT COUNT (*)
  FROM AUTEUR
  WHERE LIEU = 'PARIS' ;
  ```
  compte le nombre d'auteurs de la table `AUTEUR` nés à Paris

```
SELECT COUNT (NE_EN)
FROM AUTEUR
WHERE LIEU = 'PARIS';
```
compte le nombre d'auteurs de la table `AUTEUR` nés à Paris dont on connaît la date de naissance (colonne not null).

- `ROUND, TRUNC, FLOOR, CEIL` pour arrondir et tronquer

```
ROUND(123.27, 1)? 123.3

ROUND(123.22, 1)? 123.2

ROUND(101.8)? 102

ROUND(123.27, -2)? 100

TRUNC(123.33)? 123

TRUNC(123.567, 2)? 123.56

TRUNC(123.567, -2)? 100

FLOOR(128.7)? 128 /* entier inf */

FLOOR(129.1)? 129

CEIL (128.7)? 129 /* entier sup */

CEIL(129.1)? 130
```

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 *partitionner* 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 41

Afficher les régions avec le nombre de stations associées

```
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`. Dans l'exemple précédent, l'attribut est `region`;
2. le résultat d'une fonction d'agrégation appliquée au groupe. Dans l'exemple, `COUNT(nomStation)`.

Bien entendu, il est possible d'exprimer des ordres SQL complexes et d'appliquer un `GROUP BY` dans la clause `select`.

Exemple 42

On souhaite consulter le nombre de places réservées 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 partitionne, enfin (3) on calcule le résultat des fonctions.

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 43

On souhaite consulter le nombre de places réservées, 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.

4.3.5 Mises-à-jour

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

Insertion. L'insertion s'effectue à l'aide de 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 lesquels 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 figuer dans une clause `INSERT`.

Les `v1, ..., vn` sont les valeurs des attributs.

Exemple 44

Insertion d'un tuple dans la table `Client`.

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

Donc, à l'issue de cette assertion, les attributs `ville` et `region` seront `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 45

On a créé une table `Sites(lieu, region)` et on souhaite y copier les couples `(lieu, region)` déjà existants 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.

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

```
DELETE   FROM R
WHERE    condition;
```

`R` est la table, et `condition` est toute la 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 46

Destruction de tous les clients dont le nom commence par 'M' :

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

Modification. La modification s'effectue à l'aide de 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 47

Augmenter le prix des activités de la station Passac de 10%.

```
UPDATE  Activites
SET     prix = prix * 1.1
WHERE   nomStation = 'Passac';
```

Remarque importante : toutes les mises-à-jour ne deviennent définitives qu'à l'issue d'une validation par `commit`. Entre temps, elles peuvent être annulées par `rollback`. Nous y reviendrons dans la section sur la concurrence d'accès **??**.

Mise en pratique en utilisant PostgreSQL

- Se créer un répertoire BD : *mkdir BD*
- Se placer dans ce répertoire : *cd BD*

Utilisation de PostgreSQL

Manuel en ligne http://www.postgresql.org/docs/8.4/static/index.html

Création d'une base de données : `$ createdb mydb`

Si pas de message, c'est que ça marche.

Une fois qu'une base de données est créée, on peut y accéder en utilisant :
- l'environnement de PostgreSQL, avec la commande `psql`
- un environnement graphique comme pgAdmin ou un ODBC ou JDBC

Nous accédons à la base `mydb` par : `$ psql mydb`.

Les lignes de commandes devraient alors commencer par : `mydb=>`

Pour avoir de l'aide sur les commandes PostgreSQL : `mydb=> \h`

Pour quitter PostgreSQL : `mydb=> \q`

Créer les tables dont le schéma est en p.95 de [Elmasri et Navathe, Fr] -> Voir p.153

Remplir les tables avec les informations en p.97 de [Elmasri et Navathe, Fr]

Effectuer des requêtes pour répondre aux questions suivantes :

Requêtes simples

Requête 1 : Chercher la date de naissance et l'adresse de tous les employés dont le nom est 'Bernard Schmidt'.

```
SELECT   date_naiss, adresse
FROM     Employe
WHERE    Prenom = 'John' AND Nom = 'Smith';
```

Requête 2 : Extraire le nom et l'adresse de tous les employés qui travaillent pour le service 'recherche'.

```
SELECT   Nom, adresse
FROM     Employe, Service
WHERE    Service.Nom_Sce = 'Recherche' AND Service.No_Sce = Employe.NoSce;
```

Noms d'attributs ambigüs

Requête 3 : Pour chaque employé, extraire son prénom et son nom ainsi que le prénom et le nom de son supérieur immédiat.

```
SELECT   E.Prenom, E.Nom, S.Prenom, S.Nom
FROM     Employe AS E, Employe AS S
WHERE    E.NoSSSUPER = S.NoSS;
```

Clause `WHERE` non spécifiée et utilisation de l'astérisque

Requête 4 : Sélectionner tous les `NoSS` dans `Employe`.

```
SELECT   NoSS
FROM     Employe;
```

Requête 5 : Sélectionner toutes les combinaisons de `NoSS` d'`Employe` et de `Noms` dans `Service` dans la base de données.

```
SELECT   NoSS, Noms_Sce
FROM     Employe, Service;
```

Requête 6 : Récupérer les valeurs de tous les attributs des `Employe` qui travaillent pour le service n°5.

```
SELECT   *
FROM     Employe
WHERE    NoSce = 5;
```

Requête 7 : Récupérer les valeurs de tous les attributs de chaque `Employe` et `Service` pour lequel il travaille pour tous les employés du service 'Recherche'.

```
SELECT   *
FROM     Employe AS E, Service AS S
WHERE    Nom_Sce = 'Recherche' AND E.NoSce = S.NoSCE;
```

Requête 8 : Engendrer le produit cartésien des relations `Employe` et `Service`.

```
SELECT   *
FROM     Employe, Service;
```

Élimination des doublons : tables et ensembles

Requête 9 : Extraire le salaire de chaque employé.

```
SELECT ALL   Salaire
FROM         Employe;
```

Requête 10 : Extraire toutes les valeurs distinctes des salaires des employés.

```
SELECT DISTINCT   Salaire
FROM              Employe;
```

Requête 11 : Lister tous les numéros des projets auxquels participe un employé dont le nom de famille est 'Schmidt', que ce soit comme simple employé ou comme responsable du département en charge du projet.

```
(SELECT DISTINCT  No_P
FROM              Projet, Service, Employe
WHERE             NumS = NoSCe AND NoSSDIR = NoSS AND Nom = 'Schmidt')
UNION
(SELECT DISTINCT  NoP
FROM              Projet, Travaille_Sur, Employe
WHERE             Numero_Projet = No_Projet AND NoSS_Empl = NoSS AND Nom = 'Schmidt');
```

Sélection de sous-chaînes à l'aide de motifs et opérateurs arithmétiques

Requête 12 : Extraire les noms et les prénoms de tous les employés qui habitent Les Ulis.

```
SELECT   Prenom, nom
FROM     Employe
WHERE    Adresse LIKE '%Les Ulis%';
```

Requête 13 : Extraire les noms et les prénoms de tous les employés nés dans les années 50.

```
SELECT   Prenom, nom
FROM     Employe
WHERE    Date_Naiss LIKE '__5_______';
```

Requête 14 : Afficher les salaires des employés travaillant sur le projet 'ProduitX une fois augmentés de 10%.

```
SELECT   Prenom, nom, 1.1*Salaire as Augmentation
FROM     Employe, Travaille_Sur, Projet
WHERE    NoSS = NoSS_Empl AND Numero_Projet = No_Projet
         AND Nom_Projet = 'ProduitX';
```

Requête 15 : Extraire toutes les données relatives aux employés du service 5 dont le salaire est compris entre 30 000 et 40 000 €.

```
SELECT   *
FROM     Employe
WHERE    Salaire BETWEEN 30000 AND 40000 AND NoSce = 5;
```

Tri des résultats d'une requête

Requête 16 : Lister les employés et les projets sur lesquels ils travaillent en les classant par service et, pour chaque service, en les classant selon l'ordre alphabétique de leur nom de famille puis de leur prénom.

```
SELECT     Nom_Sce, Nom, Prenom Nom_projet
FROM       Service, Employe, Travaille_Sur, Projet
WHERE      NoSCE = NoSce AND NoSS=NoSS_Empl AND Numero_Projet = No_Projet
ORDER BY   Nom, Prenom;
```

Exercices sur les requêtes en SQL

En utilisant les tables sur les employés et les projets.

Requête 1 : Chercher la date de naissance et l'adresse de tous les employés dont le nom est 'John B. Smith'.

Requête 2 : Chercher le nom et l'adresse de tous les employés qui travaillent pour le département recherche.

Requête 3 : Pour tous les projets situés à 'Stafford', chercher les numéros de projet, le département de supervision et les nom, adresse et date de naissance des managers de ces départements.

Requête 4 : Pour chaque employé, chercher les prénoms et noms de cet employé et les prénom et nom de son supérieur immédiat.

Requête 5 : Sélectionner tous les `EMPLOYEE SSNS` de la base.

Requête 6 : Sélectionner toutes les combinaisons de `EMPLOYEE SSNS` et `DEPARTMENT DNAME` de la base.

Requête 7 : Chercher toutes les valeurs d'attribut des tuples de `EMPLOYEE` qui travaillent dans le département 5.

Requête 8 : Sélectionner tous les attributs d'un `EMPLOYEE` et les attributs du `DEPARTMENT` pour lequel il travaille, et ceci pour tous les employés du département de recherche.

Requête 9 : Chercher le produit cartésien des relations `EMPLOYEE` et `DEPARTMENT`.

Requête 10 : Chercher le salaire de tous les employés.

Requête 11 : Chercher toutes les valeurs distinctes de salaires des employés.

Requête 12 : Chercher tous les numéros de projets qui impliquent un employé dont le nom est 'Smith', soit en tant que travailleur dans ce projet ou en tant que manager du département qui contrôle ce projet.

Requête 13 : Chercher tous les employés dont l'adresse est à Houston, Texas.

Requête 14 : Chercher tous les employés nés durant les années 50.

Requête 15 : Donner le salaire résultant d'une hausse de 10% de tous les employés travaillant pour le projet 'Product X'.

Requête 16 : Chercher tous les employés dans le département 5 dont le salaire est compris entre \$30 000 et \$40 000.

Requête 17 : Chercher les employés et les projets sur lesquels ils travaillent, triés par département et, à l'intérieur de chaque département, triés par ordre alphabétique sur les noms et les prénoms.

Requête 18 : Reformuler la requête 12 en utilisant des requêtes imbriquées.

Requête 19 : Chercher les noms de tous les employés dont le salaire est plus grand que le salaire de tous les employés dans le département 5.

Requête 20 : Donner le salaire résultant d'une hausse de 10% de tous les employés travaillant pour le projet 'Product X'.

Requête 21 : Chercher le nom des employés qui ont un 'dependent' de même prénom et de même sexe que lui (ou elle).

Requête 22 : Chercher le nom des employés qui travaillent sur *tous* les projets supervisés par le département 5.

Requête 23 : Reformuler la requête 21 : (Chercher le nom des employés qui ont un 'dependent' de même prénom et de même sexe que lui (ou elle)), en utilisant un `EXISTS`.

Requête 24 : Chercher le nom des employés qui n'ont pas de 'dependent'.

Requête 25 : Chercher le nom des managers qui ont au moins un 'dependent'.

Requête 26 : Chercher le numéro de sécurité sociale de tous les employés qui travaillent sur le projet 1, 2 ou 3.

Requête 27 : Chercher le nom des employés qui n'ont pas de supérieur.

Requête 28 : Trouver le somme des salaires de tous les employés, le salaire maximal, le salaire minimal et le salaire moyen.

Requête 29 : Trouver le somme des salaires de tous les employés du département de recherche, de même que le salaire maximal, le salaire minimal et le salaire moyen dans ce département.

Requête 30 : Trouver le nombre total d'employés dans la compagnie.

Requête 31 : Trouver le nombre total d'employés dans le département de recherche.

Requête 32 : Trouver le nombre de salaires distincts dans la base.

Requête 33 : Pour chaque département, trouver le numéro de département, le nombre d'employés dans le département et leur salaire moyen.

Requête 34 : Pour chaque projet, trouver le numéro de projet, le nombre d'employés travaillant sur ce projet.

Requête 35 : Pour chaque projet sur lequel *plus de deux employés travaillent*, chercher le numéro de projet, le nom du projet, et le nombre d'employés travaillant sur ce projet.

Requête 36 : Pour chaque projet, chercher le numéro de projet, le nom du projet, et le nombre d'employés du département 5 travaillant sur ce projet.

Requête 37 : Pour chaque département qui a plus de 5 employés, trouver le numéro du département et le nombre de ses employés gagnant plus que $40 000 par an.

4.4 Le DDL (Data Description Language) en SQL

On se place 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 voyages ». Il faut donc distinguer l'*instance de schéma* de la *base de données* qui est un sur-ensemble.

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

4.4.1 Création de schémas

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 voyages ».

```
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_desspectacles
```

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 commande `CREATE TABLE` est utilisée pour définir une nouvelle relation en lui donnant un nom et en spécifiant ses attributs et ses contraintes. Les attributs sont spécifiés en premier, et on leurs donne un nom, un type de données pour spécifier leur domaine et toutes contraintes attachées (par exemple `NOT NULL`). Les clés, contraintes d'entité et contraintes d'intégrité référentielle peuvent être spécifiées - dans la commande `CREATE TABLE` - après la déclaration des attributs, ou bien elles peuvent être ajoutées plus tard en utilisant des commandes `ALTER TABLE`.

Exemple 48

```
CREATE TABLE EMPLOYEE
(FNAME VARCHAR(15) NOT NULL
MINIT CHAR,
LNAME VARCHAR(15) NOT NULL,
SSN CHAR(9) NOT NULL,
BDATE DATE,
ADDRESS VARCHAR(30),
SEX CHAR,
SALARY DECIMAL(10,2)
SUPERSSN CHAR(9),
DNO INT NOT NULL,
PRIMARY KEY (SSN)
FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN)
FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER)
```

Identificateurs. Ils doivent être de longueur maximale de 30 caractères (Oracle).

Types de données et domaines en SQL. Les types de données disponibles incluent les nombres, les chaînes de caractères, les chaînes de bits, la date et le temps.
Les types de **nombres**.

Ils incluent les *entiers* de différentes tailles (`INTEGER` ou `INT`, et `SMALLINT`), et les *réels* de précisions variables (`FLOAT, REAL, DOUBLE PRECISION`). Les nombres formatés peuvent être déclarés par `DECIMAL`(i, j) ou `DEC`(i, j) ou `NUMERIC`(i, j) où i, la précision, est le nombre total de chiffres et j, l'échelle, est le nombre de chiffres après la virgule.

Les types **chaînes de caractères**.

Ils sont soit de longueur fixe - `CHAR(`n`)` ou `CHARACTER(`n`)`-, où n est le nombre de caractères, soit de longueur variable - `VARCHAR(`n`)` ou `CHAR VARYING(`n`)` ou `CHARACTER VARYING(`n`)` -, où n est le nombre maximal de caractères.

Les types **chaînes de bits**.

Ils sont soit de longueur fixe - `BIT(`n`)` -, où n est le nombre de bits, soit de longueur variable - `BIT VARYING(`n`)` -, où n est le nombre maximal de bits. La valeur par défaut pour n est de 1.

Les **BLOBs** (Binary Long Object).

Avec le développement du multimédia, la plupart des SGBD propose un type de données qui permet de stocker des objets binaires de grande taille (documents, image, son, vidéo, ...). Actuellement deux stratégies s'opposent :

1. On stocke dans la BD la référence du fichier qui contient le BLOB.
2. On stocke l'objet directement dans une colonne de la BD.

La solution 1 fournit en principe un temps d'accès plus rapide au BLOB.

La solution 2 a l'avantage de la portabilité en cas de déplacement des fichiers et/ou de changement de plateforme.

Les types de données binaires (image, son) sont :
- RAW (jusqu'à 2000 bytes),
- LONG RAW (jusqu'à 2 giga bytes)

Le type **date**.

Oracle alloue 7 caractères pour un attribut de type `DATE`. Oracle utilise ce type pour stocker la date et l'heure. Le format par défaut est : *jj-mmm-aa* (exemple : `'01-JAN-94'`)

4.4.2 Contraintes et assertions

XXX A FAIRE

4.4.3 Vues

XXX A FAIRE

4.4.4 *Triggers*

XXX A FAIRE

4.4.5 *Les droits d'accès aux données*

L'accès à une base de données est restreint, pour des raisons 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
```

Suivies de l'entrée du mot de passe demandé 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 chémas 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 la syntaxe est :

```
GRANT   <privilege>
ON      <element du schéma>
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`.

Exemple 49

Par exemple, pour permettre à Marc de consulter 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`.

Exemple 50

```
GRANT ALL PRIVILEGES ON Film TO Marc ;
```

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

Exemple 51

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

Chapitre 4

Pratique des SGBD

Sommaire

1. Exemples de SGBD

1.1 Perspectives historiques

1.2 ORACLE

1.3 Microsoft Access

2. Techniques d'optimisation

3. Techniques d'accès partagés : le contrôle de concurrence

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**.

3.1 Problèmes potentiels et notions de base

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) :

Algorithme 1 : Algorithme de réservation

```
Programme RESERVATION
Données : Une séance c
Le nombre de places souhaité NbPlaces
Le client c
début
    Lire la séance s
    si (nombre de places libres > NbPlaces) alors
        Lire le compte du client c
        Débiter le compte du client c
        Soustraire NbPlaces au nombre de places vides
        Ecrire la séance s
        Ecrire le compte du client c
    fin Si
fin
```

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.

La nature des informations manipulées est également 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 présente donc simplement par la séquence :

r[s] r[c] w[c] w[s]

3.1.1 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 et du programme P_2.

—— Exemple ——

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.

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 $P_2(s, 2, c_2)$, en supposant que la séquence des opérations est celle donnée ci-dessus. On se concentre pour l'instant sur les évolutions du nombre de places vides.

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.

À 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 **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 **Exécution imbriquée de P_1 et P_2** ——

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

Suivons pas à pas l'exécution :

1. P_1 lit s et c_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 c_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 nouveau compte du client c_2.

Cette exécution est correcte : on obtient un résultat strictement semblable à celui issu d'une exécution en série. Il existe donc des exécutions imbriquées qui sont aussi correctes qu'une exécution en série et qui permettent une meilleure concurrence. 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**.

3.1.2 Transactions

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 entraîne 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 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.

Définition 4.1 (Transaction)

Une transaction *est l'ensemble des instructions séparant un* commit *ou un* rollback *du* commit *ou du* rollback *suivant.*

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

Les deux premières transactions ne sont pas correctes, la troisième l'est (C signifie *commit*, et R *rollback*).

1. $r(s)$ $r(c)$ $w(s)$ C $w(c)$
2. $r(s)$ $r(c)$ $w(s)$ R $w(c)$ 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.

3.1.3 Interférences possibles entre transactions

Il existe quatre types d'anomalies transactionnelles :

- L'anomalie la plus grave est la **lecture impropre** (lecture sale ou *dirty read*) : elle se produit lorsqu'une transaction lit des données qui n'ont pas encore été validées.
- Est aussi grave l'anomalie d'**écriture impropre** (écriture sale ou *dirty write*) : elle se produit lorsqu'une transaction valide une écriture d'une autre transaction alors que celle-ci va l'annuler par un rollback.
- Suit l'anomalie de **lecture non répétable** (*non repeatable read*) : deux lectures successive des mêmes données ne produisent pas le même résultat dans la même ligne.
- Enfin la **lecture fantôme** (*phantom read*) est une anomalie qui se produit lorsque des données nouvelles apparaissent ou disparaissent dans des lectures successives.

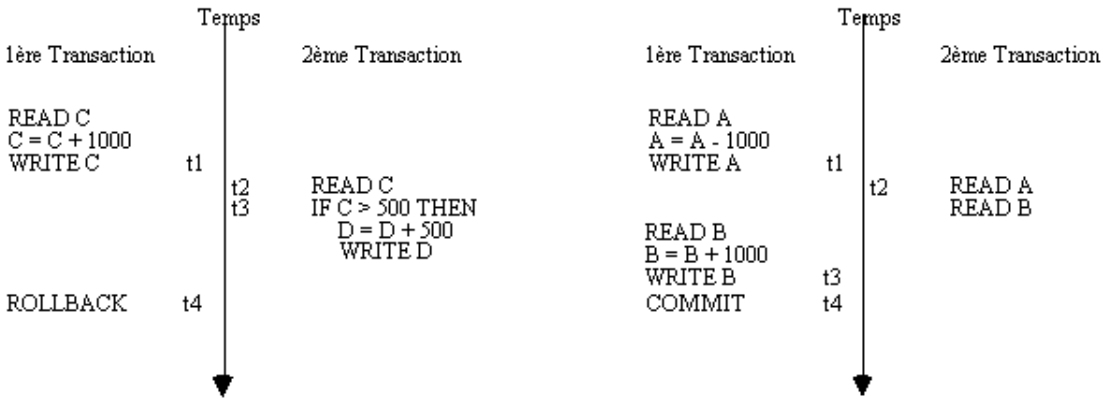

FIG. 4.1: Écriture incohérente (à gauche). Lecture incohérente (à droite).

3.1.3.1 Écriture incohérente (« *dirty write* »). Une 1ère transaction modifie le crédit C d'un compte. Une 2ème lit C dans ce nouvel état, puis, constatant que la provision est suffisante, modifie le

débit D du compte. Mais la 1ère fait un "rollback". La donnée D a désormais une valeur qui ne satisfait plus la contrainte d'intégrité D =< C c'est à dire que le solde est négatif !

3.1.3.2 Lecture incohérente (« *dirty read* »). Une 1ère transaction, pour faire un virement entre deux comptes A et B, qui satisfait la contrainte d'intégrité "somme A+B invariante", commence par débiter A. Juste à ce moment une 2ème lit A puis B et trouve A+B diminué !

Exemple de lecture impropre

L'utilisateur 1 ajoute 10 places et annule sa transaction, tandis que l'utilisateur 2 veut 7 places si elles sont disponibles...

```
                                                    T2 BEGIN TRANSACTION 2
T1 BEGIN TRANSACTION 1
                                                    T4 if (SELECT VOL_PLACES_LIBRES
T3 UPDATE T_VOL                                           FROM T_VOL WHERE VOL_ID = 2) >= 7
   SET    VOL_PLACES_LIBRES = VOL_PLACES_LIBRES + 10   then
   WHERE  VOL_ID = 2
                                                    T6 begin
                                                          UPDATE T_VOL
                                                          SET    VOL_PLACES_LIBRES = VOL_PLACES_LIBRES - 7
                                                          WHERE  VOL_ID = 2
T5 ROLLBACK TRANSACTION 1                           T7    INSERT INTO T_CLIENT_VOL VALUES (77, 2, 5)
                                                    T8    COMMIT TRANSACTION
                                                      end
                                                    T9 else
                                                          ROLLBACK TRANSACTION 2
```

FIG. 4.2: Exemple de lecture impropre.

Et les données qui sont manipulées : voir figure 4.3.

Temps	T_VOL : VOL_ID	VOL_REFERENCE	VOL_PLACES_LIBRES	T_CLIENT_VOL : CLI_ID	VOL_ID	VOL_PLACES_PRISES	TRANSACTION
T1	2	AF 812	6				DEBUT TRANSACTION POUR UTILISATEUR 1
T2	2	AF 812	6				DEBUT TRANSACTION POUR UTILISATEUR 2
T3	2	AF 812	16				
T4	2	AF 812	16				
T5	2	AF 812	6				FIN TRANSACTION POUR UTILISATEUR 1 AVEC ROLLBACK
T6	2	AF 812	-1				
T7	2	AF 812	-1	77	2	5	
T8	2	AF 812	-1	77	2	5	FIN TRANSACTION POUR UTILISATEUR 2 AVEC COMMIT

FIG. 4.3: Données manipulées lors des transactions avec lecture impropre.

Le temp d'un update avorté, la transaction 2 a lu des informations qu'elle n'aurait jamais dû voir et en a tiré la conclusion qu'elle pouvait servir les places... Conclusion *surbooking* !

3.1.3.3 Exemple de lecture non répétable Cas où notre opérateur désire toutes les places d'un vol si il y en à plus de 4...

Et les données qui sont manipulées : voir figure 4.5.

Notre utilisateur 2 voulait au moins 4 places et en à reçu 2... Conclusion, vous avez perdu un client !

3.1.3.4 Exemple de lecture fantôme Notre utilisateur 2, désire n'importe quel vol pas cher pour emmener son équipe de foot (soit 11 personnes) à une destination quelconque.

Et les données qui sont manipulées : voir figure 4.7.

```
                                                  T1 BEGIN TRANSACTION 2
                                                  T2 if (SELECT VOL_PLACES_LIBRES
                                                         FROM T_VOL WHERE VOL_ID = 2) >= 4
                                                     then
T3 BEGIN TRANSACTION 1
T4 UPDATE T_VOL
   SET    VOL_PLACES_LIBRES = VOL_PLACES_LIBRES - 4
   WHERE  VOL_ID = 2
T5 COMMIT TRANSACTION 1
                                                  T6 begin
                                                        UPDATE T_VOL
                                                        SET    VOL_PLACES_LIBRES = VOL_PLACES_LIBRES -
                                                               (SELECT VOL_PLACES_LIBRES
                                                                FROM T_VOL WHERE VOL_ID = 2)
                                                        WHERE  VOL_ID = 2
                                                  T7    INSERT INTO T_CLIENT_VOL VALUES (77, 2,
                                                                (SELECT VOL_PLACES_LIBRES
                                                                 FROM T_VOL WHERE VOL_ID = 2))
                                                  T8    COMMIT TRANSACTION
                                                     end
                                                  T9 else
                                                         ROLLBACK TRANSACTION 2
```

FIG. 4.4: Exemple de lecture non répétable.

Temps	T_VOL : VOL_ID	VOL_REFERENCE	VOL_PLACES_LIBRES	T_CLIENT_VOL : CLI_ID	VOL_ID	VOL_PLACES_PRISES	TRANSACTION
T1	2	AF 812	6				DEBUT TRANSACTION POUR UTILISATEUR 1
T2	2	AF 812	6				
T3	2	AF 812	16				DEBUT TRANSACTION POUR UTILISATEUR 2
T4	2	AF 812	2				
T5	2	AF 812	2				FIN TRANSACTION POUR UTILISATEUR 1 AVEC COMMIT
T6	2	AF 812	0				
T7	2	AF 812	0	77	2	2	
T8	2	AF 812	0	77	2	2	FIN TRANSACTION POUR UTILISATEUR 2 AVEC COMMIT

FIG. 4.5: Données manipulées lors des transactions avec lecture non répétable.

```
                                                  T1 BEGIN TRANSACTION 2
                                                  T2 if EXISTS (SELECT *
                                                                FROM T_VOL
                                                        WHERE VOL_PLACE_LIBRE >= 11)   then
T3 BEGIN TRANSACTION 1
T4 INSERT INTO T_VOL
   VALUES (5, 'AF 111', 125)
T5 COMMIT TRANSACTION 1
                                                  T6 begin
                                                        UPDATE T_VOL
                                                        SET    VOL_PLACES_LIBRES = VOL_PLACES_LIBRES - 11
                                                        WHERE  VOL_PLACES_LIBRES >= 11
                                                  T7    INSERT INTO T_CLIENT_VOL VALUES (77, 4, 11)
                                                  T8    COMMIT TRANSACTION
                                                     end
                                                  T9 else
                                                         ROLLBACK TRANSACTION 2
```

FIG. 4.6: Exemple de lecture fantôme.

11 places ont été volatilisé du vol AF 111 et c'est comme cela qu'un certain été, des avions d'Air France volaient à vide avec toutes les places réservées!!! (Histoire hélas véridique due à un bug du service informatique de réservation!!!)

3.1.4 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

Temps	T_VOL : VOL_ID	VOL_REFERENCE	VOL_PLACES_LIBRES	T_CLIENT_VOL : CLI_ID	VOL_ID	VOL_PLACES_PRISES	TRANSACTION
T1	4	AF 325	258				DEBUT TRANSACTION POUR UTILISATEUR 1
T2	4	AF 325	258				
T3	4	AF 325	258				DEBUT TRANSACTION POUR UTILISATEUR 2
T4	4 5	AF 325 AF 111	258 125				
T5	4 5	AF 325 AF 111	258 125				FIN TRANSACTION POUR UTILISATEUR 1 AVEC COMMIT
T6	4 5	AF 325 AF 111	247 114				
T7	4 5	AF 325 AF 111	247 114	77	4	11	
T8	4 5	AF 325 AF 111	247 114	77	2	11	FIN TRANSACTION POUR UTILISATEUR 2 AVEC COMMIT

FIG. 4.7: Données manipulées lors des transactions avec lecture fantôme.

et illustré par l'exemple suivant :

EXEMPLE **Exécution recouvrables**

$$r_1(s)\ \ r_1(c_1)\ \ w_1(s)\ \ r_2(s)\ \ r_2(c_2)\ \ w_2(s)\ \ w_2(c_2)\ \ C_2\ \ w_1(c_1)\ \ R_1$$

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 celui des clients ayant validé leur réservation.

Le problème ici vient du fait que la transaction T_1 est annulée **après** que la transaction T_2 a 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 mises à jour 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 **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 fait sa validation. 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*.

Cela 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 **Exécution non stricte**

$$r_1(s)\ \ r_1(c_1)\ \ r_2(s)\ \ w_1(s)\ \ w_1(c_1)\ \ r_2(c_2)\ \ w_2(s)\ \ C_1\ \ w_2(c_2)\ \ R_2$$

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

Cette fois, c'est T_1 qui annule et T_2 qui valide.

$$r_1(s)\ \ r_1(c_1)\ \ r_2(s)\ \ w_1(s)\ \ w_1(c_1)\ \ r_2(c_2)\ \ w_2(s)\ \ R_1\ \ w_2(c_2)\ \ C_2$$

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.

3.1.5 Le cahier des charges : une base ACID

Cela définit un ensemble de « bonnes » propriétés pour une transaction, qui est souvent désignée par l'acronyme **ACID**, pour :

- **Atomicité**. Soit toutes les actions d'une transaction sont exécutées, soit aucune ne l'est.
  L'utilisateur ne doit pas avoir à gérer les cas d'interruption d'une transaction quelque soit sa cause : erreur, panne, ...
- **Cohérence**. Une transaction est un ensemble de mises à jour entre deux états cohérents de la base par rapport à des contraintes d'intégrité.
  Le programmeur d'application est essentiellement le garant de cette propriété.
- **Isolation**. Le résultat d'une transaction doit être comparable à celui produit dans un environnement mono-utilisateur.
  L'utilisateur ne doit pas avoir à prendre en compte les interactions avec d'autres programmes pour écrire ses procédures de traitement.
- **Durabilité**. Une fois qu'une transaction est terminée, ses effets doivent persister malgré les pannes ou les tout autre problème rencontré par le SGBD.
  L'utilisateur ne doit pas avoir à prendre en compte ce qui peut se passer après la fin (normale) d'une transaction.

Il reste maintenant à étudier les techniques mises en œuvre pour assurer ces bonnes propriétés.

3.2 Techniques de 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** : on 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 ré-exécuter les quelques transactions qui posent problèmes.

La première approche est la plus fréquente. le mécanisme adoptée est celui du **verrouillage**. Le principe 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 se termine par *commit* ou *rollback* (« libération du verrou »).

Si tout accès en lecture ou en écriture pose un verrou bloquant les autres transactions, les anomalies ne peuvent se produire. Cependant, le blocage systématique des transactions est une contrainte trop forte qui bloque sans nécessité des transactions inoffensives. Il faut 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** (« *shared lock* ») est typiquement utilisé pour permettre à plusieurs transactions concurrentes de **lire** la même ressource.
2. Le verrou **exclusif** (« *eXclusive lock* ») 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 ou les 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.

3.2.1 Une approche pessimiste : le verrouillage à deux phases

Un protocole de verrouillage sans aucune contrainte sur l'allocation des verrous, ne permet pas de garantir la propriété de sérialisabilité. De nombreux protocoles ont été proposés, le plus classique étant le protocole de verrouillage à deux phases.

Le protocole de verrouillage à deux phases peut essentiellement se résumer à : pour une transaction, aucune action de verrouillage ne peut succéder à une libération de verrou.

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 ressource 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.

Définition 4.2 (Conflit)

Deux verrous $pl_i[x]$ et $ql_j[y]$ sont en conflit *si $x = y$ et pl ou ql est un verrou en écriture.*

Le respect du protocole est assuré par un module dit *ordonnanceur* (« *scheduler* », en anglais) qui reçoit les opérations émises par les transactions et les traite selon l'algorithme suivant :

Règle 1. L'*ordonnanceur* 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.

Règle 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.

Règle 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 fait qu'il y a d'abord **accumulation** de verrous pour une transaction T, puis **libération** des verrous.

Théorème 4.1

Toute exécution obtenue par un verrouillage à deux phases est sérialisable.

Preuve à faire en exercice (indication : démonstration par l'absurde en supposant que les transactions sont à 2 phases et qu'il existe un circuit dans le graphe de précédence). Attention, la réciproque n'est pas vraie. En exercice, trouver une exécution de transactions sérialisable mais qui ne soit pas à deux phases.

De plus, on obtient une *exécution stricte* en ne relâchant les verrous qu'au moment du *commit* ou du *rollback*. Les transactions obtenues satisfont les propriétés ACID.

Il est assez facile de se convaincre que ce protocole garantit qu'en présence de deux transactions en conflit T_1 et T_2, la dernière arrivée sera mise en attente de la première ressource conflictuelle et sera bloquée jusqu'à ce que la première commence à relâcher ses verrous (règle 1). À ce moment là, il n'y a plus de conflit possible puisque T_1 ne demandera plus de verrou (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.

Le verrouillage à deux phases (avec des variantes) est utilisé dans tous les SGBD. Il permet en effet une certaine imbrication des opérations.

Exemple 52 (Un exemple)

XXX A FAIRE

Le problème des interblocages ou encore verrous mortels (« *deadlock* »)

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.

—— Exemple ——

Soient les deux transactions suivantes :
- $T_1 : r_1[x] \rightarrow w_1[y] \rightarrow c_1$
- $T_2 : r_2[x] \rightarrow w_2[y] \rightarrow c_2$

Considérons maintenant que T_1 et T_2 s'exécutent en concurrence dans le cadre d'un verrouillage à deux phases :

$$rl_1[x]\ r_1[x]\ rl_2[y]\ r_2[y]\ wl_1[y]\ \underline{T_1 \textit{ en attente}}\ wl_2[x]\ \underline{T_2 \textit{ en attente}}$$

T_1 et T_2 sont en attente l'une de l'autre : il y a **interblocage**.

Cette situation ne peut pas être évitée et doit donc être gérée par le SGBD. Une solution est que ce dernier maintienne 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 annulées et ré-exécutée. Une autre solution, utilisée par exemple dans Oracle, consiste à tester le temps d'attente et annuler les transactions qui dépassent une limite pré-fixée.

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 les programmes, d'essayer de normaliser l'ordre d'accès aux données.

Méthodes de détection :

Pour se faire, l'ordonnanceur va maintenir un *graphe des attentes* et repérer si un circuit se forme dans le graphe des attentes. Pour comprendre le principe de construction du graphe d'attente par l'ordonnanceur, prenons l'exemple suivant. Soit T_i et T_j deux transactions. L'ordonnanceur reçoit de T_i une demande de verrou en lecture sur la donnée x mais retarde son accord car T_j a déjà posé un verrou en écriture. Alors, l'ordonnanceur peut rajouter un arc (T_i, T_j) dans le graphe d'attente. Ensuite, T_j supprime son verrou en écriture et l'ordonnanceur peut donc satisfaire T_i et retirer l'arc (T_i, T_j).

Pour maintenir le graphe d'attente, l'ordonnanceur va ajouter un arc (T_i, T_j) au graphe d'attente à chaque fois qu'une demande de pose de verrou de T_i est bloquée par un conflit avec un verrou appartenant à T_j. Un arc (T_i, T_j) est supprimé du graphe d'attente à chaque fois que l'ordonnanceur supprime le dernier verrou appartenant à T_j qui constituait un blocage pour l'attribution d'un verrou à T_i.

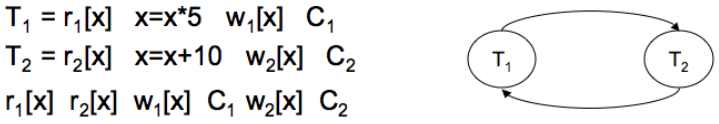

FIG. 4.8: *Exemple de graphe d'attente.*

Au niveau de la gestion de la détection de circuit dans le graphe d'attente, l'ordonnanceur peut vérifier à chaque ajout d'arc si le graphe contient un circuit, ce qui semble coûteux ou bien décider de faire des détections de « temps à autre ». Dans tous les cas, chaque fois qu'un circuit existe, il sera détecté.

Lorsqu'il détecte un circuit, l'ordonnanceur doit briser la situation d'interblocage en annulant une transaction. Cette annulation va supprimer un nœud dans le graphe d'attente et par conséquent le circuit. La transaction choisie pour être annulée s'appelle la *victime*. Parmi toutes les transactions candidates, l'ordonnanceur choisit la victime pour laquelle l'annulation est la moins coûteuse. Les facteurs de coût les plus communs sont :

- La quantité d'effort déjà investie dans la transaction. Cet effort serait perdu si la transaction est annulée.
- Le coût de l'annulation de la transaction. Ceci dépend du nombre de mises à jour déjà effectuées par la transaction.
- La quantité d'effort à investir pour terminer la transaction. L'idée est que l'ordonnanceur devrait éviter d'annuler une transaction presque achevée en essayant de prévoir le comportement futur de la transaction (par exemple en se basant sur le type de la transaction).
- Le nombre de circuits qui contient la transaction.

Il faut noter qu'une transaction peut intervenir successivement dans plusieurs interblocages. Dans chaque interblocage, la transaction est choisie comme victime, est annulée et relancée pour être impliquée encore dans un interblocage. Afin d'éviter une telle situation, le choix de la victime doit aussi prendre en compte le nombre de fois que la transaction candidate a été annulée à cause d'un interblocage. Si ce nombre est élevé alors elle ne deviendra victime que si l'on a pas d'autre choix.

3.2.2 Le contrôle par estampillage

Le mécanisme de gestion des concurrences par verrouillage à deux phases détecte *a posteriori* des attentes entre transactions. Des mécanismes de prévention basés sur un ordonnancement *a priori* des actions composantes des transactions peuvent être aussi envisagés.

L'**estampillage** est un *mécanisme du type prévention*. Les transactions T reçoivent de l'ordonnanceur à leur début - Begin Of Transaction - une estampille qui est un numéro d'ordre de présentation en entrée de l'ordonnanceur. Chaque granule G de données note l'estampille de la dernière transaction qui y a accédé. Un granule n'accepte un nouvel accès que s'il est demandé par une transaction "plus jeune", c'est-à-dire de numéro d'estampille plus grand.

```
Procédure LIRE (T,G,X)
  Si Estampille(T) >= Estampille(G) alors
    X ← lecture(G)
    Estampille(G) := Estampille(T)
  Sinon rollback(T)
```

```
Procédure ECRIRE (T, G, X)
    Si Estampille(T) >= Estampille(G) alors
      écriture(G) ← X
      Estampille(G) := Estampille(T)
    Sinon rollback(T)
```

La transaction rejetée est reprise par l'ordonnanceur après avoir reçu de lui une nouvelle estampille.

Un tel algorithme d'ordonnancement total conduit souvent à de nombreuses reprises en cascades, parfois par excès de prudence. Aussi a-t-on conçu des algorithme d'ordonnancement partiel qui distinguent pour chaque granule un compteur pour les estampilles des transactions accédant en lecture et un compteur pour les estampilles des transactions accédant en écriture, ou mieux qui distinguent des versions et les couples d'estampilles correspondantes pour chaque granule. S'ils améliorent clairement le débit transactionnel leur coût en écritures et en place mémoire n'est toutefois pas négligeable.

3.2.3 Granules et gestionnaire de transactions

Différentes granularités de contrôle de la concurrence sont possibles :

- valeur d'un attribut de n-uplet,
- un n-uplet,
- une page ou bloc,
- une relation,
- une base.

Plus la granularité est petite, plus le niveau de concurrence augmente. Par contre le niveau de complexité de gestion des verrous augmente et les performances baissent.

A priori le choix est dépendant du type des transactions (nature des informations manipulées). La plupart du temps dans les systèmes commerciaux la granularité est fixe (sauf indiquée explicitement). Certains systèmes font du verrouillage adaptatif en tenant compte de la hiérarchie des objets (plutot que de bloquer tous les attributs d'un n-uplet il vaut mieux bloquer le n-uplet et ainsi de suite). *Dans les systèmes actuels le niveau offert est celui du n-uplet.*

Dans le modèle physique les « tuples » sont des structures adressables d'octets. Un « gestionnaire de données » les charge dans un espace de la mémoire principale ou les en décharge, par des opérations élémentaires de lecture/écriture de « pages » - ou « blocs » - de données, pouvant contenir plusieurs « tuples », depuis ou vers des fichiers sur disque. Ainsi l'adresse physique de stockage d'un tuple est de la forme <fichier>,<page>,<offset>.

Le *granule* de concurrence le plus souvent gérée par les SGBD est la page ou la page et l'offset - c'est à dire le tuple.

Le *gestionnaire de transaction* enchaîne deux fonctions : (1) il découpe les transactions en actions élémentaires au niveau physique où est gérée la concurrence, puis (2) il ordonnance ces actions. Cet ordonnancement nécessite le calcul et le maintien d'un graphe de précédences *a priori* et/ou d'un graphe d'attentes *a posteriori*, puis le choix des séquences d'exécution selon le niveau d'isolation demandé et enfin la supervision de l'exécution. Un pré-processeur alimente l'ordonnanceur en flux parallèles d'actions. Ce dernier alimente le gestionnaire de données action par action et rend compte au pré-processeur de l'état instantané de chaque transaction : encours, suspendue, rejetée, validée.

3.3 La 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.

1- 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.

2- 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.

L'objectif de ces « relâchements » de contraintes plus ou moins importants est évidemment d'augmenter le parallélisme apparent et le débit en transactions.

Il est important de noter que le standard SQL ne dit pas comment choisir ces niveaux. Chaque SGBD a donc sa propre politique en la matière, y compris de ne pas en avoir, c'est-à-dire de n'implanter que l'un de ces niveaux (qui peut éventuellement être le plus laxiste).

Le tableau ci-dessous résume les différents niveaux d'isolation praticables et les anomalies qu'ils doivent interdire :

Anomalie	**READ UNCOMMITED** (niveau 0) possibilité de lire des informations qui sont en cours d'insertion mais non validées	**READ COMMITED** (niveau 1) Des données peuvent être modifiées avant la fin de la transaction	**REPEATABLE READ** (niveau 2) De nouvelles lignes peuvent apparaître avant la fin de la transaction	**SERIALIZABLE** (niveau 3) les transactions sont placées en série ou le SGBDR fait "comme ci"
Lecture impropre	possible	impossible	impossible	impossible
Lecture non répétable	possible	possible	impossible	impossible
Lecture fantôme	possible	possible	possible	impossible

FIG. 4.9: **Tableau récapitulatif des niveaux d'isolation dans SQL2.**

Les stratégies d'implantation des verrous portent sur :

1. **Les entités verrouillées** : *lignes, prédicats, pages, ...*
2. **Les modes de verrouillages** : *read* et *write*
3. **La durée des verrouillages** :
   - *courts* : les verrous acquis pour l'exécution d'une requête sont relâchés dès que la requêtes est terminée.
   - *Longs* : les verrous acquis pour l'exécution d'une requête sont relâchés seulement lorsque la transaction dont elle fait partie est terminée.
   - *Moyens* : solution intermédiaire.

Les **verrous en écriture** (« *write locks* », en anglais) sont traités de la même manière dans tous les niveaux d'isolation. Ils sont associés aux requêtes : `UPDATE, DELETE` et `INSERT`.

Les **verrous en lecture** sont traités différemment suivant les niveaux d'isolation.

- `READ UNCOMMITTED` : pas de verrou en lecture. Ainsi une transaction peut lire un objet verrouillé en écriture !
- `READ COMMITTED` : verrou de courte durée sur les lignes retournées par `SELECT`.
- `REPEATABLE READ` : verrou de longue durée sur les lignes retournées par `SELECT`.
- `SERIALIZABLE` : verrou de longue durée sur les prédicats spécifiés dans la clause `WHERE` du `SELECT`.

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
```

3.4 Le contrôle de concurrence sous Oracle

3.4.1 Niveaux d'isolation d'une transaction

Le **niveau Read Commited** est le niveau de transaction par défaut. Une requête voit les valeurs committées par les autres transactions avant le démarrage de l'exécution de la requête.

- Une requête ne lit jamais de données salies et non committées par une autre transaction.
- Une donnée peut avoir été modifiée par une autre transaction entre deux exécutions de la même requête.

Il est donc possible qu'une lecture soit non répétable et que des données fantômes apparaissent.

Le **niveau Read Only** voit seulement les modifications committées avant le démarrage de la transaction. Ce niveau garantie donc les lectures consistantes.
- On ne peut pas faire d'Insert, Update et Delete dans une transaction en mode Read Only.

Le paramétrage du mode d'isolation se fait au niveau transaction :

```
set transaction read write ; (par défaut)
set transaction read only ;
```

3.4.2 Mécanisme de verrouillage

Pour gérer les conflits d'accès concurrents aux données, Oracle a mis en place un système **automatique** de contrôle par verrouillage. Le verrouillage est réalisé au **niveau ligne**.

Un verrou exclusif est automatiquement posé sur une donnée lorsque l'on exécute un ordre `SQL SELECT ... FOR UPDATE, INSERT, UPDATE, DELETE`.

- De cette manière, aucune autre transaction ne peut modifier cette donnée tant que le verrou est posé.
- La durée du verrou est celle de la transaction.
- Le verrou est relâché sur l'exécution d'un Commit ou d'un Rollback.

Oracle détecte automatiquement les situations de verrou mortel (deadlock). L'une des transactions participant au deadlock est défaite par Oracle.

3.4.3 Les différents types de verrous utilisés par Oracle

Les ordres SQL de manipulation de données peuvent acquérir des verrous à deux niveaux :

- niveau ligne (TX)
- niveau table (TM)

Lorsqu'une transaction acquière un verrou sur la ligne qu'elle veut modifier, elle obtient également un verrou sur la table toute entière.

- Ceci empêche toute autre transaction d'exécuter un ordre DDL (data description language) tant que le verrou n'est pas relâché.

- Par exemple, une autre transaction peut vouloir supprimer (`DROP`) la table ou en modifier sa structure (`ALTER`) alors que la modification apportée par la transaction n'est pas committée.

Les verrous sur tables sont de type :

- RS (Row Share) : il est posé sur la table par une transaction qui a posé des verrous sur des lignes dans l'intention de les modifier. Ce verrou empêche toute autre transaction de verrouiller la table en mode exclusif.
- S (Share) : ce verrou est posé manuellement par l'instruction `LOCK TABLE table IN SHARE MODE ;`. Il empêche toute autre transaction de modifier le contenu de la table verrouillée. Seule la transaction qui a posé le verrou Share peut modifier le contenu de la table dans le cas où elle est la seule à détenir ce type de verrou. Dans le cas contraire, aucune des transactions détenant le verrou Share ne peut modifier le contenu de la table tant que les autres n'auront pas libérer leur verrou. Ce verrou ne garantit donc pas un accès exclusif en écriture.
- SRX (Share Mode Exclusif) : ce verrou est posé manuellement. Même effet que les verrous Share mais il ne peut être posé que par une seule transaction.
- RX (Row Exclusive) : ce verrou est en général posé sur la table par une transaction qui a réalisé une ou plusieurs mises à jour sur des lignes de la table. Il est plus restrictif que le verrou Row Share pour les autres transactions. Il empêche les verrouillages en mode Share (S) et Exclusive (X)
- X (Exclusive) : ce verrou ne peut être obtenu que par une seule transaction. Il lui confère un droit d'écriture exclusif dans la table. Il n'autorise aux autres transactions qu'un accès en lecture. Les ordres DDL de définition du schéma **drop table** et **alter table** posent implicitement un verrou X sur la table.

Les ordres DDL sont automatiquement commités.

Requête SQL	Type de verrou table posé	Modes de verrouillage compatibles				
		RS	RX	S	SRX	X
SELECT ... FROM table	aucun	oui	oui	oui	oui	oui
INSERT INTO table ...	RX	oui	oui	non	non	non
UPDATE table ...	RX	oui*	oui*	non	non	non
DELETE FROM table ...	RX	oui*	oui*	non	non	non
SELECT ... FROM table FOR UPDATE	RS	oui*	oui*	oui*	oui*	non
LOCK TABLE table IN ROW SHARE MODE	RS	oui	oui	oui	oui	non
LOCK TABLE table IN ROW EXCLUSIVE MODE	RX	oui	oui	non	non	non
LOCK TABLE table IN SHARE MODE	S	oui	non	oui	non	non
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE	SRX	oui	non	non	non	non
LOCK TABLE table IN EXCLUSIVE MODE	X	non	non	non	non	non

FIG. 4.10: **Tableau récapitulatif des verrous table posées automatiquement à l'exécution d'un ordre SQL**. Les oui* représentent des verrouillages table compatibles. Mais, s'il y a conflit sur le verrouillage ligne, la transaction qui veut poser le verrou doit quand même attendre.

4. Techniques de récupération d'erreur

Dans cette section, nous voyons comment garantir l'intégrité physique des données validées en assurant une reprise correcte dans tous les cas de panne du système.

Une panne est un évènement logique ou physique qui provoque une fin anormale de transaction - hors du cas normal d'abandon "conscient". On peut classer les pannes en :

- **Panne de transaction** due à la la transaction elle-même (si elle demande une action illégale : une division par zéro ou une modification violant une contrainte d'intégrité) ou du fait du SGBD (si il ne peut résoudre un "dead-lock" dans la concurrence de deux transactions). Dans les deux cas, un rollback de la transaction peut être automatiquement déclenché et complètement effectué.
- **Panne de système ou de mémoire principale**, qui malheureusement est toujours "volatile" : le système a en quelque sorte "perdu la mémoire" ou ne sait plus la relire correctement. Une des parties de la mémoire consacrée soit aux données, soit aux journaux, soit aux codes des gestionnaires eux-mêmes, est devenue illisible. Le système doit redémarrer "à chaud" à partir du dernier point de reprise ("checkpoint") qui correspond à une écriture en mémoire secondaire de toutes les pages mises à jour en mémoire principale.
- **Panne de média ou de mémoire secondaire** : un volume de disque ou un fichier ou certains blocs d'un fichier ne sont plus accessibles ou ne sont plus corrects. S'ils ne concernent que la base de données, on peut les reconstituer à partir du dernier "backup" (sauvegarde) qui correspond à une écriture sur un autre disque, ou sur une bande, de tout ou partie de la base de données physique et des journaux. C'est un redémarrage "à froid". S'ils concernent la base et tout ou partie des journaux depuis la dernière sauvegarde, la panne est dite "catastrophique" et seule une réparation manuelle par l'administrateur de la base est envisageable.

4.0.4 Les redondances nécessaires

4.0.5 Mémoires "sûres" et mémoires "fiables" (ou "à haute disponibilité")

Les *mémoires sûres* supposent que l'écriture physique d'une unité d'enregistrement ("page" ou "bloc") soit atomique, c'est-à-dire ou totalement et correctement exécutée ou pas exécutée du tout. Les *mémoires fiables* sont d'abord sûres et, de plus, se corrigent le plus souvent elles-mêmes des défaillances physiques, grâce à une redondance et des comparaisons systématiques.

Les mémoires secondaires fiables les plus répandues aujourd'hui utilisent la technologie RAID : Redundant Array of Inexpensive Disks.

4.0.6 Base de données, journaux, sauvegardes et archives

Les mémoires secondaires doivent être de fiabilité croissante : si la base de données physique est non sûre, les journaux doivent être sûrs et les sauvegardes doivent être fiables . Mais, bien sûr, augmenter la fiabilité de toutes les mémoires secondaires améliore la disponibilité (une reprise à froid nécessite - sauf dans le cas de dispositifs de mémoires en miroir - un arrêt d'exploitation). Dans tous les cas, on minimise les risques en utilisant au moins des disques différents pour la BD d'une part et les journaux et sauvegardes d'autre part.. Les archives sont soit des sauvegardes anciennes conservées pour usage historique, soit des déchargement partiels des parties les moins souvent accédées de la base de données, soit enfin des déchargements partiels ou des duplications partielles des journaux quand ils deviennent trop longs.

4.1 Les mécanismes utilisés

Ils sont tous basés sur le couple base de données et journaux. La base de données est toujours considérée sous son aspect physique. Parmi ses états successifs, deux sont des repères essentiels : ceux correspondant à une sauvegarde (duplication physique sur un autre volume de mémoire secondaire) et ceux correspondant à un point de reprise (recopie des pages mises à jour en mémoire principale vers la mémoire secondaire - "flush", faite si possible dans un temps mort transactionnel, c'est-à-dire sans transaction en cours). Le ou les journaux peuvent être conçus logiquement ou physiquement selon que l'on conserve la trace

des transactions (pour défaire ou refaire) ou la trace des données modifiées (images avant ou après les transactions).

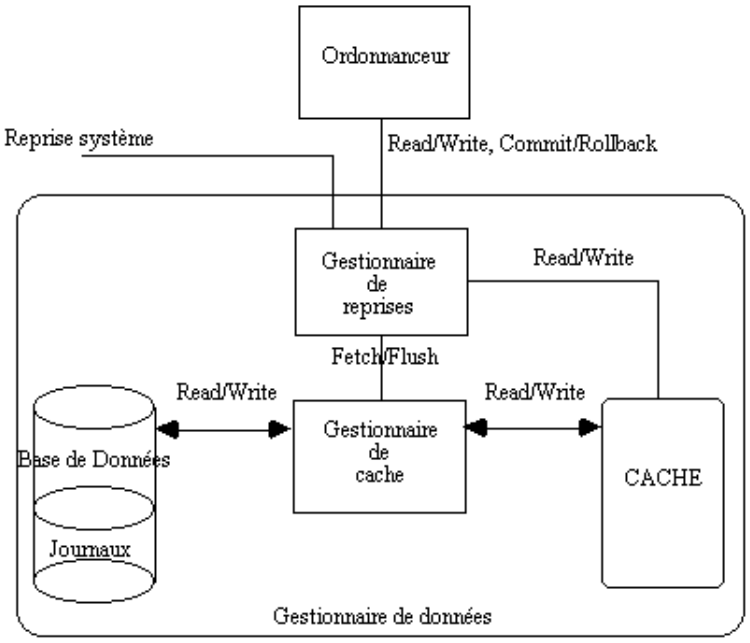

FIG. 4.11: Principe de gestion des reprises après panne.

4.1.1 "do-redo-undo" (faire-refaire-défaire)

Logiquement "faire" l'exécution d'une transaction, comme d'un flux de transactions, c'est lire et écrire des données dans le cache prévu à cet effet dans la mémoire principale puis écrire sur le ou les journaux en mémoire sûre secondaire. Quelques soient les stratégies de transfert des pages de données de la mémoire principale vers la mémoire secondaire (écriture laissée à l'initiative du gestionnaire de cache, écriture forcée au commit, écriture différée jusqu'au prochain point de reprise automatique - en général déclenchée par le constat d'un journal assez long) deux règles doivent être respectées :

1. REFAIRE toujours possible : les IMAGES APRÈS mise à jour doivent être inscrites DANS LA MÉMOIRE SECONDAIRE (journal et éventuellement base de donnée) AVANT la fin de transaction EOT
2. DÉFAIRE toujours possible : les IMAGES AVANT mise à jour doivent être inscrites dans le journal correspondant AVANT que les IMAGES APRÈS soient transférées dans la base de données.

4.1.2 Stratégies

Elles sont logiquement quatre, mais les trois premières seulement sont pratiquement envisageables, selon qu'après une panne REFAIRE et DÉFAIRE sont nécessaires, REFAIRE seul ou DÉFAIRE seul est nécessaire, ni l'un ni l'autre ne sont nécessaires.

Considérons d'abord le cas où le dernier point de reprise correspond à un point mort hors transactions. Une panne non catastrophique de media est reprise à froid par rechargement de la dernière sauvegarde et déroulement du journal des IMAGES APRÈS - ou du REDO log - jusqu'au dernier point de reprise. Ensuite on est dans le même cas qu'une reprise à chaud qui correspondrait à une panne système intervenue après ce point. Il y a deux catégories de transactions à considérer alors, toutes ayant commencé après le point de reprise : celles qui avaient été validées avant, celles qui n'étaient pas encore validées. Comme en général on ne sait pas quelles pages mises à jour par le gestionnaire de cache a ou n'a pas déjà transféré sur la base en mémoire secondaire, il faut d'abord DÉFAIRE les transactions non validées - en remontant

le journal des IMAGES AVANT - jusqu'à leur début BOT, puis REFAIRE les transactions validées - en redescendant le journal des IMAGES APRÈS - jusqu'à leur fin EOT.

Considérons maintenant le cas où le dernier point de reprise avait eu lieu alors que des transactions étaient en cours : la méthode de reprise à chaud ne diffère que par l'étendue de la relecture des journaux car il faut remonter au début BOT de la plus ancienne transaction interrompue par la panne ou commise après le point de reprise mais avant la panne, ce qui pouvait être bien avant le point de reprise ; la reprise à froid elle devra cette fois-ci nécessairement être suivie d'une reprise à chaud. Ainsi, pour garantir un redémarrage depuis une base cohérente, quelque ait pu être la cause de l'arrêt, tous les SGBD font toujours une reprise à chaud après le "startup".

5. Les bancs de mesure des performances ("benchmarks")

La question des performances transactionnelles a été déjà un peu évoquée ci-dessus en termes de temps de réponse et de "débit" de transactions. Définissons un peu plus précisément les grandeurs à mesurer :

- N = *nombre d'utilisateurs actifs simultanément* présents : dans une exploitation normale les utilisateurs ouvrent des sessions au cours desquelles ils effectuent des transactions successives. Du point de vue du SGBD, celles-ci sont seulement interrompues par les temps de lecture, de réflexion et de frappe de l'utilisateur (temps d'inactivité TI pour le système). Si les utilisateurs sont actifs, les sessions comme les transactions sont à tout instant N en parallèle.
- TR = *temps de réponse moyen par transaction* : supposons que la transaction est préparée par remplissage d'un écran qui est transmis en bloc au système, le cycle vécu par chaque utilisateur est un alternat : TI = utilisateur actif et système inactif, TR = utilisateur inactif et système actif. C'est ce deuxième qui constitue le temps réponse du système. La somme TI + TR représente un cycle transactionnel complet.
- TPS = *débit de transaction (par unité de temps)* : si l'on considère le flux global de toutes les transactions, quelque soit l'utilisateur responsable, on peut en mesurer le débit à travers le système par unité de temps — généralement la seconde, d'où le nom Transaction(s)_Par_Seconde. Ce débit est d'autant plus grand que le "taux d'arrivée" est grand et le "temps de service" court.

Il est facile de vérifier que ces trois grandeurs sont liées par la formule de Little : $TPS = N/(TI + TR)$ qui donne précisement la définition du débit d'un SGBD : nombre de transactions en cours divisé par temps de cycle moyen d'une transaction.

5.0.3 Mesures synthétiques et mesures analytiques

Le point de vue de *l'utilisateur* est toujours externe au système — "derrière" le terminal — et synthétique : il ne s'intéresse qu'à des performances globales — moyennes ou extrêmes (le "pire cas") — qui tiennent compte de tous les besoins : dispersion géographique des utilisateurs et des données, volume et distribution statistique des données, complexité du schéma et des requêtes, fiabilité des composants, etc... L'utilisateur doit prendre une décision d'achat et son cahier des charges spécifie en général une configuration pour une application type avec un volume minimum de données, des contraintes d'extrêmes sur les temps de réponse et la disponibilité. Il demande finalement des performances - et des rapports prix/performance - pour ce volume et pour plusieurs fois ce volume (dans le cas très probable où sa BD devra grandir avec le temps).

Le point de vue du *constructeur* est toujours interne au système et analytique du fonctionnement de chacun de ses composants. Pour lui, la performance globale n'est que le résultat de la performance combinée des sous-systèmes de gestion des mémoires, des reprises, d'ordonnancement des transactions, d'optimisation des plans d'exécution des requêtes et des "piles" de protocoles de communication — pour ne citer que les principaux. Pour les caractériser et pouvoir les prédire en exploitation il est amené à construire des bancs de test où, tous les autres sous-systèmes ayant des performances connues par ailleurs, un sous-système est soumis à un véritable plan expérimental. Il s'agira, pour lui, plus d'identifier les coefficients de formules

de coûts que de mesurer les coûts les plus avantageux pour une charge de travail idéale.

Quelque soit le point de vue, les qualités attendues des bancs pour réaliser la mesure de ces performances — les "benchmarks" - sont les mêmes :

- **pertinence** : le benchmark doit bien spécifier le domaine et la signification de la performance qu'il entend mesurer ; il doit être reproductible, étalonnable et le plus précis possible.
- **portabilité** : on doit pouvoir l'appliquer à une grande variété de configurations, pour comparer les résultats de plates-formes différentes sous un même SGBD, de SGBD différents sur une même plate-forme.
- **dimensionnabilité** : on doit pouvoir l'appliquer, pour une configuration donnée, à des tailles variables de BD et avec des quantités variables de ressources CPU et/ou disques, afin de tester les linéarités performance/volume BD à ressources constantes, performance/ressources à volume BD constant, volume BD/ressources à performance constante.
- **simplicité** : malgré les trois précédentes exigences, le benchmark doit rester simple à comprendre — au moins pour les "spectateurs" — et pas trop coûteux à implémenter pour les participants à la compétition !

5.0.4 Les benchmarks du Transaction Processing Council (TPC)

A fin des années 80, un "benchmark" pour les systèmes transactionnels était de plus en plus utilisé et ses résultats discutés tant par les utilisateurs et les journalistes spécialisés que par les constructeurs. Il était issu des propositions d'un article de Anon et al. de 1985 dans DATAMATION pour la mesure du "transaction processing power" des systèmes et portait le nom de "débit-crédit" ou "TP1".

Devant la bataille des chiffres sans juge ni arbitre qui fit rage alors, la plupart des grands constructeurs — une cinquantaine à ce jour — ont préféré se réunir en "concile" pour stabiliser, dans des spécifications longuement discutées et précisées dans le détail, et finalement soumises à un vote formel en 1990, les définitions de deux benchmarks inspirés du TP1 : les TPC-A et B. Ces benchmarks bien que ne représentant qu'une application très simple — un guichet automatique de banque ne faisant que des retraits ou des dépôts — ont été considérés comme des références obligées par presque tous les éditeurs de SGBD et ont eu une grande influence sur leur compétition. Les tps-A ou B, débits exprimés en transactions par seconde conformément aux spécifications des benchmarks TPC-A ou B, et les rapports prix/performance exprimés en milliers de US$ par tps-A ou B, ont fourni les chiffres de palmarès largement diffusés par la presse, bien que tout le monde s'accordait déjà pour trouver l'application "débit-crédit" comme très peu représentative des applications réelles des utilisateurs professionnels : l'accès exclusivement article par article ne donne aucun avantage aux SGBD relationnels par rapport aux SGBD réseaux ni à ces derniers par rapport aux SGF séquentiels indexés traditionnels. Ainsi ces benchmarks sont vite apparus plus dépendants d' aspects système généraux — multiplexage des terminaux pour le TPC-A, rapidité et/ou parallélisation des accès disques pour le TPC-B — que d'aspects proprement SGBD — optimisation et gestion transactionnelle des données.

C'est pourquoi le TPC a mis en chantier deux autres spécifications l'une pour le "business transaction processing" avec le TPC-C (1992), l'autre pour le "decision support" avec le TPC-D (1995). Toutes deux offrent un schéma et des requêtes plus complexes et plus représentatifs des applications réelles. Leur sémantique commune est inspirée des applications de gestion de clients, commandes et stocks. Les métriques en sont respectivement des tpm-C (transactions par minute pour le TPC-C) et des qph-D (requêtes — queries — par heure pour le TPC-D). D'autres spécifications ciblant plus particulièrement les gros serveurs BD et les configurations clients-serveurs sont en cours de discussion.

5.0.5 Le réglage ("tuning") des applications/SGBD réelles

Obtenir les meilleures performances d'une configuration complète — terminaux, système de communication, SGBD, OS, plate-forme mono ou multiprocesseurs, batterie de disques — est un art d'expert quand il s'agit des benchmarks de compétition, c'est un métier encore relativement complexe quand il ne s'agit "que" d'administrer des systèmes de bases de données réelles. Outre une excellente connaissance de la métabase — le "schéma des schémas" — et des outils de surveillance et contrôle en cours d'exploitation — "monitoring" — il faut à l'administrateur des règles et une stratégie pour les optimisations locales

aux applications et globale au système — le fameux "débit". Ce qui nécessite un apprentissage lourd, d'autant plus qu'il n'existe pas de modèles de simulation et de prévision qui soient à la fois universels et détaillés.

Les éditeurs de SGBD eux-mêmes, ou des sociétés indépendantes d'ingénierie du logiciel, proposent des interfaces dédiées aux administrateurs de SGBD qui varient de la simple manipulation visuelle des données de la métabase à de véritables systèmes-experts dont la base de modèles et de règles peut être très riche - et chère !

6. Sécurité et autorisations

Les fonctions qui opèrent sur les tables doivent être définies par catégories d'utilisateurs. À cette fin, les commandes `SQL GRANT` et `REVOKE` permettent de gérer les droits d'utilisateurs.

Les privilèges accordés à l'utilisateur par `GRANT` peuvent lui être retirés par `REVOKE` :

```
GRANT   privilege ON relation TO user ;

REVOKE  privilege ON relation FROM user ;
```

La commande `GRANT` *met à jour la liste des privilèges* pour que l'ayant droit puisse lire, insérer ou supprimer les données dans des tables ou des vues spécifiques. À l'inverse, la commande `REVOKE` permet de retirer à l'utilisateur un privilège qui lui a été accordé.

Par exemple, si nous seulement autoriser la lecture de la vue *Employe* :

```
GRANT SELECT ON Employe TO PUBLIC ;
/* droit de lecture accordé à tout le monde */
```

En spécifiant `PUBLIC` au lieu d'une liste d'utilisateurs, nous accordons le droit de lecture à tous sans aucune restriction.

Les privilèges peuvent être accordés de manière sélective. dans l'exemple suivant, le droit de mise à jour de la table *Service* est accordée exclusivement à un responsable du personnel identifié par le numéro d'utilisateur ID37289 :

```
GRANT UPDATE ON Service TO ID37289 WITH GRANT OPTION ;
```

L'utilisateur ID37289 peut alors modifier la table *Service*. En outre, avec `GRANT OPTION`, il a la compétence de transmettre ce privilège ou un droit de lecture restreint aux autres utilisateurs, et aussi de les retirer ultérieurement. Cela permet de définir et de gérer l'interdépendance des droits.

7. L'interface C/SQL

Cette section présente l'intégration de SQL et d'un langage de programmation classique, ici C. L'utilité de l'utilisation conjointe de deux langages résulte de l'insuffisance de SQL en matière de traitement de données : boucles, tests, etc. On utilise donc SQL pour extraire les données de la base, et le langage de programmation pour manipuler les données. La difficulté principale consiste à transcrire des données stockées selon des types SQL en données manipulables par le langage de programmation.

Dans la suite, nous donnons un exemple complet d'un petit programme avec des commentaires expliquant les aspects liés à l'interfaçage avec C.

7.1 Un exemple complet

On suppose qu'il existe dans la base une table `Film` dont voici le schéma :

```
CREATE TABLE film (ID_film        NUMBER(10) NOT NULL,
                   Titre          VARCHAR (30),
                   Annee          NUMBER(4),
                   ID_Realisateur NUMBER(10)) ;
```

On donne dans la suite un programme qui se connecte et recherche le film d'id `1`. Les numéros figurant en fin de ligne servent de référence 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_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 :  % 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 liens avec les librairies impliquées.

On donne ci-dessous les commentaires correspondant au programme décrit plus haut.

1. Cette ligne est spécifique à Oracle qui 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 principale 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 type `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 **suivie de la variable indicatrice**. Attention, le SGBD génère une erreur si on lit une valeur nulle sans utiliser de variable indicatice.

9. Gestion des erreurs : le champ `sqlcode` de la structure `sqlca` est mis à jour après chaque ordre SQL. Quand il vaut 0, c'est que l'on n'a pas rencontré d'erreur. La valeur 1403 (spécifique d'Oracle) indique qu'aucune ligne n'a été trouvée. Toute valeur négative indique une erreur, auquel cas le message se trouve dans `sqlca.sqlerrm.sqlerrmc`.

À 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'à EXEC SQL WHENEVER SQL ERROR ...  */

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 préparer le code gérant les accès à la base du code implantant l'application proprement dite. Quelques suggestions sont données dans la section suivante.

7.1.1 Développement en C/SQL

La recherche d'information dans une table est une bonne occasion d'isoler une partie bien spécifique du code créant une fonction chargée d'accéder à cette table, de vérifier la validité des données extraites de la base, d'effectuer les conversions nécessaire, 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 des stratégies qui semblent 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 paramètres les critères de recherche.
2. Gérer l'ouverture et la feermeture 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 varaible 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
    }
}
```

Et 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 fonction 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.

7.2 Autres commandes SQL

Voici, à titre de complément, les principales fonctionnalités d'accès à une base de données et leur expression en C/SQL.

Validation et annulation

1. Validation : `EXEC SQL COMMIT WORK...`
2. Annulation : `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`. En 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 scène est inconnu */
    ...
```

8. Les environnements de programmation d'applications sur les bases de données

Tout commence et tout finit par l'utilisateur. Trois fois dans les chapitres précédents le point de vue de l'utilisateur a été explicitement au départ des progrès demandés au SGBD :

- comme concepteur et développeur : ses exigences conduisent à des modèles et langages de richesse sémantique de plus en plus grande et indépendants de l'implémentation physique ;
- comme usager non informaticien : la prise en compte de sa vision propre des données - liée à ses droits et responsabilités - conduit à la définition des vues externes, logiquement indépendantes du schéma du concepteur ;
- comme opérateur sur terminal : son temps propre - de lecture, de réflexion, de frappe et/ou "cliqué" - définit le "temps réel" que devra respecter au mieux le système et lui fixe ainsi ses performances.

8.1 La "programmation visuelle"

8.1.1 Les interprêteurs de SQL ou de QBE

Quand l'administrateur d'un SGBD, ou le développeur d'application, veut interroger directement la BD, il utilise un interprêteur de SQL. Celui-ci admet généralement l' entrée de texte de requête en mode ligne ou en mode plein écran et peut formater les sorties en tableau ; des instructions supplémentaires au SQL standard permettent le contrôle du texte entré et de l'affichage du résultat. Mais on ne peut pas mettre une telle interface entre les mains d'un non-informaticien. Aussi les éditeurs de SGBD proposent-ils des interprêteurs de requêtes qui guident l'utilisateur par une succession de choix à faire sur un schéma graphique des relations (rectangles) et de leurs possibles jointures (arcs joignant les rectangles). Des menus permettent de compléter logiquement les prédicats WHERE de restriction du SELECT. L'utilisateur peut, avant ou après exécution de sa requête, visualiser le texte SQL automatiquement généré en tâche de fond.

Le langage QBE avait, au contraire du SQL, été conçu pour être graphique dès l'origine, mais malgré sa puissance d'expression et les enrichissement picturaux qui sont proposés encore à ce jour par des chercheurs, il représente encore une voie peu suivie par les éditeurs de SGBD.

8.1.2 Les outils QBF et dérivés OO

Presque tous les SGBD offrent des générateurs de formulaires d'affichage de type "Query By Form" faits de champs interactifs qui acceptent des valeurs ou des prédicats restrictifs mono-table et des jointures entre tables, mais implicitement prédéfinies par le choix des tables affichées. C'est, bien sûr, sémantiquement inférieur au QBE, mais acceptable pour des applications simples et stables.

Enrichis avec la panoplie des objets graphiques et avec les mêmes principes de guidage que pour les aides à la formulation de requêtes SQL, on trouve, aujourd'hui, des dérivés "orientés objet" du QBF très ergonomiques.

8.1.3 Les "langages visuels"

On appelle ainsi un ensemble de langages dont les éléments terminaux sont des objets graphiques dans le plan de l'écran et les expressions des assemblages construits par déplacements et "cliqués" de la souris, les identifiants d'objets ou d'attributs et leurs valeurs restant des éléments textuels ajoutés. Si certaines de leurs règles de construction sont reprises dans les outils QBF et dérivés, leur utilisation exclusive pour les GUI reste du domaine de la recherche : leur conception autant que la vérification de leur richesse sémantique prend toujours comme référence un des langages relationnels SQL ou QBE ou un modèle sémantique général comme l'Entité-Association.

8.2 Langages de quatrième génération

Ces langages, mi-visuels mi-textuels, le plus souvent interprétés, sont nés en même temps que les SGBD Relationnels. Ils ont pour principal objectif de permettre le développement d'applications sur terminaux semi-graphiques par maquettage-prototypage beaucoup plus rapidement que par programmation et compilation avec les langages classiques - COBOL, PL/1, Pascal, etc... - relégués au rang de "langages de 3ème génération" par les promoteurs de ces nouveaux outils. Leur simplicité apparente d'utilisation avait pour revers leur grande diversité de syntaxe et de capacité sémantique. Aujourd'hui l'existence et l'utilisation par ces outils de bibliothèques semi-graphiques ou graphiques standards et le renforcement de la norme SQL ont créé une convergence notable.

Concernant directement les développeurs d'applications, les langages de quatrième génération visent à faire gagner du temps et à baisser les coûts d'écriture des programmes. Ils utilisent largement les possibilités des SGBD relationnels à travers une manipulation essentiellement ensembliste des données. Ils ne constituent cependant pas simplement un langage de manipulation de données comme SQL, mais intègrent un ensemble complet de possibilités de traitement en fournissant des structures de contrôle d'un langage structuré de haut niveau (boucle, condition, parcours...). Ils s'efforcent, de surcroît, d'intégrer un ensemble d'outils (générateurs de rapports, de graphiques, d'applications...) souvent disponibles comme des logiciels à adjoindre au SGBD. Orientés vers la programmation visuelle, ils constituent parfois un véritable environnement de développement et sont aujourd'hui l'objet d'une forte demande.

Il ne faut cependant pas perdre de vue que ces langages modernes ne constituent qu'une couche de manipulation des informations stockées dans un SGBD. La qualité, les performances, la puissance d'expression résulte, en dernière analyse, essentiellement à la qualité du SGBD relationnel sous-jacent.

Signalons de surcroît qu'un L4G ne fournit aucun support à une gestion de transactions (voir chapitre 6), nécessaire à tout usage multi-utilisateurs : cette gestion de transactions doit alors être prise en compte au niveau de l'application. Les L4G ne sont pas l'objet d'une définition standard acceptée par tous : à ce jour, le choix d'un L4G particulier condamne l'utilisateur à une dépendance totale envers son fournisseur. Certains L4G génèrent cependant du SQL standard, élément favorable qui peut être pris en compte dans le choix.

8.3 Les interfaces au standard SQL et la portabilité

8.3.1 Une portabilité encore bien relative

Un développeur d'application ne peut pas ne pas avoir le soucis de la portabilité, par rapport à la variété des terminaux et stations semi-graphiques et graphiques, d'une part, et par rapport à la variété des SGBD, d'autre part. Il y a deux façons de le faire : soit il programme, par exemple en C, en utilisant une bibliothèque de procédures standards pour les entrées-sorties graphiques et l'Embedded SQL pour les entrées-sorties de données de la base, soit il programme en L4G et récupère, après mise au point complète et traduction, un programme intermédiaire en C incluant des appels de procédures de X11 Motif et des blocs codés en Embedded SQL. Mais deux limitations de portabilité subsistent : - les spécifications des procédures de la bibliothèque semi-graphique curses(3v) ou de la bibliothèque graphique X11 Motif assurent la portabilité, et même, dans le monde Unix standard, l'indépendance physique par rapport à l'écran. Elles ne valent néanmoins pas pour le monde PC sous MS Windows, ni pour le monde Macintosh.

- la précompilation, qui traduit les blocs de code Embedded SQL en appels de procédures de définition ou de manipulation des données de la base, ne peut se faire qu'en présence d'un SGBD et le code C obtenu reste ensuite lié par ces appels de procédures à ce SGBD particulier.

8.3.2 Vers une ouverture plus grande

Selon que l'application regarde vers l'interface graphique ou vers la BD l'effort pour une plus grande ouverture n'est pas de même nature : - le code de l'application s'exécute sur le PC ou la station donc avec l'unique window manager de ce système, le problème pour le développeur est donc seulement un problème classique de compilation multi-cibles à partir d'un code source commun. La spécification d'une bibliothèque source graphique commune relève donc du choix de l'éditeur de L4G et de son client pour le temps du développement seulement.

- par contre une compilation d'Embedded SQL ne pouvant se faire qu'avec un SGBD donné, ce PC ou cette station ne peut plus, pendant l'exploitation, qu'accéder à ce SGBD. Il faudrait simultanément lancer l'exécution d'un autre code compilé - du même source, mais avec un autre SGBD - pour accéder à un autre SGBD depuis le même PC ou la même station. Ce besoin d'interconnectivité a été très tôt exprimé par les clients, aussi les éditeurs de L4G se sont-ils concertés dès après l'adoption du SQL1 pour spécifier un niveau commun d'appels de procédures pour manipuler les BD. Il a été adopté en 1991 par le consortium X/Open sous le nom de Call Level Interface (ou CLI) et est candidat à la normalisation par l'ISO. On voit ainsi se préfigurer symétriquement deux bibliothèques programmatiques standards - ou Application Programmatic Interfaces (API) - une pour les entrées-sorties côté terminal, l'autre pour les entrées-sorties côté SGBD, qui, ensemble, donneront le maximum de portabilité et d'interconnectivité aux applications sur BD.

8.4 Les ateliers de génie logiciel (AGL / CASE)

Les cibles pour le développement d'applications multi-utilisateurs sur BD apparaissent maintenant complètement :

- la métabase - Information Schema - du SGBD : c'est le réceptacle des définitions logiques et physiques des tables, vues, contraintes d'intégrité, droits, index, procédures et d'une façon générale de tout ce qui peut constituer le dictionnaire de définition et de manipulation cataloguée des données ;
- les PC ou stations et leurs bibliothèques graphiques : ils recevront les codes exécutables des programmes applicatifs, de préférence par téléchargement et configuration automatique ;
- la base elle-même : avec nécessairement son "peuplement" initial de données, mais aussi, dans des "annexes" spécialement prévues à cet effet, les "images" des applications qui seront sauvegardées ou exportées en même temps que les données, ce qui, somme toute, peut rendre de grands services pour la sécurisation du système applicatif tout entier.

Quels outils rassembler pour une telle production logicielle et dans quel atelier les trouver ou les intégrer ?

8.4.1 Outils détachés

Quel est la panoplie minimale d'outils ? En général, on souhaite au moins y trouver, dans l'ordre des productions citées ci-dessus :

- un outil d'aide à la conception du schéma logique des données, doté de plus ou moins d'expertise, éventuellement étendu pour l'aide à la conception du schéma physique ;
- des générateurs d'applications pour interfaces graphiques mais aussi des générateurs de rapports imprimés ;
- des chargeurs de données pour le peuplement initial de la BD à partir de fichier externes de formats variés ; auquels il faut ajouter les outils d'exploitation pour les administrateurs du SGBD et des BD : - un moniteur des ressources statiquement ou dynamiquement allouées ;
- des outils d'export et d'import .

8.4.2 Ateliers intégrés

Que l'on suive ou non des méthodes générales de gestion des projets de développement logiciel - MERISE ou l'une des nouvelles méthodes Orientée Objet - on peut vouloir se doter d'un environnement intégré pour rassembler de façon cohérente, avec une gestion minimale des versions des configurations - SCCS, SRCS ou mieux - toutes les pièces de la production d'une application sur BD, sans oublier l'aide en ligne, la documentation papier et les jeux et procédures de validation.

Trois stratégies sont alors possibles :

- l'achat de l'atelier d'un éditeur de SGBD qui fait de l'intégration verticale, avec le confort d'une validation garantie par celui-ci pour toutes les étapes du cycle de production d'applications sur ce SGBD, et, également, avec les garanties de portabilité et d'ouverture offertes par cet éditeur, mais avec le danger d'une dépendance grandissante par rapport à cet éditeur (ses limites seront vos limites) ;

- l'achat d'un atelier indépendant par rapport aux différents SGBD comme par rapport aux différentes interfaces graphiques, mais aussi par rapport à l'intégration des autres outils détachés de production, comme les outils de "middleware" pour les architectures clients-serveurs ;
- la constitution de son propre atelier sur un standard ouvert comme par exemple le "Portable Common Tool Environment" (PCTE) sous Unix, permettant l'intégration de tous les outils souhaités dont aucun n'imposera son environnement aux autres.

Chapitre 5

Réalisation physique des bases de données

Sommaire

1. Introduction

Généralement, les bases de données sont trop volumineuses pour tenir en mémoire centrale.

Par ailleurs, les opérations routinières d'un SGBD incluent :

1. *Insertion* d'un enregistrement ;
2. *Recherche* d'un enregistrement ;
3. *Mise à jour* d'un enregistrement ;
4. *Destruction* d'un enregistrement.

Ces opérations peuvent donc impliquer un nombre, parfois très grand, d'accès à la mémoire secondaire qui est beaucoup plus lente que la mémoire centrale (dans un rapport d'environ 100 000).

Afin d'accélérer l'ensemble du processus, on essaye d'améliorer les performances des supports mémoires. Mais cela serait largement insuffisant pour changer la nature du problème, et on a également recours à des techniques d'organisation particulières des informations permettant de diminuer les accès en mémoire secondaire. L'une des structures les plus employée consiste à définir des index.

Ces index sont essentiellement à la charge de l'administrateur ou du concepteur de la base de données. Il est donc important de bien les comprendre.

Pour résumer : Les données stockées sur le disque sont organisées en fichiers d'enregistrements.
- Il faut pouvoir les localiser rapidement
- et minimiser le nombre d'accès car la mémoire secondaire est lente.

2. Techniques de stockage

2.1 Stockage de données

2.1.1 Les supports physiques

- Mémoire centrale ou primaire. DRAM et RAM.
- Mémoire secondaire. Disques magnétiques, CD-ROM, bandes magnétiques, ...

2.1.2 Fonctionnement d'un disque et enregistrement des données

Un disque est divisé en *secteurs*, un secteur constituant la plus petite surface d'adressage. On sait donc lire ou écrire des zones débutant sur un secteur et couvrant un nombre entier de secteurs. La taille d'un secteur est le plus souvent de 512 octets.

Le système d'exploitation fixe une unité d'entrée/sortie éventuellement supérieure à la taille d'un secteur, et multiple de cette dernière. On obtient des *blocs*, dont la taille est typiquement 512 octets (un secteur), 1024 octets (deux secteurs) ou 4096 octets (huit secteurs).

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

Toute lecture ou toute écriture sur les disques s'effectue par blocs, et ceci même si la donnée manipulée ne concerne que 4 octets par exemple. Tout le bloc contenant ces 4 octets sera transmis en mémoire centrale. Un des objectifs des SGBD est de s'arranger pour que lorsqu'un bloc est transmis, les données *a priori* superflues transmises en même temps que la donnée désirée, aient cependant de grandes chances d'être

utiles a court terme alors qu'elles seront encore chargées en mémoire centrale. Ainsi des informations corrélées logiquement devraient être stockées sur des segments proches physiquement. Cette motivation est à la base du mécanisme de *regroupement* qui fonde, notamment, les structures d'index et de hachage.

Compromis sur la taille des pages
- Grande taille -> les données liées à x ont des chances d'être stockées sur la même page, d'où réduction des accès aux pages
- Petite taille -> réduit le temps de transfert et réduit la taille du buffer en mémoire centrale
- Taille typique : 4096 octets

Contrairement à une bande magnétique, par exemple, les disques sont à *accès direct*.

La lecture d'un bloc, étant donnée 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 fixes et que 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 de ces 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 peut devenir important quand des milliers de blocs doivent être lus. Le mécanisme d'écriture est à peu près semblable, mais peut prendre un peu plus de temps si le contrôleur vérifie que l'écriture s'est faite correctement.

2.1.3 Technologie RAID

Un système RAID (« *Redundant Array of Independent Disks* ») est un ensemble de disques configurés de telle manière qu'ils apparaissent comme un seul disque avec :
- Une *vitesse de transfert accrue*
  - Des demandes à des disques différents peuvent être traitées indépendamment
  - Si une requête concerne des données stockées séparément sur plusieurs disques, les données peuvent être transférées en parallèle
- Une *plus grande fiabilité*
  - Les données sont stockées de manière redondante
  - Si un disque tombe en panne, le système peut continuer à opérer

2.2 Organisation des fichiers

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*.

Sauf exception (par exemple, MysQL qui a choisi le parti-pris d'une simplicité maximale), les SGBD ont leur propre module de gestion de fichiers et de mémoire cache, à côté de celui utilisé par le système d'exploitation hôte.

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 peut, ou pas, d'espace vide dans le fichier.
- **Temps.** Une bonne organisation doit favoriser les opérations sur un fichier. En pratique, on s'intéresse plus particulièrement à la recherche d'un enregistrement, notamment parce que cette opération conditionne l'efficacité de la mise à jour et de la destruction. Il ne faut cependant pas négliger le coût des insertions.

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

L'efficacité en temps d'une organisation de fichier se définit en fonction d'une opération donnée (par exemple, l'insertion ou la recherche) et se mesure par le rapport entre le nombre de blocs lus et la taille totale du fichier. Pour une recherche par exemple, il faut, dans le pire cas, lire tous les blocs du fichier

pour trouver une enregistrement, ce qui donne une complexité linéaire en la taille du fichier. Certaines organisations de fichier permettent d'effectuer des recherches en temps sous-linéaire : arbres-B (temps logarithmique) et hachage (temps constant), par exemple.

Il existe plusieurs **organisations primaires de fichiers**, qui déterminent la façon dont les enregistrements sont *placés physiquement* sur le disque, *et donc la façon dont il possible d'accéder à ces enregistrements.*

- Dans un *fichier plat* (ou *fichier non ordonné*), les enregistrements sont placés sur le disque sans observer d'ordre particulier et les nouveaux enregistrements sont simplement ajoutés en fin de fichier.
- Dans un *fichier trié* (ou *fichier séquentiel*), ils sont ordonnés en fonction de la valeur d'un champ particulier nommé *clé de tri.*
- Un *fichier haché* utilise une fonction de hachage appliquée à un champ particulier (la clé de hachage).
- D'autres organisations primaires, les arbres-B par exemple, utilisent des structures arborescentes.

Une **organisation secondaire**, ou **structure d'accès auxiliaire**, permet d'accéder rapidement aux enregistrements en se basant sur d'*autres champs* que ceux qui ont servi pour l'organisation primaire.

2.3 Organisation primaire des fichiers

2.3.1 Fchiers séquentiels non ordonnés (« *heap files* »)

Type d'organisation le plus simple. Les enregistrements sont placés dans le fichier dans l'ordre de leur insertion.

Insertions/modifications d'un tuple :
- En moyenne $F/2$ pages (ou blocs) sont lues (donc transférées) si la page existe déjà
- $F + 1$ pages transférées si la page n'existe pas

Effacement d'un tuple :
- En moyenne $F/2$ pages sont lues (donc transférées) si la page existe déjà
- F pages transférées si la page n'existe pas

2.3.2 Fchiers séquentiels ordonnés (« *sorted files* »)

Les enregistrements sont ordonnés physiquement sur disque en fonction des valeurs d'un de leurs champs (*champ d'ordonnancement*).

Si le champ d'ordonnancement est la clé primaire, le champ est appelé *clé d'ordonnancement.*

Avantages :
- Les requêtes basées sur le champ d'ordonnancement ont un coût moyen de $\log_2 F$ par *recherche binaire* ;
- Si les requêtes concernent des tuples successifs, l'*utilisation de cache* est très efficace

Aucun avantage si l'accès se fait par d'autres attributs que le champ d'ordonnancement.

Problème (*maintien de l'ordre*) :
- Coût moyen : $\lfloor \log_2 F \rfloor$ lectures (pour trouver l'endroit par recherche dichotomique) + $F/2$ écritures (afin de pousser les enregistrements suivants)

Solution partielle 1 : Laisser des espaces libres

Solution partielle 2 : Utilisation de pages d'*overflow* (et tri de temps en temps)

Désavantages
- Les pages successives ne sont plus nécessairement stockées à la suite
- Les pages overflow ne sont pas dans l'ordre

On attend des périodes de moindre activité de la base de données pour re-trier les enregistrements.

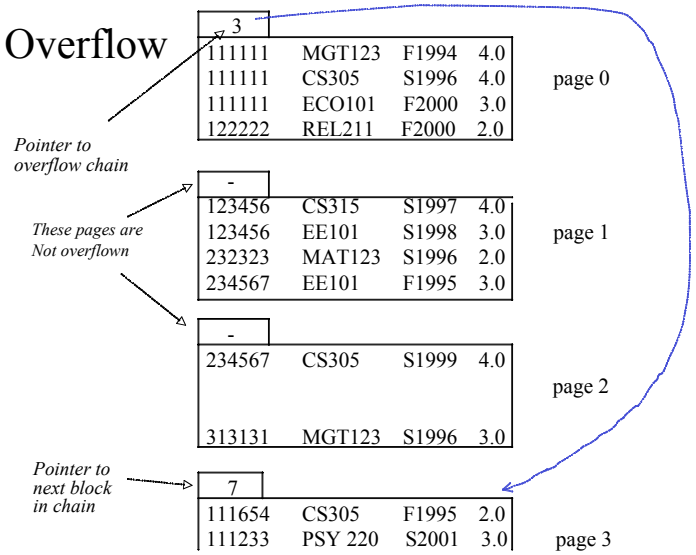

FIG. 5.1: *Organisation de fichier séquentiel avec Overflow.*

2.4 L'exemple du SGBD Oracle

2.4.1 Fichiers et blocs

2.4.2 Les *tablespaces*

2.4.3 Création des tables

3. Structures de données pour optimiser les accès

Étant donné que les bases de données sont généralement de taille importante, elles doivent être stockées en mémoire secondaire, même lors des traitements. De ce fait, et pour optimiser les temps de réaction, le but est en particulier de minimiser le nombre d'accès aux unités de mémoire externe lors des opérations de lecture ou d'écriture dans les tables.

Une organisation de fichier réussie devrait pouvoir exécuter *efficacement* les opérations que nous pensons *appliquer fréquemment* au fichier.

EXEMPLE

- Si condition de recherche fréquente sur le numéro de Sécurité Sociale, il faut favoriser la localisation par rapport à la valeur de `NoSS`.
  ?==> Soit trier par rapport à NoSS, soit définir un index sur NoSS.
- Si, en plus, recherche d'employés par service, il faudrait stocker tous les enregistrements des employés dont la valeur de `SERVICE` est identique de façon contiguë. Mais cet arrangement est en conflit avec les enregistrements classés sur la valeur de `NoSS` ?

==> Il faut trouver un compromis : **optimisation**

3.1 Indexation de fichiers

Types de recherches :

- Recherche par clé
- Recherche par intervalle
- Recherche multi-attributs

3.1.1 Index primaires, de regroupement et secondaires

Un *index ordonné* est un index, généralement construit sur un attribut (ou champ) de la table à indexer (e.g. sa clé primaire), et classé selon le même ordre que les éléments de la table. Par exemple, la table des matières d'un livre est une sorte d'index ordonné. Le fichier d'index étant beaucoup plus petit que le fichier principal, il est beaucoup plus facile à consulter. Le fait qu'il soit ordonné permet d'effectuer une recherche binaire.

Un index est dit **primaire** si le champ de tri sert à ordonner *phyiquement* les enregistrements en mémoire secondaire. Une conséquence est qu'il ne peut y avoir au plus qu'un index primaire.

Si plusieurs enregistrements peuvent avoir la même valeur pour le champ de tri (qui n'est donc pas la clé primaire), on parle d'**index de regroupement**.

Un **index secondaire** sert à accélérer les opérations sur les tables, mais il peut être spécifié sur n'importe quel champ et son classement ne correspond pas à l'organisation physique des enregistrements.

Il ne peut donc y avoir plus d'un seul index primaire alors qu'il peut y avoir plusieurs index secondaires.

Index primaire

Un index primaire est un fichier ordonné dont les enregistrements sont de taille fixe et comportant deux champs $\langle K(i), P(i)\rangle$:

1. Le premier champ est du même type de donnée que celui du champ de la clé de tri (par exemple le `nom` ou le `numero_SS`) ;
2. Le second champ est un pointeur sur un bloc du disque (l'adresse d'un bloc).

Il existe une entrée d'index pour chaque bloc du fichier de données. Chacune de ces entrées contient donc deux champs, d'une part, la valeur du champ de la clé primaire du *premier* enregistrement du bloc adressé, d'autre part le pointeur sur ce bloc (son adresse).

La figure 5.2 montre un tel index. Le nombre total d'entrées dans l'index est identique au nombre de blocs de disque dans le fichier de données ordonné. Le premier enregistrement de chaque bloc est appelé *enregistrement d'ancrage* du bloc ou encore *ancre du bloc*.

Les index peuvent aussi être caractérisés par leur densité. Un *index dense* contient une entrée d'index pour *chacune des valeurs de la clé de recherche* (et donc pour chaque enregistrement) du fichier de données. A l'inverse, un *index non dense* ne contient d'entrée d'index que pour certaines des valeurs sur lesquelles il y a des recherches à effectuer. En conséquence, *un index primaire n'est pas dense* puisqu'il contient une entrée pour chaque bloc de disque du fichier de données et non pour chaque valeur de recherche (ou chaque enregistrement).

Le fichier d'un index primaire nécessite beaucoup moins de blocs que le fichier de données pour deux raisons :

- D'abord, il y a moins d'entrées d'index qu'il n'y a d'enregistrements dans le fichier de données (index non dense) ;
- Ensuite, la taille de chaque enregistrement d'index est normalement bien inférieure à celle d'un enregistrement de données.

Pour extraire un enregistrement étant donné la valeur K du champ de la clé primaire, on entreprend une recherche binaire sur le fichier de l'index pour retrouver l'entrée d'index i appropriée et on récupère le bloc du fichier de données dont l'adresse est $P(i)$ (pointeur associé).

—— Exemple **Taille de l'index primaire** ——

Soit un fichier de 30 000 enregistrements, chacun de 100 octets. Soit des blocs de 1024 octets. Il tient donc 10 enregistrements par bloc, avec une perte de 24 octets pour chaque bloc.

En tout, il faut donc 3 000 blocs pour contenir les données.

Une recherche binaire sur le fichier nécessite à peu près $\lceil \log_2 3000 \rceil = 12$ accès blocs.

Soit un index primaire construit à partir d'un champ de clé de tri de 9 octets et d'un pointeur de bloc de 6 octets, soit 15 octets par entrée d'index. Il y a donc $\lfloor 1024/15 \rfloor = 68$ entrées d'index par bloc. Et il y a 3 000 entrées d'index puisque c'est un index dense. D'où le besoin de $\lceil 3000/68 \rceil = 45$ blocs pour contenir l'index.

Une recherche binaire sur ces blocs nécessite à peu près $\lceil \log_2 45 \rceil = 6$ accès bloc, auxquels il faut ajouter un accès bloc pour aller chercher le bloc de données.

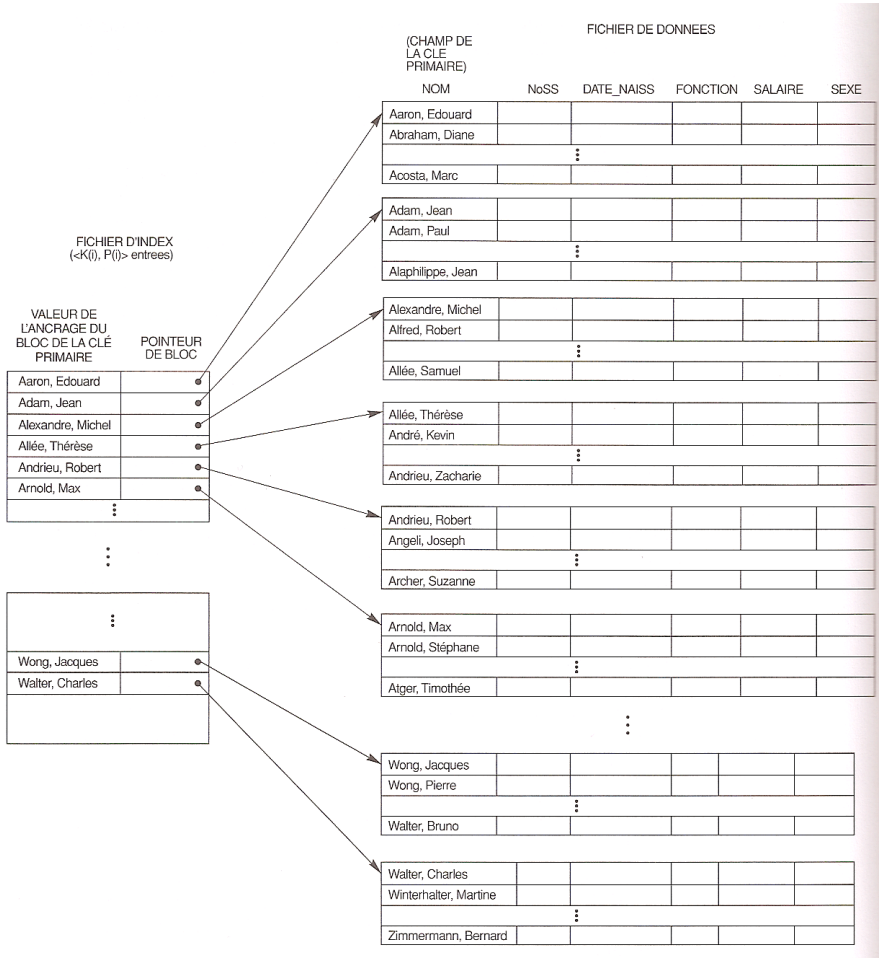

FIG. 5.2: *Exemple d'index primaire.*

Le gain est donc de $5 = 12 - 7$ accès sur cet exemple de taille assez réduite.

Le grand problème posé par un index primaire (comme pour n'importe quel fichier ordonné) est celui de l'insertion et de la suppression d'enregistrements. Un index primaire accroît le problème parce que l'insertion d'un enregistrement à sa position appropriée dans le fichier provoque non seulement le déplacement des enregistrements pour faire de la place à un nouvel enregistrement, mais affecte aussi des entrées d'index, puisque le déplacement d'enregistrements modifie les enregistrements d'ancrage de plusieurs blocs.

L'emploi d'un fichier de débordement (*unordered overflow file*) peut résoudre en partie ce problème.

Index de regroupement

On peut créer des index sur des clés non spécifiques à chaque enregistrement mais qui dont l'ordre est l'ordre physique d'enregistrement des données dans le fichier.

Un index de regroupement est aussi un fichier ordonné contenant des articles à deux champs :

1. Le premier champ est du même type de données que le champ de regroupement qui sert de *champ d'indexation* ;
2. Le second champ contient un pointeur de *bloc* pointant sur le premier enregistrement du bloc où est situé un champ de donné comportant la valeur spécifiée par le premier champ.

L'insertion et la suppression posent les mêmes problèmes que pour un index primaire puisque les enregistrements de données sont physiquement ordonnés selon la clé de regroupement.

Un index de regroupement est un autre exemple d'index non dense.

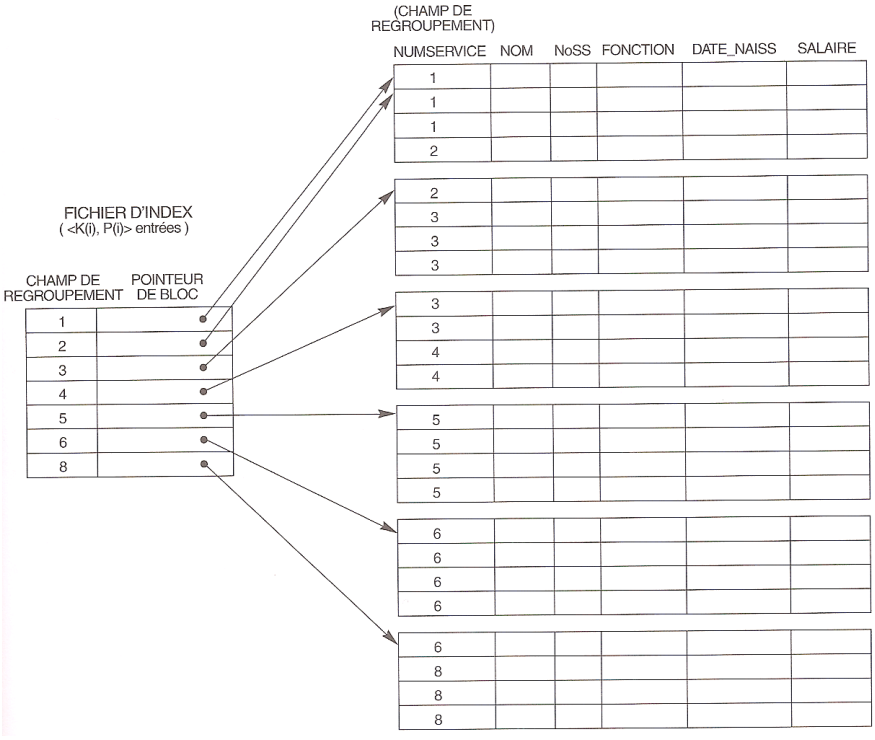

FIG. 5.3: *Exemple d'index de regroupement.*

Index secondaires

Un index secondaire peut porter sur un champ clé qui est une clé candidate et a une valeur unique pour chaque enregistrement, ou bien sur un champ non-clé comportant des valeurs dupliquées.

L'index est un fichier trié avec deux champs :

1. Le premier champ est du même type de données que le champ non ordonné du fichier de données qui sert de *champ d'indexation* ;
2. Le second champ est soit un pointeur de *bloc* (index non dense), soit un pointeur d'*enregistrement* (index dense).

Il peut y avoir plusieurs index secondaires (et par suite plusieurs champs d'indexation) pour un même fichier.

Nous nous intéressons d'abord à une structure d'index secondaire reposant sur un champ clé ayant une valeur distincte pour chaque enregistrement, c'est-à-dire reposant sur une clé secondaire. Il s'agit d'un index dense.

Notons $\langle K(i), P(i)\rangle$ les deux valeurs des champs d'une entrée i d'index. Les entrées sont triées en fonction de la valeur de $K(i)$, ce qui rend possible une recherche binaire. Comme les enregistrements du fichier de données ne sont pas physiquement ordonnés en fonction de la valeur du champ de la clé secondaire, il n'est pas possible d'employer des ancrages de blocs. C'est la raison pour laquelle une entrée d'index est crée pour chaque enregistrement du fichier de données. Mais le pointeur $P(i)$ donne l'adresse du bloc dans lequel se trouve la donnée recherchée. Ce n'est donc que lorsque le bloc approprié est transféré en mémoire centrale qu'il est possible d'entreprendre la recherche de l'enregistrement souhaité à l'intérieur de celui-ci.

Normalement, un index secondaire nécessite davantage d'espace de stockage qu'un index primaire à cause de son plus grand nombre d'entrées (si il est dense). Cependant, l'amélioration du temps de recherche est significative, car sinon, il faudrait entreprendre une recherche séquentielle sur le fichier de données lui-même.

—— EXEMPLE **Taille de l'index secondaire** ——

Idem exemple précédent, sauf que recherche séquentielle sur fichier des données demande en moyenne $3000/2 = 1500$ accès blocs.

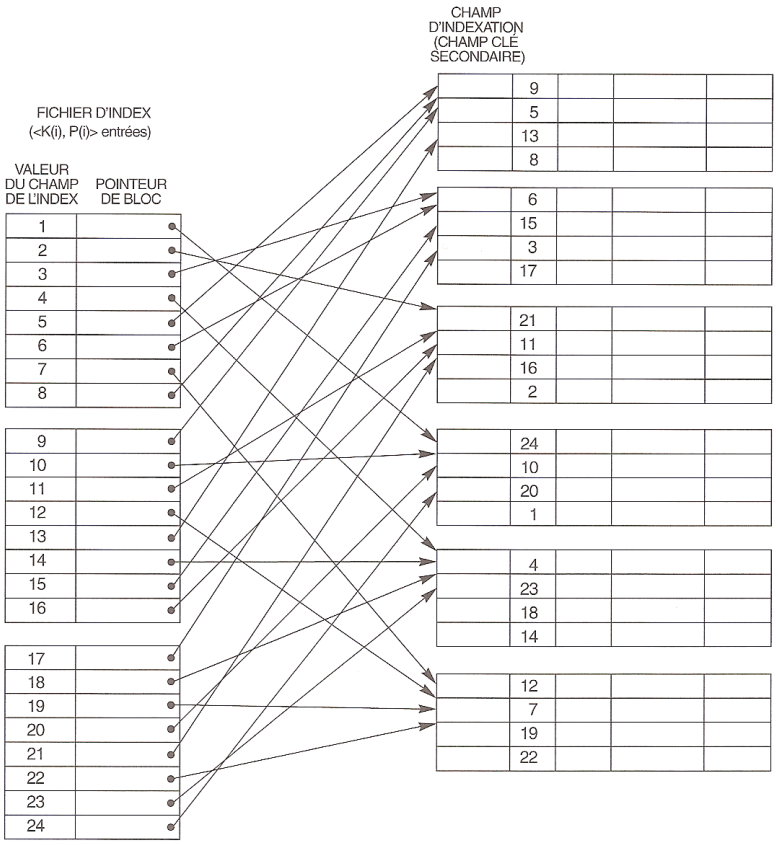

FIG. 5.4: *Exemple d'index secondaire dense (avec des pointeurs de blocs) sur un champ clé non utilisé pour le tri d'un fichier.*

Ave index secondaire (dense), on a 30 000 entrées d'index de 15 bits chacune, et 68 entrées par bloc, soit $\lceil 30000/68 \rceil = 442$ blocs.

Une recherche binaire nécessite donc à peu près $\lceil \log_2 442 \rceil = 9$ accès bloc, auquel il faut ajouter l'accès au bloc de données.

Le gain est donc substanciel : de 1500 à 10.

Il est également possible de créer un index secondaire sur un champ non-clé. Dans ce cas, plusieurs enregistrements du fichier de données peuvent avoir la même valeur pour le champ d'indexation. Plusieurs options existent pour implémenter un tel index. Nous renvoyons le lecteur à [EN00] pp.375-376, par exemple pour obtenir davantage de détails.

Remarque

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

3.1.2 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 soient pénalisées. C'est le cas particulièrement si l'index lui-même est si grand qu'il doit être stocké en mémoire secondaire. La solution naturelle est alors d'indexer le fichier d'index lui-même.

Un index multi-niveaux permet de généraliser l'approche de la recherche par dichotomie. Chaque niveau de l'index est composé d'entrées d'index comprenant deux champs (voir figure 5.7) :

1. Le premier champ est du même type de données que le champ du fichier de données qui sert de *champ d'indexation* ;

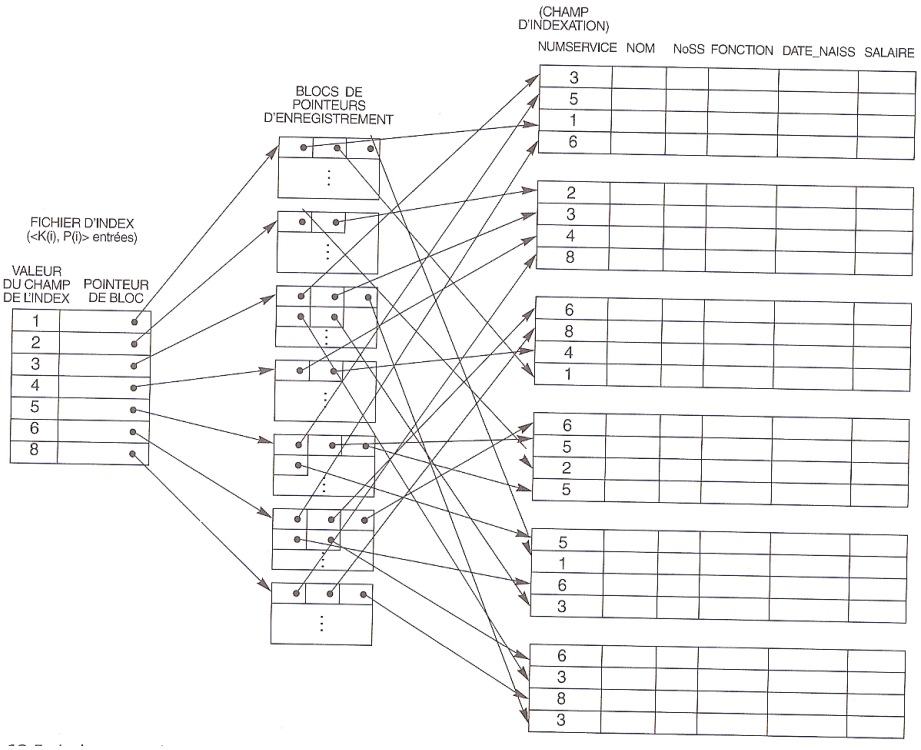

FIG. 5.5: *Exemple d'index secondaire (avec des pointeurs d'enregistrement) sur un champ non clé implémenté avec un niveau d'indirection de manière à ce que les entrées d'index aient une valeur fixe et que la valeur de leur champ soit unique.*

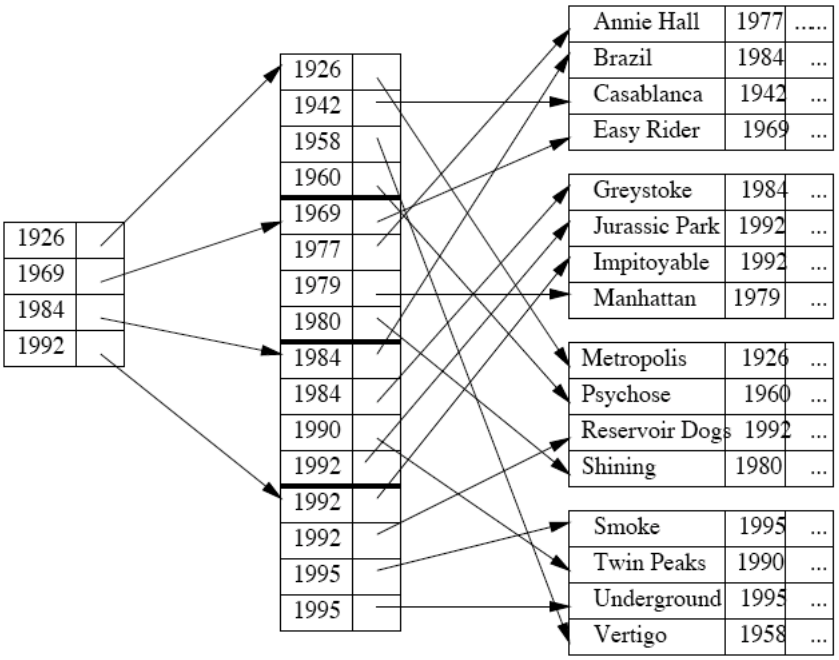

FIG. 5.6: Un index multi-niveaux.

2. Le second champ est un pointeur de *bloc* (index non dense).

À chaque niveau de l'index, le facteur de branchement *br* est contrôlé par le nombre d'entrées d'index pouvant être stockées dans un bloc. Dans la figure 5.7, le facteur de branchement est de 4.

Supposons que le fichier contienne F enregistrements (tuples) et que le nombre d'entrées d'index par bloc soit de br, alors, avec un seul niveau d'indexation, il suffit d'un seul accès à la mémoire secondaire (dans

ce cas, en effet, l'index est supposé tenir en mémoire principale, et il suffit de positionner la condition de requête par rapport aux valeurs de clé dans l'index pour connaître le bloc à charger et à explorer en mémoire secondaire. Un seul niveau est suffisant tant que le nombre de blocs associé à la table en mémoire secondaire est $< br$. Dans le cas contraire, il faut avoir recours à au moins un niveau d'index supplémentaire. Par exemple, avec deux niveaux d'index, il est possible d'accéder à br^2 blocs en mémoire secondaire, et le nombre d'accès pour une requête est de 2 : un accès pour atteindre le bloc pertinent au premier niveau de l'index, puis un accès pour accéder au bloc contenant le ou les enregistrements recherchés.

Le nombre de niveaux nécessaires pour un fichier contenu dans b blocs est ainsi de $\lceil \log_{br} b \rceil$. (E.g. avec un facteur de branchements $br = 10$, il faut 6 niveaux pour indexer un fichier de $1\,000\,000 = 10^6$ blocs).

EXEMPLE **Taille de l'index multi-niveaux**

On suppose que l'index secondaire dense de l'exemple 2 est converti en index multi-niveaux.

Le facteur de branchement est donc de 68 entrées d'index par bloc.

Le nombre de blocs de premier niveau est de $b_1 = 442$. Le nombre de blocs du deuxième niveau de l'index sera donc de $b_2 = \lceil (b_1/br) \rceil = \lceil (442/68) \rceil = 7$ blocs. Le nombre de blocs de troisième niveau sera donc de 1 ($68 > 7$).

Il en résulte que le nombre d'accès nécessaire pour atteindre un bloc de la table indexée est de 3 + 1 (un accès par niveau de l'index (en supposant que le troisième niveau soit lui-même en mémoire secondaire) + 1 accès au bloc de la table).

Ceci est à comparer aux 10 accès nécessaire avec une recherche binaire sur un index à un niveau.

Les index multi-niveaux sont très efficaces en recherche, et ce même pour des jeux de données de très grande taille. Le problème réside, comme toujours, dans la difficulté de maintenir des fichiers triés sans dégradation de performances. L'arbre-B 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'insertions et de suppressions dans les données. Nous y reviendrons dans la section 3.3.

3.2 Hachage

Une *fonction de hachage* (« *hash function* ») permet de transformer une clé en une adresse.

Par exemple, pour un nom d'employé donné, la fonction retournera l'adresse d'un bloc en mémoire où se trouve l'enregistrement associé à cet employé.

Une méthode de transformation simple consiste à attribuer une adresse (représentée par un nombre naturel de 1 à n) à chaque valeur de clé choisie dans une table. Cette adresse est interprétée comme un numéro de page relatif. Chaque page contient un nombre fixe de valeurs de clé, avec ou sans les enregistrements de données correspondants.

La transformation des clés doit satisfaire les critères suivants :

- La méthode de transformation doit comporter des *opérations simples et non onéreuses.*
- Les adresses réservées doivent être *uniformément distribuées* dans leur espace d'adressage.
- La *probabilité d'attribuer une même adresse à plusieurs valeurs de clé* doit être identique pour toutes les valeurs de clés.

Il est à noter que la condition de recherche ne peut être qu'une condition d'égalité sur le champ de hachage.

Il existe une grande variété de fonctions de hachage avec leurs avantages et leurs inconvénients. La méthode la plus connue et la plus simple est le hachage utilisant le reste d'une division.

3.2.1 Principes de base

Supposons que nous ayons M emplacements (e.g. des blocs) dans lesquels stocker nos données. Afin d'utiliser une fonction de hachage, nous devrons disposer d'une table associant à l'entier $\in \{0, \ldots, M-1\}$ une adresse de bloc.

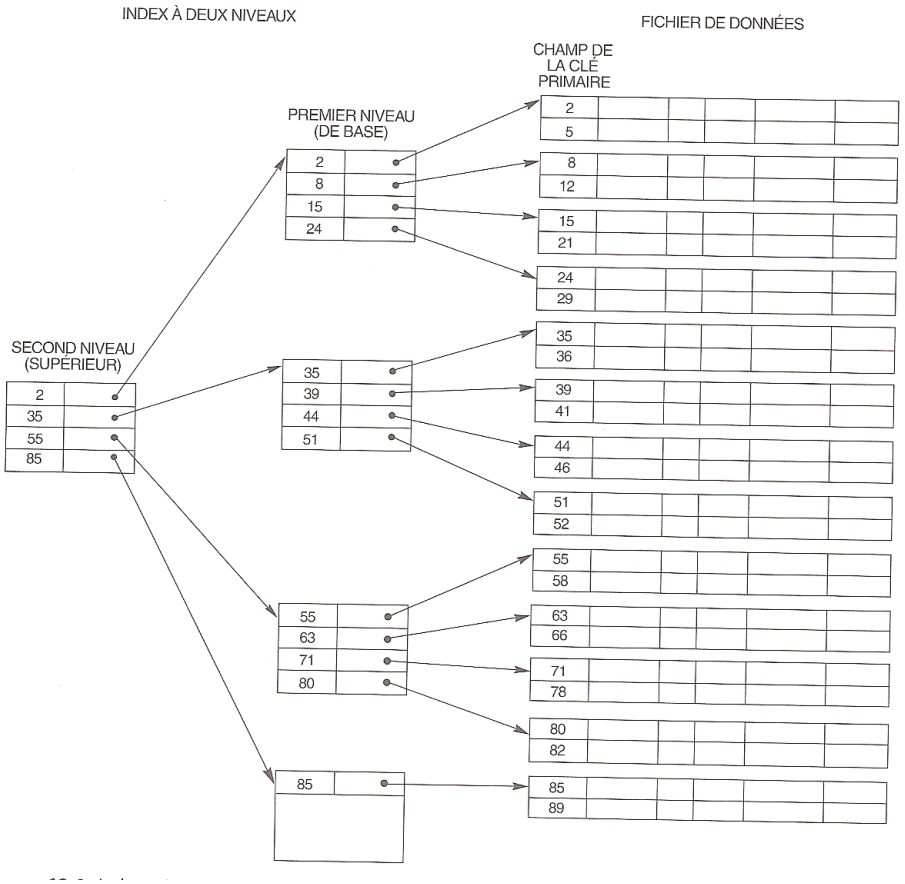

FIG. 5.7: *Exemple d'index primaire à deux niveaux.*

Une fonction de hachage couramment utilisée est :

$$\boxed{h(K) \;=\; f(K) \mod M} \tag{5.1}$$

où K est la clé de hachage et f est une fonction transformant la clé en un entier. On pourrait par exemple imaginer que f calcule la somme des caractères ASCII correspondant à la valeur de la clé de hachage.

En général, on choisit M comme un nombre premier, ce qui assure une meilleure répartition des valeurs de la fonction h.

EXEMPLE **Table de hachage**

Soit un ensemble de films auquel nous voulons accéder grâce à une table de hachage sur le titre.

On suppose que chaque bloc contient au plus quatre films et que l'ensemble des 16 films de la base actuelle occupe donc au moins 4 blocs. Afin d'avoir un plus grand degré de liberté, on affecte 5 blocs, numérotés de 0 à 4, à la collection de films.

On définit une fonction prenant en entrée le titre d'un film et produisant en sortie un numéro de bloc. Cette fonction doit satisfaire aux critères :

1. le résultat de la fonction, pour n'importe quelle chaîne de caractères, doit être une adresse de bloc, soit, ici, un entier compris entre 0 et 4 :
2. la distribution du résultat de la fonction doit être uniforme sur l'intervalle $[0, 4]$.

Si le premier critère est relativement facile à satisfaire, le second est plus problématique car l'ensemble des chaînes de caractères auxquelles s'applique la fonction de hachage possède souvent des propriétés statistiques spécifiques. par exemple, dans le cas des titres de films en français, les chaînes de caractères commencent souvent par "La" ou "Le".

Pour notre exemple, nous utiliserons une méthode simple. On affecte à chaque lettre son rang dans l'alphabet, soit 1 pour 'a', 2 pour 'b', etc. Puis, pour se ramener à une valeur dans $[0, 4]$, on utilisera la fonction *modulo*.

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

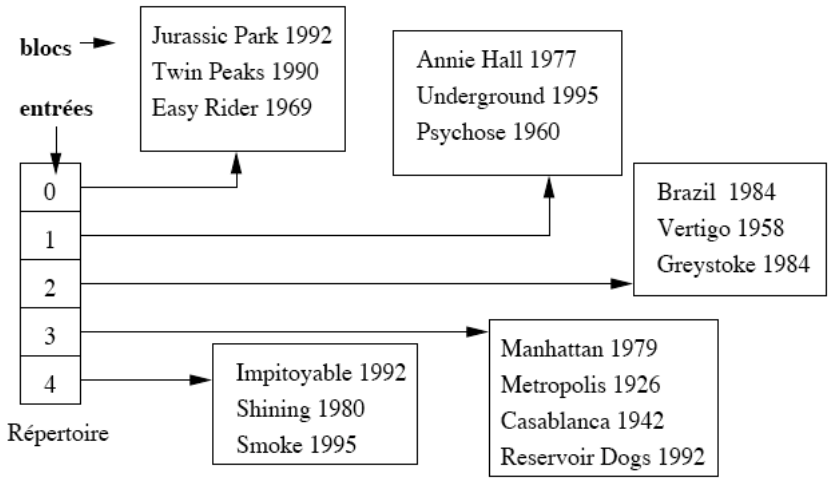

FIG. 5.8: *Exemple de table de hachage.*

La figure 5.8 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, les lettres *b, g, l, q* et *v* sont affectées au bloc 2, etc.
La figure 5.8 présente, outre les 5 blocs stockant les films, un répertoire à 5 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. Ce répertoire est généralement de taille suffisamment limitée pour tenir en mémoire principale.
Cette fonction est garantie de retourner une valeur entre 0 et 4, mais la distribution risque de ne pas être uniforme. Ainsi, le bloc 2 auquel les films commençant par la lettre *l* sont affectés, risque en effet d'être débordé. C'est pourquoi en pratique on utilise un calcul moins sensible à ce type de biais. Par exemple, on considérera les 4 ou 8 premiers caractères du titre. Après avoir associé à ces caractères leur rang dans l'alphabet, on prend la somme et on utilise la fonction de hachage sur cette somme.

3.2.2 Recherche dans une table de hachage

La structure de hachage permet les recherches par titre, ou par le début d'un titre. Par exemple :

```
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* ici, 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 tienne en mémoire centrale, la recherche a donc pu s'effectuer en lisant un seul bloc, ce qui est optimal. On voit donc ici les deux avantages d'une table de hachage :

1. La structure n'occupe pas d'espace mémoire secondaire, contrairement à l'arbre-B, comme nous verrons :
2. Elle permet d'effectuer des recherches par clé par accès direct (calculé) au bloc susceptible de contenir les enregistrements.

En revanche, le hachage ne permet pas d'optimiser des requêtes par intervalle puisque l'organisation des enregistrements ne s'appuie pas sur l'ordre des clés. Ainsi la requête suivante entraîne l'accès à tous les blocs de la structure.

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

Cette incapacité à effectuer efficacement des recherches par intervalle doit mener à préférer l'arbre-B dans tous les cas où ce type de recherche est envisageable. C'est le cas, souvent, d'une clé représentant une date.

3.2.3 Mise à jour d'une table de hachage

La structure en table de hachage est mal adaptée, dans sa version simple, aux mises à jour. Ainsi, dans le cas des insertions, l'espace initial calculé peut ne plus être suffisant et conduite au débordement de blocs. Par exemple, dans la figure 5.9, l'insertion du nouveau film "Citizen Kane" doit se faire dans le bloc 3, mais celui-ci est déjà plein. Il faut alors chaîner de nouveaux blocs. À l'adresse 3 dans le répertoire correspondent maintenant 2 blocs chaînés. Cela signifie qu'au lieu d'un seul accès en mémoire secondaire, il peut maintenant arriver que deux soient nécessaires (voir figure 5.9).

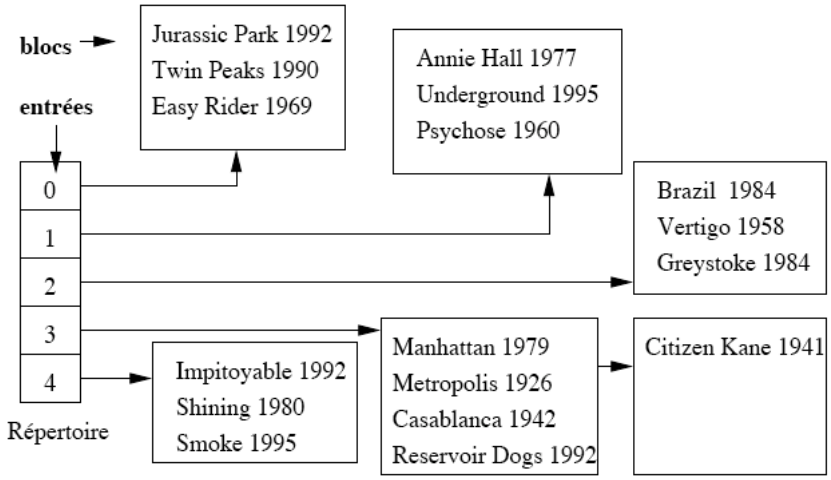

FIG. 5.9: *Exemple de table de hachage.*

Dans le pire cas où on a mal évalué les propriétés statistiques de distribution des lettres des films, on pourrait se retrouver avec un simple fichier séquentiel, avec comme en conséquence une dégradation catastrophique des performances.

Ce type de structure n'est pas dynamique et ne permet pas une évolution en fonction de la croissance de la base de données.

Par ailleurs, le hachage est une structure dite *plaçante*, c'est-à-dire qui dicte le placement en mémoire des enregistrements. Il ne peut donc pas y avoir plusieurs index organisés selon une table de hachage.

3.2.4 Hachage extensible

Le hachage extensible permet d'adapter le placement des données dans les blocs en fonction des insertions passées de manière à éviter les chaînages de blocs préjudiciables aux performances.

Plutôt que d'en donner une description générale, nous en illustrons le principe à l'aide d'un exemple.

Soit une fonction de hachage dont le résultat est un octet, soit 8 bits. La table suivante donne les résultats pour 8 films.

Titre	h(titre)
Vertigo	01110010
Brazil	10100101
Twin Peaks	11001011
Underground	01001001
Easy Rider	00100110
Psychose	01110011
Greystocke	10111001
Shinning	11010011

On suppose que seuls trois films peuvent être stockés par bloc.

La figure 5.10 (à gauche) montre le résultat de l'insertion des 5 premiers films et leur affectation à l'un de deux blocs. Tous les films dont la valeur de hachage commence par 0 sont affectés au bloc 0, tandis que ceux dont la veleur commence par 1 sont affectés au bloc 1.

L'insertion de "Psychose", dont la valeur de hachage est '01110011', entraîne le débordement du bloc 0.

On va alors doubler la taille du répertoire, et le nombre de blocs potentiels. Les nouvelles valeurs seront 00, 01, 10 et 11, avec la nouvelle organisation :

- Les films associée à l'ancienne valeur 0 sont maintenant répartis sur les valeurs 00 et 01 en fonction de la valeur de leurs deux premiers bits de hachage. Ainsi, "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 placés dans le deuxième bloc.
- les films associés à l'ancienne valeur 1 restent sur un seul bloc, de double adresse 10 et 11, puisqu'il n'y a pas de débordement.

Le résultat est illustré sur la figure 5.10 (à droite).

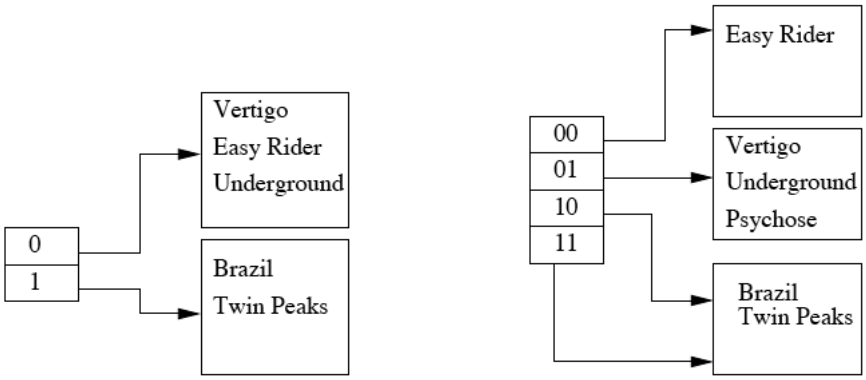

FIG. 5.10: *Exemple de table de hachage extensible à deux entrées (à gauche) ; Exemple de table de hachage extensible après doublement du répertoire (à droite).*

Si les films "Greystocke" (valeur 10111001) et "Shinning" (valeur 11010011) sont alors insérés , il y a débordement du troisième bloc, qui doit alors être scindé en deux. L'un des nouveaux blocs sera associé à la valeur 10, avec les films "Brazil" et "Greystocke", l'autre sera associé avec la valeur 11, avec les films "Twin Peaks" et "Shinning" (voir figure 5.11).

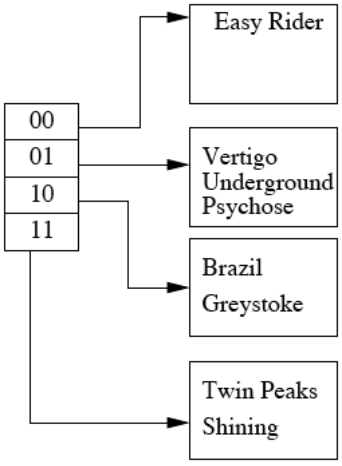

FIG. 5.11: *Étape finale du hachage extensible.*

3.3 L'arbre-B

Un **arbre-B** est un arbre trié et équilibré (B pour *Balanced tree*).

Arbre-B d'ordre m :

- Toutes les *feuilles* sont au même niveau (donc tous les chemins de la racine aux nœuds feuilles ont la même longueur).

- Un *nœud non feuille* ou interne a un nombre de fils $\in [m+1, 2m+1]$.
- Le *nœud racine* a un nombre de fils $\in [0, 2m+1]$.

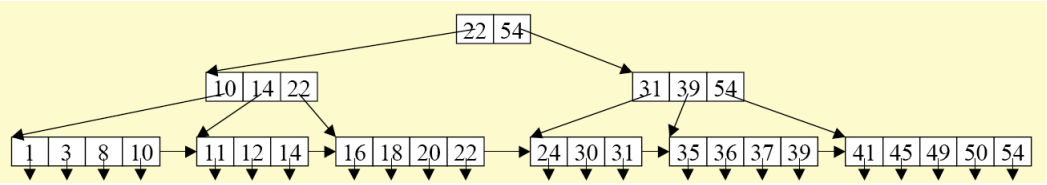

FIG. 5.12: Un arbre B d'ordre 2.

À la différence d'un arbre-B, un arbre-B+ stocke les valeurs (les clés) uniquement dans les feuilles.

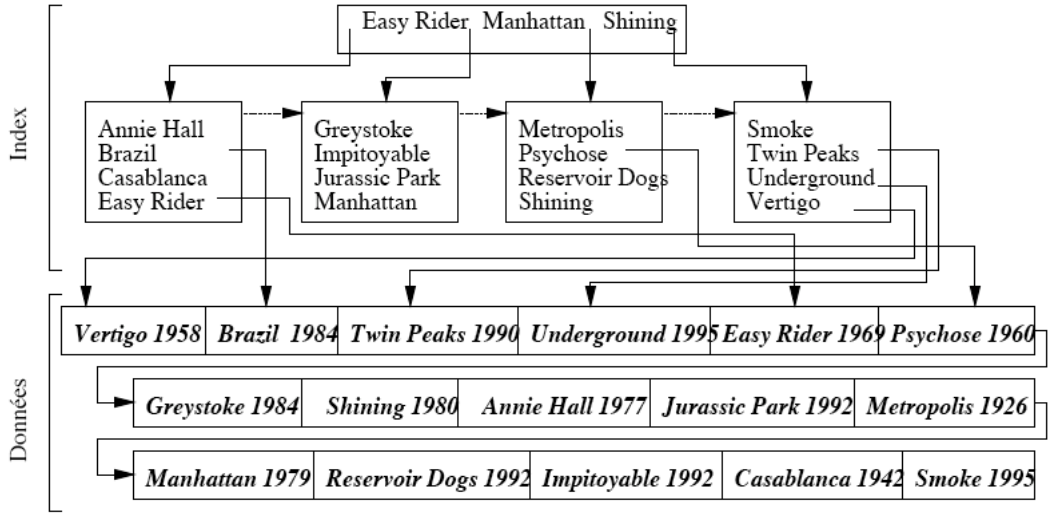

FIG. 5.13: Arbre-B+ pour le fichier des films avec un index unique sur le titres.

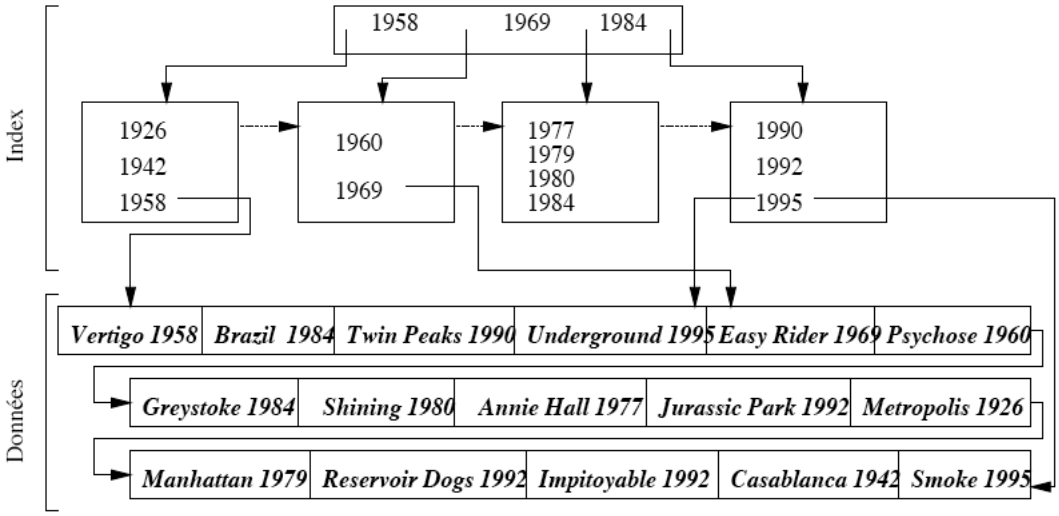

FIG. 5.14: Arbre-B+ pour le fichier des films avec un index unique sur l'année.

3.3.1 Structure d'un arbre-B+

Dans un arbre-B+, les **nœuds feuille** contiennent chacun deux champs :

1. Le premier champ est du même type de données que le champ du fichier de données qui sert de *champ d'indexation* ;

2. Le second champ est soit un pointeur d'*enregistrement* si le champ de recherche est une clé (index dense), soit un pointeur de *bloc* (index non dense).

Les nœuds feuille d'un arbre-B+ sont habituellement chaînés entre eux de manière à offrir un accès ordonné au champ de recherche des enregistrements. Ces nœuds feuilles sont donc équivalents au premier niveau d'un index multi-niveaux.

Les **nœuds internes** correspondent aux autres niveaux d'un index multi-niveaux. Leur structure est la suivante :

1.

3.3.2 Recherches avec un arbre B+

Lors d'une requête spécifiant une valeur K_r du champ de recherche, il y a comparaison de cette valeur avec les valeurs de comparaison K_i du nœud racine, puis récursivement sur chacun des noeuds racine des sous-arbres rencontrés, jusqu'à ce que le nœud soit un nœud feuille. À chaque fois, il y a recherche (éventuellement par dichotomie) du pointeur p_i tel que : $K_{i-1} < K_r \leq K_i$, et suivi de ce pointeur. (Voir figure 5.15).

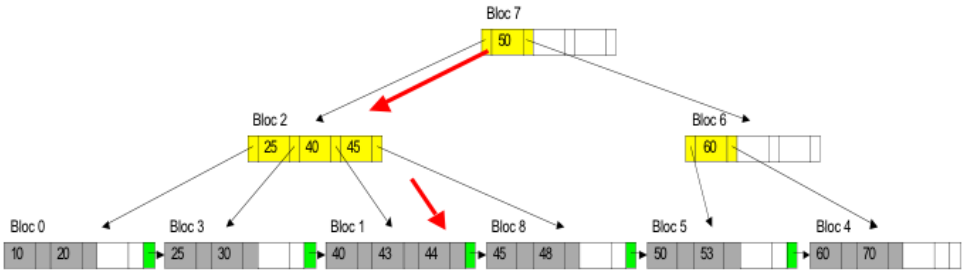

FIG. 5.15: *Recherche d'un élément, ici 43, dans un arbre B+.*

3.3.3 Mise à jour d'un arbre B+

Algorithme d'insertion dans un arbre B+

1. Trouvez la feuille où l'on doit insérer la nouvelle valeur en suivant la règle de <= à gauche (ou >= à droite).
2. Insérer dans la feuille en ordre de tri.
3. *Si la feuille est pleine*, alors il y a débordement (« Overflow »), Il faut *créer un nouveau nœud* :
   a) Insérer les $j = \lfloor (p_{leaf} + 1)/2 \rfloor$ premières entrées dans le nœud original et les autres entrées dans le nouveau nœud.
   b) La $j$ième valeur (+1 pour la règle du >= à droite) de clé est répliquée dans le parent et un nouveau pointeur au nouveau nœud est aussi ajouté au parent.
4. *Si il y a débordement dans le parent*, Il faut *créer un nouveau nœud* :
   a) Les entrées dans le nœud interne jusqu'à P_j restent en place. P_j est le $j$ième pointeur après insertion de la nouvelle valeur et pointeur, où $j = \lfloor (p+1)/2 \rfloor$.
   b) La $j$ième valeur (+1 pour la règle du >= à droite) de clé est déplacée, et non pas répliquée, dans le parent.
   c) Le nouveau nœud va contenir les entrées à partir de $P_j + 1$ jusqu'à la fin des entrées.
   d) Les division peuvent se propager jusqu'à créer une nouvelle racine et par conséquent un nouveau niveau pour l'arbre B+.

Algorithme de suppression dans un arbre B+

1. Lorsqu'une entrée est effacée, elle est d'abord retirée de son nœud terminal (sa feuille).
2. Si cette valeur est aussi dans un nœud interne, elle doit aussi y être retirée. La valeur à sa gauche (à sa droite pour la règle du >= à droite) dans le nœud terminal la remplace dans le nœud interne.

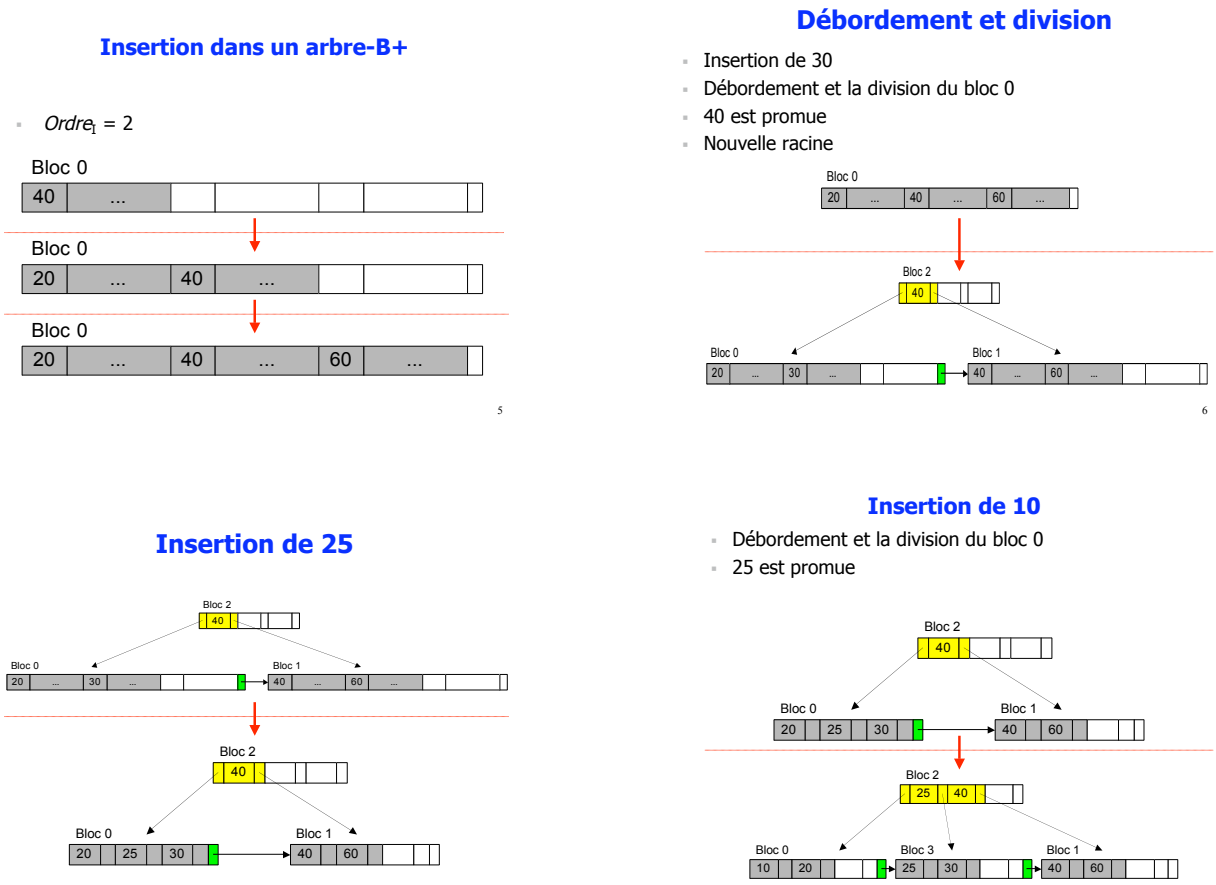

FIG. 5.16: *Séquence d'insertions dans un arbre B+.*

3. L'effacement peut causer une insuffisance de valeur dans le nœud terminal. Dans ce cas, il faut faire une redistribution des valeurs avec les feuilles voisines afin que toutes les feuilles aient le nombre minimum de valeur requis.
4. Si la redistribution est impossible, il faut fusionner des feuilles.
5. Lorsque des feuilles sont fusionnées, les insuffisances de valeur peuvent se propager dans les nœuds internes puisque moins de pointeurs sont requis. Il faut alors comprimer les nœuds internes en suivant les règles de base.
6. Notez que la propagation peut entraîner l'élimination d'un niveau.

3.4 Autres méthodes d'indexation

3.4.1 Les index *bitmap*

Dans un index *bitmap*, on construit un tableau de bits correspondant à un attribut avec autant de colonnes que de valeurs possibles de cet attribut (il est préférable ce que ce nombre soit réduit), et autant de lignes que de tuples dans la table indexée. Un bit de ce tableau sera à 1 si la valeur correspondante de l'attribut fait partie du tuple désigné.

Ainsi, supposons que l'on considère l'attribut *genre* dans la table des films (voir table 5.1) :

L'index *bitmap* correspondant est donné dans le tableau 5.2. Il ne peut y avoir qu'un seul 1 par film (et donc ici par colonne) puisqu'une seule valeur est possible pour chaque attribut pour un tuple.

En général, un index *bitmap* est de très petite taille. De plus, certaines requêtes peuvent être exécutées très efficacement, parfois sans même recourir à la table. Ainsi, par exemple, la réponse à la requête "*Combien de films y a-t-il dont le genre est "Drame" ou "Comédie" ?* peut être calculée directement à partir de l'index *bitmap*, en comptant le nombre de bits à 1 dans les colonnes (ici lignes) correspondant aux valeurs considérées de l'attribut.

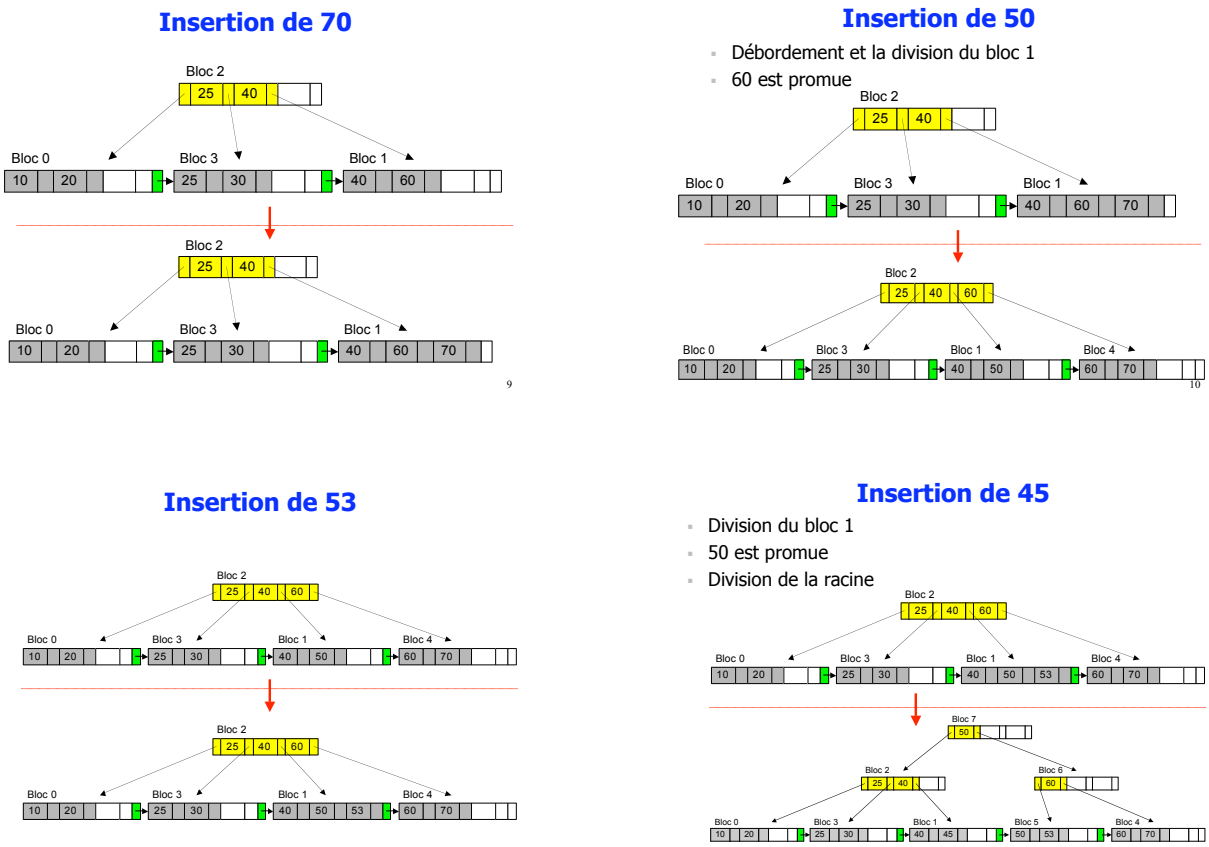

FIG. 5.17: *Séquence d'insertions dans un arbre B+ (suite).*

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

3.4.2 Structures de données multidimensionnelles

3.5 Bilan

3.6 Comparaison

- Les arbre B offrent des recherches rapide en fonction d'une valeur de la clé ; utiles pour les colonnes ayant un très grand nombre de valeurs différentes ; mise à jour ralentie
- Le hachage permet d'accéder à un tuple en une seule E/S
- Les index bitmap sont utiles pour des colonnes comptant peu de valeurs distinctes

3.7 Quelques règles

- Ne pas créer des index pour des tables de moins de 200-300 lignes (l'accès séquentiel est plus rapide)
- Ne pas créer d'index sur une colonne qui ne possède que quelques valeurs différentes (utiliser éventuellement bitmap)
- Indexer les colonnes qui interviennent souvent dans les clauses `WHERE` et `ORDER BY`
- Indexer les colonnes de jointure

3.8 Indexation dans Oracle

3.9 Index en SQL

- Indexation implicite : par défaut, un arbre B+ est créé sur la clé primaire de chaque table

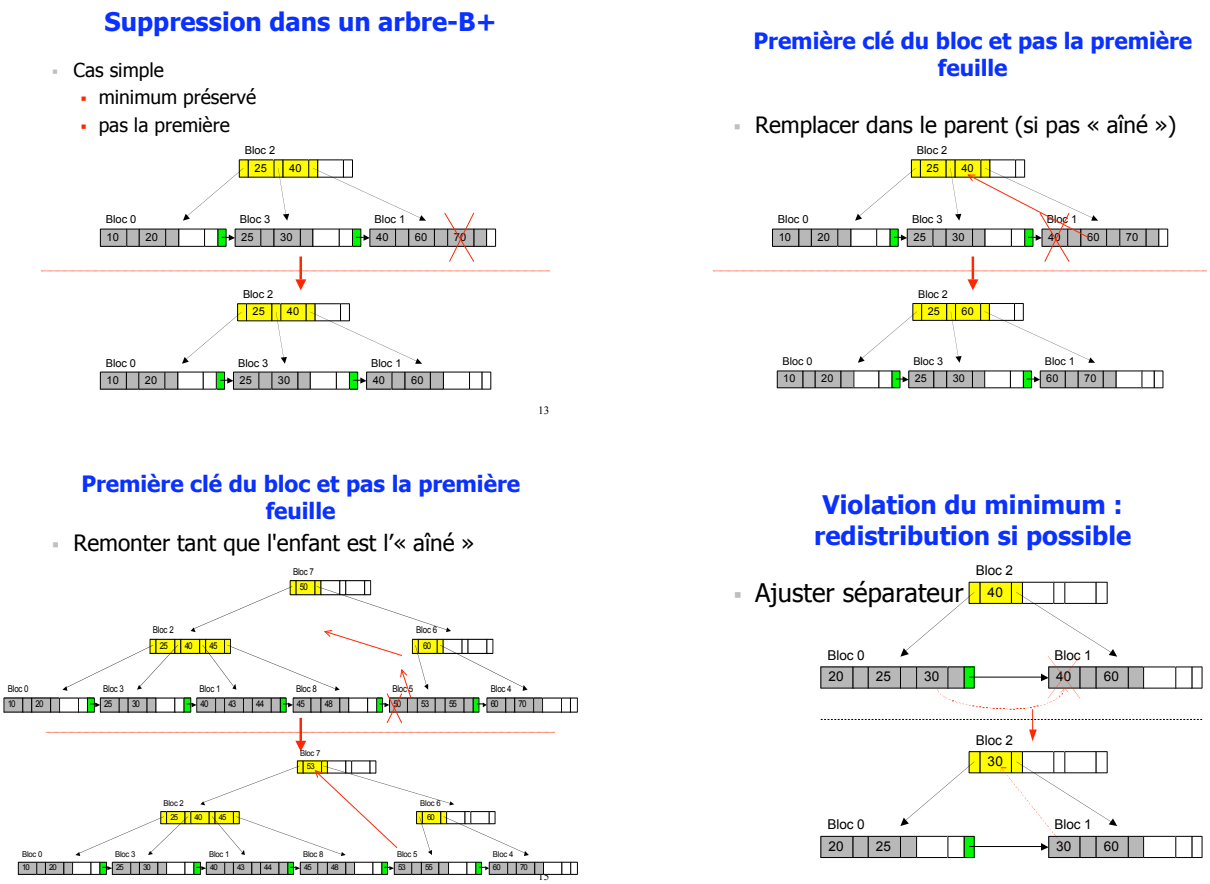

FIG. 5.18: *Séquence de suppressions dans un arbre B+.*

- Indexation explicite : les recherches sur de critères qui ne font pas partie de la clé peuvent être optimisées en créant un nouvel index
  `CREATE [UNIQUE] INDEX nom_index ON nom_table [(nom_colonne)[ASC|DESC], ... ] ;`
  `UNIQUE` : sert à éliminer les doublons / `ASC` : par défaut
  Ex : `CREATE INDEX nom_et ON Etudiants(Nom, Moyenne DESC)`
  `DROP INDEX nom_index`
- Oracle propose plusieurs techniques d'indexation : arbre B, B+, tables de hachage, etc.

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

3.9.1 Arbres B+

La principale structure d'indexation utilisée par Oracle est l'arbre B+. Par défaut, un arbre B+ est créé sur la clé primaire de chaque table. Cet arbre B+ est ensuite automatiquement maintenu lors des insertions, suppressions et mises à jour affectant la table.

On peut aussi optimiser des recherches sur des attributs non clé 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 Artite (non, prenom)
```

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

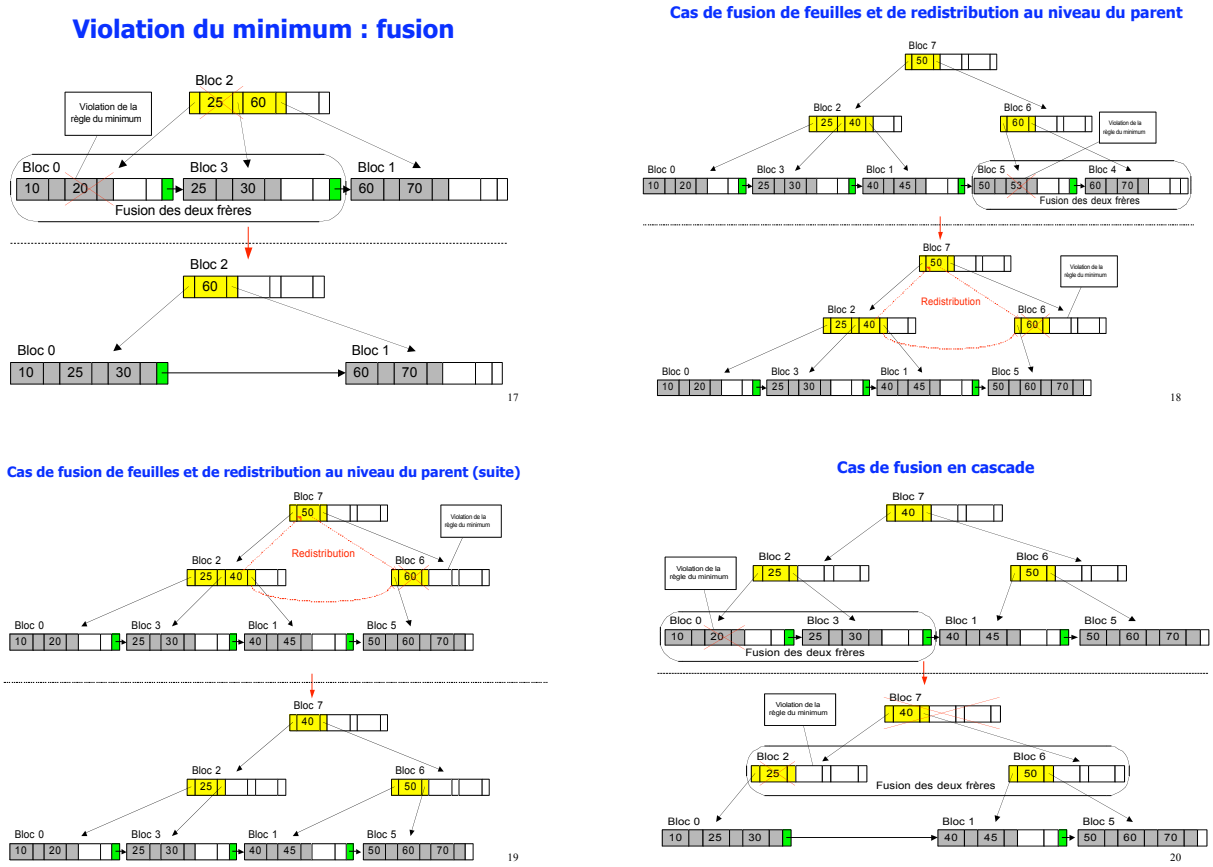

FIG. 5.19: *Séquence de suppressions dans un arbre B+ (2).*

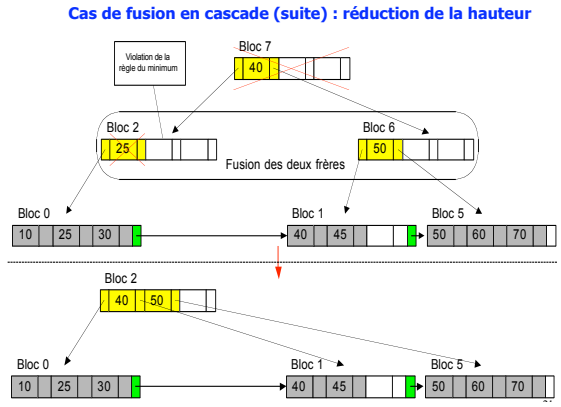

FIG. 5.20: *Séquence de suppressions dans un arbre B+ (3).*

3.9.2 Tables de hachage

La création d'une table de hachage s'effectue en deux étapes. Ondéfinit 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 HashFilms (id INTEGER)
SIZE 500
HASHKEYS 500;

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

Rang	Titre	h(titre)
1	Vertigo	Suspense
2	Brazil	Science-fiction
3	Twin Peaks	Fantastique
4	Underground	Drame
5	Easy Rider	Drame
6	Psychose	Drame
7	Greystocke	Aventures
8	Shinning	Fantastique
9	Annie Hall	Comédie
10	Jurassic Park	Science-fiction
11	Metropolis	Science-fiction
12	Manhattan	Comédie
13	Reservoir dogs	Policier
14	Impitoyable	Western
15	Casablanca	Drame
16	Smoke	Comédie

TAB. 5.1: *Table des films qualifiés par genre.*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Aventures	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Comédie	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1
Drame	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0
Fantastique	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
Science-fiction	0	2	0	0	0	0	0	0	0	1	1	0	0	0	0	0
Suspense	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Western	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

TAB. 5.2: *Index bitmap sur les films classés par genre.*

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

1. *la clé de hachage* est spécifiée dans l'exemple comme étant un `id` de type `INTEGER` ;
2. *le nombre d'entrée* 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 types). Il est cependant possible pour l'administrateur de fournir sa propore 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 4096 octets, Oracle affectera alors $\lfloor \frac{4096}{500} \rfloor = 4$ entrées de la table de hachage à un même bloc. Cela étant, si une entrée occupe à elle seule tout le bloc, les autres seront insérées dans un bloc de débordement.

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

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

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

3.10 Structures de données multidimensionnelles

4. Évaluation des requêtes

4.1 Introduction à l'optimisation des performances

4.1.1 Opérations exprimées par SQL

4.1.2 Traitement d'une requête

4.1.3 Mesure de l'efficacité des opérations

4.2 Algorithmes de base

4.2.1 Recherche dans un fichier (sélection)

4.2.2 Quand doit-on utiliser un index ?

4.2.3 Le tri externe

4.3 Algorithmes de jointure

4.3.1 Jointures par boucles imbriquées

4.3.2 Jointure par tri-fusion

Chapitre 6

Perspectives des SGBD

Sommaire

1. Nouveaux modèles de données pour nouvelles applications

La technologie des bases de données relationnelles domine largement le marché actuellement. Cependant, de nouveaux besoins avec l'élargissement des domaines d'application révèle les limites du modèle relationnel classique et des systèmes de bases de données relationnelles.

- Intégration d'information provenant de la Toile
  - Nombreuses sources de données
  - Hétérogénéité des sources et des formats de données (HTML, XHTML, PDF, ...)

2. Bases de données réparties

Les bases de données réparties ou distribuées (« *distributed databases* ») permettent de réaliser des applications qui nécessitent le stockage, la maintenance et le traitement des données en plusieurs endroits différents. Une base de données est décentralisée ou répartie lorsqu'elle est modélisée par un seul schéma logique de base de données, mais implémentée dans plusieurs fragments de tables physiques sur des sites

géographiquement dispersés et reliés par un réseau. L'utilisateur d'une base de données répartie se focalise sur sa vue logique des données et n'a pas besoin de se préoccuper des fragments physiques. C'est le SGBD qui se charge d'exécuter les opérations, soit localement, soit en les distribuant sur plusieurs ordinateurs en cas de besoin.

2.1 Motivations

- Limiter le transfert d'information entre sites
- Meilleure répartie de charge
- Augmenter la fiabilité (grâce à la duplication des informations)
- Résultat d'une fusion de systèmes d'information initialement séparés.

Exemples.

2.2 Fragmentation d'une BD

Fragmentation horizontale

Fragments définis par sélection. Reconstruction par union des fragments. (Voir figure 6.1).

Pertinente si des sous-tables sont spécifiques de sites séparés (e.g. les employés d'un site de production).

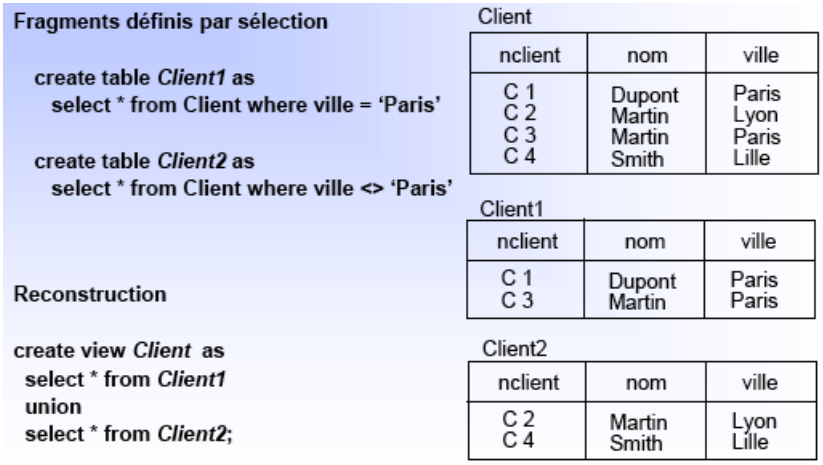

FIG. 6.1: Fragmentation horizontale.

Fragmentation verticale

Fragments définis par projection. Reconstruction par jointure. (Voir figure 6.2).

Pertinente si forte affinité des attributs.

Il faut alors penser à conserver un identificateur de tuple pour chaque vue.

Fragmentation hybride

On parle de *fragmentation disjointe* si il n'y a pas de duplication des informations, et de *fragmentation non disjointe* sinon.

2.3 Les problèmes spécifiques

Coût des communications

Cohérence des données

Contrôle de concurrence

Reprise après panne

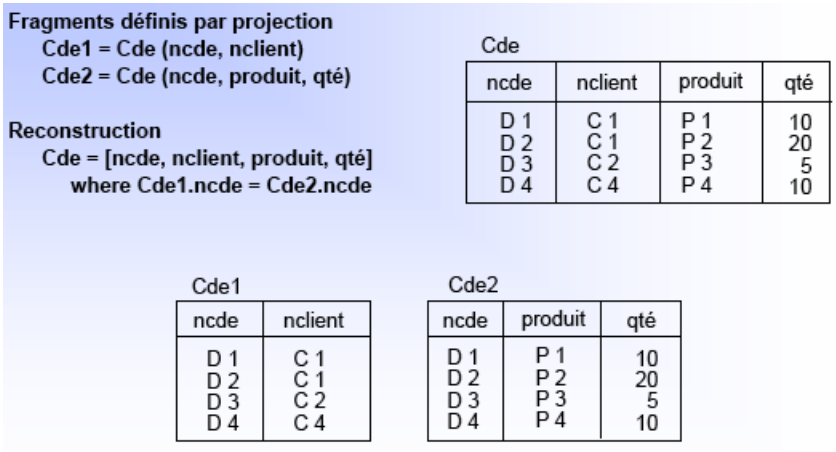

FIG. 6.2: Fragmentation verticale.

Optimisation des requêtes

Répartition des charges.

2.4 Transaction répartie

2.4.1 Transaction en deux phases

2.4.2 Traitement optimal des requêtes réparties

3. Bases de données déductives

4. Entrepôts de données et fouille de données

4.1 Perspectives historiques

Bibliographie générale

Bibliographie

[EN00] R. Elmasri and S. Navathe. *Fundamentals of dabase systems.* Addison-Wesley, 2000.

Index