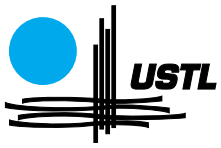


Bases de Données

(De Merise à JDBC)

Philippe Mathieu

Version 1.4

le 16/07/1999

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. 03 20 43 47 24 Télécopie 03 20 43 65 66

Préface

Présentation du livre

J'ai entrepris la réalisation de ce livre lorsque j'ai pris la responsabilité du cours de Bases de données à l'IUT informatique de l'université de Lille 1 en septembre 1997. Je cherchais alors à conseiller un livre à mes étudiants et je me suis vite rendu à l'évidence qu'il n'existait aucun recueil qui leur convenait. Il y a pourtant de nombreux livres très bien faits dans ce domaine (la bibliographie en cite quelques uns) mais ils sont généralement trop spécialisés. Il aurait fallu en acheter 10 et ne lire que 3 ou 4 chapitres de chacun ! Les étudiants n'avaient pas besoin de connaître tous les détails de la méthode Merise comme dans [Gab89], toutes les subtilités de SQL2 comme dans [Del95], toutes les notions théoriques liées aux langages d'interrogation comme dans [Ull88]ou le détail d'implémentation d'un serveur d'applications comme dans [Ree97]. Il leur fallait un livre d'introduction (en français :-) qui présente la réalité des bases de données d'aujourd'hui et leur mette le pied à l'étrier pour être ensuite capables de tirer pleinement parti des livres cités en bibliographie. Ce livre est donc volontairement incomplet dans chacun des chapitres qui le constituent. Ne vous attendez pas à y trouver les notions de "modèle conceptuel de traitement", la syntaxe complète de l'ordre SQL CREATE dans les différents SGBD du marché, l'ensemble des ordres VBA permettant l'écriture de curseurs sous MS-ACCESS ou même les méthodes nécessaires à l'implémentation d'un serveur d'application à l'aide des RMI.

A qui s'adresse ce livre ?

Ce livre s'adresse à tous les étudiants et développeurs dans le domaine des bases de données qui cherchent à avoir une vue d'ensemble des différentes technologies liées à ce sujet, sans pour autant rentrer dans les détails profonds de chaque sous-domaine. Il peut être indifféremment lu et étudié par des étudiants d'IUT, de MIAGE ou même de DESS selon les spécialités. Les rappels de base en informatique ne sont pas faits dans ce livre, aussi un niveau minimum d'au moins une année en informatique est nécessaire. Ce livre ne s'adresse pas aux personnes ayant déjà une bonne vue d'ensemble sur le domaine et qui cherchent une spécialisation poussée dans un sous-domaine, aux personnes qui débutent en informatique et aux personnes qui cherchent un n-ième guide "comment faire mes formulaires avec MS-ACCESS".

En quoi sont implémentés les exemples ?

J'ai cherché, en écrivant ce livre, à rendre tous les exemples accessibles au plus grand nombre. A priori vous n'avez pas besoin d'un SGBD particulier pour comprendre et traiter les exemples détaillés. Néanmoins, afin d'illustrer les différentes parties j'ai utilisé MS-ACCESS pour sa convivialité et sa simplicité d'utilisation et ORACLE v7 (et parfois un AS400) pour les accès en Client-Serveur. Cela dit, je pense m'être détaché au maximum d'un quelconque système et avoir fourni des exemples pouvant être réalisés sur n'importe quel SGBDR bien conçu.

Pourquoi est-ce disponible sur le WEB ?

Depuis l'avènement d'Internet, une vague de gratuité sans précédents est apparue. On trouve de plus en plus de systèmes d'exploitation, de logiciels et même de livres remarquables disponibles gratuitement.

Pour le lecteur, ce monde idyllique a un revers : vous devez savoir charger ce fichier, imprimer un fichier Postscript et relier le résultat vous-même avec une belle couverture cartonnée pour avoir l'équivalent d'un livre en librairie. Pour l'auteur, aucun espoir de royalties n'est à espérer de ce travail. Ce monde du freeware a par contre plusieurs avantages : tous mes étudiants vont l'obtenir (tous n'achèteront pas un livre à 350F !), je peux espérer toucher de nombreux internautes intéressés, mais surtout je peux espérer avoir des retours permettant de l'améliorer ! Peut-être qu'un jour, quand je serai persuadé que ce livre est devenu assez complet, je me déciderai à contacter un éditeur, mais ça n'est pas pour demain :-)

Le domaine des bases de données est en évolution permanente, ce livre n'est certainement pas complet et de plus, il y a sans doute des parties difficiles à lire pour certains lecteurs qui nécessitent d'être plus détaillées. En le laissant sur le WEB, j'ai donc la possibilité en permanence, d'améliorer, ajouter et corriger des passages de ce livre. Pour cela j'ai besoin de vous.

N'hésitez pas : Si vous avez des remarques et critiques (même positives :-) à formuler sur ce livre, envoyez les à l'adresse email `mathieu@lifl.fr`. La dernière version de ce livre est disponible sur Internet à l'URL `http://www.lifl.fr/~mathieu/bdd`

et si vous utilisez ce poly, envoyez moi juste un petit mail, ça fait toujours plaisir ...

Remerciements

Merci à Pierre-Yves Gibello (INRIA) pour son aide sur RMI-JDBC, à Anne-cécile Caron, Mireille Clerbout, Christophe Gransart et Fabrice Descamps (LIFL) pour la relecture et leurs idées d'améliorations et à vous ... si vous contribuez à améliorer ce livre..

Bugs

Cette section est destinée à disparaître :-)

- Il manque un chapitre sur les techniques de Hachage et d'indexation. Il manque un chapitre sur les BDD objets et déductives (P Mathieu).
- Chapitre 2; Intersection; la remarque en bas de page sur intentionnel-extensionnel est mal exprimée, elle nécessite d'être réécrite et détaillée (E. Wegrzyno).
- Chapitre 10; la partie ODBC traite d'une table toto alors que la partie JDBC traite d'une table client; Il faudrait traiter le même exemple dans les deux cas (C. Gransart).
- Chapitre 5 SQL; quelques résultats devraient être donnés notamment pour les requêtes complexes (A. Taquet).

Saisi avec LaTeX

Table des matières

Table des figures

Chapitre 1

Présentation générale

1.1 Qu'est ce qu'une base de données

Il est assez difficile de définir ce qu'est une base de données si ce n'est que de dire trivialement que tout système d'information peut être qualifié de base de données. Il semble plus facile de définir l'outil principal de gestion d'une base de données : le système de gestion de bases de données (SGBD) [1]

- C'est un outil permettant d'insérer, de modifier et de rechercher efficacement des données spécifiques dans une grande masse d'informations.
- C'est une interface entre les utilisateurs et la mémoire secondaire facilitant le travail des utilisateurs en leur donnant l'illusion que toute l'information est comme ils le souhaitent. Chacun doit avoir l'impression qu'il est seul à utiliser l'information.

Le SGBD est composé de trois couches successives :

- Le système de gestion de fichiers.
  Il gère le stockage physique de l'information. Il est dépendant du matériel utilisé (type de support, facteur de blocage, etc ...).
- Le SGBD interne.
  Il s'occupe du placement et de l'assemblage des données, gestion des liens et gestion de l'accès rapide.
- Le SGBD externe.
  Il s'occupe de la présentation et de la manipulation des données aux concepteurs et utilisateurs. Il s'occupe de la gestion de langages de requêtes élaborés et des outils de présentation (états, formes, etc...)

1.2 Objectifs et avantages

Un système d'information peut toujours être réalisé sans outil spécifique. On peut alors se demander quels sont les objectifs et avantages de l'approche SGBD par rapport aux fichiers classiques. La réponse tient en neuf points fondamentaux :

1. Indépendance physique.
   Les disques, la machine, les méthodes d'accès, les modes de placement, les méthodes de tris, le codage des données ne sont pas apparents. Le SGBD offre une structure canonique permettant la représentation des données réelles sans se soucier de l'aspect matériel.

1. ce qui se traduit en anglais par DBMS : DataBase Management System

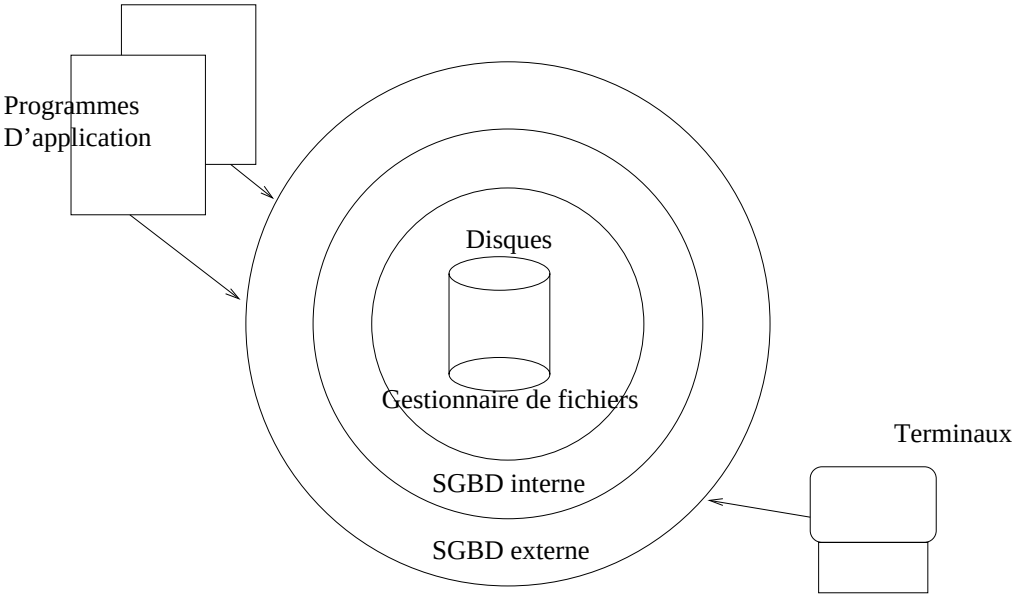


FIG. 1.1 – *Le modèle à 3 couches*

2. Indépendance logique.
   Chaque groupe de travail doit pouvoir se concentrer sur ce qui l'intéresse uniquement. Il doit pouvoir arranger les données comme il le souhaite même si d'autres utilisateurs ont une vue différente. L'administrateur doit pouvoir faire évoluer le système d'informations sans remettre en cause la vue de chaque groupe de travail.
   Exemple : une base de données contient les informations suivantes :

   ```
   véhicule(num-véhicule, marque, type, couleur)
   personne(num-ss, nom, prénom)
   propriétaire(num-ss, num-véhicule, date-achat)
   ```

   Un groupe de travail ne s'intéressera qu'aux personnes qui possèdent une voiture :
   `personne(num-ss, nom, prénom, num-véhicule)`
   Un autre groupe ne s'intéressera qu'aux véhicules vendus à une certaine date :
   `voiture(num-véhicule, type, marque, date-achat)`

3. Manipulable par des non-informaticiens.
   Le SGBD doit permettre d'obtenir les données par des langages non procéduraux. On doit pouvoir décrire ce que l'on souhaite sans décrire comment l'obtenir.
4. Accès aux données efficace.
   Les accès disque sont lents relativement à l'accès à la mémoire centrale. Il faut donc offrir les meilleurs algorithmes de recherche de données à l'utilisateur.
   Remarque: le système de gestion de fichiers y répond parfois pour des mono-fichiers (ISAM, VSAM etc...) mais dans le cas d'intercroisements entre différents fichiers cela devient beaucoup plus complexe et parfois même contextuel à la recherche effectuée.
5. Administration centralisée des données.
   Le SGBD doit offrir aux administrateurs des données des outils de vérification de cohérence des données, de restructuration éventuelle de la base, de sauvegarde ou de réplication. L'administration est centralisée et est réservée à un très petit groupe de personnes pour des raisons évidentes de sécurité.

6. Non redondance des données.
   Le SGBD doit permettre d'éviter la duplication d'informations qui, outre la perte de place mémoire, demande des moyens humains importants pour saisir et maintenir à jour plusieurs fois les mêmes données.
7. Cohérence des données.
   Cette cohérence est obtenue par la vérification des contraintes d'intégrité. Une contrainte d'intégrité est une contrainte sur les données de la base, qui doit toujours être vérifiée pour assurer la cohérence de cette base. Les systèmes d'information sont souvent remplis de telles contraintes, le SGBD doit permettre une gestion automatique de ces contraintes d'intégrité sur les données. Par exemple :
   - Un identifiant doit toujours être saisi.
   - Le salaire doit être compris entre 4000 et 100000F.
   - Le nombre de commandes du client doit correspondre avec le nombre de commandes dans la base.
   - L'emprunteur d'un livre doit être un abonné du club.

   Dans un SGBD les contraintes d'intégrité doivent pouvoir être exprimées et gérées dans la base et non pas dans les applications.
8. Partageabilité des données.
   Le SGBD doit permettre à plusieurs personnes (ou applications) d'accéder simultanément aux données tout en conservant l'intégrité de la base. Chacun doit avoir l'impression qu'il est seul à utiliser les données.
9. Sécurité des données.
   Les données doivent être protégées des accès non autorisés ou mal intentionnés. Il doit exister des mécanismes permettant d'autoriser, contrôler et enlever des droits d'accès à certaines informations pour n'importe quel usager. Par exemple un chef de service pourra connaître les salaires des personnes qu'il dirige, mais pas de toute l'entreprise. Le système doit aussi être tolérant aux pannes : si une coupure de courant survient pendant l'exécution d'une opération sur la base, le SGBD doit être capable de revenir à un état dans lequel les données sont cohérentes.

Remarque : ces neuf points, bien que caractérisant assez bien ce qu'est une base de données, ne sont que rarement réunis dans les SGBD actuels. C'est une vue idéale des SGBD.

1.3 Différents types de bases de données

Il existe actuellement 5 grands types de bases de données :

- Les bases hiérarchiques.
  Ce sont les premiers SGBD apparus (notamment avec IMS d'IBM). Elles font partie des bases navigationnelles constituées d'une gestion de pointeurs entre les enregistrements. Le schéma de la base doit être arborescent.
- Les bases réseaux.
  Sans doute les bases les plus rapides, elles ont très vite supplanté les bases hiérarchique dans les années 70 (notamment avec IDS II d'IBM). Ce sont aussi des bases navigationnelles qui gèrent des pointeurs entre les enregistrements. Cette fois-ci le schéma de la base est beaucoup plus ouvert.
- Les bases relationnelles.
  A l'heure actuelle les plus utilisées. Les données sont représentées en tables. Elles sont basées sur l'algèbre relationnelle et un langage déclaratif (généralement SQL).
- Les bases déductives.
  Les données sont aussi représentées en tables (prédicats), le langage d'interrogation se base sur le calcul des prédicats et la logique du premier ordre.

– Les bases objets.
  Les données sont représentées en tant qu'instances de classes hiérarchisées. Chaque champ est un objet. De ce fait, chaque donnée est active et possède ses propres méthodes d'interrogation et d'affectation. L'héritage est utilisé comme mécanisme de factorisation de la connaissance.

La répartition du parc des SGBD n'est pas équitable entre ces 5 types de bases. 75% sont relationnelles, 20% réseaux, les 5% restants étant partagés entre les bases déductives et objets. Ces chiffres risquent néanmoins d'évoluer d'ici quelques années et la frontière entre les bases relationnelles et objets risque d'être éliminée par l'introduction d'une couche objets sur les bases relationnelles.

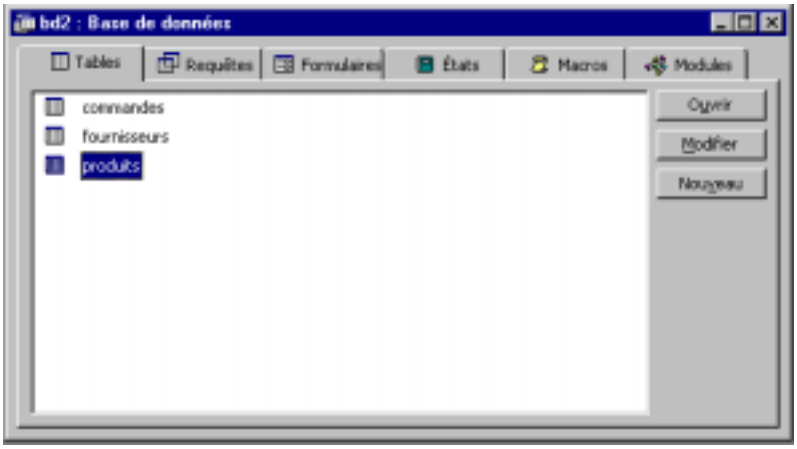


FIG. 1.2 – *Une base Access*

1.4 Quelques systèmes existants

– Oracle (http://www.oracle.com)
– DB2 (IBM, http://www.software.ibm.com)
– Ingres
– Informix
– Sybase (http://www.sybase.com)
– SQL Server (Microsoft, http://www.microsoft.com)
– O2
– Gemstone
– et sur micros : Access 97 (http://www.microsoft.com), Paradox 8.0 (http://www.corel.com), Visual Dbase (http://www.borland.com), FoxPro (http://www.microsoft.com), FileMaker 4.0 (http://www.claris.fr), 4D, Windev (http://www.pcsoft.fr)

Il existe aussi quelques sharewares que l'on peut trouver sur Internet pour s'initier aux bases de données relationnelles comme MySQL (`http://web.tryc.on.ca/mysql/`), MSQL (`http://Hughes.com.au/`), Postgres (`http://www.postgresql.org`) ou InstantDB (`http://www.instantdb.co.uk`) qui est entièrement écrit en Java.

1.5 Les niveaux ANSI/SPARC

Pour assurer les objectifs précédemment décrits, 3 niveaux de description ont été distingués par le groupe ANSI/X3/SPARC en 1975.

– Le niveau conceptuel.

Il correspond à ce que l'on retrouve dans la méthode Merise avec les modèles de données (MCD[2], MLD[3]).

- Le niveau interne.
  Il correspond à la structure de stockage des données: types de fichiers utilisés, caractéristiques des enregistrements (longueur, composants), chemins d'accès aux données (types d'index, chaînages etc..).
- Le niveau externe.
  Il est caractérisé par l'ensemble des vues externes qu'ont les groupes d'utilisateurs.

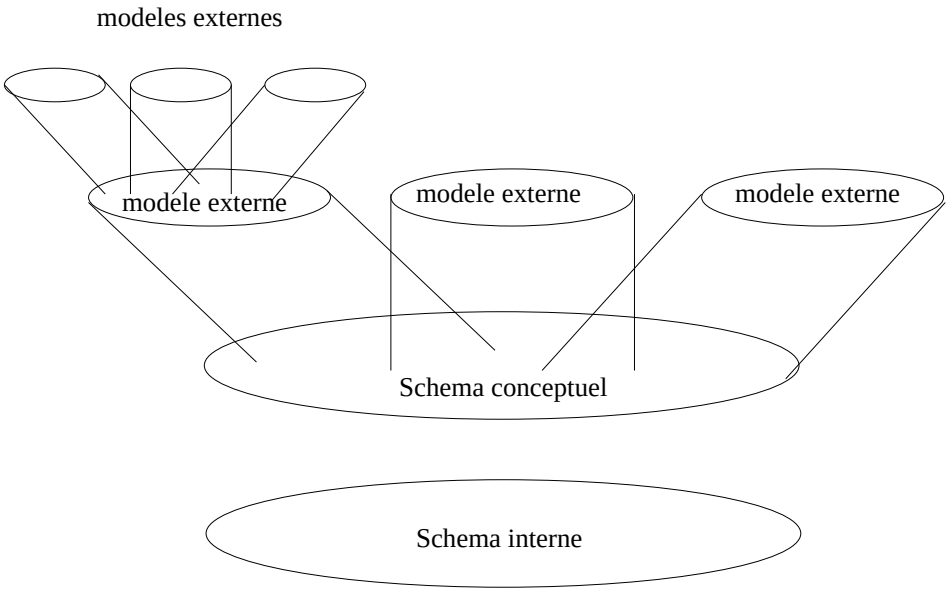


FIG. 1.3 – *Le modèle Ansi/Sparc*

1.6 Modéliser les données

Avant de s'attaquer à tout problème, il est toujours nécessaire de réfléchir profondément aux tenants et aboutissants de ce que l'on veut réaliser. La phase de conception nécessite souvent de nombreux choix qui auront parfois des répercussions importantes par la suite. La conception de bases de données ne fait pas exception à la règle. Les théoriciens de l'information ont donc proposé des méthodes permettant de structurer sa pensée et présenter de manière abstraite le travail que l'on souhaite réaliser. Ces méthodes ont donné naissance à une discipline, l'analyse, et un métier, l'analyste.

L' **analyse** est la discipline qui étudie et présente de manière abstraite le travail à effectuer.

La phase d'analyse est très importante puisque c'est elle qui sera validée par les utilisateurs avant la mise en œuvre du système concret. Il existe de nombreuses méthodes d'analyse (AXIAL, OMT etc ...), la plus utilisée en France étant la méthode Merise. Merise sépare les données et les traitements à effectuer avec le système d'information en différents modèles conceptuels et physiques. Celui qui nous intéresse particulièrement ici est le MCD.

2. Modèle conceptuel de données
3. Modèle logique de données

Le **MCD** (modèle conceptuel de données) est un modèle abstrait de la méthode Merise permettant de représenter l'information d'une manière compréhensible aux différents services de l'entreprise. Il permet une description statique du système d'informations à l'aide d'entités et d'associations.

Le travail de conception d'une base de données par l'administrateur commence juste après celui des analystes qui ont établi le MCD.

Commençons par quelques définitions propres au MCD:

La propriété est une donnée élémentaire et indécomposable du système d'information. Par exemple une date de début de projet, la couleur d'une voiture, une note d'étudiant.

L'entité est la représentation dans le système d'information d'un objet matériel ou immatériel ayant une existence propre et conforme aux choix de gestion de l'entreprise. L'entité est composée de propriétés. Par exemple une personne, une voiture, un client, un projet.

FIG. 1.4 – *Représentation d'une entité*

L'association traduit dans le système d'information le fait qu'il existe un lien entre différentes entités. Le nombre d'intervenants dans cette association caractérise sa dimension :

- réflexive sur une même entité.
- binaire entre deux entités.
- ternaire entre trois entités.
- n-aire entre n entités.

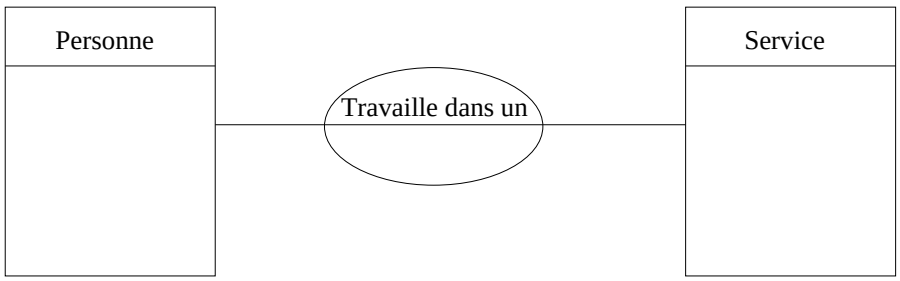


FIG. 1.5 – *Représentation d'une association*

Des propriétés peuvent être attachées aux associations. Par exemple, un employé peut passer 25% de son temps dans un service et 75% de son temps dans un autre. L'association "travaille dans" qui relie une personne à un service portera la propriété "volume de temps passé" (fig. 1.6).

Les cardinalités caractérisent le lien entre une entité et une association. La cardinalité d'une association est constituée d'une borne minimale et d'une borne maximale :

- minimale : nombre minimum de fois qu'une occurrence d'une entité participe aux occurrences de l'association, généralement 0 ou 1.
- maximale : nombre maximum de fois qu'une occurrence d'une entité participe aux occurrences de l'association, généralement 1 ou n.

Les cardinalités maximales sont nécessaires pour la création de la base de données. les cardinalités minimales sont nécessaires pour exprimer les contraintes d'intégrités.

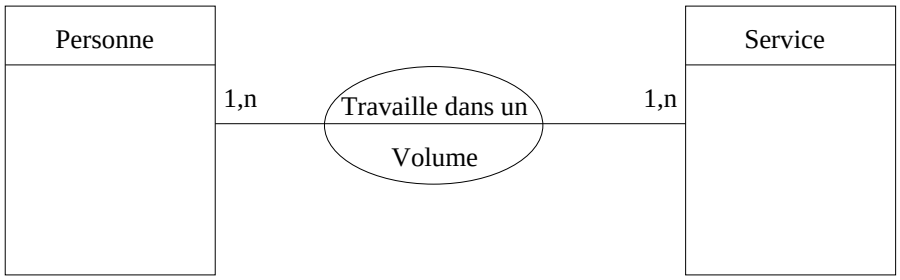


FIG. 1.6 – *Représentation des cardinalités*

De ce schéma on en déduit que "une personne peut travailler dans plusieurs services". On constate de plus que "dans chaque service il y a au moins 1 personne mais qu'il peut y en avoir plusieurs". Enfin, une mesure du "volume de travail" est stockée pour chaque personne travaillant dans un service donné.

Remarque: il existe une notation "à l'américaine" dans laquelle on ne note que les cardinalités maximum. Dans le schéma Fig 1.6 la notation américaine serait `n:m`

un lien hiérarchique est un lien 1:n en notation américaine.
un lien maillé est un lien n:m en notation américaine.

Identifiant : l'identifiant d'une entité est constitué d'une ou plusieurs propriétés de l'entité telles qu'à chaque valeur de l'identifiant corresponde une et une seule occurrence de l'entité.
L'identifiant d'une association est constitué de la réunion des identifiants des entités qui participent à l'association.

Remarque : l'identifiant est représenté en souligné dans le MCD.

1.7 Exemple de MCD

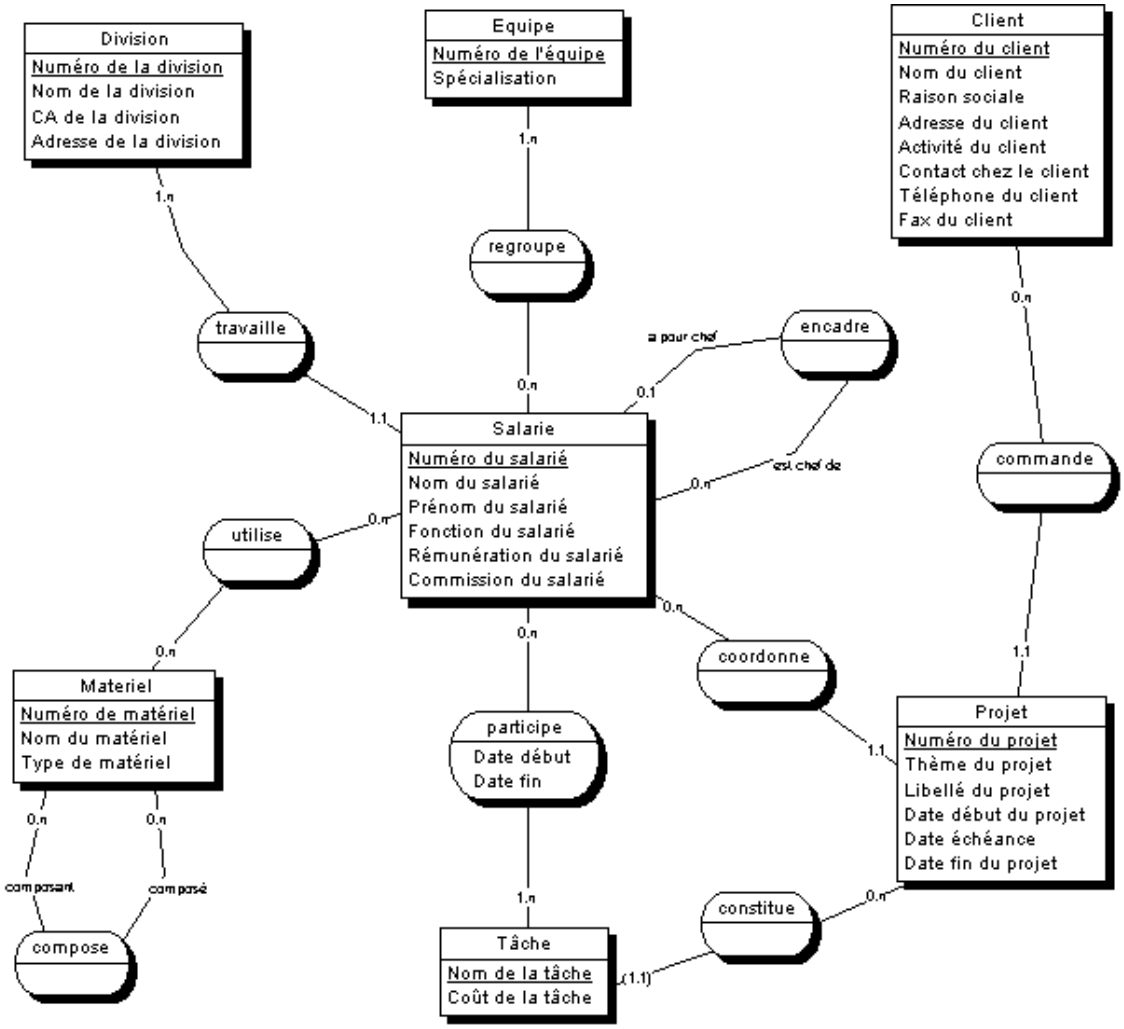


FIG. 1.7 – *Exemple de MCD*

1.8 Le modèle relationnel

Les systèmes de gestion de bases de données relationnelles organisent les données en tables (à la manière d'un tableur).

Il est simple, facile à comprendre et fidèle à un cadre mathématique (l'algèbre relationnelle).

Le concept mathématique sous-jacent est celui de relation de la théorie des ensembles, qui se définit comme un sous-ensemble du produit cartésien de plusieurs domaines.

Un domaine est un ensemble fini ou infini de valeurs possibles. Le domaine des entiers, le domaine des booléens, le domaine des couleurs du drapeau français { bleu, blanc, rouge } etc ...

On utilise alors le produit cartésien d'un ensemble de domaines pour définir un ensemble de n-uplets.

Le produit cartésien d'un ensemble de domaines $D_1, D_2, \ldots, D_n$. que l'on écrit $D_1 * D_2 * \ldots * D_n$ est l'ensemble des n-uplets (ou tuples) $< V_1, V_2, \ldots, V_n >$ tels que $V_i \in D_i$.

Exemple:
Le produit cartésien des domaines D1={durand,lefebvre,martin} et D2={christian,franck} donne

durand	christian
durand	franck
lefebvre	christian
lefebvre	franck
martin	christian
martin	franck

Le produit cartésien est une opération plus générale que la simple application à des domaines, on peut par exemple l'appliquer aussi à des ensembles de tuples, ce que nous ferons en algèbre relationnelle.

Une table relationnelle [4] est un sous-ensemble du produit cartésien d'une liste de domaines. Elle est généralement caractérisée par un nom permettant de l'identifier clairement.

Afin de rendre l'ordre des colonnes sans importance tout en permettant plusieurs colonnes de même domaine, on associe un nom à chaque colonne.

personne	D1	D2
	lefebvre	christian
	martin	franck
	durand	franck

Les colonnes constituent ce que l'on appelle les attributs de la table relationnelle.

Remarque: comme pour toute définition d'ensembles, il existe en fait deux possibilités pour définir une relation: en intention ou en extension. La forme intentionnelle est utilisée dans les bases de données déductives, par exemple $\{(x, y, z) \in (N, Z, Q)$ tels que $x + y > z\}$, mais pas dans les bases relationnelles ni objets. La forme extensionnelle qui consiste à spécifier un à un tous les tuples de la relation est *la seule forme utilisable dans les bases de données relationnelles.* Elle est bien sûr utilisée aussi dans tous les autres types de bases.

Le schéma d'une table relationnelle est constitué de l'ensemble des attributs de la table. Par extension, le schéma de la base de données est constitué de l'ensemble de toutes les tables.

Une base de données relationnelle est une base de données dont le schéma est un ensemble de schémas de tables relationnelles et dont les occurrences sont des tuples de ces tables.

1.9 Exemple de base de données relationnelle

A titre d'exemple voici une base de données relationnelle rudimentaire, utilisant 3 tables représentant les commandes de produits à des fournisseurs.

Cet exemple sera utilisé de nombreuses fois par la suite pour illustrer l'écriture de requêtes.

produits	pno	design	prix	poids	couleur
	102	fauteuil	1500	9	rouge
	103	bureau	3500	30	vert
	101	fauteuil	2000	7	gris
	105	armoire	2500	35	rouge
	104	bureau	4000	40	gris
	107	caisson	1000	12	jaune
	106	caisson	1000	12	gris
	108	classeur	1500	20	bleu

4. On dira parfois uniquement “table” ou “relation” selon le contexte.

fournisseurs	fno	nom	adresse	ville
	10	Dupont		Lille
	15	Durand		Lille
	17	Lefebvre		Lille
	12	Jacquet		Lyon
	14	Martin		Nice
	13	Durand		Lyon
	11	Martin		Amiens
	19	Maurice		Paris
	16	Dupont		Paris

commandes	cno	fno	pno	qute
	1001	17	103	10
	1003	15	103	2
	1005	17	102	1
	1007	15	108	1
	1011	19	107	12
	1013	13	107	5
	1017	19	105	3
	1019	14	103	10
	1023	10	102	8
	1029	17	108	15

Le schéma de cette base est donc :

```
produits(pno, design, prix, poids, couleur)
fournisseurs(fno, nom, adresse, ville)
commandes(cno, fno, pno, qute)
```

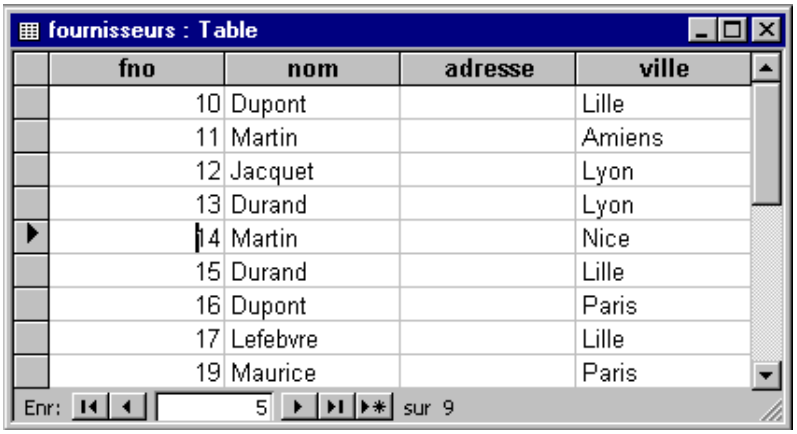


FIG. 1.8 – *Visualisation d'une table Access*

1.10 Passage du MCD aux tables relationnelles

Une fois le MCD écrit par les analystes, le travail du concepteur de bases de données consiste à traduire ce modèle en un modèle plus proche du SGBD utilisé: le MLD (modèle logique de données). Dans le MLD relationnel, l'unique type d'objet existant est la table. La méthode de passage d'un MCD Merise aux tables relationnelles est simple et systématique :

Traitement des entités :

- chaque entité devient une table.
- chaque propriété d'une entité devient une colonne de cette table.
- l'identifiant d'une entité devient la clé primaire de la table correspondante (création d'un index).

Traitement des associations :

- une association (0,n)-(0,1) (lien hiérarchique) provoque la migration d'une clé étrangère (l'identifiant côté 0,n) vers la table de l'entité côté (0,1). Si des propriétés étaient sur l'association elles migrent côté (0,1) (Fig. 1.9 et 1.10).

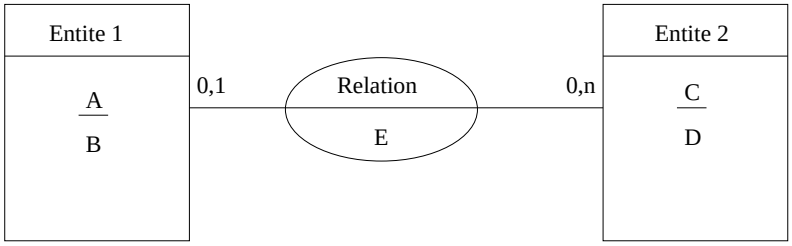


FIG. 1.9 – *MCD avec lien hierarchique*

- une association (0,n)-(0,n) (lien maillé) donne naissance à une nouvelle table. Les identifiants des entités auxquelles l'association est reliée migrent dans cette table. la clé primaire de cette nouvelle table est constituée de la réunion de ces identifiants. Si des propriétés étaient portées par l'association, elles migrent dans la nouvelle table (Fig. 1.10 et 1.11).

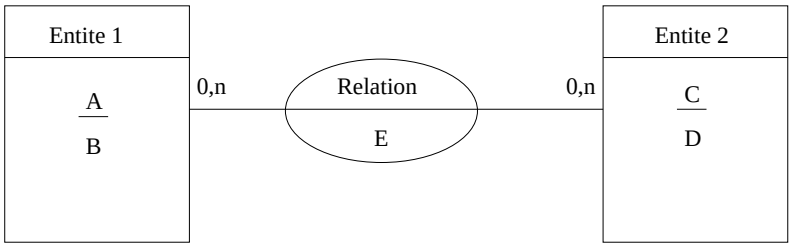


FIG. 1.10 – *MCD avec lien maillé*

- Les associations n-aires sont gérées comme précédemment avec la naissance d'une nouvelle table.

Les schémas du MLD contiennent généralement des flèches indiquant les reports de clé (clés étrangères). Ces flèches sont présentes à titre informatif, en aucun cas ces flèches ne correspondent à un pointeur physique. Les tables relationnelles sont toutes physiquement indépendantes.

Dans le cas Fournisseur-Produits-Commandes (fig. 1.12), l'association `commandes` est un lien maillé porteur d'une propriété. Il y a donc création d'une table `commandes` avec report des clés des entités liées, ce qui nous donne bien les 3 tables vues précédemment.

Rappel: Seules les cardinalités maximales servent à définir le nombre de tables et les reports de clés. Les cardinalités minimales ne servent qu'à préciser par la suite si les colonnes peuvent prendre la valeur `null` ou pas.

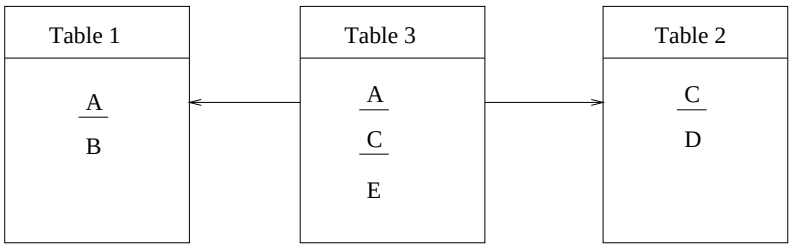


FIG. 1.11 – *MLD correspondant à la Fig.1.10*

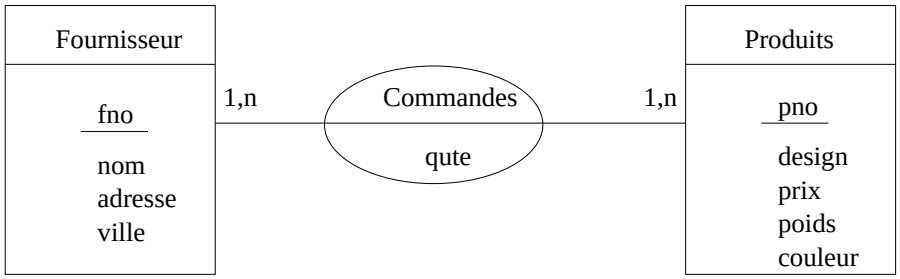


FIG. 1.12 – *MCD Fournisseur-Produits-Commandes*

FIG. 1.13 – *Structure d'une table sous Access*

Chapitre 2

Présentation des données

Une fois la base et les tables créées, il faut ensuite les exploiter. L'utilisateur final aura besoin de visualiser et saisir des données, d'effectuer des calculs et d'imprimer des résultats. La réponse à ces différents problèmes de présentation de données est fournie par les `formulaires` destinés à être affichés à l'écran et les `états` destinés à être imprimés. Ce sont les formulaires et états qui forment la partie visible d'une base de données. Si le concepteur crée, remplis et administre les tables, l'utilisateur, lui, ne sera en contact qu'avec les formulaires et états. Il lui faudra donc des présentations agréables, la présence du logo de l'entreprise et de l'aide à la manipulation du logiciel. C'est à partir des formulaires et états qu'une application tire son ergonomie, c'est dire leur importance.

Il n'y a en général rien de très technique dans leur conception. Malheureusement, la création de formulaires et d'états n'est pas normalisée et chaque SGBD fournit ses propres outils plus ou moins ergonomiques. Oracle fournit SQL-Forms ou Developper 2000. Access et Windev fournissent des outils intégrés.

De plus les logiciels comme Access ou Windev fournissent des assistants qui génèrent automatiquement des formulaires et des états sans que le concepteur n'ait à connaître quoi que ce soit en programmation. Bien que l'utilisation d'assistants soit un avantage indéniable, ils restent d'un usage limité et il faut souvent mettre la main à la pâte pour retoucher ceux-ci afin d'obtenir l'ergonomie que l'on souhaite.

Il n'en reste pas moins, qu'un certain nombre de points fondamentaux sont à détailler pour bien comprendre la création de formulaires et d'états.

2.1 Les formulaires

Un accès aux tables vous donne la possibilité de voir plusieurs enregistrements simultanément et les modifier directement. Malheureusement, si la table contient de nombreuses colonnes, vous devez faire défiler les colonnes pour visualiser un enregistrement en entier. De plus vous ne pouvez pas mettre à jour plusieurs tables simultanément. Enfin la présentation tabulaire n'est pas ergonomique et est source de nombreuses erreurs de saisie.

Si vous souhaitez afficher, saisir ou modifier des données dans une table, les SGBD offrent des objets spécialisés que l'on appelle formulaires. Un formulaire est destiné à être affiché à l'écran pour offrir à l'utilisateur une interface agréable à la mise à jour des tables.

Il y a deux types de formulaires :

- Les formulaires de présentation de données.
  Leur objectif est de présenter, saisir ou modifier les données d'une ou plusieurs tables.
- Les formulaires de distribution.
  Ils ne sont attachés à aucune table, et servent uniquement de page de menu pour orienter l'utilisateur vers d'autres formulaires ou états. Ils ne contiennent que du texte et des boutons d'orientation.

En principe un formulaire de présentation affiche un seul enregistrement à la fois mais peut contenir des champs provenant de plusieurs tables. Le formulaire de présentation est toujours attaché à une table ou une requête principale.

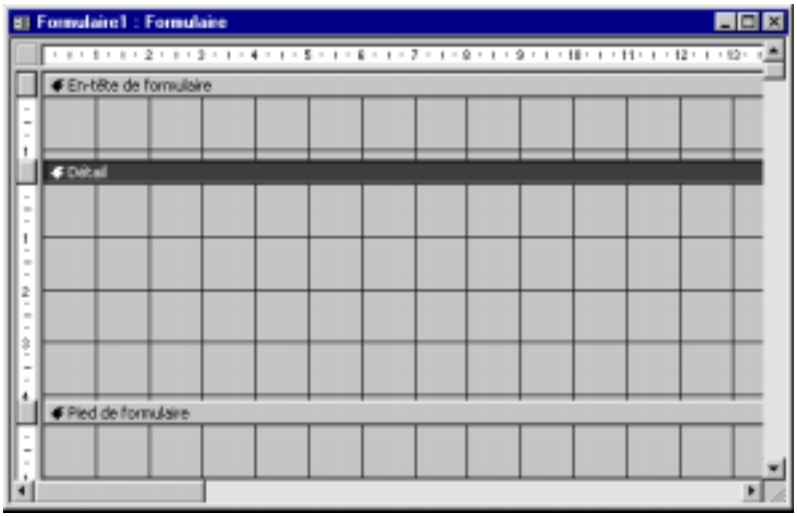


FIG. 2.1 – *Structure d'un formulaire sous Access*

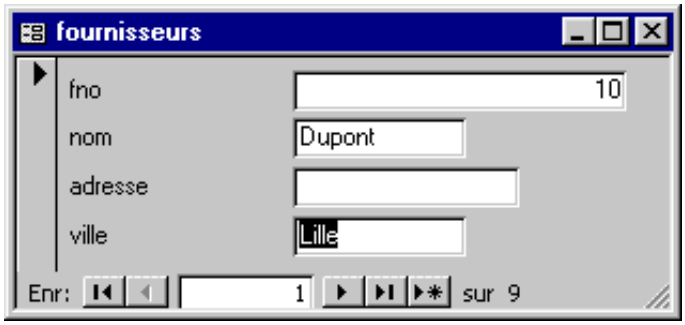


FIG. 2.2 – *Formulaire rudimentaire*

Sous sa forme la plus simple, un formulaire se présente comme une page blanche dans laquelle on place des composants graphiques permettant l'affichage d'un enregistrement.

On dispose en général d'objets de type :

- Label, permettant d'afficher du texte
- Zone de texte, permettant de saisir ou afficher la valeur d'un champs de la table
- Boutons d'options (Radio button)
- Cases à cocher (CheckBox)
- Zones de listes, pour afficher des listes de valeurs
- Boutons, pour tester le click de souris par exemple
- Bascules, pour un champs de type booléen par exemple.
- Images
- Onglets
- Attibuts graphiques de type rectangles, segments, ovales etc... pour agrémenter le formulaire.

Il est bien sûr possible d'effectuer des calculs qui seront affichés dans les formulaires sans que le résultat du calcul soit explicite dans aucune table.

FIG. 2.3 – *Structure d'un formulaire avec sous-formulaire*

Il arrive fréquemment que l'on souhaite afficher des données de deux tables qui sont en relation l'une avec l'autre. Ceci se produira si l'on souhaite afficher une commande avec la liste des produits commandés, un compte bancaire avec les écritures en cours sur ce compte, ou les vols avec le personnel de bord.

On utilise dans ce cas un formulaire qui contient les données de la table principale, et un sous-formulaire qui contient les données de la table liée. Le sous-formulaire est alors placé dans le formulaire principal créant ainsi une relation père-fils entre formulaires.

Le sous-formulaire n'affiche alors que les enregistrements relatifs à l'enregistement en cours du formulaire principal. Si l'utilisateur change d'enregistrement principal, le sous-formulaire est automatiquement mis à jour. Avec les outils les plus ergonomiques dans ce domaine, un simple *glisser* à la souris permet de placer un sous-formulaire dans un formulaire principal avec mise en place automatique du lien entre les deux.

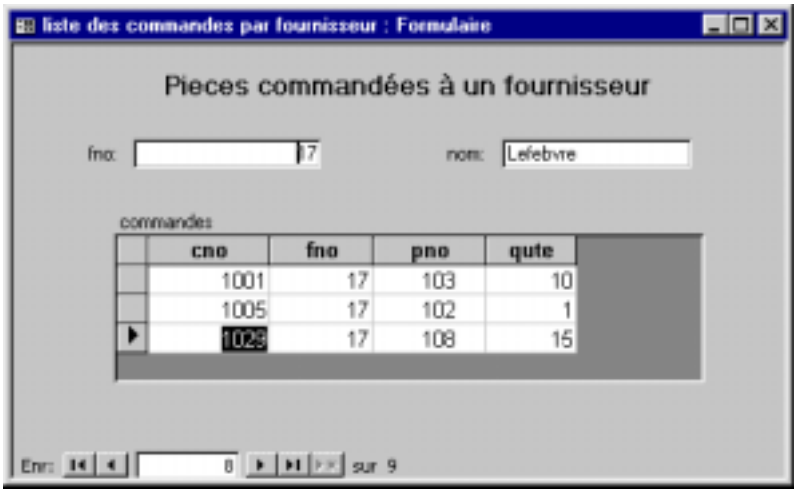


FIG. 2.4 – *Formulaire avec sous-formulaire*

Un formulaire principal peut contenir autant de sous-formulaires que l'on souhaite. De plus, un sous-formulaire peut lui aussi contenir un autre sous-formulaire. On pourra par exemple afficher dans le formulaire principal les coordonnées d'un client avec dans le sous-formulaire les

commandes de ce client et à l'intérieur de ce sous-formulaire un autre sous-formulaire détaillant chaque commande.

Pour agrémenter la présentation chaque formulaire possède ses propres propriétés commme la couleur de fond, la présence ou non de boutons de déplacement ou de barres de défilement. De même, chaque objet placé dans le fomulaire possède lui aussi ses propres propriétés comme sa couleur, la taille de la police utilisée. Il est aussi possible dans de nombreux SGBD de préciser un mode d'utilisation d'un formulaire; On pourra préciser par exemple que tel formulaire ne doit servir que pour effectuer des saisies et tel autre ne pourra être utilisé que pour faire de l'affichage de données sans mise à jour possible. Ces fonctionalités augmentent grandement la sécurité d'utilisation au niveau de l'utilisateur. Cette partie bien qu'étant très importante pour l'ergonomie d'une application professionnelle, n'est pas détaillée dans ce livre car elle est propre à chaque outil.

2.2 Les états

Si les formulaires offrent une réponse élégante à la question "comment pésenter mes données à l'écran", ils ne sont pas adaptés à la présentation papier. Pour analyser les données et les mettre en page pour l'impression on utilise alors un état. Celui-ci permet bien sûr l'affichage de plusieurs enregistrements simultanés mais aussi le calcul de cumuls, de regroupements et l'affichage de synthèses. Il permet aussi l'élaboration de graphiques de présentation, la gestion d'étiquettes ou de publipostage.

En général un état est créé pour présenter les données contenues dans une ou plusieurs tables. Il est donc toujours lié au moins à une table. Comme pour les formulaires, un état peut très bien présenter des calculs qui ne sont pas explicites dans aucune table.

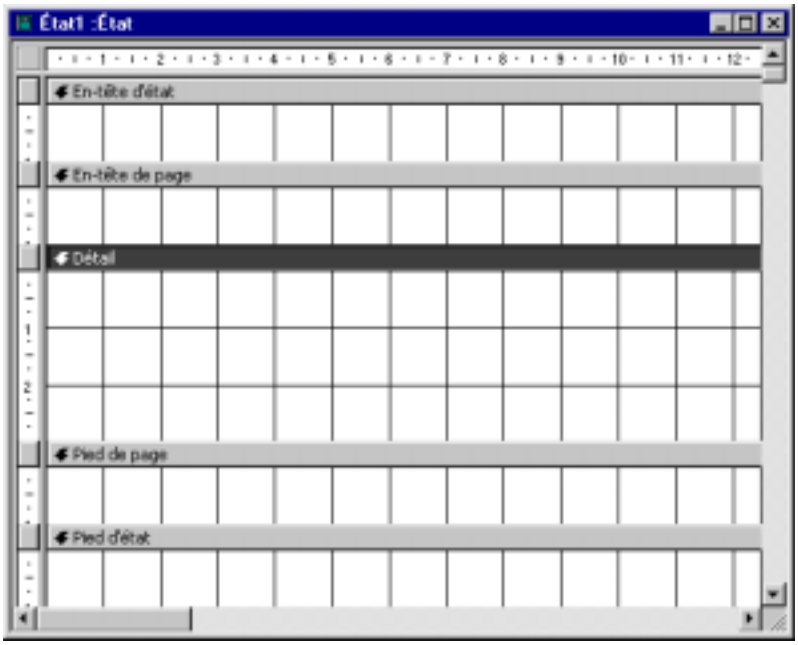


FIG. 2.5 – *Structure d'un état sous Access*

Un état est un objet structuré qui comprend plusieurs niveaux. Dans sa forme la plus simple, il contient :

– Un entête de formulaire
  L'entête de formulaire n'est affiché qu'une seule fois en début d'état. Il contient en général

le titre du formulaire avec toutes les informations permettant d'identifier précisément ce qui sera ensuite présenté. Il contient aussi le logo de la société et la date et l'heure d'impression.

- Un entête de page
  L'entête de page est affiché en haut de chaque page. Si l'état s'imprime sur n pages, il sera affiché n fois. Il contient en général les intitulés des colonnes qui sont imprimées afin que le lecteur puisse se repérer facilement dans l'état.
- Une zone de détail
  La zone de détail contient les données proprement dites. Elle est répétée pour chaque enregistrement qui sera affiché. Elle peut bien sûr tenir sur plusieurs lignes et afficher des champs calculés.
- Un Pied de page
  Le pied de page est le pendant de l'entête de page. Il est affiché en bas de chaque page et contient en général au moins le numéro de page.
- Un pied de formulaire
  Le pied de formulaire est le pendant de l'entête de formulaire. Il n'est affiché qu'une fois en fin de formulaire pour par exemple imprimer une synthèse de ce qui a été affiché dans l'état.

Mise à part la zone de détail, toutes les autres zones sont facultatives et peuvent éventuellemennt être vides.

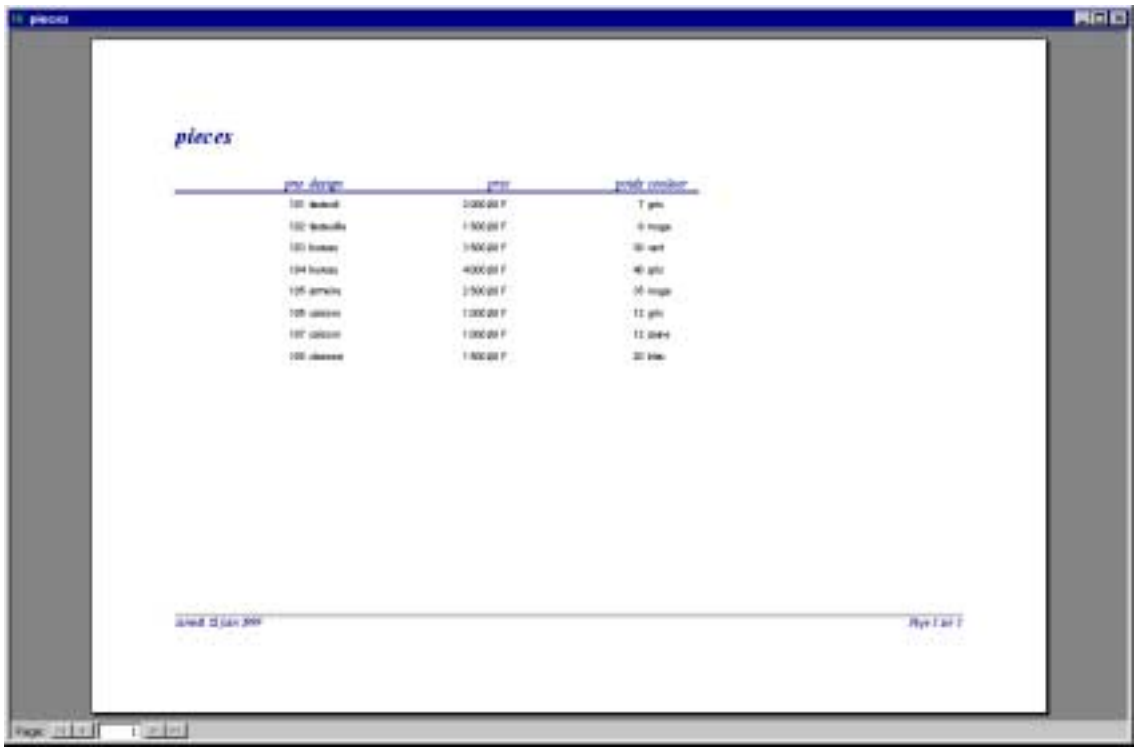


FIG. 2.6 – *état rudimentaire*

Comme pour les formulaires, la notion de sous-état existe mais présente peu d'intérêt. Lors de l'affichage de données provenant de plusieurs tables, on préférera la solution qui consiste à créer une requête reprenant tous les champs à imprimer, puis à créer un état sur cette nouvelle requête. Cette solution, très générale est beaucoup plus pratique et ergonomique que la solution basée sur des sous-états.

Quand les données proviennent (indirectement) de plusieurs tables, il arrive fréquemment que l'on souhaite afficher des informations compémentaires en cas de rupture de logique dans

la présentation de ces données. Par exemple, afficher le somme des commandes après chaque détail d'une commande ou afficher le solde d'un compte après la liste des opérations sur ce compte. Dans un tel cas, les opérations effectuées regroupent plusieurs enregistrements sur une ou plusieurs valeurs communes comme le numérode commande ou le numéro de compte.

Pour afficher sur l'état des informations complémentaires en cas de changement de zone de regroupement les états contiennent deux nouvelle structures :

- Entête de regroupement
  L'entête de regroupement contient les informations à afficher avant que le détail des écritures qui seront regroupées soit affiché. On y place en général un sous-titre ou un cadre.
- Pied de regroupement
  Le pied de regroupement contient les information qui seront affichées après le détail des écritures regroupées. On y place en général les opérations de cumul et de synthèse (nombre, somme, moyenne etc...) des informations afficheées

Entête et pied de regroupement sont affichés pour chaque groupe de données présenté dans l'état.

Pour agrémenter la présentation chaque état possède ses propres propriétés. Chaque zone constituant un état possède elle aussi ses propriétés comme la couleur de fond, la police utilisée. Enfin, chaque objet placé dans l'état possède lui aussi ses propres propriétés comme sa couleur, la taille de la police utilisée etc ... Cette partie bien qu'étant très importante pour l'ergonomie d'une application professionnelle, n'est pas détaillée dans ce livre car elle est propre à chaque outil.

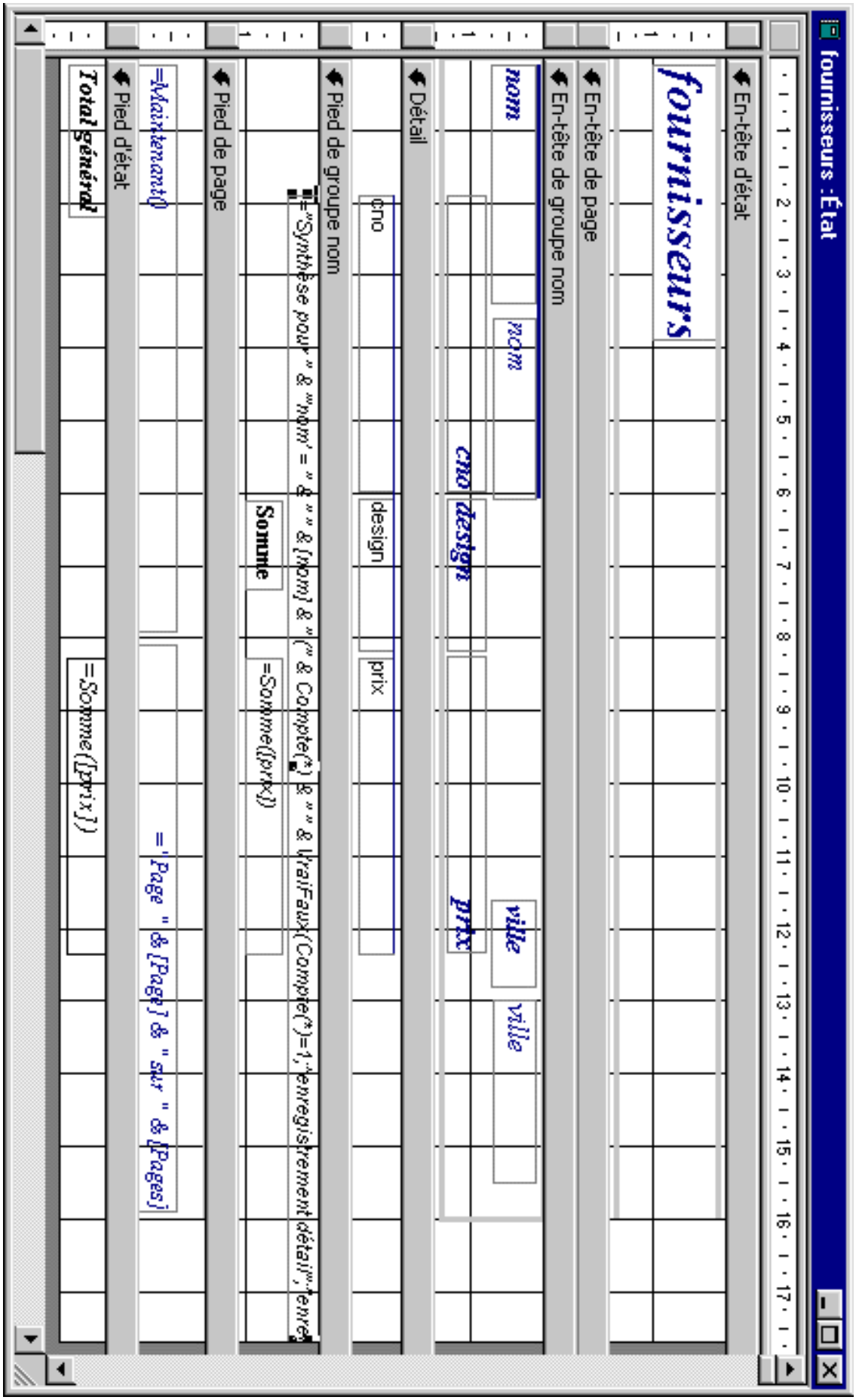


FIG. 2.7 – *structure d'état avec regroupement*

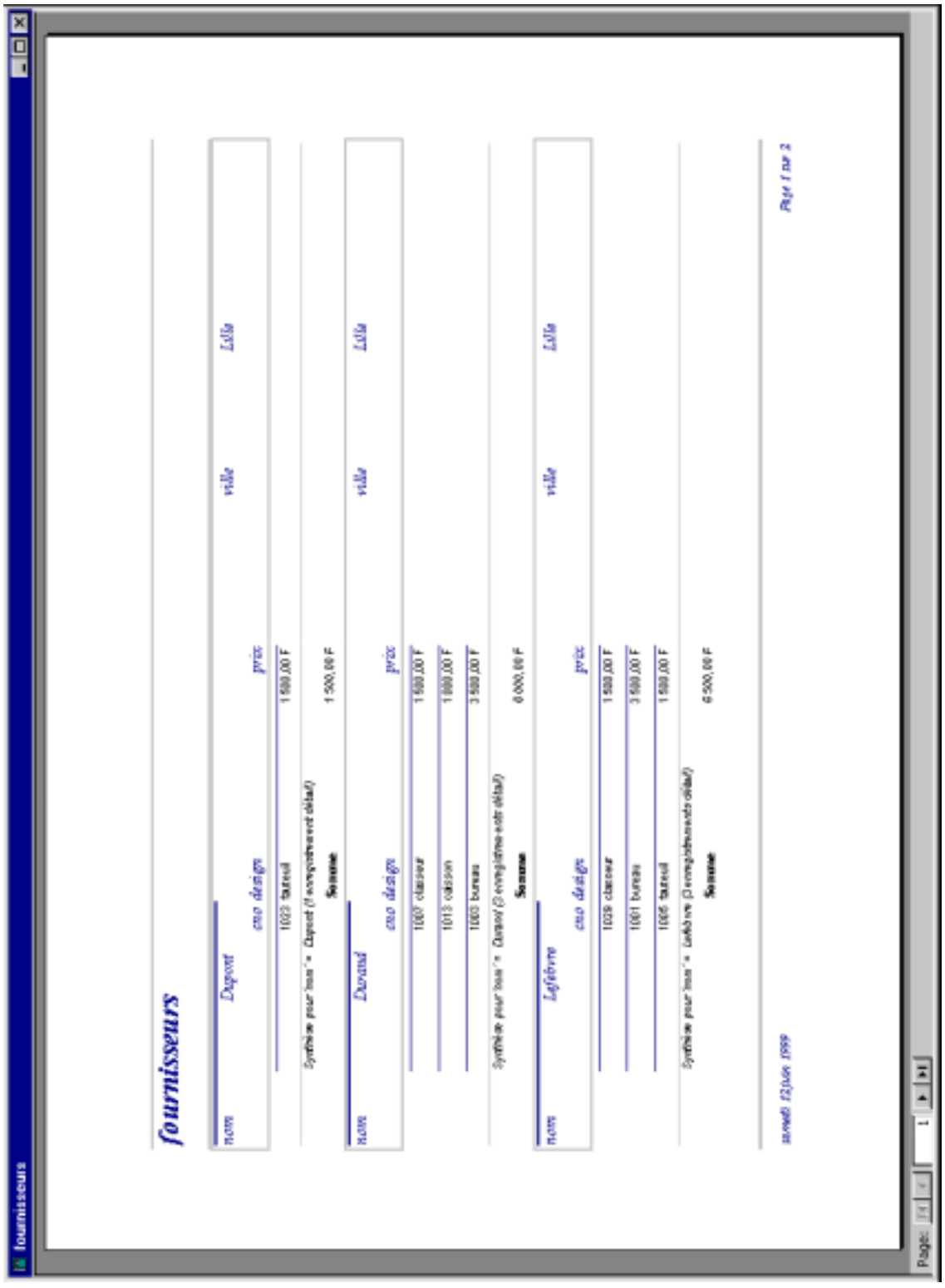


FIG. 2.8 – *état avec regroupement et champs calculés*

Chapitre 3

L'algèbre relationnelle

L'algèbre relationnelle a été introduite par Codd en 1970 pour formaliser les opérations sur les ensembles. Il existe deux familles d'opérations: les opérations ensemblistes et les opérations unaires.

En guise d'exemple nous illustrerons les opérations décrites dans les prochaines pages à l'aide des deux relations R et S suivantes:

R	A	B	C
	a	b	c
	d	a	f
	c	b	d

S	A	B	C
	b	g	a
	d	a	f

3.1 Les opérations de base

Trouver l'ensemble des opérations de base consiste à trouver un ensemble minimal d'opérations au sens où aucune d'entre elles ne peut s'écrire par combinaison des autres. Il existe plusieurs ensembles minimaux pour l'algèbre relationnelle. Celui que nous présentons se base sur 3 opérations ensemblistes et 2 opérations unaires.

3.1.1 Opérations ensemblistes

Les opérations ensemblistes de base sont l'union, la différence et le produit.

L'union de deux relations R et S de même schéma est une relation T de même schéma contenant l'ensemble des tuples appartenant à R, à S ou aux deux.
On notera $T = (R \cup S)$ ou T=union(R,S).

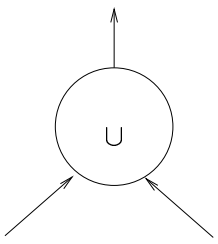


FIG. 3.1 – *L'opérateur Union*

$R \cup S$	A	B	C
	a	b	c
	d	a	f
	c	b	d
	b	g	a

La différence entre deux relations R et S de même schéma dans l'ordre $(R - S)$ est la relation T de même schéma contenant les tuples appartenant à R et n'appartenant pas à S.

On notera $T = (R - S)$ ou minus(R,S)

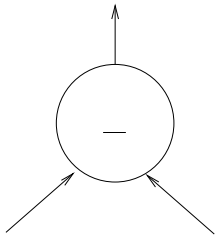


FIG. 3.2 – *L'opérateur Différence*

$R - S$	A	B	C
	a	b	c
	c	b	d

Le produit cartésien de deux relations R et S de schéma quelconque est une relation T ayant pour attributs la concaténation des attributs de R et de S et dont les tuples sont constitués de toutes les concaténations d'un tuple de R à un tuple de S.

On notera $T = (R * S)$ ou T=product(R,S)

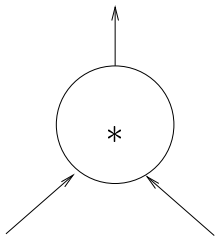


FIG. 3.3 – *L'opérateur Produit*

$R * S$	A	B	C	D	E	F
	a	b	c	b	g	a
	a	b	c	d	a	f
	d	a	f	b	g	a
	d	a	f	d	a	f
	c	b	d	b	g	a
	c	b	d	d	a	f

3.1.2 Opérations unaires

Les opérations unaires ont pour objectif de permettre l'élimination de colonnes ou de lignes dans la table relationnelle. Ces deux opérations sont la projection et la restriction.

La projection d'une relation R de schéma $(A_1, A_2, \ldots, A_n)$ sur les attributs $A_{i_1}, A_{i_2}, \ldots, A_{i_p}$ (avec $i_j \neq i_k$ et $p < n$) est une relation R' de schéma $(A_{i_1}, A_{i_2}, \ldots, A_{i_p})$ dont les tuples sont obtenus par élimination des attributs de R n'appartenant pas à R' et par suppression des doublons.

On notera $T = \Pi_{X_1,\ldots,X_n}(R)$ ou $T = proj_{X_1,\ldots,X_n}(R)$

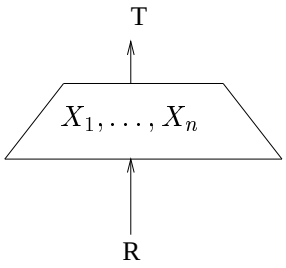


FIG. 3.4 – *L'opérateur Projection*

$\Pi_{A,C}(R)$	A	C
	a	c
	d	f
	c	d

La restriction (ou sélection) de la relation R par une qualification Q est une relation R' de même schéma dont les tuples sont ceux de R satisfaisant la qualification Q.

La qualification Q peut être exprimée à l'aide de constantes, comparateurs arithmétiques ($>$, $\geq, <, \leq, =, \neq$) et opérateurs logiques ($\vee, \wedge, \neg$).

On notera $T = \sigma_Q(R)$ ou $T = select_Q(R)$

FIG. 3.5 – *L'opérateur Restriction*

$\sigma_{B='b'}(R)$	A	B	C
	a	b	c
	c	b	d

Les cinq opérations précédentes (union, différence, produit, projection, restriction) forment un ensemble cohérent et minimal. Aucune d'entre-elles ne peut s'écrire à l'aide des autres. A partir de ces cinq opérations élémentaires, d'autres opérations (sans doute plus pratiques) peuvent être définies.

3.2 Opérations dérivées

Les opérations dérivées sont construites à partir des cinq opérations de base. Nous présentons ici quelques opérations additionnelles mais cette liste n'est pas limitative. Bien d'autres pourraient être ajoutées.

L'intersection de deux relations R et S de même schéma est une relation T de même schéma contenant les tuples appartenant à la fois à R et à S.
On notera $T = (R \cap S)$ ou $T = inter(R, S)$.

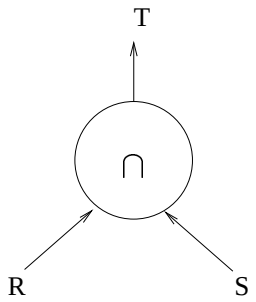


FIG. 3.6 – *L'opérateur Intersection*

$R \cap S$	A	B	C
	d	a	f

L'intersection peut s'écrire à l'aide des opérations de base :

$$R \cap S = R - (R - S) = S - (S - R)$$

Remarque : il existe 3 types d'intersection dans la théorie des ensembles. L'intersection entre deux ensembles extensionnels, entre deux ensembles intentionnels et entre un intentionnel et un extensionnel. L'algèbre relationnelle n'en code que deux : l'intersection entre deux ensembles extensionnels (par l'intersection précédemment définie) et l'intersection entre un ensemble extensionnel et un ensemble intentionnel (par la restriction avec une qualification). On ne peut pas faire, par exemple, l'intersection entre l'ensemble des nombres pairs et l'ensemble des multiples de 3.

Le quotient (ou division) de la relation R de schéma $R(A_1, A_2, \ldots, A_n)$ par la sous-relation S de schéma $S(A_{p+1}, ..., A_n)$ est la relation T de schéma $T(A_1, A_2, A_p)$ formée de tous les tuples qui concaténés à chaque tuple de S donnent toujours un tuple de R.
On notera $T = (R/S)$ ou $T = div(R, S)$

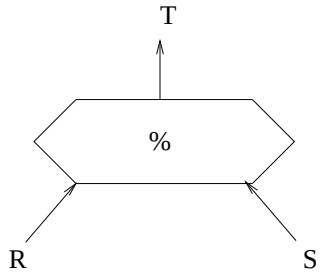


FIG. 3.7 – *L'opérateur Quotient*

R	A	B	C	D
	a	b	c	d
	a	b	e	f
	b	c	e	f
	e	d	c	d
	e	d	e	f
	a	b	d	e

S	C	D
	c	d
	e	f

R/S	A	B
	a	b
	e	d

Le quotient peut s'écrire à l'aide des opérations de base: si l'on note $V = A_i, A_2, \ldots, A_p$ alors

$$R/S = \Pi_V(R) - \Pi_V((\Pi_V(R) * S) - R)$$

Le quotient permet concrètement de rechercher l'ensemble de tous les sous-tuples d'une relation satisfaisant une sous-relation décrite par l'opération diviseur. Il permet de chercher "les tuples de R qui vérifient tout les tuples de S. Chercher par exemple tous les produits qui existent dans une gamme de couleurs et/ou une gamme de prix.

La $\theta-$jointure de deux relations R et S selon une qualification Q est l'ensemble des tuples du produit cartésien $R * S$ satisfaisant la qualification Q.

La qualification Q peut être exprimée à l'aide de constantes, comparateurs arithmétiques ($>$, $\geq, <, \leq, =, \neq$) et opérateurs logiques ($\vee, \wedge, \neg$).

On notera $T = (R \bowtie_Q) S$ ou $T = join_Q(R, S)$

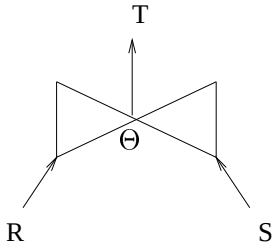


FIG. 3.8 – *L'opérateur Jointure*

R	A	B	C
	1	2	3
	4	5	6
	7	8	9

S	D	E
	3	1
	6	2

$R \bowtie_{B<D} S$	A	B	C	D	E
	1	2	3	3	1
	1	2	3	6	2
	4	5	6	6	2

La $\theta-$jointure peut s'écrire à l'aide des opérations de base:

$$R \bowtie_Q S = \sigma_Q(R * S)$$

Cette opération est essentielle dans les systèmes relationnels et permet l'utilisation raisonnable du produit cartésien.

L'équi-jointure est une $\theta-$jointure avec pour qualification l'égalité entre deux colonnes.

La jointure naturelle est une équi-jointure de R et S sur tous les attributs de même nom suivie de la projection qui permet de conserver un seul de ces attributs égaux de même nom.

La jointure naturelle est la jointure la plus utilisée de manière pratique, elle s'écrit simplement $R \bowtie S$ et peut se définir avec les opérations de base:

$$R \bowtie S = \Pi_V(\sigma_C(R * S))$$

R	A	B	C
	a	b	c
	d	b	c
	b	b	f
	c	a	d

S	B	C	D
	b	c	d
	b	c	e
	a	d	b

$R \bowtie S$	A	B	C	D
	a	b	c	d
	a	b	c	e
	d	b	c	d
	d	b	c	e
	c	a	d	b

3.3 Les opérations de calcul

En plus des opérations décrites précédemment, certains auteurs ajoutent des opérations de calcul sur les relations. Cet ajout est justifié par le fait que de nombreuses requêtes ont besoin de ce type d'opérations qui d'ailleurs sont implémentées dans tous les langages d'interrogation. Ces opérateurs de calcul forment donc une extension aux opérateurs de base et ne peuvent pas être exprimés à l'aide de ceux-ci.

Compte est une opération courante qui permet de dénombrer les lignes d'une relation qui ont une même valeur d'attributs en commun. La relation résultante ne contient que les attributs de regroupement X_i choisis avec leurs occurrences dans la relation.

On notera $T = Compte_{X_1,\ldots,X_n}(R)$ ou $T = Count_{X_1,\ldots,X_n}(R)$, les $X_1, \ldots, X_n$ étants les attributs de regroupement.

Si aucun attribut de regroupement n'est précisé, l'opération renvoie alors uniquement le nombre de tuples de la relation.

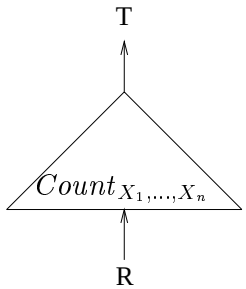


FIG. 3.9 – *L'opérateur Compte*

R	A	B	C
	a	n	17
	b	o	14
	c	n	9
	d	p	13
	e	m	20
	f	m	10

$Compte_B(R)$	B	Compte
	n	2
	m	2
	o	1
	p	1

$Compte(R)$	Compte
	6

Somme est une opération qui permet de faire la somme cumulée des valeurs d'un attribut Y pour chacune des valeurs différentes des attributs de regroupement $X_1, \ldots, X_n$. Y doit bien sûr être numérique. La relation résultante ne contient que les différentes valeurs des attributs X_i de regroupement choisis ainsi que la somme cumulée des Y correspondants.

On notera $T = Somme_{X_1,\ldots,X_n}(R, Y)$ ou $T = Sum_{X_1,\ldots,X_n}(R, Y)$

Si aucun attribut de regroupement n'est précisé, l'opération renvoie alors la somme de toutes les valeurs de la colonne Y.

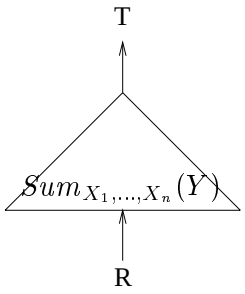


FIG. 3.10 – *L'opérateur Somme*

R	A	B	C
	a	n	17
	b	o	14
	c	n	9
	d	p	13
	e	m	20
	f	m	10

$Somme_B(R, C)$	B	Somme
	n	26
	m	30
	o	14
	p	13

$Somme(R, C)$	Somme
	83

Cette opération se généralise facilement à d'autres opérations comme le calcul de la moyenne, du minimum ou du maximum d'une colonne.

3.4 Expressions de l'algèbre relationnelle

Une fois les opérateurs relationnels identifiés il est alors facile de combiner ces opérations élémentaires pour construire des expressions de l'algèbre relationnelle permettant de fournir les réponses à des questions complexes sur la base.

Nous reprendrons comme exemple les trois tables représentant les commandes de produits à des fournisseurs présentées précédemment.

```
fournisseurs(fno,nom,adresse,ville)
produits(pno,design,prix,poids,couleur)
```

`commandes(cno,fno,pno,qute)`

– Déterminer les numéros de fournisseurs des différents 'Dupont'.

$$proj_{fno}(select_{nom='Dupont'}(fournisseurs))$$

– Déterminer les numéros de fournisseurs qui ont moins de 3 commandes.

$$proj_{fno}(select_{compte<3}(compte_{fno}(commandes)))$$

De la même manière que l'on dessine des arbres d'expressions pour les expressions arithmétiques, on peut représenter les expressions de l'algèbre relationnelle sous forme d'arbres. Les feuilles sont étiquetées par les tables à exploiter tandis que chaque nœud est constitué d'un opérateur relationnel.

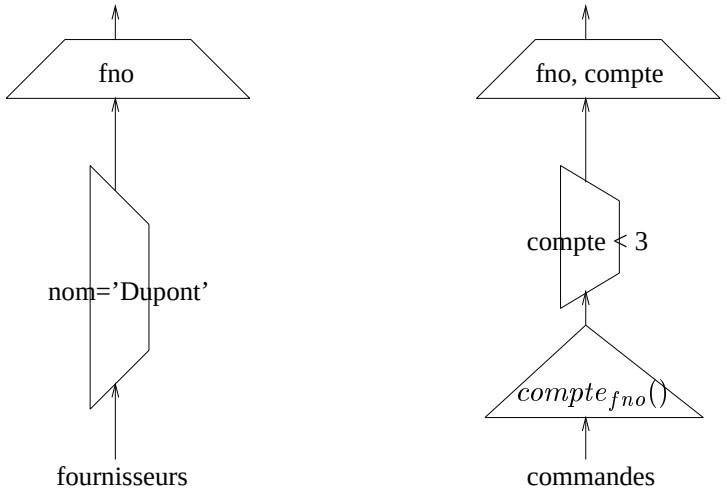


FIG. 3.11 – *Arbres de requêtes*

3.4.1 Pourquoi une requête est-elle meilleure qu'une autre?

Une requête n'est en général pas l'unique solution d'un problème comme le montre la question précédente. Dans ce cas, si la relation obtenue est identique avec les deux requêtes, l'efficacité est rarement la même. D'où l'utilité d'algorithmes d'optimisation automatique de requêtes que l'on trouve dans les SGBD dignes de ce nom.

Par exemple la requête consistant à "déterminer les références, prix et quantités des produits commandés en plus de 10 exemplaires par commande" peut s'écrire de plusieurs manières différentes plus ou moins efficaces.

– **(A)** $proj_{pno,prix,qute}(select_{qute>10}(join_{produits.pno=commandes.pno}(commandes, produits)))$
– **(B)** $proj_{pno,prix,qute}(join_{produits.pno=commandes.pno}(produits, select_{qute>10}(commandes)))$
– **(C)** $proj_{pno,prix,qute}(join_{p.pno=c.pno}(proj_{pno,prix}(produits), proj_{pno,qute}(select_{qute>10}(commandes))))$

Prenons pour hypothèse simplificatrice que chaque colonne nécessite 10 caractères en mémoire. La table `produit` nécessite donc 8 lignes $*$ 5 colonnes $*$ 10 caractères $=$ 400 caractères, tandis que la table `commandes` nécessite $10 * 4 * 10 = 400$ caractères.

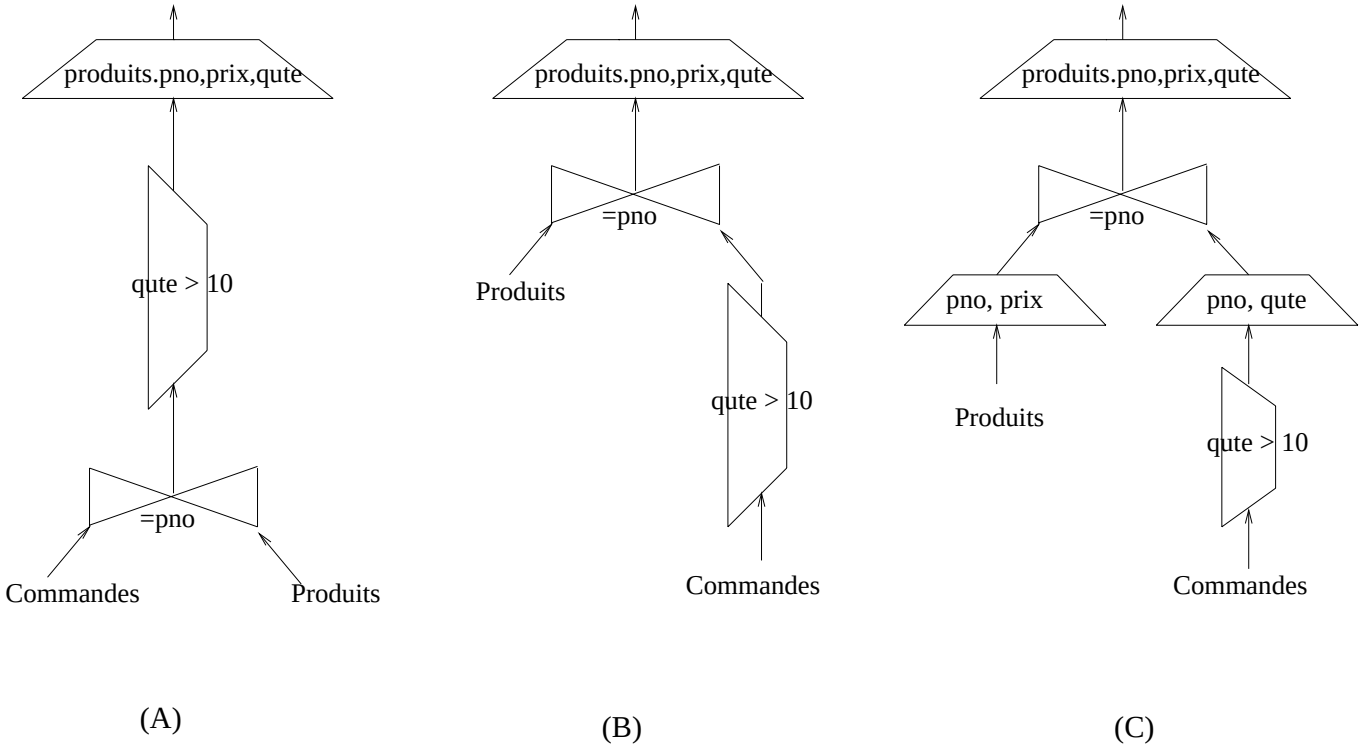


FIG. 3.12 – *Différents arbres pour la même requête*

Intéressons nous à la requête (A). Elle nécessite une opération de Jointure qui est en général coûteuse. En effet, la table temporaire la plus volumineuse pour effectuer cette requête est engendrée par le produit cartésien des deux tables `produit` et `commandes` nécessaire au calcul de la jointure. Cette table temporaire possède un volume de $400 * 400 = 160000$ caractères.

Si l'on effectue les projections et restrictions le plus tôt possible comme dans la requête (C), on ne conserve alors que le couple `(pno,prix)` de la table `produit` ce qui donne $8 * 2 * 10 = 160$ caractères et le couple `(pno,qté)` de la table `commandes` pour les quantités supérieures à 10 ce qui donne $2 * 2 * 10 = 40$ caractères. La jointure engendre une table temporaire de $40 * 160 = 6400$ caractères, d'où un gain de 75% (facteur 25) en taille mémoire nécessaire (et un gain en vitesse évidemment proportionnel).

Les tables utilisées dans cet exemple sont très petites et la requête encore très simple, mais on imagine facilement ce que ce genre d'optimisation permet de gagner sur des requêtes à 3 ou 4 jointures sur des tables de milliers d'enregistrements!

3.5 Quelques remarques sur l'algèbre relationnelle

- L'algèbre relationnelle permet l'étude des opérateurs entre eux (commutativité, associativité, groupes d'opérateurs minimaux etc ...). Cette étude permet de démontrer l'équivalence de certaines expressions et de construire des programmes d'optimisation qui transformeront toute demande en sa forme équivalente la plus efficace.
- D'une manière pratique, les opérations les plus utilisées sont la projection, la restriction et la jointure naturelle.
- L'opération de jointure est en général très coûteuse. Elle est d'ailleurs proportionnelle au

nombre de tuples du résultat et peut atteindre $m * n$ tuples avec m et n les nombres de tuples des deux relations jointes.

- Afin d'avoir des requêtes efficaces en temps il est toujours préférable de faire les restrictions le plus tôt possible afin de manipuler des tables les plus réduites possibles[1].
- Les opérations de l'algèbre relationnelle sont fidèles à certaines lois algébriques : commutativité, associativité etc... pour peu que l'ordre des colonnes soit sans importance. C'est pourquoi les colonnes sont toujours nommées.

1. Cette remarque est valable pour les personnes qui développent des interpréteurs de langages d'interrogation; les utilisateurs de QBE ou SQL n'ont pas vraiment le choix !

Chapitre 4

Les contraintes d'intégrité

La gestion automatique des contraintes d'intégrité est l'un des outils les plus importants d'une base de données. Elle justifie à elle seule l'usage d'un SGBD. Selon les contraintes, différentes anomalies que nous allons passer en revue peuvent se déclencher dès qu'un accès aux données (saisie, modification, effacement) est effectué.

Dès qu'un accès non conforme aux contraintes spécifiées dans la base survient, l'action effectuée est automatiquement rejetée puis soit présentée à l'utilisateur si le traitement se fait en interactif, soit rangée dans une table d'erreurs si le traitement se fait en batch.

4.1 Contraintes de clé

Le premier type de contraintes d'intégrité traité par un SGBD permet de vérifier la présence de clés uniques pour chacune des tables. Cette clé est nommée clé primaire de la table. Il ne peut y en avoir qu'une seule par table. Une clé primaire peut bien sûr être constituée de plusieurs colonnes. Quelle que soit la table, si une clé primaire est définie, elle doit être présente pour chaque enregistrement, elle doit être unique et aucun de ses constituants ne peut être `NULL`. Si la clé est omise, si elle est à la valeur `NULL` ou si elle a déjà été saisie pour un autre enregistrement de la table, une anomalie de clé est déclenchée.

Plusieurs regroupements d'attributs peuvent en général prétendre à être clé primaire. Ce sont les clés candidates. Les clés candidates qui n'ont pas êté choisies comme clé primaire sont alors déclarées avec la contrainte `UNIQUE`.

4.2 Contraintes de types de données

Le second type de contraintes d'intégrité traité par un SGBD permet ensuite de vérifier les types des données saisies (entiers, réels, dates, chaînes de caractères, booléens etc ...) et les domaines de validité pour chacune de ces données (date postérieure au 01/01/1980 pour la gestion des commandes d'une entreprise créée à cette date, entier compris entre 0 et 20 pour représenter une note d'étudiant, etc ...).

4.3 Contraintes d'intégrité référentielle

La gestion des contraintes d'intégrité permet aussi de vérifier automatiquement la présence de données référencées dans des tables différentes. Ce type de contrainte est appelé contrainte d'intégrité référentielle. Une contrainte d'intégrité référentielle peut s'appliquer dès qu'une clé primaire d'une table est utilisée comme référence dans une autre table. On la nomme alors clé étrangère de la seconde table. Par exemple, l'identifiant d'un produit est une clé étrangère dans la table des commandes, de même l'identifiant d'un bus est une clé étrangère de la table des lignes

de bus. Concrètement, les clés étrangères se trouvent dans toutes les tables possédant un champ issu d'associations du MCD. Comme pour la clé primaire, une clé étrangère peut être constituée de plusieurs colonnes. Contrairement aux clés primaires, la valeur `NULL` est acceptée dans une clé étrangère. Ce cas se produit notamment avec une contraintes du type `ON DELETE SET NULL` que nous verrons par la suite.

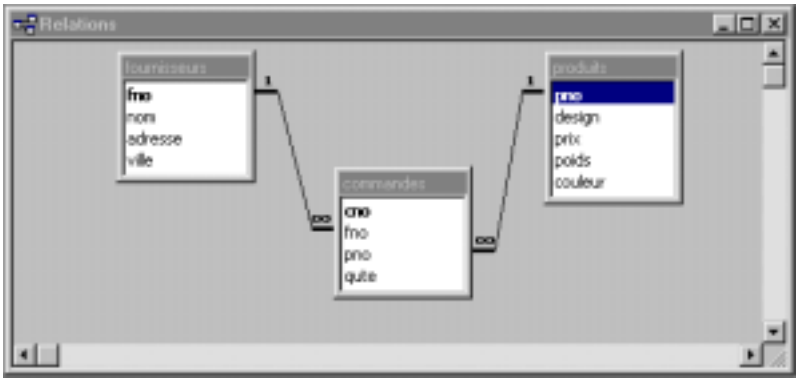


FIG. 4.1 – *Affichage des contraintes référentielles dans Access*

Bien évidemment, il est anormal qu'une clé étrangère apparaisse dans la base de données tandis que la clé primaire associée n'est pas présente. On n'imagine pas saisir une commande d'un produit dont la référence n'est pas présente dans la table des produits. De la même manière il ne serait pas très logique de mettre une note à un étudiant qui ne figure pas dans la table des étudiants. Si un tel cas se produit, la base est dite **incohérente**.

Reprenons par exemple les 3 tables `fournisseurs`, `commandes`, `produits` présentées page 19.

Le numéro de produit est une clé primaire de la table des produits et clé étrangère de la table des commandes. De même, le numéro de fournisseur est clé primaire de la table des fournisseurs et clé étrangère de la table des commandes.

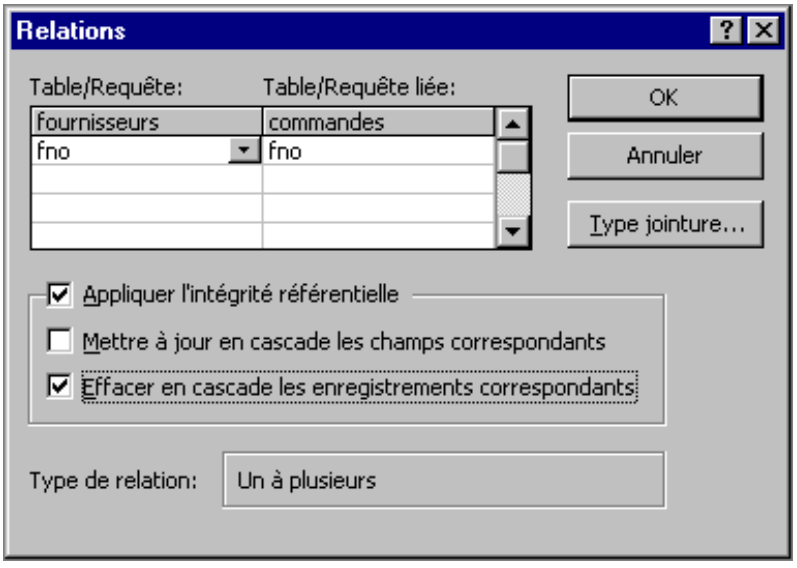


FIG. 4.2 – *Gestion des contraintes référentielles dans ACCESS*

Différentes anomalies peuvent se produire :

- **Anomalie de suppression :** si on supprime le fournisseur Lefebvre (numéro 17), 3 commandes de la table des commandes deviennent incohérentes. Elles doivent donc être supprimées.
- **Anomalie de modification :** si on modifie le numéro du fauteuil rouge que l'on transforme de 102 à 112, les deux enregistrements de la table des commandes le concernant doivent être modifiés.
- **Anomalie d'ajout :** si on tente d'ajouter une commande qui référence le produit 111, l'ajout est refusé puisque ce produit n'existe pas dans la table des produits. Il en ira de même si le fournisseur de la commande référence le numéro 14 qui n'existe pas non plus dans la table des fournisseurs.

Grâce à la gestion des contraintes d'intégrité, le SGBD s'occupe automatiquement, à chaque action sur les données (saisie, modification, effacement), de vérifier la cohérence de la base de données.

Cette vérification peut se faire de trois manières différentes selon les souhaits du concepteur :

1. Simple signalement d'une anomalie de présence. Dans ce cas un message apparaît et la mise à jour est refusée.
2. Effacement automatique des tuples qui référencent un objet qui n'existe plus dans la table principale.
3. Mise à jour automatique des tuples utilisant la clé étrangère qui référence une clé primaire venant de changer de valeur.

Selon les SGBD utilisés, d'autres types de contraintes peuvent encore être gérées, notamment via des procédures lancées automatiquement en cas d'ajout, de modification ou d'effacement de données (triggers). L'important est de noter que l'avantage qu'apporte un SGBD sur un système classique est que ces contraintes sont traitées au niveau des données et non pas placées dans les traitements.

Remarque : La sémantique de vérification des contraintes est différent selon que l'on considère un environnement transactionnel ou non. Quand une transaction est en cours d'exécution, les contraintes doivent-elles être vérifiées après chaque mise à jour ou bien uniquement à la fin de la transaction? Nous reparlerons de ce problème lorsque nous aborderons les transactions.

Chapitre 5

Le langage QBE

Le langage QBE, abréviation de "Query by Example" a été inventé par IBM en 1978 (Zloff) pour faciliter la construction de requêtes relationnelles grâce à un aspect graphique.

L'idée de base consiste à faire formuler la question à l'utilisateur par un exemple d'une réponse possible à la question souhaitée. Le principe de construction suivi est constitué de deux étapes :

1. Afficher le schéma des tables nécessaires à l'expression de la requête.
2. Remplir les colonnes avec les critères recherchés.

L'expression d'une requête élémentaire dans les colonnes se fait en respectant les points suivants [1]

- Les attributs à projeter sont définis par un `P` (Print) dans la colonne associée.
- Les constantes sont tapées directement dans la colonne de l'attribut concerné, précédées de l'opérateur de comparaison nécessaire ($=, \leq, \geq, >, <, \neq$). Si aucun opérateur n'est spécifié, le système considère qu'il s'agit d'une égalité (le symbole = peut donc être omis).
- La liaison entre deux tables se fait par l'utilisation de valeurs exemples soulignées (sorte de variables du calcul des prédicat). QBE associe alors automatiquement deux valeurs exemples identiques dans deux tables différentes.
- Par défaut, QBE élimine les doublons. Si on souhaite conserver les doublons, le mot clé `ALL` doit être placé sur la ligne du `P` concerné (c'est le contraire de `SQL`).
- Les opérateurs logiques `ET`, `OU`, `NON`, peuvent être utilisés dans les colonnes pour exprimer des liaisons entre critères de restriction.
- Les opérateurs arithmétiques `Somme`, `Moyenne`, `Compte`, `Maximum`, `Minimum` peuvent être utilisés dans les colonnes comme des constantes. Dès qu'une de ces opérations est utilisée, les autres colonnes sélectionnées sans opération arithmétique sont automatiquement considérées comme des colonnes de regroupement.
- les tris des résultats se font en mettant l'attribut `AO` (Ascendent order) ou `DO` (descendent order) sur la ligne du `P` correspondant.

Grâce à ce simple mécanisme graphique, il est alors possible d'exprimer très facilement un grand nombre de requêtes. En voici quelques exemples.

1. lister les numéros et noms des fournisseurs.

fournisseurs	fno	nom	adresse	ville
	P	P		

1. Nous n'avons conservé dans cette présentation de QBE que les points communs à toutes les implémentations.

2. lister les désignations de produits dont le poids est supérieur à 15.

produits	pno	design	prix	poids	couleur
		P		> 15	

3. lister les noms des fournisseurs avec les numéros de produits commandés ainsi que la quantité commandée.

commandes	cno	fno	pno	qute
		X	P	P

fournisseurs	fno	nom	adresse	ville
	X	P		

4. afficher les produits avec leurs couleurs respectives, triés sur le nom croissant et la couleur décroissante.

produits	pno	design	prix	poids	couleur
		P.AO			P.DO

5. Afficher la somme des quantités commandées par numéro de fournisseur.

commandes	cno	fno	pno	qute
		P		somme()

Contrairement à SQL, QBE n'est pas standardisé et chaque SGBD en fait sa propre implémentation. Chaque implémentation permet l'expression de toutes les requêtes de l'algèbre relationnelle avec une syntaxe qui lui est propre. Il est donc difficile de présenter ici ne serait-ce que les approches les plus fréquentes.

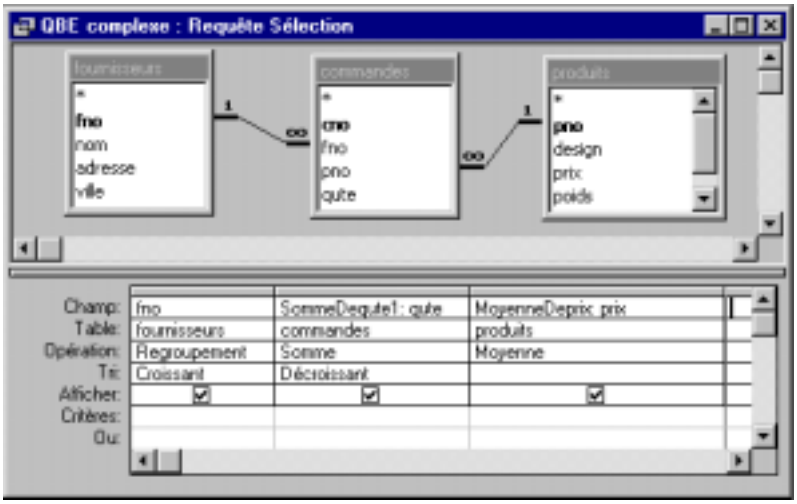


FIG. 5.1 – *Requête QBE sous Access*

Si les requêtes courantes (projections, restrictions et jointures) s'expriment facilement en `QBE`, il faut bien reconnaître que dès que la requête se complique, l'expression en `QBE` devient très délicate. D'autres langages prennent alors toute leur puissance ; c'est le cas de SQL.

Chapitre 6

Le langage SQL

SQL est un langage de définition et de manipulation de bases de données relationnelles. Son nom est une abréviation de "Structured Query Language" (langage d'interrogation structuré).

SQL est un standard qui a été normalisé par l'organisme `ANSI`. Il existe pour chaque produit `SQL` réalisé des différences ou des compléments par rapport à la norme.

Historiquement, après la découverte du modèle relationnel par E.F. Codd en 1970, plusieurs langages relationnels sont apparus dans les années suivantes dont SEQUEL d'IBM, puis SEQUEL/2 pour le System/R d'IBM en 1977, langage qui donna naissance à SQL. L'année 1981 a vu la sortie du premier SGBD relationnel implémentant SQL, le système Oracle. Ce n'est qu'en 1983 qu'IBM sortit DB2, héritier du System/R, lui aussi avec un langage de type SQL. Le langage a ensuite été normalisé en 1986 pour donner `SQL/86` puis légèrement modifié en 1989 pour donner `SQL/89` la version la plus utilisée actuellement. En 1992 de nombreuses améliorations ont été apportées à la norme pour donner `SQL/92` que l'on nomme généralement `SQL2`, beaucoup plus verbeuse que la précédente. Actuellement une version `SQL3` est en cours de rédaction. Elle intègre une couche objets supplémentaire mais n'est pas encore reconnue comme norme à la date d'aujourd'hui.

SQL contient un langage de définition de données, le DDL[1], permettant de créer, modifier ou supprimer les définitions des tables de la base par l'intermédiaire des ordres `Create`, `Drop`, et `Alter`. Il contient aussi un langage de manipulation de données, le DML[2], par l'intermédiaire des ordres `Select`, `Insert`, `Update`, `Delete`. Il contient enfin un langage de gestion des protections d'accès aux tables en environnement multi-utilisateurs par l'intermédiaire des ordres `Grant`, `Revoke`, le DCL[3].

DDL	DML	DCL
ALTER	DELETE	GRANT
CREATE	INSERT	REVOKE
COMMENT	SELECT	
DESCRIBE	UPDATE	
DROP		
RENAME		

FIG. 6.1 – *Ordres SQL principaux*

Une requête SQL peut être utilisée de manière interactive ou incluse dans un programme d'application (quel que soit le langage).

Toutes les instructions `SQL` se terminent par un point-virgule (;). Un commentaire peut être introduit dans un ordre `SQL` par les signes /* et */ ou par le caractère % qui traite toute la fin de ligne comme commentaire.

1. Data Definition Language
2. Data Manipulation Language
3. Data Control Language

6.1 La manipulation des données : le DML

Tout d'abord, pourquoi commencer par le DML ? La réponse est simple. De plus en plus d'outils comme MS-ACCESS permettent de créer des tables "à la souris". Pour de nombreux utilisateurs, le DML est donc le premier contact qu'ils ont avec SQL, c'est pourquoi nous commencerons par lui. Il est néanmoins évident que l'ordre chronologique d'utilisation d'une base de données nécessite l'usage du DDL avant le DML.

Dans cette section on parlera tout d'abord de l'ordre `SELECT` qui est sans doute le plus utilisé de tous les ordres `SQL`, puis des ordres `INSERT`, `UPDATE` et `DELETE`.

6.1.1 L'obtention des données

L'obtention des données se fait exclusivement par l'ordre `SELECT`. La syntaxe minimale de cet ordre est :

```
SELECT <liste des noms de colonnes> FROM <liste des noms de tables> ;
```

Nous verrons qu'elle peut être enrichie, de très nombreuses clauses permettant notamment d'exprimer les projections, les restrictions, les jointures, les tris etc ...

FROM
WHERE
GROUP BY
HAVING
ORDER BY

FIG. 6.2 – *Ordres des clauses du SELECT*

Expression des projections La projection d'une table en vue de l'obtention d'un ensemble de colonnes de cette table se fait simplement en spécifiant les noms des colonnes désirées. Si l'on souhaite obtenir toutes les colonnes d'une table le caractère '*' peut être utilisé avantageusement à la place de la liste des noms de colonnes.

R1 *lister la table fournisseurs*

```
SELECT * FROM fournisseurs ;
```

R2 *lister les numéros et noms des fournisseurs*

```
SELECT fno, nom FROM fournisseurs ;
```

SQL n'élimine pas les doubles à moins que ça ne soit explicitement précisé par le mot clé `DISTINCT`.

R3 *lister les différentes désignations de produits*

```
SELECT DISTINCT design FROM produits;
```

result	design
	fauteuil
	bureau
	armoire
	caisson
	classeur

Expression des restrictions La restriction s'exprime à l'aide de la clause `WHERE` que l'on ajoute à la requête juste après la liste des tables utilisées.

R4 *lister les données sur les produits dont le poids est supérieur à 15*

```
SELECT * FROM produits
WHERE poids>15 ;
```

Bien sur, l'association projection-restriction est triviale :

R5 *lister les désignations de produits différentes dont le poids est supérieur à 15*

```
SELECT DISTINCT design FROM produits
WHERE poids > 15 ;
```

Il est possible d'exprimer des qualifications complexes à l'aide des comparateurs arithmétiques `>,>=,<,<=,=,<>` et des opérateurs logiques `AND, OR, NOT`.

R6 *lister les produits dont le poids est compris entre 15 et 40*

```
SELECT * FROM produits
WHERE poids > 15 AND poids < 40 ;
```

result	pno	design	prix	poids	couleur
	103	bureau	3500	30	vert
	105	armoire	2500	35	rouge
	108	classeur	1500	20	bleu

D'autres prédicats peuvent encore être utilisés. Ils servent à définir des ensembles ou des valeurs approximatives. Ce sont les prédicats `IN,BETWEEN,LIKE` qui peuvent tous trois être préfixés par `NOT` pour exprimer la négation.

R7 *lister les fournisseurs habitant Lille, Lyon ou Nice*

```
SELECT * from fournisseurs
WHERE ville IN ('Lille','Lyon','Nice') ;
```

R8 *lister les produits dont le poids n'est pas compris entre 15 et 35*

```
SELECT * FROM produits
WHERE poids NOT BETWEEN 15 AND 35 ;
```

Le caractère de soulignement '_' remplace n'importe quel caractère tandis que le pourcentage '%' remplace n'importe quelle séquence de n caractères (n pouvant être égal à 0) [4].

R9 *lister les fournisseurs dont le nom ne commence pas par 'd'*

```
SELECT * FROM fournisseurs
WHERE nom NOT LIKE 'D%'
```

result	fno	nom	adresse	ville
	17	Lefebvre		Lille
	12	Jacquet		Lyon
	14	Martin		Nice
	11	Martin		Amiens
	19	Maurice		Paris

Le test de valeurs manquantes est aussi possible. Une valeur manquante est représentée par une valeur spéciale notée `NULL`. Le test d'une valeur `NULL` ne peut pas se faire avec l'égalité classique mais doit se faire obligatoirement avec les prédicats `IS NULL` ou `IS NOT NULL`.

R10 *lister les fournisseurs dont l'adresse n'est pas renseignée*

```
SELECT * FROM fournisseurs
WHERE adresse IS NULL;
```

4. MS-ACCESS prend à cet égard quelques libertés par rapport à la norme puisque les chaînes de caractères sont entre guillemets et que le symbole du joker est l'astérisque

R11 *lister les fournisseurs dont l'adresse est renseignée*

```
SELECT * FROM fournisseurs
WHERE adresse IS NOT NULL ;
```

Tri et présentation des résultats SQL permet de trier les tuples obtenus par la requête selon différents critères grâce à la clause `order by` . Les mots clés `asc` et `desc` permettent de préciser si le tri est croissant ou décroissant.

R12 *afficher les produits rouges ou verts ou bleus triés sur le nom croissant et la couleur décroissante*

```
SELECT design, couleur FROM produit
WHERE couleur IN ('rouge','vert','bleu')
ORDER BY design ASC, couleur DESC ;
```

Par défaut, les noms des colonnes de la table résultat de la requête sont les noms des colonnes spécifiées dans la liste des colonnes du SELECT. Il est néanmoins possible de changer le nom de la colonne à l'affichage en accolant au nom de colonne choisi le prédicat `AS` suivi du nouveau nom de colonne devant apparaître à l'affichage.

R13 *afficher les produits et leurs couleurs avec des noms de colonnes compréhensibles.*

```
SELECT design AS [désignation du produit], couleur AS [couleur du produit]
FROM produits;
```

result	designation du produit	couleur du produit
	armoire	rouge
	bureau	vert
	bureau	gris
	caisson	gris
	caisson	jaune
	classeur	bleu
	fauteuil	rouge
	fauteuil	gris

Expression des jointures Les jointures se font en spécifiant les tables sur lesquelles la jointure sera faite dans la liste des tables utilisées et en spécifiant la qualification de jointure dans la clause `WHERE`. On rappelle qu'une jointure sans qualification est un produit cartésien et qu'une jointure avec égalité est une équi-jointure.

R14 *lister le produit cartésien fournisseur* produits*

```
SELECT * FROM fournisseurs, produits ;
```

Jusqu'à présent nous n'utilisions qu'une seule table. Il n'y avait donc pas d'ambiguïté de nom sur les colonnes utilisées. Dès que deux tables sont utilisées, des ambiguïtés peuvent apparaître. Dans ce cas il faut préfixer le nom de colonne par le nom de la table concernée suivi d'un point.

R15 *lister les noms des fournisseurs avec les numéros de produits commandés ainsi que la quantité commandée*

```
SELECT fournisseur.nom, commandes.pno, commandes.qute
FROM fournisseurs, commandes
WHERE fournisseurs.fno = commandes.fno ;
```

Si les noms de tables qui servent de préfixes deviennent pénibles à taper, SQL offre de plus une possibilité de synonymes dans la clause `FROM`. Il suffit pour cela de préciser le synonyme juste après le nom de la table à l'aide du mot clé `AS` (facultatif) comme pour les synomymes de noms de colonne. La requête précédente devient alors

R16 *utilisation des synonymes de noms de tables*

```
SELECT f.nom, c.pno, c.qute
```

```
FROM fournisseurs AS f, commandes AS c
WHERE f.fno = c.fno ;
```

result	f.nom	c.pno	c.qute
	Lefebvre	103	10
	Durand	103	2
	Lefebvre	102	1
	Durand	108	1
	Maurice	107	12
	Durand	107	5
	Maurice	105	3
	Martin	103	10
	Dupont	102	8
	Lefebvre	108	15

La requête précédente ne faisait qu'une seule jointure, mais bien sûr, plusieurs jointures peuvent être exécutées en cascade.

R17 *lister les couples (nom de fournisseur, nom de produit) en commande*

```
SELECT f.nom AS fournisseur, p.design AS produit
FROM fournisseurs AS f, produits AS p, commandes AS c
WHERE f.fno = c.fno
AND   p.pno = c.pno ;
```

Rien n'empêche bien sûr de faire une jointure d'une table sur elle même. Dans ce cas on parle alors d'auto-jointure.

R18 *Lister les couples de références de fournisseurs situés dans la même ville.*

```
SELECT f1.fno, f2.fno
FROM fournisseurs AS f1, fournisseurs AS f2
WHERE f1.ville = f2.ville
AND f1.fno < f2.fno;
```

Les apports de SQL 2. SQL 2 (parfois appelé SQL/92) a apporté quelques nouvelles possibilités quant à l'expression des jointures. Avec SQL 2 il est possible de préciser les attributs de jointure dans la clause FROM du SELECT. Par exemple `FROM fournisseurs f JOIN commandes c ON f.fno=c.fno` permet d'exprimer une jointure entre les tables `fournisseur` et `commandes` sur l'attribut `fno`.

Outre la jointure classique, SQL 2 offre aussi la possibilité de définir des jointures externes. Une jointure interne ne fournit dans la table résultante que les tuples qui satisfont au critère de jonction. Dans une jointure externe l'une des tables utilisées dans la jointure est considérée comme directrice et aura tous ses enregistrements dans la table résultante même s'ils n'ont pas de valeurs correspondantes avec l'autre table de la requête. En effet il se peut parfois que l'on souhaite aussi conserver dans la table résultante des tuples qui ne peuvent pas être complétés par l'opération de jointure. Contrairement à la jointure interne, les jointures externes permettent de les conserver en complétant les colonnes manquantes par des valeurs NULL.

Par exemple la clause `FROM fournisseurs f JOIN LEFT commandes c ON f.fno=c.fno` permet d'exprimer une jointure entre les tables `fournisseur` et `commandes` sur l'attribut `fno` et conserver dans la table résultat les fournisseurs qui n'ont pas de commande en cours.

Pour exprimer les variantes de jointures externes, il existe notamment dans SQL 2 les ordres `JOIN LEFT` pour conserver les tuples de la table de gauche, `JOIN RIGHT` pour conserver les tuples de la table de droite et `JOIN FULL` pour conserver les deux. L'ordre `JOIN`peut être préfixé par `OUTER` pour marquer le fait que c'est une jointure externe, ou par `INNER` pour la jointure interne classique.

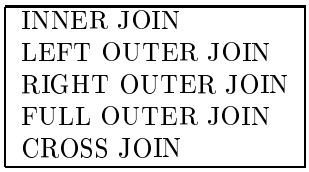


FIG. 6.3 – *Ordres de jointure appportés par SQL 2*

Il est à noter que de nombreux fournisseurs de SGBD prennent à cet égard quelques distances avec la norme. On trouvera par exemple la syntaxe `(+)` sur Oracle, la syntaxe `*=` sur Sybase. ACCESS quant à lui est fidèle à la norme.

R19 *afficher tous les produits de moins de 20Kg avec les quantités en cours de commande si possible*

```
SELECT p.pno, design, qute , poids
FROM produits AS p LEFT JOIN commandes AS c
ON c.pno = p.pno
WHERE poids < 20 ;
```

result	p.pno	design	qute	poids
	101	fauteuil		7
	102	fauteuil	1	9
	102	fauteuil	8	9
	106	caisson		12
	107	caisson	12	12
	107	caisson	5	12

R20 *afficher les produits qui ne sont pas commandés*

```
SELECT p.pno
FROM produits AS p LEFT JOIN commandes AS c ON c.pno = p.pno
WHERE c.pno is NULL ;
```

Les expressions de manipulation de données SQL permet bien sûr d'effectuer des calculs arithmétiques à l'affichage des colonnes. Il suffit pour cela de placer l'expresion à calculer comme champ d'affichage du `SELECT`. Ces valeurs calculées peuvent bien sûr être testées dans les clauses `HAVING` ou `WHERE`.

R21 *Prix des produits avec une TVA à 20,6%*

```
SELECT DISTINCT pno, design, prix*1.206 AS prixTTC
FROM produits ;
```

R22 *volume financier commandé pour les commandes de moins de 10 articles*

```
SELECT DISTINCT cno, qute*prix AS volume, qute
FROM produits INNER JOIN commandes ON produits.pno = commandes.pno
where qute <10 ;
```

result	cno	volume	qute
	1003	7000	2
	1005	1500	1
	1007	1500	1
	1013	5000	5
	1017	7500	3
	1023	12000	8

D'autre opérations sur chaque valeur sélectionnée peuvent être effectuées notamment sur les dates (`YEAR`, `MONTH`, `DAY`) et les chaînes de caractères (`SUBSTRING`, `UPPER`, `LOWER`, `CHARACTER_LENGTH`). C'est ainsi par exemple que les responsables systèmes obtiennent les noms de login des utilisateurs en écrivant une requête affichant une combinaison des caractères du nom et du prénom chaque l'utilisateur.

Ces fonctions donent bien sûr de la puissance au SGBD er c'est souvent sur ce point que les différentes implémentations ne respectent pas la norme, chacun voulant apporter des fonctionnalités supplémentaires que l'autre n'a pas !

Les fonctions statistiques Les fonctions précédentes permettent de manipuler les données d'un tuple à la fois. Mais l'utilisateur a souvent besoin d'autres fonctions que celles présentées précédemment. Notamment, il lui faut parfois pouvoir manipuler des fonctions de regroupement "inter tuples". SQL offre la possibilité de récupérer des données chiffrées sur les tables ou des parties de tables. Il est par exemple possible d'obtenir le nombre de tuples répondant à un critère, la valeur moyenne d'une colonne, la valeur maximum ou minimum et la somme d'une colonne.

AVG	moyenne
COUNT	nombre d'éléments
MAX	maximum
MIN	minimum
SUM	somme

FIG. 6.4 – *fonctions statistiques principales*

R23 *compter le nombre de commandes*

```
SELECT COUNT(*) FROM commandes ;
```

result	Nbre
	2

R24 *afficher la somme de toute les quantités commandées*

```
SELECT SUM(qute)
FROM commandes ;
```

R25 *compter le nombre de livraisons du produit numéro 102.*

```
SELECT COUNT(*)
FROM commandes
WHERE pno=102 ;
```

Attention, dans certains SGBD comme ACCESS, si un nom de colonne est spécifié dans `COUNT` au lieu de l'étoile, le mot clé `DISTINCT` doit impérativement être utilisé. La norme SQL contient aussi le mot clé `ALL` qui peut être utilisé avec la fonction `COUNT` et qui permet de compter les doublons par opposition à `DISTINCT`. Néanmoins, compter toutes les valeurs, doublons compris, revient à compter toutes les lignes. La forme `COUNT(ALL attribut)` est donc généralement remplacée par `COUNT(*)`.

R26 *compter les noms de fournisseurs différents*

```
SELECT COUNT(DISTINCT nom)
FROM fournisseurs ;
```

R27 *récupérer toutes les statistiques sur les quantités en commande*

```
SELECT COUNT(*),SUM(qute),MAX(qute),MIN(qute),AVG(qute)
FROM commandes ;
```

Il est important de noter que les opérations statistiques précédentes ne peuvent pas s'imbriquer. Il est par exemple interdit d'écrire quelque chose comme `AVG(SUM(montant))`. Ce type de calcul ne pourra s'obtenir qu'avec des sous-requêtes ou des vues comme nous le verrons plus tard.

Les regroupements Il est souvent intéressant de regrouper les données d'une table en sous-tables pour y faire des opérations par groupes. Par exemple compter les commandes par fournisseur. Ceci se fait avec la clause `GROUP BY` suivie de la colonne à partitionner.

R28 *afficher la somme des quantités commandées à chaque fournisseur*

```
SELECT SUM(qute)
FROM commandes
GROUP BY fno ;
```

R29 *lister le nombre de commandes par fournisseur*

```
SELECT fno,COUNT(*) as Nbre
FROM commandes
GROUP BY fno ;
```

result	fno	Nbre
	10	1
	13	1
	14	1
	15	2
	17	3
	19	2

Remarque : si dans un `SELECT` certaines colonnes sélectionnées utilisent des fonctions de calcul et d'autres non, alors toutes les colonnes sélectionnées qui n'utilisent pas de calcul doivent être spécifiées dans le `GROUP BY` (c'est le cas dans la requête précédente de l'attribut `fno`).

La clause `HAVING` étroitement liée au `GROUP BY` permet d'exprimer une restriction sur les lignes de la table obtenue avec les fonctions de calcul. Les tests sur les colonnes simples se font dans la clause `WHERE` tandis que les tests sur les fonctions de regroupement se font dans la clause `HAVING`.

R30 *lister les fournisseurs qui ont moins de 3 commandes.*

```
SELECT fno, count(*)
FROM commandes
GROUP BY fno
HAVING COUNT(*)<3 ;
```

Les fonctions arithmétiques et statistiques testées dans une clause `HAVING` n'ont pas à être forcément sélectionnées comme colonne résultat. Il est possible d'utiliser la fonction uniquement dans la clause `HAVING` sans qu'elle soit projetée.

R31 *référence des fournisseurs qui fournissent plus d'un produit.*

```
SELECT fno
FROM fournisseurs
GROUP BY fno
HAVING COUNT(*) > 1 ;
```

Nous en arrivons finalement à la forme la plus générale du `SELECT` avec toutes les clauses possibles pouvant figurer dans une requête de sélection:

R32 *On ne s'intéresse ici qu'aux commandes dont le numéro est supérieur à 6 et, pour ces commandes, on souhaite lister les noms des fournisseurs avec les quantités maximum commandées, dont la moyenne des quantités commandées est supérieure à 50. Le résultat doit être trié par ordre décroissant du nom de fournisseur.*

```
SELECT nom, max(qute)
FROM commandes AS c , fournisseurs AS f
WHERE cno > 6 AND c.fno = f.fno
GROUP BY fno
HAVING AVG(qute) > 50
ORDER BY nom DESC ;
```

Les sous-requêtes SQL donne la possibilité d'utiliser des sous-requêtes afin de décrire des requêtes complexes permettant d'effectuer des opérations dépendant d'autres requêtes.

La sous-requête est toujours placée dans la clause `WHERE` ou `HAVING` en lieu et place d'une constante et doit renvoyer soit une valeur unique (qui est alors utilisée avec les opérateurs classiques du where comme n'importe quelle constante), soit une colonne unique (qui est alors utilisée avec les opérateurs `IN` et `EXISTS`, ou avec les comparateurs classiques suivis de `ALL` ou `ANY`). Attention : la sous-requête doit suivre obligatoirement l'opérateur de comparaison, le contraire est interdit.

Les différentes formes de tests avec sous-requêtes sont donc les suivants (dans les exemples suivants le comparateur > peut évidemment être remplacé par n'importe quel autre comparateur).

- `SELECT ... WHERE fno > 125`
  Sélection classique, par comparaison avec une constante. Cette constante peut être remplacée par une sous-requête qui renvoie une seule valeur.
- `SELECT ... WHERE fno > (SELECT fno from ...)`
  Sous-requête dans laquelle un champ est comparé au résultat de la requête. La sous-requête doit renvoyer une seule valeur (une table d'une seule ligne et une seule colonne).
- `SELECT ... WHERE fno IN (SELECT fno from ...)`
  Ici la sous-requête renvoie plusieurs valeurs (une table de plusieurs lignes et une seule colonne). Pour obtenir un succès, la valeur testée doit être présente dans le résultat de la sous-requête. La forme `NOT IN` est aussi autorisée.
- `SELECT ... WHERE fno > ALL (SELECT fno from ...)`
  Sous-requête qui renvoie plusieurs valeurs. Pour obtenir un succès, la valeur testée doit être supérieure à toutes les valeurs obtenues par la sous-requête.
- `SELECT ... WHERE fno > ANY (SELECT fno FROM ...)`
  Sous requête qui renvoie plusieurs valeurs. Pour obtenir un succès, la valeur testée doit être supérieure à au moins une valeur obtenue par la sous-requête. La forme `= ANY` est équivalente à la forme `IN`
- `SELECT ... WHERE EXISTS (SELECT .... WHERE fno = ...)`
  Sous-requête qui renvoie plusieurs valeurs. Pour obtenir un succès la sous-requête doit donner un succès. Ce type de sous-requête est dit "corrélée", car un champ de la requête principale est utilisé dans la sous-requête. La forme `NOT EXISTS` est aussi autorisée.

R33 *lister les fournisseurs qui habitent la même ville que le fournisseur 10.*

```
SELECT nom
FROM fournisseurs
WHERE ville = (SELECT ville
               FROM fournisseurs
               WHERE fno=10) ;
```

R34 *lister les fournisseurs d'au moins un des produits fournis aussi par un fournisseur d'un produit rouge.*

```
SELECT DISTINCT fno
FROM commandes
WHERE pno IN (SELECT DISTINCT pno % produit fourni par
              FROM commandes
              WHERE fno IN (SELECT DISTINCT fno % fournisseurs de produits rouges
                            FROM commandes
                            WHERE pno IN (SELECT pno
                                          FROM produits
                                          WHERE couleur='rouge' )));
```

Pour résoudre certaines requêtes, il est parfois plus facile de traiter la forme inverse. Par exemple, pour obtenir les noms des fournisseurs livrant tous les produits, il est plus facile de rechercher les fournisseurs pour lesquels il n'existe aucun produit non livré. Cette sélection s'écrit

avec une magnifique requête à deux sous requêtes corrélées qui travaillent sur 3 tables simultanément !

R35 *lister les noms des fournisseurs livrant tous les produits.*

```
SELECT fnom
FROM fournisseurs f
WHERE NOT EXISTS (SELECT *
                  FROM produits p
                  WHERE NOT EXISTS (SELECT *
                                    FROM commandes c
                                    WHERE f.fno=c.fno
                                    AND c.pno=p.pno) ) ;
```

Cette requête est dit "corrélative" car l'une des données du sous-select est comparée à une donnée du select principal. Ce type de requête prend évidemment beaucoup de temps de calcul puisque la sous-requête doit être réévaluée pour chaque tuple de la requête principale.

Remarque Les prédicats `ALL` et `ANY` peuvent toujours être reformulés comme des tests d'existence. En effet, si l'on note $ le comparateur arithmétique utilisé on voit immédiatement que :

`... WHERE x $ ANY (SELECT y FROM t WHERE p)` est sémantiquement équivalent à
`... WHERE EXISTS (SELECT * FROM t WHERE (p) AND x $ t.y)`

De même,

`... WHERE x $ ALL (SELECT y FROM t WHERE p)` est sémantiquement équivalent à
`... WHERE NOT EXISTS (SELECT * FROM t WHERE (p) AND NOT (x $ t.y)`

R36 *Lister les références des fournisseurs livrant au moins un produit en quantité supérieure à chacun des produits livrés par le fournisseur 19.*

```
SELECT DISTINCT fno
FROM commandes
WHERE qute > ALL
      ( SELECT qute
        FROM commandes
        WHERE fno=19 ) ;
```

Ou aussi

R37

```
SELECT DISTINCT c1.fno
FROM commandes c1
WHERE NOT EXISTS
          ( SELECT *
            FROM commandes c2
            WHERE c2.fno = 19
            AND c2.qute >= c1.qute) ;
```

Une sous-requête comme celle de l'exemple précédent est dite corrélée. Cela signifie que sa valeur dépend d'une variable recevant sa valeur de la requête principale. Ce type de sous-requête doit être réévalué de manière itérative pour chaque nouvelle valeur de la variable en question et non pas une fois pour toutes.

Les opérateurs ensemblistes Comme dans l'algèbre relationnelle, ils sont au nombre de trois et s'intercalent entre deux `SELECT`.

– `UNION`: fournit la réunion des tuples des 2 select

- `INTERSECT` : fournit l'intersection des tuples des 2 select
- `EXCEPT` : fournit les tuples du premier select qui ne sont pas dans le second.

Ces opérateurs ne sont valides qu'à la condition que les i^{emes} colonnes de chacune des tables aient exactement la même description (type et longueur). On notera néanmoins que `INTERSECT` et `EXCEPT` ne font pas partie de la norme SQL et qu'ils ne sont pas implémentés dans tous les SGBD.

L'opérateur `UNION` supprime par défaut les doublons de l'opération. Si les doublons sont souhaités il faut utiliser la forme `UNION ALL`.

R38 *lister les numéros de produits dont le poids est supérieur à 20 ainsi que les produits commandés par le fournisseur 15*

```
SELECT pno FROM produits WHERE poids>20
UNION
SELECT pno FROM commandes WHERE fno=15 ;
```

Remarque Tout arbre relationnel peut être codé avec SQL (puisque les opérateurs de base le sont) mais pas systématiquement en une seule requête. C'est par exemple le cas pour l'opérateur Quotient qui nécessite pour être implémenté, la création de deux requêtes.

6.1.2 Récapitulatif

Union
$table1(a,b,c) \cup table2(d,e,f)$
	`SELECT * FROM table1`
	`UNION`
	`SELECT * FROM table2;`

Différence
$table1(a,b,c) - table2(d,e,f)$
	`SELECT a,b,c FROM table1`
	`EXCEPT`
	`SELECT d,e,f FROM table2;`
	
	et si EXCEPT n'est pas implémenté
	`SELECT a,b,c`
	`FROM table1`
	`WHERE NOT EXISTS`
	`(SELECT d,e,f`
	`FROM table2`
	`WHERE a = d`
	`AND b = e`
	`AND c = f);`

Produit cartésien $table1(a,b,c) * table2(d,e,f)$	`SELECT * FROM table1, table2;`
Projection $\pi_{a,b} table1(a,b,c)$	`SELECT DISTINCT a,b FROM table1;`
Restriction $\sigma_{a>b} table1(a,b,c)$	`SELECT * FROM table1` `WHERE a>b;`

Intersection
$table1(a,b,c) \cap table2(d,e,f)$
	`SELECT a,b,c FROM table1`
	`INTERSECT`
	`SELECT d,e,f FROM table2;`
	
	et si INTERSECT n'est pas implémenté
	
	`SELECT a,b,c FROM table1`
	`WHERE EXISTS`
	`( SELECT d,e,f FROM table2`
	`WHERE a = d`
	`AND b = e`
	`AND c = f);`

Quotient
$table1(a,b,c,d)/table2(c,d)$
	`CREATE VIEW v AS`
	`SELECT a,b,table2.c,table2.d`
	`FROM table1,table2`
	`EXCEPT`
	`SELECT a,b,c,d FROM table1 ;`
	
	`SELECT a,b FROM table1`
	`EXCEPT`
	`SELECT a,b FROM v ;`

$\theta-$Jointure $\theta_{a>d}(table1(a,b,c), table2(d,e,f))$	`SELECT * FROM table1, table2` `WHERE a > d;`
Equi-Jointure $\theta_{a=b}(table1(a,b,c), table2(d,e,f))$	`SELECT * FROM table1, table2` `WHERE a = b;`
Compte $Compte_b(table1(a,b,c))$	`SELECT count(*) FROM table1` `GROUP BY b;`
Somme $Somme_b(table1(a,b,c),c)$	`SELECT SUM(c) FROM table1` `GROUP BY b;`

6.1.3 La mise à jour d'informations

Trois types de mise à jour sont nécessaires pour le contenu d'une table relationnelle. L'ajout de nouveaux tuples, le changement de certains tuples et la suppression de certains tuples.

Insertion. L'ordre `INSERT` permet d'ajouter des lignes dans une table. Dans sa forme la plus générale, SQL demande que les noms des colonnes soient explicitement citées. Les valeurs insérées sont alors en correspondance avec l'ordre dans lequel les colonnes sont citées. Si l'ordre `INSERT` contient une clause `VALUE` alors une seule ligne est insérée dans la table ; si l'ordre `INSERT` contient une clause `SELECT` alors plusieurs lignes peuvent être simultanément insérées dans la table. L'insertion de tuples peut donc se faire soit en extension par la clause `VALUE` soit en intension par un ordre `SELECT` imbriqué.

R39 *ajouter un nouveau produit*

```
INSERT INTO produits (pno,design,prix,poids,couleur)
VALUES (8,ecrou,5,12,vert) ;
```

R40 *ajouter dans la table* `petites_commandes` *les numéros et quantités des produits commandés en moins de 5 exemplaires*

```
INSERT INTO petites_commandes(pno,qute)
       SELECT pno,qute FROM commandes
       WHERE qute < 5 ;
```

Si une valeur est insérée dans toutes les colonnes de la table, l'énumération des colonnes est facultatif. La requête précédente peut donc s'écrire de la manière suivante :

R41 *ajouter un nouveau produit (2eme méthode)*

```
INSERT INTO produits
VALUES (8,ecrou,5,12,vert) ;
```

SQL accepte que tous les noms de colonne de la table dans laquelle les tuples seront insérés ne soient pas précisés. Dans ce cas, les colonnes non précisées sont alors automatiquement remplies avec la valeur `NULL`. Il est néanmoins conseillé de toujours préciser toutes les colonnes quitte à préciser les valeurs `NULL` manuellement, afin d'éviter de fâcheuses erreurs de programmation.

R42 *ajouter un nouveau produit dont on ne connait pas le poids et la couleur*

```
INSERT INTO produits (pno,design,prix)
VALUES (8,ecrou,5) ;
```

ou aussi

```
INSERT INTO produits (pno,design,prix,poids,couleur)
VALUES (8,ecrou,5,NULL,NULL) ;
```

ou encore

```
INSERT INTO produits
VALUES (8,ecrou,5,NULL,NULL) ;
```

On notera que dans un ordre `INSERT` utilisant un `SELECT`, une constante peut être placée dans la requête en lieu et place d'un nom de colonne. Ceci permet d'inserer facilement des tuples dans la base avec des valeurs par défaut.

R43 *ajouter dans la table* `braderie` *les numéros de produits entre 100F et 500F et les afficher à un prix de 100F*

```
INSERT INTO braderie(pno,prix)
       SELECT pno, 100 FROM produits
       WHERE produit.prix < 500  and produit.prix > 100 ;
```

Une table citée dans le champ `INTO` de l'ordre `INSERT` ne peut pas être citée dans le champ `FROM` du sous-select de ce même `INSERT`. Il n'est donc pas possible d'insérer des éléments dans une table à partir d'une sous-sélection de cette même table.

R44 *Requête interdite : la duplication des éléments d'une table par un insert avec sous-select sur la même table.*

```
INSERT INTO produits
       SELECT * FROM produits ;
```

Mise à jour L'ordre `UPDATE` permet de modifier des lignes dans une table. L'expression caractérisant la modification à effectuer peut être une constante, une expression arithmétique ou le résultat d'un select imbriqué.

R45 *Augmenter le prix de tous les produits rouges de 5%*

```
UPDATE produits
SET prix = prix*1.05
WHERE couleur='rouge' ;
```

L'ordre `UPDATE` peut parfois poser des problèmes d'intégrité de la base. Par exemple, si une table possède comme clé unique un entier de 1 à n, que doit faire le SGBD lors d'une incrémentation de 1 de cette clé?

R46 *UPDATE qui peut poser des problèmes d'intégrité si clé est unique*

```
UPDATE table
SET cle=cle+1;
```

Certains SGBD vérifient la cohérence de la base uniquement après chaque ordre SQL. C'est le cas d'Oracle ou DB2. Dans ce cas cette requête ne pose aucun problème. D'autres en revanche vérifient la cohérence de la base après chaque modification de ligne. C'est le cas d'Access. Dans ce cas il risque d'y avoir un problème d'unicité de la clé durant l'exécution de la requête. Ce choix du mode de vérification de l'intégrité de la base n'est toujours pas tranché et vous trouverez selon les SGBD les deux modes d'exécution. D'une manière générale il n'est pas conseillé de faire des `UPDATE` sur des colonnes utilisées dans une clé primaire.

Suppression. L'ordre `DELETE` permet de supprimer des lignes dans une table selon une qualification fixée.

R47 *Supprimer les produits dont le prix est supérieur à 100F*

```
DELETE FROM produits WHERE prix>100;
```

La qualification peut être constituée d'un test simple comme précédemment, ou d'un test plus complexe comme le test d'existance d'un attribut dans une autre table.

R48 *Supprimer les produits dont le numéro figure dans la table TEMP*

```
DELETE FROM produits
WHERE pno IN (SELECT pno from TEMP);
```

L'ordre `DELETE FROM <table>;` permet de vider complètement une table. Néanmoins, dans ce cas, la table existe toujours bien qu'elle soit vide.

La procédure d'effacement de données dans une table est toujours une opération délicate. Une fois les données détruites, impossible de les retrouver. Il faut donc travailler avec méthode ! tout d'abord il faut tester la clase `WHERE` dans un `SELECT` simple pour s'assurer que les tuples sélectionnés sont bien ceux qu'il faut effacer. Ensuite, il est conseillé de sauvegarder les tuples à effacer dans une table temporaire. Nous étudierons dans la section suivante l'instruction `CREATE ... AS SELECT` qui permet de créer une table à partir d'une sélection. Une fois ces opérations faites, vous pouvez détruire les tuples sans crainte ! un retour arrière sera toujours possible.

6.2 La définition des données : Le DDL

Avant de parler précisément des ordres de description de données (le DDL) précisons brièvement les types de données autorisés.

6.2.1 Les types de données

Les types de données que l'on peut représenter dans un SGBD sont fortement dépendants de l'architecture de l'ordinateur sur lequel il tourne. Il existe donc de nombreuses variantes de types selon les SGBD. Néanmoins certains types se retrouvent dans la plupart des SGBD. Voici les principaux :

Le type alphanumérique :

CHAR (n)	Longueur fixe de n caractères. n_max : 16 383
VARCHAR(n)	Longueur variable, n représente le maximum

Le type numérique :

NUMBER (n,[d])	Nombre de n chiffres dont d après la virgule.
SMALLINT	Mot signé de 16 bits (-32768 à 32767)
INTEGER	Double mot signé de bits (-2E31 à 2E31 -1)
FLOAT	Numérique flottant

Le type gestion du temps:

DATE	Champ date (ex 24/02/1997)
TIME	Champ heure (ex 14:45:10.95)
TIMESTAMP	regroupe DATE et TIME

6.2.2 La création de tables

Une fois les types de données définis il est alors possible de créer les tables. La commande `CREATE TABLE` permet de définir des colonnes, de leur associer un type de données et d'y ajouter des contraintes à vérifier. La syntaxe la plus simple est la suivante :

```
CREATE TABLE <nom-de-table> ( <nom-de-colonne> <Type de données>, ...);
```

Dans la plupart des SGBD, le nom de la table doit commencer par une lettre et on autorise 254 colonnes maximum par table.

R49 *création de la table département avec un numéro et un nom :*

```
CREATE TABLE département
         (numdept NUMBER(3),
          nomdept CHAR(10));
```

Il est aussi possible de créer une table en insérant directement des lignes à la création. Dans ce cas, les colonnes existant ailleurs, il est inutile de les spécifier à nouveau.

```
CREATE TABLE <nom-de-table>
       ( <nom-de-colonne> <Type de données>, ...)
         AS SELECT <nom-de-champ>,... FROM <nom-de-table>
         WHERE <prédicat>;
```

Dans le cas d'une telle requête, seules les données du `SELECT` sont insérées dans la nouvelle table. Il est bien évident que les contraintes de types, de clé ou d'intégrités de l'ancienne table ne sont pas recopiées dans la nouvelle.

R50 *création de la table* `bonus` *avec insertion des noms et des salaires des chefs de services de la table* `employé`.

```
CREATE TABLE BONUS (nom,salaire)
AS SELECT nom,salaire FROM employé
WHERE métier = 'Chef de service';
```

L'instruction `CREATE ... AS SELECT ...` est en pratique très importante. En effet, la suppression de tuples dans une table peut aussi se faire avec une sous-requête `SELECT`. Or il est toujours conseillé avant de détruire des tuples d'une table de tester la sous-requête en sélection seule puis de sauvegarder temporairement les tuples à détruire. C'est donc l'instruction `CREATE ... AS SELECT...` qui résoud ce problème.

Les SGBG contiennent rapidement de très nombreuses tables. Il est alors difficile de connaître les structures de toutes les tables. La description d'une table peut être obtenue par l'ordre `DESCRIBE` qui affiche à l'écran la structure complète de la table choisie.

R51 *Afficher la structure de la table Fournisseurs*

```
DESCRIBE fournisseurs;
```

6.2.3 Expression des contraintes d'intégrité

Nous avons vus dans les chapitres précédents les différentes contraintes d'intégrité qu'il était possible d'exprimer dans un SGBD. Le DDL doit bien sûr offrir la possibilité d'exprimer ces contraintes dans la requête de création de table. Associée à la définition d'une colonne, il est possible d'accoler différentes clauses gérant ces contraintes, bien qu'aucune ne soit obligatoire :

CONSTRAINT	permet de nommer une contrainte
DEFAULT	précise une valeur par défaut
NOT NULL	force la saisie de la colonne
UNIQUE	vérifie que toutes les valeurs sont différentes
CHECK	vérifie la condition précisée

FIG. 6.5 – *contraintes de colonne*

Il est notamment possible de :

- Nommer une contrainte de colonne : `CONSTRAINT nom` permet de nommer une contrainte. Ce nom est automatiquement affiché par le SGBD dès que la contrainte n'est pas vérifiée. Si cette clause n'est pas spécifiée, le SGBD attribue à la contrainte un nom interne peu explicite. Il est donc conseillé de nommer notamment les contraintes d'intégrité référentielle.
- Définir une valeur par défaut : `DEFAULT` permet de préciser une valeur par défaut qui sera automatiquement introduite si l'utilisateur ne fournit pas de valeur par `INSERT`. Les valeurs suivantes sont généralement admises : une constante numérique ou alphanumérique, `USER` pour le nom de l'utilisateur, `NULL`, `CURRENT_DATE` pour la date de saisie, `CURRENT_TIME` pour l'heure de saisie, `CURRENT_TIMESTAMP` pour le moment de saisie.
- Refuser une valeur nulle : `NOT NULL` permet de vérifier à la saisie d'un tuple si la valeur est effectivement renseignée.
- Tester la validité d'une valeur : `CHECK(condition)` permet de spécifier un test à effectuer pour valider la saisie. La condition peut aller du simple test jusqu'au `SELECT` sophistiqué. Cette contrainte n'est vérifiée que quand la valeur saisie est différente de `NULL`.

En plus des contraintes de colonnes, il est possible spécifier des contraintes qui s'appliquent globalement à la table. Ceci est notamment nécessaire lorsque la contrainte s'applique sur plusieurs colonnes simultanément comme pour la définition de clés primaires ou étrangères.

Il est donc possible à la création d'une table de:

- Nommer une contrainte de table: `CONSTRAINT nom`.
- Définition d'une clé: `PRIMARY KEY(liste de cols)` permet de spécifier que la colonne définit la clé primaire de la table. La clé primaire peut porter sur plusieurs colonnes. Il n'y a bien sûr qu'une clause de ce genre par table. Automatiquement les colonnes qui constituent la clé primaire ne peuvent plus être nulles et chaque clé doit être unique. Dans la majorité des SGBD un index est automatiquement créé sur cette clé.
- Intégrité référentielle: `FOREIGN KEY (liste-col1) REFERENCES table(liste-col2)` permet de spécifier que les colonnes liste-col1 de la table en cours de définition référencent la clé primaire liste-col2 de la table étrangère spécifiée. La clé étrangère peut porter sur plusieurs colonnes. Il peut bien sûr y avoir plusieurs clés étrangères dans une même table. Les modifications automatiques à faire sur les clés étrangères en cas de changement de la clé primaire associée sont précisées par les clauses `ON DELETE` et `ON UPDATE`.
  - `ON DELETE {RESTRICT | CASCADE | SET NULL | SET DEFAULT}`
    `RESTRICT` donne un échec si la valeur effacée correspond à la clé de la table référencée, `CASCADE` signifie que les lignes correspondantes seront effacées en cascade en cas d'effacement d'une ligne de la table référencée, `SET NULL` place une valeur nulle dans la ligne de la table en cas d'effacement d'une ligne de la table référencée et `DEFAULT` place dans ce cas la valeur par défaut.
  - `ON UPDATE {RESTRICT | CASCADE | SET NULL | SET DEFAULT}`
    `RESTRICT` donne un échec si la valeur modifiée correspond à la clé de la table référencée, `CASCADE` signifie que les lignes correspondantes seront modifiées en cascade en cas de modification d'une ligne de la table référencée, `SET NULL` place une valeur nulle dans la ligne de la table en cas de modification d'une ligne de la table référencée et `DEFAULT` place dans ce cas la valeur par défaut.

  Dans les deux cas, `RESTRICT` est l'option par défaut. Notons que la valeur `NULL` est acceptée dans les colonnes qui constituent la clé étrangère bien que cela soit certainement signe d'un défaut d'analyse au départ. Les contraintes de référence ne sont vérifiées pour un tuple que quand toutes les valeurs qui constituent la clé étrangère sont différentes de `NULL`.

Une convention généralement admise consiste à nommer les contraintes d'intégrité référentielle par un nom préfixé par `PK` pour la clé primaire et `FK` pour les clés étrangères.

CONSTRAINT	permet de nommer une contrainte
PRIMARY KEY	déclare que la colonne est clé primaire
FOREIGN KEY	déclare que la colonne est clé étrangère

FIG. 6.6 – *contraintes de table*

On notera que les contraintes de tables peuvent aussi s'exprimer sous forme de contraintes de colonnes si elles ne portent que sur des colonnes uniques. En effet `SQL` autorise, bien que cette définition fasse perdre la lisibilité, à spécifier `PRIMARY KEY` aussi bien à la définition de la colonne qu'à la fin de l'ordre de création de la table.

`SQL2` offre aussi la possibilité de déclarer des assertions, formules qui doivent toujours être satisfaites. Ces assertions sont encore peu implémentées dans les SGBD actuels, notamment à cause du coût qu'elles impliquent. En effet, sous leur forme générale elles doivent être vérifiées par le SGBD à chaque modification des données ce qui implique un traitement très lourd en temps de calcul.

R52 *Toujours s'assurer que l'on a au moins 5 fournisseurs*

```
CREATE ASSERTION macontrainte CHECK
((SELECT count(*) FROM fournisseurs ) > 5) ;
```

6.2.4 Script DDL de création de base

```
-- ================================================
--   Creation Table : Fournisseurs
-- ================================================
create table fournisseurs
(       fno             integer  not null , check (fno < 20),
        nom             char(20) not null,
        adresse         char(50),
        ville           char(10) default 'Lille',
        constraint pk_fournisseur primary key (fno)
)
/
-- ================================================
--   Creation Table : Produits
-- ================================================
create table produits
(       pno     integer , check (pno < 200) ,
        design  char(10),
        prix    integer , check (prix between 1000 and 9000),
        poids   integer , check (poids < 100),
        couleur char(10), check (couleur in ('rouge', 'vert',
                                    'jaune', 'bleu', 'gris')) ,
        constraint pk_produits primary key (pno)
)
/
-- ================================================
--   Creation Table : Commandes
-- ================================================
create table commandes
(       cno integer,
        fno integer,
        pno integer,
        qute integer,
        constraint fk_fournisseur foreign key (fno)
                   references fournisseurs(fno)
                   on delete cascade ,
        constraint fk_produits foreign key (pno)
                   references produits(pno)
                   on delete cascade
)
/
-- ================================================
--   Creation données : Produits
-- ================================================
insert into produits(pno,design,prix,poids,couleur)
       values (102, 'fauteuil' ,1500 , 9 ,'rouge')
/
insert into produits(pno,design,prix,poids,couleur)
       values (103, 'bureau'   ,3500 ,30 ,'vert')
```

6.2.5 La création de vues

Une vue est une table dont les données ne sont pas physiquement stockées mais qui se réfère à des données stockées dans d'autres tables. C'est une fenêtre sur la base de données permettant à chacun de voir les données comme il le souhaite. On peut ainsi définir plusieurs vues à partir d'une même table et en faire varier la composition en fonction des utilisateurs. Il est aussi possible de combiner des données venant de plusieurs tables, d'autres vues ou des données calculées dans une vue. Une vue est utilisée exactement comme s'il s'agissait d'une table classique bien que les données ne soient pas dupliquées. La vue est calculée dynamiquement (interprétée) à chaque exécution d'une requête qui y fait référence. Les données ne sont stockées que dans les tables d'origine. La définition d'une vue se fait simplement à l'aide de l'ordre `SELECT` pour sélectionner les colonnes.

```
CREATE VIEW <nom-de-vue>
       ( <nom-de-colonne> , ... )
         AS <sélection>;
```

La construction de vues présente plusieurs avantages :

- Les vues permettent de rendre les programmes moins dépendants de l'évolution des tables.
- Une vue permet de restreindre l'accès d'une table à un sous-ensemble de colonnes et un sous-ensemble de lignes, ceci pouvant aller jusqu'à la donnée élémentaire.
- Il est possible de rassembler dans un seul objet des données "éparpillées" dans plusieurs tables.
- La programmation est simplifiée pour l'utilisateur puisqu'une partie de l'ordre `SELECT` est déjà codée dans la vue.
- Les vues permettent de simplifier les ordres SELECT avec sous-requêtes complexes. Une vue est souvent créée pour chaque sous-requête, ce qui facilite son écriture et sa compréhension.
- Les vues permettent de protéger finement l'accès aux tables en fonction des utilisateurs. On utilise l'utilisation d'une vue partielle sur une ou plusieurs tables et on interdit l'accès aux tables. C'est donc l'un des moyens les plus efficaces de protection des données.

Dans MS-ACCESS, dès qu'une requête a été exécutée, le système propose de sauver cette requête sous un nom donné. Les requêtes sauvegardées de MS-ACCESS sont donc des vues bien qu'elles n'en portent pas le nom.

R53 *création d'une vue sur la jointure fournisseurs-commandes ne permettant de voir que les fournisseurs avec leur total de commandes.*

```
CREATE VIEW vuefournisseur
AS SELECT f.nom,
          f.adresse,
          sum(c.qute) as somme
FROM fournisseurs f, commandes c
WHERE f.fno = c.fno
group by f.nom, f.adresse;
```

L'utilisateur peut ensuite écrire ses propres requêtes sur la vue. Cette requête sera "traduite" par le SGBD en une requête sur les tables puis exécutée effectivement.

R54 *Requête sur une vue*

```
SELECT * FROM vuefournisseur WHERE somme > 50 ;
```

sera traduite en ...

```
SELECT f.nom, f.adresse, sum(qute)
FROM fournisseurs f, commandes c
WHERE f.fno=c.fno
GROUP BY f.nom, f.adresse
HAVING sum(qute) > 50 ;
```

Afin de restreindre la visibilité des données on peut imaginer que pour des raisons de confidentialité, certains commerciaux de l'entreprise ne doivent accéder qu'aux fournisseurs de Lille avec leurs commandes, et d'autres uniquement aux fournisseurs de Lyon avec leurs commandes. On créera donc à cet effet deux vues dont l'accès sera uniquement donné à chacun.

```
CREATE VIEW fourniss-lille
AS SELECT f.nom, f.pno, f.qute
   FROM fournisseurs f , commandes c
   WHERE f.fno=c.fno
   AND f.ville='Lille';

CREATE VIEW fourniss-lyon
AS SELECT f.nom, f.pno, f.qute
   FROM fournisseurs f , commandes c
   WHERE f.fno=c.fno
   AND f.ville='Lyon';
```

De cette manière, aucun commercial ne voit ni les tables réelles, ni les données de ces tables qui ne lui sont pas destinées.

La majeure partie des commandes `SQL` qui s'appliquent à une table peuvent être appliquées à une vue moyennant certaines restrictions. En effet, pour faire une mise à jour au travers d'une vue il doit être possible de propager la mise à jour sur les tables source. La mise à jour d'une vue ne peut donc se faire que si :

- La clause FROM principale ne fait référence qu'à une seule table ou une vue accessible en mise à jour.
- Elle ne comporte pas de `DISTINCT` ou une fonction sur colonne.
- Elle ne contient pas de clause `GROUP BY` ou `HAVING`.
- Elle n'utilise pas les opérateurs ensemblistes `UNION`, `INTERSECT` ou `EXCEPT`
- Elle ne contient que des références aux colonnes de la table source (pas de `COUNT`, `SUM` etc ...).
- Elle ne contient pas de sous-requête dont la clause `FROM` contient la même table que la clause `FROM` principale.

De manière simplifiée on peut dire que les ordres `INSERT`, `DELETE` et `UPDATE` ne peuvent s'appliquer qu'à une vue n'utilisant qu'une seule table avec restrictions et sélections.

Une vue qui ne respecte pas ces critères est dite en lecture seule.

Illustrons ce problème sur un exemple à l'aide des deux tables R et S suivantes avec la vue T obtenue par équi-jointure sur R et S.

R	A	B
	a	1
	b	2
	c	3

S	C	D
	v	1
	x	2
	y	2
	z	3

T	A	C
	a	v
	b	x
	b	y
	c	z

```
CREATE VIEW T
AS SELECT A,C FROM R,S
      WHERE R.B=S.D ;
```

Comment répercuter la suppression du tuple `(b,y)` de la vue T?

De même, il est clair que l'ordre SQL suivant, qui tente une mise à jour de la vue `vuefournisseur` précédemment créée n'a aucun sens. Cette vue ne peut être utilisée qu'en lecture seule.

```
UPDATE vuefournisseur
```

```
set somme=somme*2
where nom ='DUPONT';
```

Dans les SGBD purement interactifs, comme ACCESS, si une vue est ouverte et que durant sa consultation des données sont modifiées ou ajoutées sur les tables sources, cette modification ne sera répercutée dans la vue qu'à sa prochaine ouverture. Les SQBD les plus puissants offrent des options permettant de préciser si les modifications faites parallèlement sont répercutées immédiatement dans la vue ou à sa prochaine ouverture.

6.2.6 La création d'index

Les objets que nous avons décrit précédemment suffisent théoriquement à l'élaboration d'une base de données. Néanmoins, en pratique, il est nécessaire de disposer de méthodes d'accès efficace. C'est le rôle des index. Un index permet d'accéder rapidement et directement par des techniques de hachage aux colonnes et aux lignes des tables. Dans la plupart des SGBD, la déclaration d'une clé primaire pour une table implique la création automatique d'un index sur celle ci, mais d'autres colonnes peuvent aussi justifier la création d'autres index. On peut indexer une table sur un ou plusieurs noms de colonne cités par ordre d'importance. La création d'index ne peut pas se faire sur une vue. Un index est un objet différent de la table auquel il se rapporte. Il est mis à jour automatiquement lors de chaque mise à jour d'une table.

```
CREATE [UNIQUE] INDEX <nom-de-l'index>
       ON <nom-de-table>
       (<Nom-de-champ> [ASC / DESC]
        , ...   );
```

L'option `ASC` ou `DESC` précise si les valeurs de la colonne doivent être triées de façon ascendante ou descendante. L'option `UNIQUE` précise si les valeurs des colonnes indexées sont uniques ou non.

Si dans la base de données, de nombreuses requêtes recherchent les fournisseurs par leurs noms et prénoms, il peut être intéressant de créer un index sur ces colonnes afin d'optimiser les recherches :

R55 *création d'un index sur les colonnes* `nom` *et* `prénom` *de la table fournisseurs :*

```
CREATE INDEX i_founiss ON fournisseurs (nom ASC, prénom DESC);
```

Il est important de noter que l'existence d'un index alourdit et ralentit la saisie de données dans la base. Il doit donc être judicieusement choisi au regard des résultats fournis par les analyseurs de requêtes. Par exemple, si toutes les requêtes recherchent les fournisseurs par leurs noms, il peut être judicieux de créer un index sur le nom de fournisseur. Afin que les techniques d'indexation soient efficaces, on s'arrange en général pour mettre des index sur des champs dont les valeurs sont la plupart du temps sans trop de doublons.

6.2.7 Modification et suppression de table, de vue ou d'index

L'ordre `DROP` permet d'effectuer des suppressions sur chacun des objets précédemment cités :

```
DROP TABLE <nom-de-table>;
DROP VIEW <nom-de-la-vue>;
DROP INDEX <nom-de-l'index>;
```

La commande `ALTER` ne fonctionne que sur les tables. Elle permet de rajouter, modifier ou supprimer des colonnes à une table. Elle peut être utilisée avec les mots clés `ADD` , `MODIFY` et `DROP` selon l'action à effectuer.

- ADD ajoute la colonne en fin de table

```
ALTER TABLE <nom-de-table>
      ADD (<nom-de-colonne> <type-de-données> ,
           ...);
```

R56 *ajout d'une colonne "prénom" dans la table Fournisseurs :*

```
ALTER TABLE fournisseurs
      ADD (prenom CHAR(15));
```

La valeur initiale des champs rajoutés est nulle. Il n'est donc pas possible de préciser la condition `NOT NULL` dans la définition d'un attribut avec la commande `ALTER`.

La commande `ALTER` permet aussi bien sûr d'ajouter ou de supprimer des contraintes sur la table.

- MODIFY permet de modifier les types de données.

R57 *modification du type quantité dans la table commandes :*

```
ALTER TABLE commandes
      MODIFY (qute NUMBER (8,2));
```

- DROP permet de supprimer les colonnes de tables existantes.

```
ALTER TABLE <nom-de-table>
      DROP <nom-de-colonne>;
```

Dans la majorité des SGBD, si vous changez le type en réduisant la taille de la donnée, `SQL` tronque les valeurs existantes, s'il y a compatibilité entre l'ancien et le nouveau type. On ne peut pas modifier ou supprimer une colonne si :

- elle est présente dans une vue.
- elle sert dans un index.
- une contrainte y fait référence.

Il est parfois possible dans certains SGBD de renommer une table. C'est le cas d'Oracle avec l'ordre `RENAME <ancien-nom> TO <nouveau-nom>;`

6.2.8 Commenter une table

Il est possible de commenter une table, une vue ou une colonne.

```
COMMENT on TABLE <Nom-de-table> IS <'commentaires'>;
COMMENT on TABLE <Nom-de-vue> IS <'commentaires'>;
COMMENT on TABLE <Nom-de-table>.<Nom-de-colonne> IS <'commentaires'>;
```

6.3 L'aspect multi-utilisateurs : le DCL

Les systèmes de gestion de l'information actuels nécessitent tous un accès concurrent : Différentes personnes peuvent travailler sur la même base de données. Cette possibilité implique deux problèmes majeurs dans la conception d'un système d'information : la gestion de la confidentialité des données qui permet de préciser "qui peut faire quoi" et l'accès concurrent aux données qui règle le problème des accès simultanés à l'information.

6.3.1 La confidentialité des données

L'administrateur de la base de données (DBA) est une personne qui non seulement doit gérer la base, ses schémas, ses contraintes et ses index mais aussi les différents utilisateurs. Il doit vérifier que l'accès à la base se fait par des utilisateurs identifiés et doit ensuite définir pour ces utilisateurs leurs droits et privilèges. Certains auront accès à certaines tables, d'autres ne pourront faire que des ajouts, d'autres encore ne pourront faire que des lectures. Il y a en général deux types de privilèges (gérés par le SGBD de manière différente) : les privilèges Système qui donnent des droits

de connexion à l'utilisateur et les privilèges sur les données qui donnent les autorisations d'accès aux données.

La gestion des autorisations comprend deux mécanismes : l'octroi/annulation des autorisations et le contrôle de ces autorisations. Ces deux problèmes sont résolus par le passage obligatoire de toute requête à travers le dictionnaire (ou méta-base). C'est dans le dictionnaire que l'on note puis que l'on retrouve par la suite les droits et privilèges de chacun dans des tables dont le nom est ou ressemble à `TABLE_PRIVILEGES`, `COLUMN_PRIVILEGES`, `USAGE_PRIVILEGES`.

Pour accorder ou retirer un ou plusieurs privilèges sur un objet (table ou vue) SQL dispose de 2 commandes : `GRANT` pour accorder les privilèges et `REVOKE` pour retirer les privilèges. Ces ordres s'adressent à un seul utilisateur ou parfois, pour certains SGBD à des groupes d'utilisateurs.

Les privilèges principaux sont :

- `SELECT` : privilège permettant de lire le contenu d'une table ou vue.
- `INSERT` : privilège permettant d'insérer des valeurs dans une table ou une vue.
- `DELETE` : privilège permettant d'effacer des tuples d'une table ou vue.
- `UPDATE` : privilège permettant de modifier le contenu d'une table ou vue.
- `REFERENCES` : privilège permettant de faire référence à une table ou vue dans une contrainte d'intégrité.
- `SELECT(x)`, `INSERT(x)`, `DELETE(x)`, `UPDATE(x)`, `REFERENCES(x)` : même chose mais uniquement sur la liste de colonnes `x`.

Aux privilèges, s'ajoutent des rôles dans la plupart des SGBD. Un rôle correspond à un ensemble de droits. Il est utilisé dans les commandes comme un droit classique. Son objectif est avant tout de simplifier et clarifier les commandes d'attribution de droits aux utilisateurs. Parmi les rôles principaux on trouvera :

- DBA : qui permet de spécifier d'un coup tous les droits du DBA
- Connect : qui regroupe les droits nécessaires à la connexion
- Resource : qui regroupe les droits nécessaires à la création et modification de ses propres tables.

L'une des tâches de l'administrateur consiste à définir des rôles adaptés à chaque type d'utilisateur de l'entreprise.

6.3.2 Accorder les droits

Seul le créateur d'un objet possède tous les privilèges sur cet objet. A tout instant un utilisateur qui a le droit de transmettre des privilèges sur un objet peut utiliser la commande `GRANT` pour accorder ce privilège.

```
GRANT liste_privilèges ON liste_objets TO liste_utilisateurs
      [WITH GRANT OPTION] ;
```

`WITH GRANT OPTION` permet de préciser que l'utilisateur qui reçoit les droits peut aussi les transmettre. Le mot clé `PUBLIC` à la place du nom d'utilisateur permet de préciser que tous les utilisateurs connus du système sont concernés. Le mot clé `ALL` référence d'un coup tous les privilèges que le donneur peut accorder sur l'objet.

R58 *Donner les droits de mise à jour à la secrétaire Anne sur la colonne qute de la table des commandes avec possibilité de transmettre ce droit.*

```
GRANT UPDATE(qute) ON commandes TO anne WITH GRANT OPTION;
```

Imaginons un système sur lequel peuvent se connecter 5 individus, Pierre, Paul, Jean, Jacques, Luc. Initialement seuls Pierre et Paul ont tous les droits sur la table T (Fig.6.7).

donneur d'ordre	date	ordre		
pierre	5	`GRANT select, insert`	`on T TO jean`	`WITH GRANT OPTION;`
jean	10	`GRANT select, insert`	`on T TO jacques`	`WITH GRANT OPTION;`
paul	15	`GRANT select`	`on T TO jean`	`WITH GRANT OPTION;`
jean	20	`GRANT select, insert`	`on T TO luc`	`WITH GRANT OPTION;`

En fin de session chaque utilisateur aura les droits select et insert. Jean les aura d'ailleurs obtenu par deux personnes différentes.

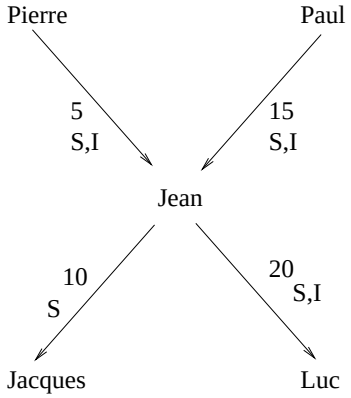


FIG. 6.7 – *attribution de droits*

6.3.3 Retirer les droits

A tout instant, tout utilisateur ayant donné un privilège peut retirer ce privilège à l'aide de la commande `REVOKE`. La syntaxe de cet ordre est la suivante :

```
REVOKE [GRANT OPTION FOR] liste_privileges ON liste_objets
FROM liste_utilisateurs ;
```

R59 *retirer les droits de mise à jour à Anne sur la colonne qutés*

```
REVOKE UPDATE(qute) ON commandes FROM anne;
```

R60 *retirer tous les droits à Anne sur les commandes*

```
REVOKE ALL ON commandes FROM anne ;
```

Lors d'un REVOKE, comme le receveur d'un droit a pu transmettre à son tour ce privilège il faut donc l'enlever à toutes les personnes à qui il a été transmis. Les choses se compliquent encore puisqu'un individu peut recevoir un privilège par l'intermédiaire de plusieurs personnes différentes.

Pour résoudre ce problème, une solution communément implémentée dans les SGBD actuels, consiste à associer à chaque attribution de privilèges une estampille dont la valeur indique quand ce privilège a été accordé. Le SGBD stocke dans des tables système toute attribution d'un droit en conservant le donneur de ce droit, le receveur, le type de droit ainsi que la date précise à laquelle il a été transmis.

Pour que la base reste cohérente, il faut que le SGBD s'assure qu'un utilisateur ne se trouve pas en possession d'un droit obtenu par quelqu'un qui ne l'a plus !

Cette contrainte est difficile à respecter notamment quand un utilisateur a reçu le même droit de plusieurs personnes.

Reprenons l'exemple précédent avec Pierre, Paul, Jean, Jacques et Luc

donneur d'ordre	estampille	ordre SQL
pierre	30	`REVOKE insert on T FROM jean ;`
pierre	35	`REVOKE select on T FROM jean ;`

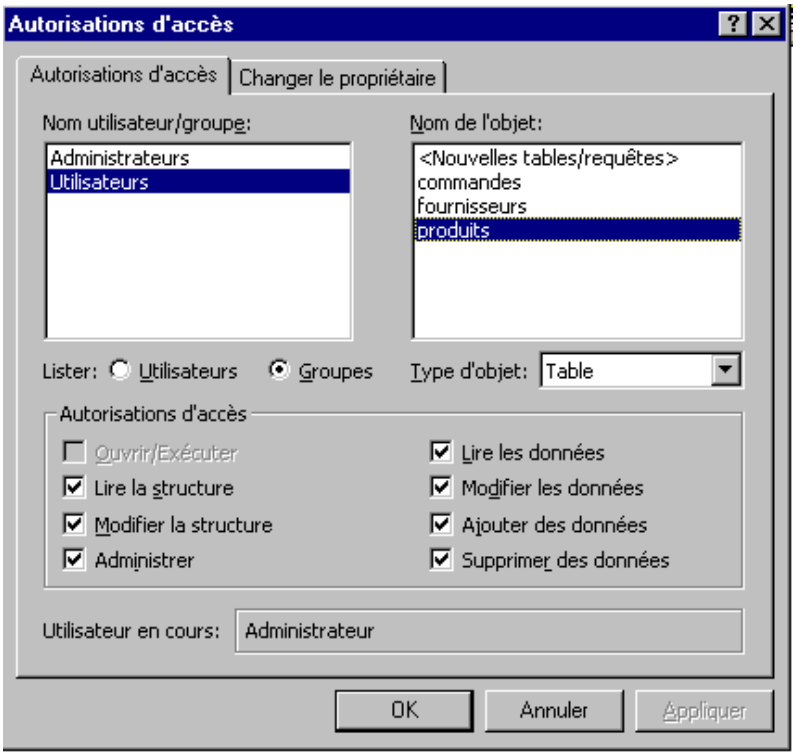


FIG. 6.8 – *Les droits sous Access*

En fin de session Jean, Jacques et Luc auront perdu le droit INSERT. Par contre seul Jacques aura perdu le droit SELECT.

Si l'on note `ui` l'utilisateur i et que l'on considère le triplet (ui,uj,droit) comme clé dans la table des droits, un algorithme permettant d'implémenter un ordre `REVOKE` (ui enleve le droit à uj à l'instant t) pourrait être le suivant :

```
% ui enleve le droit à uj à l'instant t. Au premier appel t=sysdate()
revoke(ui,uj,t)
        estamp_mini = +infini
        Si uj est beneficiaire de ui à un temps tj ET tj < t
                - supprimer ce droit à uj
                - trouver estamp_mini(uj) = tmin
                - Pour tout uk tel que uj a donné le droit à uk en tk < tmin
                        revoke(uj,uk,tmin)
        fsi
fin
```

On notera que l'implémentation du REVOKE varie beaucoup d'un SGBD à l'autre et qu'il est souvent nécessaire d'expérimenter son fonctionnement sur quelques exemples afin de s'assurer de son mode de fonctionnement.

6.4 Le catalogue

La majorité des SGBD (Oracle, DB2, mais pas MSACCESS) gèrent leurs propres informations dans un catalogue de tables. Cette manière d'utiliser des tables pour représenter les informations

du SGBD dans des tables (donc des tables sur des tables) permet de voir ce catalogue comme une méta-base.

La mise à jour des informations du catalogue se fait par le SGBD lui-même en fonction des requêtes de création ou de gestion de droits que chaque utilisateur fait.

Le catalogue gère tous les objets utilisés par le SGBD. Il est alors possible grâce à la méta-base d'obtenir par un simple `SELECT` des informations sur les tables, les vues, les utilisateurs, les droits etc ... Le catalogue est l'outil privilégié du DBA (DataBase Administrator). C'est à travers lui que le DBA crée de nouveaux utilisateurs, gère les droits d'accès aux tables ou crée de nouveaux clusters.

Quel que soit le SGBD utilisant un catalogue, celui–ci est toujours très volumineux (plus de 500 tables pour les systèmes actuels). Les liens entre les tables des catalogues sont jalousement gardés par chaque concepteur. Il n'existe pas de description détaillée publique[5] de son fonctionnement. De plus, les catalogues ne sont pas normalisés et sont en constante évolution. Présenter le cas d'un SGBD particulier à l'instant t n'aurait pas beaucoup de sens. Nous avons donc choisi de présenter un MCD montrant ce que pourrait être une méta-base restreinte aux objets principaux (Fig.6.9).

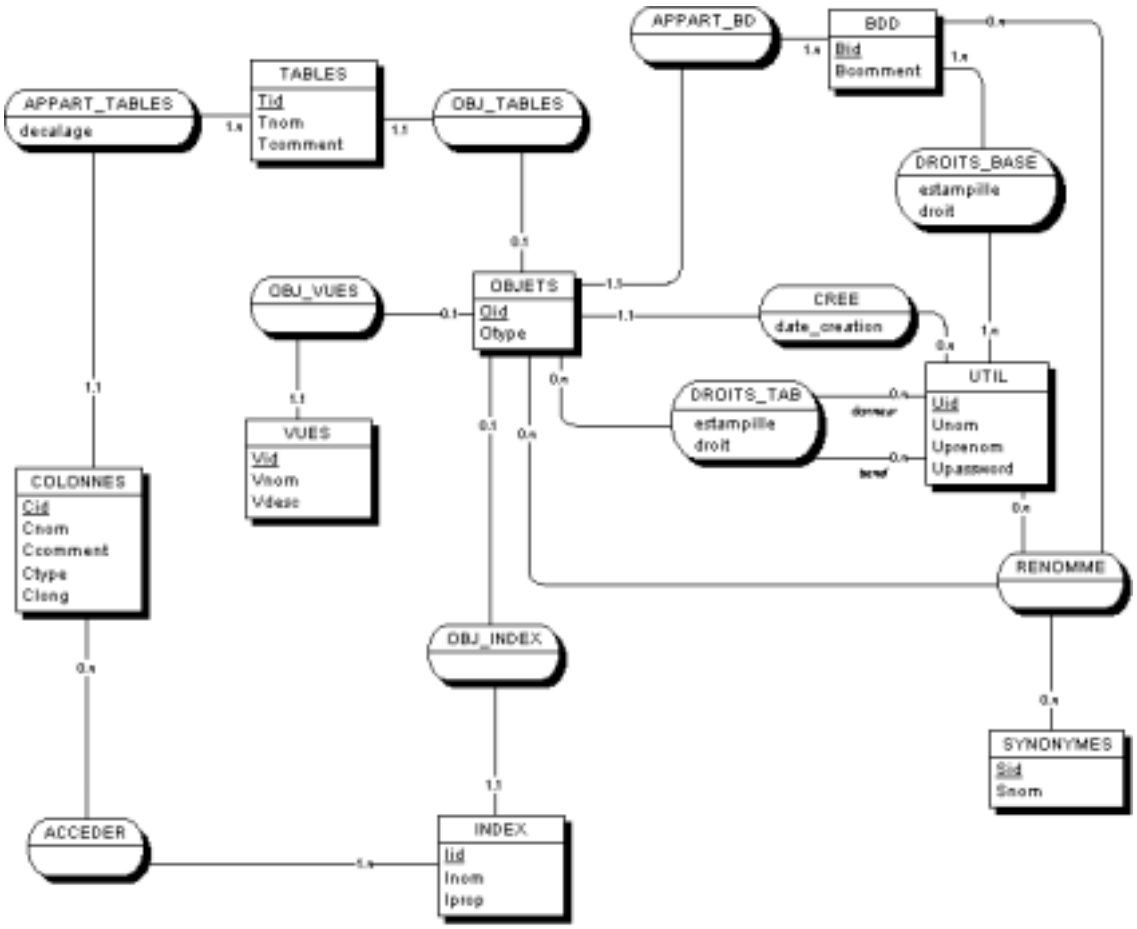


FIG. 6.9 – *Proposition de méta-base*

Les tables de ce MCD se retrouvent "grossièrement" dans chaque SGBD qui utilise une méta-base mais cette présentation est néanmoins très restreinte. On trouve aussi dans la méta-base les

5. à part bien sûr la liste exhaustive des tables et vues du catalogue

tables de gestion des contraintes d'intégrité, les table de gestion de la représentation physique de la base (clusters, blocks), les tables d'audit etc ... De plus, de très nombreuses vues sont créées pour chaque type d'utilisateur afin de restreindre ses informations à ses propres objets. Si on prend le cas du SGBD Oracle, les tables du catalogue se dupliquent en trois catégories de vues préfixées par `DBA` (pour ce qui concerne l'administrateur), `USER` (pour l'utilisateur en cours), `ALL` (pour tout le monde). Voici quelques exemples de tables basiques.

Nom MCD	Nom Oracle	Nom DB2	signification
objets	obj$	syscatalog	carac. de tous les objets
colonnes	col$	syscolumns	carac. de toutes les colonnes
tables	tab$	systaballoc	carac. de toutes les tables
vues	view$	sysviews	carac. de toutes les vues
index	ind$	sysindex	carac. de tous les index
synonymes	syn$	synonyms	carac. de tous les synonymes
util	user$	sysuserlist	carac. de tous les utilisateurs
droits-tab	tabauth$	systabauth	droits des utilisateurs sur les tables
droits-base	sysauth$	sysuserauth	droits des utilisateurs sur les bases

L'interrogation du catalogue se fait à l'aide d'un ordre `SELECT` comme sur n'importe quelle table. Les descriptions des champs de chacune des tables du catalogue peuvent être obtenues par l'ordre `DESCRIBE`.

R61 *Mes propres vues et tables (Oracle v7).*

```
SELECT * FROM user_catalog;
```

R62 *Liste des utilisateurs du SGBD (Oracle v7).*

```
SELECT * FROM all_users;
```

R63 *Droits accordés sur la table Fournisseurs (Oracle v7).*

```
SELECT * FROM user_tab_grants WHERE table_name='FOURNISSEURS';
```

R64 *Retrouver les tables de la base KBDD avec leurs propriétaires (DB2)*

```
SELECT table_name, creator FROM systables
WHERE dbname='KBDD' ;
```

R65 *Retrouver les tables que j'ai créées et les bases concernées (DB2)*

```
SELECT table_name, dbname FROM systables
WHERE creator='MATHIEUP' ;
```

R66 *Retrouver la constitution d'une de mes tables (DB2)*

```
SELECT * FROM syscolumns
WHERE tbcreator='MATHIEUP' AND tbname='FOURNISSEURS' ;
```

Chapitre 7

SQL Intégré

Le langage SQL est insuffisant à lui seul pour écrire des applications complètes. Certains calculs sont très difficiles voire impossibles à faire uniquement avec SQL. Il est donc important de pouvoir l'intégrer dans d'autres langages de programmation.

C'est notamment le cas pour des traitements à faire sur un grand nombre de tuples en tâche de fond (Batch), ou pour des traitements qui nécessitent à la fois des données de la base de données et des données de fichiers classiques (sequentiels, VSAM etc ...)

Jusqu'à présent, le mode SQL sur lequel nous nous sommes basés se nomme mode interactif car le SGBD répond directement aux sollicitations de l'utilisateur. Nous allons décrire maintenant le mode intégré et la manière de le mettre en œuvre.

7.1 Liens entre SQL et le langage applicatif

Tout d'abord, chaque ordre SQL inclus dans un programme doit commencer par un `EXEC SQL`. La fin de chaque ordre est dépendante du langage. En C et en ADA il doit se terminer par un point virgule, tandis qu'en COBOL il faut utiliser le terme `END-EXEC`. Il ne doit y avoir qu'une seule instruction SQL entre le `EXEC SQL` et le `END-EXEC`. Un préprocesseur, fourni avec le SGBD, se chargera à la compilation, de convertir chaque ordre ainsi défini en une série d'ordres propres au SGBD.

Il faut ensuite déclarer les variables de communication avec SQL pour chaque type de donnée à récupérer. Ces variables sont appelées variables hôtes. A chaque type de donnée SQL correspond bien sûr un type de donnée dans le langage hôte. Voici la conversion des types les plus courants:

Type SQL	Exemple	COBOL	C
CHAR(n)	'ma chaine'	pic x(n)	char(n+1)
SMALLINT	227	pic s9(4) comp	short int
INTEGER	127345	pic s9(9) comp	long int
FLOAT	3.1415	comp-1	float
NUMERIC(p,e)	12.66	pic s9(p-e)v(e) comp-3	-
DATE	1964-02-24	pic x(10)	char(11)
TIME	17:12:56	pic x(8)	char(9)
TIMESTAMP	1964-02-24:17:12:56	pic x(19)	char(20)

Voici par exemple la structure nécessaire pour recevoir une ligne d'une table fictive contenant des colonnes de plusieurs types différents en COBOL:

```
create table  essai
(      nom     char(20),
       annee   smallint,
       salaire integer,
       heure   date,
       jour    time
)
```

```
01 ligne
   02 nom     pic x(20).
   02 annee   pic s9(4) comp.
   02 salaire pic s9(9) comp.
   02 heure   pic x(8).
   02 jour    pic x(10).
```

Quelques règles sont à respecter pour l'utilisation correcte d'une variable hôte :

- Elle doit être déclarée comme toute autre variable avec un type connu du langage hôte.
- Elle doit être compatible avec la définition des données de la colonne qu'elle est destinée à recevoir.
- Elle doit être précédée du caractère ':' quand elle est utilisée dans un ordre SQL.
- Une structure peut être utilisée mais en COBOL elle ne doit pas dépasser 2 niveaux.
- Pour des raisons de lisibilité et de compatibilité il est conseillé de déclarer les variables hôtes dans une partie du programme délimitée par `BEGIN DECLARE SECTION` et `END DECLARE SECTION`.

Une fois que les structures de données nécessaires au programme sont déclarées, il faut déclarer les structures de retour de codes d'erreur propres à la base de données utilisée. Cela se fait automatiquement par la récupération du fichier `SQLCA`.

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

La structure définie par `SQLCA` contient de nombreuses variables renseignées à chaque ordre SQL passé par le programme. Par exemple la variable SQLERRMC qui contient le libellé en clair du message d'erreur. Parmi toutes les variables déclarées dans `SQLCA`, la plus importante est sans aucun doute `SQLCODE`[1].

Le contenu de `SQLCODE` obéit aux conventions suivantes :

- valeur égale à 0 : L'ordre SQL s'est correctement exécuté.
- Valeur négative : Une erreur est intervenue.
- Valeur positive : Message d'information suite à un ordre correctement exécuté. Le code d'information le plus important étant +100 qui correspond au positionnement en fin de table.

Pour tester les différents codes retour possibles suite à l'exécution d'un ordre `SQL` on peut soit tester manuellement le code retour juste après chaque instruction, soit utiliser un échappement défini une fois pour toutes en début de programme. Cette méthode, beaucoup plus élégante, rencontre généralement l'approbation des programmeurs. Ceci se fait avec l'ordre `WHENEVER` qui permet de tester les 3 conditions `NOT FOUND` (équivalent au `SQLCODE`=+100) , `SQLERROR` (équivalent à `SQLCODE` négatif) ou `SQLWARNING` (équivalent à `SQLCODE` positif et différent de +100). Dans chacun de ces trois cas on spécifie ensuite l'action à exécuter. Cette action peut être `CONTINUE` pour exécuter en séquence la prochaine instruction ou `GOTO` pour effectuer un branchement à une étiquette.

On trouvera par exemple en début de `PROCEDURE DIVISION` les lignes suivantes :

```
EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
EXEC SQL WHENEVER SQLERROR GOTO caserreur END-EXEC.
```

Les ordres `SQL` classiques (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) peuvent ensuite être exécutés en respectant le fait que les noms des variables du langage hôte utilisés dans la requête doivent être préfixés par le caractère `:`.

1. Les nouveaux SGBD basés sur la norme SQL2 utilisent la variable SQLSTATE ainsi qu'une zone de diagnostic pour tester les codes retour, mais le principe reste le même

Le cas du SELECT est un peu à part. Il peut soit retourner une seule ligne, soit une table qui sera exploitée ligne à ligne. Pour ce dernier cas SQL offre la notion de curseur que nous détaillerons ensuite. Dans le cas d'une seule ligne à récupérer, une clause spéciale nommée INTO doit être introduite avant la clause FROM. Elle permet de spécifier les variables du langage hôte qui récupéreront chaque ligne d'une sélection. On peut par exemple écrire :

R67 *Déclaration d'un select pour récupération d'une ligne unique*

```
EXEC SQL
     SELECT SUM(salaire),COUNT(*)
     INTO :somme :nombre
     FROM essai
END-EXEC.
```

Dans le cas précédent, la requête ne renvoie qu'une seule ligne, mais il est évidemment possible de traiter une table ligne à ligne. L'outil principal pour effectuer ce traitement est le curseur. Un curseur est un nom symbolique permettant de matérialiser la position courante sur une table résultant d'une requête. La déclaration d'un curseur se fait par l'ordre DECLARE CURSOR suivi de la description de l'ordre SELECT concerné (la clause INTO n'est plus nécessaire ici).

La syntaxe générale de cet ordre est la suivante :

```
DECLARE nom_curseur [INSENSITIVE] [SCROLL]
CURSOR FOR requête
[FOR READ ONLY | FOR UPDATE [OF liste_colonnes]];
```

SCROLL indique que le curseur peut être utilisé dans tous les types de parcours (avant, arrière etc ...). Par défaut, seul le parcours séquentiel est autorisé.
INSENSITIVE indique qu'une copie de la sélection est faite et que toute modification sur cette copie ne sera visible des autres utilisateurs qu'à la fermeture du curseur. UPDATE précise que l'on souhaite faire des mises à jour sur la sélection. Par défaut, le curseur est en READ ONLY.

R68 *Déclaration d'un select pour récupération d'une table*

```
EXEC SQL
    DECLARE moncurseur CURSOR FOR
    SELECT *
    FROM essai
END-EXEC.
```

Il faut ensuite ouvrir le curseur par l'ordre OPEN

```
EXEC SQL
     OPEN moncurseur
END-EXEC.
```

L'ordre FETCH permet quant à lui d'effectuer une lecture dans la table résultante à l'emplacement du curseur. C'est lui qui remplit les variables du langage hôte, il remplace donc la clause INTO précédemment introduite dans le SELECT. Un sélecteur peut être associé au FETCH pour préciser l'ordre de parcours du résultat. Ce sélecteur peut prendre les valeurs NEXT, PRIOR, FIRST, LAST, ABSOLUTE n, RELATIVE n. La valeur par défaut est NEXT.

La syntaxe générale de cette instruction est la suivante

```
FETCH [[selecteur] [FROM]] nom_curseur INTO liste_cible ;
```

On pourra écrire par exemple

```
EXEC SQL
     FETCH LAST moncurseur
     INTO :ligne
END-EXEC.
```

ce qui aura pour effet de placer le curseur en dernière position puis de remplir la variable `ligne` avec les données correspondantes.

Les mises à jours et effacements sont toujours effectués sur la position courante. La syntaxe générale est la suivante :

```
EXEC SQL
     UPDATE nom_table
     SET liste_colonne_valeur
     WHERE CURRENT OF nom_curseur
END-EXEC.

EXEC SQL
     DELETE FROM table
     WHERE CURRENT OF nom_curseur
END-EXEC.
```

`UPDATE` et `DELETE` ne modifient pas la position du curseur. Si plusieurs tuples doivent être modifiés il est donc nécessaire de réexecuter un `FETCH` pour positionner à nouveau le curseur.

Une fois le traitement de toutes les données effectué il est nécessaire de fermer le curseur après le traitement par l'ordre `CLOSE`.

```
EXEC SQL
    CLOSE moncurseur
END-EXEC.
```

INCLUDE SQLCA
WHENEVER
SELECT ... INTO ...
DECLARE curseur CURSOR FOR ...
OPEN
FETCH curseur INTO ...
CLOSE curseur

FIG. 7.1 – *Ordres de gestion de curseurs*

La structure d'un programme avec curseur est alors la suivante :

- Déclarer les routines à appeler en cas d'erreur (WHENEVER).
- Déclarer un curseur qui servira à parcourir la table résultante (DECLARE).
- Définir l'opération de sélection qui définit la table résultante (SELECT).
- Ouvrir le curseur (OPEN).
- Amener une nouvelle ligne dans les variables du programme hôte tant que nécessaire (FETCH).
- Terminer le traitement après avoir détecté le code retour, soit par un test sur SQLCODE=+100, soit par déclenchement de l'ordre WHENEVER correspondant.
- Fermer le curseur (CLOSE).

7.2 Exemple de programme SQL Intégré

A l'aide des tables précédemment utilisées on souhaite effectuer une extraction des commandes faites au fournisseur Dupont dans un fichier nommé Extract. Rappelons d'abord la structure de nos tables :

```
fournisseurs(fno,nom,adresse,ville)
produits(pno,design,prix,poids,couleur)
commandes(cno,fno,pno,qute)
```

Nous détaillerons ce traitement dans 2 langages différents (C, Cobol) et un langage d'application[2] (VBA sous Access).

2. La nuance peut sembler subtile mais elle ne l'est pas. Les langages d'applications comme VBA pour ACCESS, PL-SQL pour Oracle ou Transac-SQL pour Sybase ne sont pas portables, pas normalisés et sont des langages SQL propriétaires

7.2.1 Programme C

```
#include "stdlib.h"
#include "stdio.h"

exec sql include SQLCA;

main()
{
FILE *sortie;
char mynom[]="DUPONT";
long commande;
char designation[10+1];
long quantite;

     sortie=fopen("extract","w");
     exec sql whenever SQLWARNING continue;
     exec sql whenever SQLERROR goto maj-erreur;
     exec sql whenever not found goto termine ;

     exec sql
          declare c1 cursor for
          select cno,design,qute
          from fournisseurs f, commandes c, produits p
          where c.fno = f.fno
          and c.pno = p.pno
          and f.nom = :mynom ;

     exec sql open c1 ;

     for (;;)
     {
          exec sql fetch c1 into :commande, :designation, :quantite ;
          fprintf(sortie,"%4ld %10s %4ld \n",
                         commande, designation,quantite);
     }

maj-erreur:
     printf("Problème d'accès à la base de données : \%d", SQLCODE);

termine:
     exec sql close c1;
     fclose(sortie);
     exit(0);
}
```

7.2.2 Programme COBOL

```
configuration section.
source-computer. IBM-AS400.
object-computer. IBM-AS400.

file-control.
    select sortie assign to database-extract
         organization is sequential.

data division.

file section.

fd  sortie
    block contains 1 records
    label records are omitted.
01  enreg.
    10 fich-cmd    pic x(4).
    10 fich-design pic x(10).
    10 fich-qute   pic x(4).

working-storage section.

77  mynom        pic x(10).
77  commande     pic s9(9) comp.
77  designation  pic x(10).
77  quantite     pic s9(9) comp.

procedure division.

    move ‘‘DUPONT’’ to mynom.
    exec sql include SQLCA
    end-exec.

    open output sortie.

    exec sql whenever SQLWARNING continue
    end-exec.
    exec sql whenever SQLERROR goto maj_erreur
    end-exec.
    exec sql whenever not found goto termine
    end-exec.

    exec sql
         declare c1 cursor for
         select cno,design,qute
         from fournisseurs f, commandes c, produits p
         where c.fno = f.fno
         and c.pno = p.pno
         and f.nom = :mynom
     end-exec.

     exec sql open c1
     end-exec.


debut.
     perform traitement until sqlcode equal to +100.
```

```
     go to termine.

maj-erreur.
     display ‘‘Problème d’’ ‘‘accès à la base de données’’ SQLCODE.

termine.
     exec sql close c1
     end-exec.
     close fichier.
     stop run.

traitement.
     exec sql fetch c1 into :commande, :designation, :quantite
     end-exec.
     move commande to fich-cmd.
     move designation to fich-design.
     move quantite to fich-qute.
     write enreg.
```

7.3 Programme VBA sous ACCESS

Dans les langages d'application le principe de programmation est le même que pour COBOL ou C. La base doit être ouverte et la requête définie avant exécution. Il faut ensuite créer un curseur qui permet de parcourir le résultat de la requête. Le curseur doit ensuite être fermé. Malheureusement les langages d'applications suivent rarement la norme SQL, les ordres sont donc un peu différents bien que la philosophie soit identique. Prenons l'exemple de Visual Basic pour Applications (VBA) qui permet d'accéder à une base ACCESS ou ODBC.

Précisons tout d'abord que le code VBA sous Access peut être placé à différents endoits :

- Fenêtre de Débug
  C'est l'endroit idéal pour tester de petits morceaux de code. On l'obtient en tapant `Cntl+G` au clavier. Ce qui est tapé dans cette fenêtre n'est pas permanent ! Dès que la fenêtre est fermée, le code est perdu.
- Module
  Dans un module VBA il est possible d'enregistrer des fonctions, des procédures et des classes qui seront utilisées dans différents endroits de l'application. Tout ce qui est tapé ici est permanent. Toute fonction rangée dans un module peut être appelée n'importe où dans Access. Durant la mise au point la touche F5 qaund le curseur est placé sur l'une des lignes d'une procédure ou fonction permet d'exécuter le code associé.
- Procédure évènementielle
  Lors de l'édition des propriétés d'un objet, il est possible d'accéder aux procédures qui seront lancées lorsque l'évènement concerné se produira. Celles ci sont rangées dans un module spécial. Par exemple `OnClick` sur un bouton permettra d'associer du code qui sera exécuté quand l'utilisateur effectuera un click sur ce bouton à l'aide de la souris.

VBA ne fait pas de distinction entre majuscules et minuscules. On peut donc sous Access taper les mots clés comme on le souhaite, l'interpréteur se charge de les réécrire lui-même dans une syntaxe plus "normalisée".

Quelque soit l'endroit où le code est tapé, un système de complétion automatique des noms de méthodes ou d'attributs est accessible. Ceci offre là aussi une aide précieuse au développeur.

L'une des instructions les plus utiles en VBA est l'instruction `MsgBox` qui permet d'afficher un message à l'écran dans une boite de dialogue. N'hesitez pas à l'utiliser pour visualiser les étapes intermédiaires lors de l'élaboration de code d'accès aux données et notamment les requêtes construites sous forme de chaînes de caractères.

Une autre instruction capitale en VBA est l'instruction`DoCmd` qui permet d'exécuter toutes les macros que l'on exécute habituellement à la souris. Que vous souhaitiez ouvrir un formulaire, maximiser une fenêtre ou afficher une table, tout ceci peut se faire avec `DoCmd`. Toute macro Access est une méthode de l'objet `DoCmd`. Par exemple `DoCmd.OpenTable("table1")` ouvrira la table citée.

Il existe aussi sous Access un système d'aide en ligne très performant qui peut être obtenu en tapant F1 sur n'importe quel mot clé entré dans une fenêtre contenant du code.

Ces différents points étant cités nous allons maintenant pouvoir nous attaquer à notre problème : l'accès à une base Access à partir d'un programme VBA.

La base de données courante est référencée par la variable `CurrentDb`.

L'instruction `Print CurrentDb.Name` dans la fenêtre Debug affiche par exemple le nom de la base chargée. L'instruction `Print CurrentDb.TableDefs.Count` affiche le nombre de tables de cette base.

afin de donner un petit exemple illustrant la syntaxe et les méthodes principales d'accès aux tables, voici une procédure qui permet de visualiser les informations principales sur les tables créées par l'utilisateur. Cette procédure affiche les noms des tables de l'utilisateur avec la liste des colonnes et le nombre de tuples pour chaque table.

```
Public Sub listerTables()
Dim T As TableDef, F As Field, chaine
    ' parcourir toutes les tables
    For Each T In CurrentDb.TableDefs
        ' tester si ce sont des tables utilisateur
        If T.Attributes = 0 Then
            chaine = "Table " & T.Name & Chr(13) & Chr(13)
            chaine = chaine & "    nb cols " & T.Fields.Count
            chaine = chaine & Chr(13)
            ' parcourir toutes les colonnes
            For Each F In T.Fields
                chaine = chaine & "              : " & F.Name
                chaine = chaine & Chr(13)
            Next
            chaine = chaine + Chr(13)
            chaine = chaine & "    nb tuples " & T.RecordCount
            MsgBox (chaine)
        End If
    Next
End Sub
```

Pour exécuter ce code, il suffit de taper F5 si le curseur est sur l'une des lignes dans le panneau Modules, ou de taper `listerTables` à tout autre endroit d'Access comme dans la fenêtre Debug (Cntl + G)

7.3.1 Principes d'accès aux données

4 objets principaux sont définis en VBA pour accéder aux tables : `Workspace` qui définit les propriétés de l'espace de travail, `Database` qui définit les propriétés de la base de données, `QueryDef` qui permet de définir une requête et `RecordSet` qui correspond à la sélection proprement dite. Pour chacun de ces objets il existe un nombre important d'attributs et de méthodes utilisables. Ceci rend sans doute l'accès à VBA complexe pour le profane : il existe de nombreuses manières d'obtenir la même chose ! nombreuses propriétés et des méthodes. Nous ne présenterons dans cette partie que les méthodes principales de ces objets.

Nous ne décrirons pas ici les autres caractéristiques de VBA qui sont très nombreuses et sortent du cadre de ce livre. Pour plus d'informations le lecteur pourra se référer à l'aide en ligne qui est très complète ou aux sites Web suivants : `http://www.microsoft.com.accesshttp://www.microsoft.com/accessdev.devc` ou aux groupes de News sur le sujet.

- `CreateWorkspace` : permet de définir une session de travail, l'utilisateur et le type de base qui sera utilisé (JET ou ODBC). Il est donc très facile contrairement à l'idée reçue, d'accéder à une base DB2 ou Oracle via VBA.
- `W.OpenDatabase` : précise le nom de base à ouvrir. L'exemple de code sivant montre comment créer un espace de travail et ouvrir une base du moteur Jet nommée `bd1.mdb`

  ```
  Set wrkCurrent = CreateWorkspace("current", "admin", "", dbUseJet)
  Set dbCurrent = wrkCurrent.OpenDatabase("bd1.mdb")
  ```

  En général, les instructions précédentes sont inutiles. Dans la grande majorité des cas le programme doit s'exécuter sur la base courante qui est obtenue par `set dbCurrent = CurrentDB`
- `CreateQueryDef` : permet de définir la sélection à effectuer. Cette méthode s'applique toujours à un objet Database. Les paramètres en entrée doivent être spécifiés par une clause spéciale (équivalent à DECLARE CURSOR). Le premier argument est le nom de la requête à créer. Si une chaîne vide est fournie, la requête est temporaire. Si un nom est fourni, la requête devient permanente et est ajoutée à la liste de requêtes de la base.

- `OpenRecordSet`: Effectue la sélection (équivalent à OPEN CURSOR). Cette méthode peut s'appliquer à un objet QueryDef si celui-ci a été créé, ou directement à un objet Databas.
- `MoveFirst`: permet de se positionner sur le premier tuple. Il existe de la même manière MoveNext, MovePrevious, MoveLast (équivalent au FETCH)
- `Execute`: Permet d'exécuter une requête SQL qui ne renvoie pas de résultat (Insert, Update, Delete). Cette méthode peut s'appliquer à un objet QueryDef s'il existe ou directement à un objet de type Database. L'attribut `recordsAffected` de l'objet `QueryDef` permet de savoir combien de tuples on été affectés par la requête.
- `Close`: Ferme la sélection, le curseur, la base et l'espace de travail selon l'objet auquel il est appliqué.

En résumé la philosophie d'écriture de code VBA pour Access est donc très classique: La base courante est fixée dans la variable `CurrentDb`. En général c'est sur celle là que l'on travaille. Si ce n'est pas le cas, la méthode `openDataBase` permet d'ouvrir une autre base. Soit on crée la requête par la méthode `CreateQueryDef` qui permet notamment de la rendre persistante, soit on exécute directement la requête sur l'objet `Database` par `Execute` ou `OpenRecordSet`.

Les noms de champs peuvent parfois être composés de plusieurs mots ou même être constitués de mots clés. Microsoft préconise donc de toujours placer les noms de colonnes entre les caractères `[]`. L'accès à partir d'un `resultSet` se fait par le caractère `!`. Par exemple `R![nom du client]` permet d'accéder à la colonne `nom du client` de l'objet `resultSet` nommé `R`.

En guise d'exemple, Voici deux petites procédures qui permettent respectivemment d'afficher dans une `MsgBox` la liste des noms des clients et de saisir au clavier le nom et l'adresse d'un client pour l'ajouter à la base.

```
Sub ListeClients()
    Dim R As Recordset, chaine as String

    Set R = CurrentDb.OpenRecordSet("SELECT distinct nom from client")

    R.MoveFirst
    While Not R.EOF
        chaine = chaine + R![nom] + Chr(13)
        R.MoveNext
    Wend
    MsgBox (chaine)
    R.Close
End Sub
```

Pour ce premier programme aucun objet `QueryDef` n'a été utilisé. Dans le second nous illustrons l'utilisation d'un tel objet.

```
Sub ajout()
    Dim Q As QueryDef  ' la question
    Dim nom, adresse, req As String

    nom = InputBox("Nom de la personne à ajouter")
    adresse = InputBox("Adresse de mr " & nom)
    req = "insert INTO table1 (nom,prenom) " _
          & " values ('" & nom & "','" & adresse & "') "
    Set Q = CurrentDb.CreateQueryDef("", req)
    Q.Execute
    MsgBox Q.RecordsAffected & " enregistrement ajouté"

End Sub
```

Nous pouvons maintenant écrire le programme de sélection des commandes de Mr Dupont

comme nous l'avions fait en Cobol et en C. Profitons-en pour montrer comment on peut passer des paramètres à une requète pré-compilée.

```
Public Sub liste()
    Dim Q As QueryDef  ' la requête
    Dim R As Recordset ' le résultat

    Set Q = CurrentDB.CreateQueryDef("",
        "PARAMETERS [mynom] string;
        SELECT cno,design,qute
        FROM fournisseurs f, commandes c, produits p
        WHERE f.fno=c.fno
        AND p.pno=c.pno
        AND nom = [mynom]")

    Q.Parameters("mynom") = "Dupont"
    Set R = Q.OpenRecordset

    R.MoveFirst

    While Not R.EOF
        Debug.Print R![cno], R![design], R![qute]
        R.MoveNext
   Wend

    R.Close
End Sub
```

Chapitre 8

La gestion des transactions

8.1 La tolérance aux pannes

Un système informatique, comme tout autre appareil est sujet à des pannes diverses, l'une des plus fréquentes étant sans doute la coupure d'alimentation.

Dans un tel cas, la perte de données peut s'avérer catastrophique. Le SGBD doit donc proposer un mécanisme permettant de reprendre des traitements interrompus en cours d'exécution.

Il n'est bien sûr pas concevable, dans le cas d'une mise à jour interrompue suite à une panne, ni de relancer la requête à nouveau, car dans ce cas les tuples modifiés avant la panne seraient modifiés une seconde fois, ni d'abandonner son traitement. Dans les deux cas, la base deviendrait alors incohérente.

L'exemple classique concerne une application bancaire dans laquelle on doit transférer une somme S d'un compte A à un compte B. L'une des contraintes du système d'information consiste à toujours vérifier que la somme des soldes des comptes A et B est constante avant et après transfert, ce qui garantit de ne pas perdre d'argent lors d'un virement. Deux ordres doivent être exécutés pour ce traitement :

```
update compte
set solde = solde - S
where num_compte = A ;

update compte
set solde =  solde + S
where num_compte = B ;
```

Si la contrainte est bien vérifiée avant et après ces deux ordres, la base passe évidemment par une phase incohérente entre les deux ordres UPDATE. Une panne entre l'exécution de ces deux ordres aurait des conséquences dramatiques.

La règle à suivre est simple : la base doit toujours être dans un état cohérent. Il est donc nécessaire de pouvoir définir des suites d'opérations considérées par le système comme **atomiques** (indivisibles), au sens où soit toutes les actions de cette suite sont exécutées, soit aucune ne l'est. Ce qui nous conduit à la notion de transaction dont le souci essentiel est de préserver la cohérence de la base de données.

Définition : Une **transaction** est une unité de traitement séquentiel constitué d'une suite d'instructions à réaliser sur la base de données, et qui appliquée à une base cohérente, restitue une base cohérente.

La transaction est en fait un programme qui accède à la base de données et qui peut être constitué d'une requête simple avec les ordres du DML ou être élaborée à partir du langage hôte.

Une transaction peut se trouver dans 4 états différents.

- Active; état initial conservé tant qu'aucune anomalie ne se produit.
- Partiellement validée; lorsque la dernière instruction de la transaction a été atteinte.
- Echouée; suite à une anomalie logique ou physique.
- Validée; après une exécution totalement terminée.

L'action de validation d'une transaction est effectuée par l'ordre **COMMIT**. Cet ordre peut être explicite dans une transaction où, comme c'est la plupart du temps le cas, implicite et exécuté à la fin de la transaction par le SGBD. Une fois l'ordre COMMIT exécuté, les modifications faites sont irrémédiables et la transaction est arrêtée.

Si la transaction a échouée, le SGBD doit alors revenir à l'état précédant le début de la transaction puisque c'est le dernier point de garantie de cohérence. Ceci est fait automatiquement par le SGBD par la commande **ROLLBACK**[1]. L'administrateur a alors la possibilité soit de relancer la transaction si le problème était physique, soit de tuer définitivement cette transaction si le problème était logique (mauvaise conception de la transaction).

Pour assurer la reprise sur panne, les SGBD actuels utilisent la notion de journal.

Définition : Un **journal** est un fichier texte dans lequel le SGBD inscrit, dans l'ordre, toutes les actions de mise à jour[2] qu'il effectue.

Il existe plusieurs méthodes de gestion d'atomicité d'une transaction à l'aide d'un journal. L'une des plus utilisées est l'utilisation d'un journal incrémental avec mise à jour immédiate[3]. Ce journal doit être stocké en mémoire sûre, généralement sur mémoire non volatile avec réplication. En effet le journal ne doit pas pouvoir être détruit, même en cas de panne. La mémoire volatile lui est évidemment interdite !

Le principe d'utilisation de ce type de journal est le suivant :

Chaque ordre inscrit dans le journal est préfixé par le nom de la transaction. Au début de la transaction l'ordre **Start** est inscrit au journal. Ensuite, tout ordre d'écriture dans la base est précédé par l'inscription d'un nouvel enregistrement dans le journal contenant notamment le nom de l'article concerné, l'ancienne valeur de cet article et la nouvelle valeur à affecter. Lorsque la transaction est partiellement validée, un ordre **Commit** est inscrit au journal.

En cas d'arrêt intempestif du système deux cas se présentent :

- Si le journal contient l'enregistrement Start sans enregistrement Commit correspondant, la transaction est annulée par la commande **undo** qui restaure tous les articles mis à jour par la transaction à leur ancienne valeur.
- Si le journal contient les enregistrements Start et Commit, la commande **redo** remet tous les articles mis à jour par la transaction à leur nouvelle valeur.

Pour illustrer cette technique[4], considérons deux transactions de mise à jour de comptes bancaires A et B. La première effectue un transfert de 50F de A vers B. Initialement A=1000F et B=2000F . La seconde effectue un retrait de 100F sur le compte C. Initialement C=700F.

T1
x=read(A)
x=x-50
write(A,x)
y=read(B)
y=y+50
write(B,y)

T2
z=read(C)
z=z-100
write(C,z)

1. COMMIT et ROLLBACK peuvent aussi être utilisés en mode interactif.
2. Pour retrouver un point de cohérence, seules les modifications effectuées sont importantes.
3. D'autres méthodes pratiquent une mise à jour différée, basée sur un principe similaire.
4. Afin d'alléger l'écriture des exemples nous n'utiliserons que des ordres read, write et affiche pour décrire le contenu des transactions.

On suppose que les transactions T1 et T2 sont exécutées l'une à la suite de l'autre, en séquence. Considérons maintenant trois cas de pannes :

1. Le crash intervient juste après l'inscription sur le journal de l'étape write(B,y). Comme T1 possède un Start sans Commit, la commande undo(T1) est effectuée.
2. Le crash intervient juste après la saisie sur le journal de l'étape write(C,z). Comme T1 possède un Start avec Commit, la commande redo(T1) est effectuée. T2 possède un Start sans Commit, la commande undo(T2) est effectuée.
3. Le crash intervient juste après l'inscription du Commit de T2. Les deux transactions possèdent un Start avec Commit, les commandes redo(T1) et redo(T2) sont effectuées.

Illustrons l'état du journal dans ces trois cas :

Cas (1)
T1, Start
T1, A,1000,950
T1, B,2000,2050

Cas (2)
T1, Start
T1, A,1000,950
T1, B,2000,2050
T1, Commit
T2, Start
T2, C,700,600

Cas (3)
T1, Start
T1, A,1000,950
T1, B,2000,2050
T1, Commit
T2, Start
T2, C,700,600
T2, Commit

En cas d'avarie du système il faut alors parcourir l'intégralité du journal pour identifier les transactions à itérer ou à annuler. Ceci demande du temps et est inutile pour toutes les transactions qui ont déjà rafraîchi la base. Afin d'éviter de parcourir tout le journal et de retraiter toutes les transactions, le SGBD jalonne régulièrement le journal de points de contrôle. A chaque exécution d'un point de contrôle, le SGBD s'assure d'écrire tous les blocs modifiés sur disque puis il inscrit le point le contrôle au journal. La reprise sur panne se fait alors à partir de la première transaction précédant le dernier point de contrôle marqué au journal.

De nombreux problèmes système subsistent pour détailler en profondeur le fonctionnement du système de reprise sur panne. Notamment la réalisation d'un gestionnaire de mémoire tampon, de pages d'ombres et d'une mémoire sûre. Ces problèmes sortant du cadre de ce livre, nous ne les aborderons pas ici.

On notera que les ordres COMMIT et ROLLBACK peuvent aussi être utilisés manuellement en mode interactif. Dans ce mode, certains SGBD effectuent automatiquement un commit dans certaines circonstances. Oracle par exemple effectue un Commit implicite après chaque ordre DDL.

8.2 L'accès concurrent aux données

Jusqu'à présent nous avons étudié l'ensemble des primitives d'accès à la base sans tenir compte du fait, qu'en général, plusieurs utilisateurs accèdent simultanément à la base. Les transactions décrites précédemment étaient considérées séquentielles. L'accès aux données par plusieurs transactions simultanées est un point capital pour l'efficacité d'une base de données. L'accès concurrent permet de partager les ressources machines et d'optimiser ainsi le temps CPU.

De nombreux problèmes gérant de gros volumes d'informations nécessitent ce type d'accès concurrent. Voici quelques exemples :

- opérations comptables dans les agences bancaires.
- commandes dans les magasins de vente par correspondance.
- enregistrement des places d'un vol dans un aéroport.
- inscription des étudiants à l'université.

8.2.1 Les problèmes liés à l'accès concurrent

Le problème principal de l'accès concurrent réside dans les opérations de mise à jour. Que se passe t-il lorsque plusieurs utilisateurs tentent de modifier la même donnée au même moment ? Si le système ne prend pas de précautions, des incohérences peuvent apparaître dans la base.

On dit que deux transactions sont concurrentes si elles accèdent simultanément aux mêmes données. L'exécution d'un ensemble de transactions concurrentes est représenté par un interclassement des diverses actions de chaque transaction. on considère bien sûr qu'une exécution doit préserver l'ordre d'apparition des instructions de chacune des transactions.

Bien sûr, les ordres dans chaque transaction ne s'exécutent pas forcément tous à la même vitesse, ne serait-ce qu'à cause de la position de la donnée sur le disque. Il s'en suit donc un décalage entre les ordres des différentes transactions. Illustrons quelques problèmes classiques qui peuvent survenir avec des transactions concurrentes :

- La perte de mise à jour.
  Soient les deux transactions suivantes T1 et T2 qui effectuent une mise à jour sur la même donnée A:

T1
x=read(A)
x=x+10
write(A,x)

T2
y=read(A)
y=y+20
write(A,y)

  si au départ A=50, après exécution séquentielle de T1 puis T2 ou de T2 puis T1 on obtient A=80. Si les transactions sont exécutées de manière concurrente, nous pouvons alors obtenir la séquence suivante :

T1	T2
x=read(A)	
x=x+10	
	y=read(A)
write(A,x)	
	y=y+20
	write(A,y)

  Après cette exécution, A=70 . En d'autres termes, la mise à jour de T2 a écrasé celle de T1 . Il y a perte de mise à jour.
- La création d'incohérences.
  Soient les deux transactions T1 et T2 qui utilisent une base sur laquelle on suppose que la contrainte A=B doit toujours être vérifiée :

T1
x=read(A)
x=x+1
write(A,x)
y=read(B)
y=y+1
write(B,y)

T2
z=read(A)
z=z*2
write(A,z)
t=read(B)
t=t*2
write(B,t)

  Si initialement A=50 et B=50, après exécution de T1 puis T2(resp. T2 puis T1) on obtient A=102 et B=102 (resp. A=101 et B=101) . La contrainte A=B est toujours vérifiée. Si les

transactions sont exécutées de manière concurrente, nous pouvons alors obtenir la séquence suivante :

T1	T2
x=read(A)	
x=x+1	
write(A,x)	
	z=read(A)
	z=z*2
	write(A,z)
	t=read(B)
	t=t*2
	write(B,t)
y=read(B)	
y=y+1	
write(B,y)	

Après cette exécution, A=102 et B=101 . Cette fois ci la base est devenue incohérente !

– Les lectures non reproductibles.
Soient les deux transactions T1 et T2 telles que la transaction T1 effectue deux fois une même lecture sur la base.

T1
x=read(A)
affiche x
x=read(A)
affiche x

T2
y=read(A)
y=y+1
write(A,y)

si initialement A=50, après exécution de T1 puis T2 ou de T2 puis T1, la transaction T1 aura affiché deux fois la même valeur. Si les transactions sont exécutées de manière concurrente, nous pouvons alors obtenir la séquence suivante :

T1	T2
x=read(A)	
affiche x	
	y=read(A)
	y=y+1
	write(A,y)
x=read(A)	
affiche x	

Cette fois-ci les 2 valeurs affichées ne sont plus identiques, les lectures ne sont plus reproductibles.

8.2.2 Caractériser les exécutions correctes

Le contrôle de concurrence est la partie du SGBD qui contrôle l'exécution simultanée de transactions de manière à produire les mêmes résultats qu'une exécution séquentielle. Cette propriété essentielle des systèmes distribués est nommée sérialisabilité.

Définition : Une exécution d'un ensemble de transactions est dite **sérialisable** si elle donne pour chaque transaction participante, le même résultat que l'exécution en série de ces mêmes transactions.

Une condition suffisante pour assurer l'absence de conflits consiste à s'assurer que le mécanisme de contrôle de concurrence ne peut générer que des exécutions sérialisables.

Le rôle du contrôle de concurrence consiste donc à n'autoriser lors de l'exécution de transactions concurrentes que des exécutions sérialisables.

Prenons l'exemple de deux transactions qui effectuent des virements compte à compte. Si les deux comptes traités sont A et B, quelle que soit la somme virée, la somme A+B doit être constante. Selon la manière dont le contrôle de concurrence effectue l'ordonnancement des opérations, l'exécution obtenue sera correcte (sérialisable) ou non.

T1	T2
x=read(A)	
x=x-50	
	y=read(A)
	y=y-100
	write(A,y)
	z=read(B)
write(A,x)	
t=read(B)	
t=t+50	
write(B,t)	
	z=z+100
	write(B,z)

T1	T2
x=read(A)	
x=x-50	
write(A,x)	
	y=read(A)
	y=y-100
	write(A,y)
z=read(B)	
z=z+50	
write(B,z)	
	t=read(B)
	t=t+100
	write(B,t)

Avec A=1000 et B=2000, la première exécution fournit A=950 et B=2100, la contrainte initiale n'est pas préservée. La seconde exécution fournit A=850 et B=2150 ce qui respecte la contrainte.

Il est impossible en général de déterminer si deux programmes sont équivalents quelles que soient les données initiales et donc de tester si deux exécutions de transactions concurrentes ont les mêmes effets. Heureusement, pour savoir si deux transactions risquent d'entrer en conflits il est simplement nécessaire de connaître l'ordonnancement des ordres de lectures et d'écritures sur la base. Les systèmes de gestion des transactions ne regardent pas quelles sont les valeurs affectées aux données mais seulement la date à laquelle les données ont été accédées. On considère donc que des valeurs ne peuvent être identiques que si elles ont été produites exactement par les mêmes séquences d'opérations (ce qui n'est pas toujours exact). Moyennant cette interprétation abstraite de la transaction il existe une méthode de test qui permet de déterminer si une exécution est sérialisable. Précisément, quand on dira qu'une exécution est sérialisable, elle l'est effectivement. Par contre il se peut que des exécutions démontrées non sérialisables le soient quand même (grâce aux propriétés de l'arithmétique par exemple). Ce type d'erreur n'implique pas un dysfonctionnement du système, mais simplement un ralentissement.

Dans nos exemples, afin de pouvoir illustrer les problèmes qui peuvent surgir, nous inscrivons toujours des opérations sur les données. Les opérations sont précisées uniquement à titre pédagogique mais ne sont en aucun cas nécessaires aux algorithmes que nous présentons.

Pour savoir si des transactions risquent de poser des problèmes, il est nécessaire de calculer un graphe de dépendances entre les transactions participant à une même exécution. Les sommets du graphe sont constitués de toutes les transactions participant à l'exécution.
Un arc relie une transaction Ti à une transaction Tj si et seulement si

- Ti exécute un write(X) avant que Tj ne fasse un read(X).
- Ti exécute un read(X) avant que Tj ne fasse un write(X).

Sémantiquement, le fait qu'un arc existe entre Ti et Tj dans le graphe de précédence signifie que dans toute exécution équivalente à celle traitée, Ti doit précéder Tj. On constate que dans le cas d'une exécution séquentielle Ti puis Tj, il n'y a qu'un seul arc entre Ti et Tj.

On démontre que si le graphe de dépendance possède un cycle alors l'exécution n'est pas sérialisable. Dans le cas contraire elle l'est. L'ordre de sérialisation peut être obtenu par tri topologique du graphe. On notera que ce tri n'est en général pas unique.

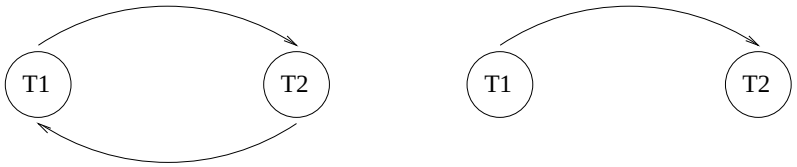


FIG. 8.1 – *graphe des transactions précédentes*

Si nous reprenons les deux exécutions précédentes, on constate après le calcul du graphe de dépendance de chaque exécution (fig. 8.1) que la première exécution possède un cycle tandis que la seconde n'en possède pas. Le graphe indique par son dépliage que la seconde exécution est équivalente à T1 puis T2.

Le test de sérialisabilité d'un ensemble de transactions est coûteux puisqu'il est basé sur la recherche de cycles dans un graphe (On^2 dans un graphe possédant n nœuds). De plus ce calcul ne peut être fait qu'à posteriori quand on connaît l'ordre d'exécution des transactions. Les concepteurs de SGBD se sont donc tournés vers d'autres solutions.

8.2.3 Le système de verrouillage

Afin d'assurer que seules des exécutions sérialisables seront exécutées, plusieurs mécanismes peuvent être mis en place. l'outil le plus répandu[5] pour mener à bien cette tâche est basé sur l'utilisation de verrous. On impose que l'accès aux données se fasse de manière mutuellement exclusive. Un verrou est posé sur chaque donnée accédée par une transaction afin d'empêcher les autres transactions d'y accéder. Cette technique est utilisée dans tous les systèmes commerciaux actuels.

On considère qu'un verrou est placé avant une lecture ou une écriture d'une donnée. Si une transaction tente de verrouiller une donnée déjà verrouillée, elle est obligée d'attendre que l'autre transaction débloque cette donnée. C'est le gestionnaire de transactions qui gère les files d'attentes des transactions. CICS (Customer Information Control System) est sans doute le plus connu de ces outils sous système IBM MVS.

La taille de la donnée verrouillée est appelée granularité. C'est au gestionnaire de verrous qu'incombe la tâche de gérer l'accès aux données. Sous DB2 avec le système MVS il se nomme IRLM (IMS Resource Lock Manager). Pour gérer les verrous le système utilise une table des verrous posés. A chaque verrou correspond une ligne définissant la donnée verrouillée, le type de verrou posé et le nom de la transaction qui l'a posé. Quand un verrou est levé, la ligne correspondante dans la table des verrous est supprimée. Une granularité forte implique un faible parallélisme, tandis qu'une granularité faible implique la gestion d'une table des verrous importante. On peut imaginer bloquer la table entière durant la transaction ou ne bloquer que la page physique (unité de 4096 octets ou 32768 octets) contenant les données. Les meilleurs systèmes actuels parviennent à travailler de manière idéale en ne bloquant que les lignes des données accédées par la transaction.

Dans la majorité des systèmes il existe trois types de verrous :

- Les verrous posés en cas de lecture. Si une transaction lit une donnée, aucune autre transaction ne doit pouvoir la modifier tant que cette transaction n'est pas terminée. C'est le mode partagé (*Shared*).
- Les verrous posés en cas de sélection pour mise à jour. D'autres transactions peuvent lire les données à condition qu'elles n'aient pas l'intention de les modifier. C'est le mode protégé (*Protected*).

5. bien qu'assez restrictif. D'autres mécanismes, basés sur l'utilisation d'estampilles ou sur l'utilisation de graphes de sérialisibilité sont aussi possibles.

– Les verrous posés en cas d'écriture. Si une transaction vient d'écrire une donnée, aucune autre ne doit pouvoir la lire tant que cette transaction n'est pas terminée. C'est le mode exclusif (*eXclusive*).

La compatibilité entre ces trois modes est résumée Fig.8.2. Deux modes sont incompatibles si Non est inscrit à leur intersection dans la matrice de verrouillage. Par exemple, si une transaction a obtenu un verrou de type *Share* sur une donnée, un autre verrou de type *Share* sera accepté mais un verrou *eXclusive* sera refusé.

Verrou	S	P	X
S	Oui	Oui	Non
P	Oui	Non	Non
X	Non	Non	Non

FIG. 8.2 – *Matrice des compatibilités de verrouillage*

On utilisera par la suite les notations lockS, lockP et lockX pour les différents verrous ainsi que unlock pour le déverrouillage.

Voici deux exemples d'exécutions concurrentes possibles de transactions avec verrous. T1 effectue un virement compte à compte, T2 et T3 affichent la somme de deux comptes (elles ne font que des ordres de lecture).

T1	T2
lockX(B)	
x=read(B)	
x=x-10	
write(B,x)	
unlock(B)	
	lockS(A)
	y=read(A)
	unlock(A)
	lockS(C)
	z=read(C)
	unlock(C)
	affiche(y+z)
lockX(A)	
t=read(A)	
t=t+10	
write(A,t)	
unlock(A)	

T1	T3
lockX(B)	
x=read(B)	
x=x-10	
write(B,x)	
unlock(B)	
	lockS(A)
	y=read(A)
	unlock(A)
	lockS(B)
	z=read(B)
	unlock(B)
	affiche(y+z)
lockX(A)	
t=read(A)	
t=t+10	
write(A,t)	
unlock(A)	

L'usage de verrous n'est néanmoins pas suffisant pour assurer la sérialisabilité d'une exécution. Là encore il existe un test permettant de savoir si une exécution avec verrous est sérialisable. Elle se base toujours sur la recherche de cycles dans un graphe de dépendance.

Les sommets du graphe sont constitués de toutes les transactions participant à l'exécution. Un arc relie une transaction Ti à une transaction Tj si et seulement si

– il existe une donnée X telle que Ti a posé un verrou dans un mode A sur X et Tj pose ultérieurement un verrou sur X dans un mode B incompatible avec A.

Sémantiquement, le fait qu'un arc existe entre Ti et Tj dans le graphe de précédence signifie que dans toute exécution équivalente à celle traitée, Ti doit précéder Tj.

Si le graphe de dépendance possède un cycle alors l'exécution n'est pas sérialisable. Dans le cas contraire elle l'est. L'ordre de sérialisation peut être obtenu par tri topologique du graphe. On notera que ce tri n'est en général pas unique.

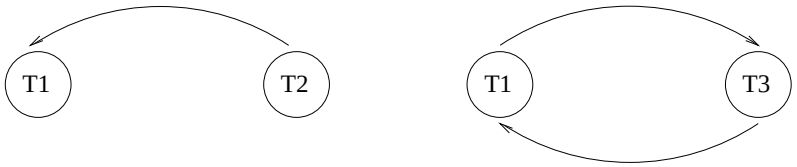


FIG. 8.3 – *graphe des transactions précédentes avec verrou*

Comme on peut le voir sur le graphe Fig. 8.3 l'exécution précédente entre T1 et T2 est sérialisable tandis que T1 et T3 ne l'est pas. Cet exemple montre par ailleurs que même si l'auteur d'une transaction n'effectue que des lectures, l'exécution de celle ci n'est pas sûre si un protocole de verrouillage adéquat n'est pas appliqué.

8.2.4 Le protocole à deux phases

S'il est aisé de savoir quand poser les verrous, la levée des verrous est beaucoup plus délicate. En effet d'autres précautions sont encore à prendre. Si on tente d'augmenter le rendement du traitement simultané des transactions en déverrouillant les données dès que possible, la base peut se trouver en état inconsistant. Si on ne déverrouille pas un article avant d'en verrouiller un autre, on crée des interblocages. Il faut donc imposer aux transactions un protocole de déverrouillage. Le plus fréquent est le protocole de verrouillage à deux phases, qui garantit la sérialisabilité de l'exécution des transactions. Ce protocole implique que chacune des transactions émette ses demandes de verrouillage et de déverrouillage en deux phases distinctes :

- Verrouillage croissant. La transaction peut obtenir de nouveaux verrouillages mais pas de nouveaux déverrouillages
- Verrouillage décroissant. La transaction peut obtenir des déverrouillages mais pas de nouveaux verrouillages.

Initialement une transaction est en phase de verrouillage croissant. Lorsqu'elle libère un verrou, elle entre en phase de décroissance et aucun verrouillage ne peut plus être demandé.

Définition : Une transaction est dite à **deux phases** si elle n'effectue aucun LOCK après avoir exécuté un UNLOCK.

Voici les deux transactions T1 et T3 précédentes transformées pour être compatibles avec le protocole de verrouillage à deux phases.

T1'
lockX(B)
x=read(B)
x=x-10
write(B,x)
lockX(A)
t=read(A)
t=t+10
write(A,t)
unlock(B)
unlock(A)

T3'
lockS(A)
y=read(A)
lockS(C)
z=read(C)
display(y+z)
unlock(A)
unlock(C)

Bien que ce verrouillage à deux phases assure la sérialisabilité il n'évite pas pour autant les interblocages. Il se peut en effet que plusieurs transactions s'attendent mutuellement. C'est le cas avec la transaction à deux phases T4 qui fonctionne comme T3' (lui-même transformation deux phases de T3) mais sur la donnée B à la place de la donnée C.

T1'
lockX(B)
x=read(B)
x=x-10
write(B,x)
lockX(A)
t=read(A)
t=t+10
write(A,t)
unlock(B)
unlock(A)

T4
lockS(A)
y=read(A)
lockS(B)
z=read(B)
display(y+z)
unlock(A)
unlock(B)

Avec ces deux transactions deux phases, des interblocages[6] peuvent se produire. Il y a interblocage si plusieurs transactions s'attendent mutuellement pour accéder aux données. En voici un exemple :

T1'	T4
lockX(B)	
x=read(B)	
x=x-10	
write(B,x)	
	lockS(A)
	y=read(A)
	lockS(B)
lockX(A)	
⋮	⋮

Il existe plusieurs manières d'éviter les verrous mortels :

- Obliger les transactions à demander tous les verrous simultanément. Le système les accepte alors tous à la fois ou aucun. Des données peuvent donc être verrouillées longtemps sans être utilisées.
- Fixer une relation d'ordre sur les données et obliger les transactions à demander leurs verrous dans cet ordre.
- Quand une situation d'interblocage se produit, arrêter l'une transactions incriminée et la réexécuter ultérieurement. Pour cela on affecte à chaque transaction une estampille unique et on utilise ces estampilles pour décider si en cas de conflit, une transaction doit être mise en attente ou rejetée. Si elle est rejetée elle reste affectée à la même estampille, ce qui la fait "vieillir". Deux schémas existent pour régler les conflits entre transactions plus jeunes et plus vieilles : Wait-Die et Wound-Wait.

  1. Avec le schéma Wait-Die (attendre ou mourir), si une transaction Ta détient une donnée et que Tb la réclame dans un mode non compatible, on autorise Tb à attendre ssi Tb est plus vieille que Ta. Dans le cas contraire, Tb est avortée, elle sera relancée avec la même estampille. Donc plus Tb vieillit plus elle est autorisée à attendre.
  2. Avec le schéma préemptif Wound-Wait (Blesser ou attendre) si une transaction Ta détient une donnée et que Tb la réclame dans un mode non compatible, on autorise Tb à attendre si elle est plus jeune que Ta. Dans le cas contraire, Ta est blessée (avortée et relancée avec la même estampille). Donc plus Tb est vieille plus elle est autorisée à accéder aux données.

Que ce soit dans le schéma Wait-Die ou dans le schéma Wound-wait, la plus vieille des transactions ne peut être rejetée. Ces deux shémas évitent les situations de deadlock.

6. parfois appelés verrous mortels ou situation de deadlock

Prenons l'exemple de 3 transactions Ta, Tb et Tc d'estampilles respectives 5,10 et 15.

1. Wait-Die :
   Si Ta réclame un article traité par Tb, Ta attend.
   Si Tc réclame un article traité par Tb, Tc est rejetée.
2. Wound-Wait :
   Si Ta réclame un article traité par Tb, Tb est rejetée et Ta prend la donnée.
   Si Tc réclame un article traité par Tb, Tc attend.

Selon les types d'applications gérées par l'entreprise, on préfèrera soit le schéma par estampillage de type Wound-Wait (c'est le cas pour le contrôle de processus ou la gestion bancaire) soit le schéma Wait-Die.

Pour prendre en compte l'ordonnancement des transactions, la gestion des files d'attentes entre transactions, la pose et la levée des verrous il faut un outil spécial : Le moniteur transactionnel. Le plus connu est sans doute CICS fourni par IBM pour systèmes MVS.

C'est aussi à lui de vérifier et de gérer les points suivants :

- Quand une transaction ne peut pas accéder à une donnée à cause d'un verrou, elle est mise en attente. Dès que la transaction concurrente libère les verrous, elle se remet en exécution.
- Si une transaction est en attente depuis une durée supérieure à une durée limite, la transaction est rejetée.
- Si plusieurs transactions s'attendent mutuellement (deadlock), le système choisit une transaction qui sera rejetée.

Pour conclure ce chapitre il est important de dire que la manière dont le journal, les verrous et l'ordonnancement des transactions sont effectués restent transparents pour l'utilisateur. Tout se passe comme si l'utilisateur était seul avec sa base de données. Néanmoins la compréhension des mécanismes profonds permettant l'accès concurrent est important pour écrire des transactions efficaces. On prendra garde notamment et autant que faire se peut à

- grouper les lectures ensembles et les écritures ensembles dans la transaction, ou mieux, dans des transactions séparées.
- éviter des traitements longs entre les lectures et les écritures.
- regrouper les interventions de l'opérateur en début de transaction et les lectures-écritures en fin de transaction.

Chapitre 9

Normalisation des Relations

Lors de la phase de conception d'une base de données relationnelle, la question que le concepteur se pose le plus souvent est : *Quels sont, parmi les ensembles de relations modélisant mon schéma conceptuel, les plus à mêmes à éviter les problèmes de cohérence pouvant survenir lors d'opérations de mise à jour ?* . Les relations constituant ces bases "bien faites" seront dites "normalisées"; nous verrons qu'il existe une échelle de nuances dans cette notion de "normalisation".

9.1 Le besoin de Normalisation.

Considérons une administration dont le parc de machines à écrire est représenté par la relation unique suivante: *Parc(numero_série, type, caractéristiques_techniques, lieu_d'installation, ...)*. Cette relation est naturellement soumise à la contrainte d'intégrité suivante: *des machines de même type ont les mêmes caractéristiques techniques*. On peut intuitivement relever des redondances d'informations : plusieurs occurrences du même couple *(type, caractéristiques_techniques)* peuvent être présentes dans cette relation. Ces redondances engendreront immanquablement des difficultés de mise à jour:

- **Anomalie d'Insertion.** Telle qu'elle se présente, cette relation ne permet pas de mémoriser les caractéristiques techniques d'un type de machine n'existant pas dans le parc.
- **Anomalie de Suppression.** La disparition d'une machine, qui est l'unique représentante de son type, entraîne également la perte des informations techniques dont on disposait sur ce type de machine.
- **Anomalie de modification.** Toute modification portant sur les caractéristiques techniques d'un type de machine doit être répercutée sur tous les tuples correspondant à des machines de ce type; ce qui allonge les temps de mise à jour, et augmente les risques d'incohérence.

La théorie de la normalisation repose sur l'analyse des dépendances entre attributs qui sont à l'origine des phénomènes de redondances, et propose des méthodes systématiques visant à décomposer les relations incriminées afin qu'il soit toujours possible de reconstituer la relation originelle par une opération de l'algèbre relationnelle : la jointure.

Ces méthodes conduisent à des résultats "normaux", c'est-à-dire conformes à ceux que l'on aurait obtenus empiriquement pourvu que l'on ait une bonne perception de la problématique liée aux redondances. Dans la majorité des cas, les relations échafaudées directement par le concepteur de bases de données sont normalisées. Certains $\mathcal{AGL}$ (ateliers de génie logiciel) produiront directement les tables normalisées à partir du modèle conceptuel de données (MCD). La normalisation des tables n'est pas non plus un dogme totalitaire, si bien que, dans certaines situations, souvent pour des raisons d'efficacité, on est amené à *dénormaliser*.

9.2 La Première Forme Normale.

Une relation est dite en **Première forme normale - 1NF** si tous ses attributs sont atomiques (i.e. ne peuvent pas être décomposés du point de vue du contexte dans lequel est envisagée la relation).

En d'autres termes, pour qu'une relation soit en 1NF aucun des attributs qui la composent ne doit être lui-même une relation entre "sous-attributs". Cette condition correspond pratiquement à l'idée de "table" (les attributs en colonne, et les tuples de la relation en ligne).

Exemple. La relation suivante :

Nom	*Notes*
Einstein	8, 12.5
Freud	2.5, 0, 18
....	.....

n'est pas en 1NF. Pour qu'elle le soit, on peut soit créer un attribut par note à la condition que le nombre maximal de notes soit connu et que le nombre moyen de notes par personne soit voisin de ce maximum,

Nom	*Note1*	*Note2*	*Note3*
Einstein	8	12.5	**NULL**
Freud	2.5	0	18
....	.....	.....	.....

soit fabriquer autant de tuples que de couples (nom, note).

Nom	*Note*
Einstein	8
Einstein	12.5
Freud	2.5
Freud	0
....	.....

9.3 Notion de dépendance fonctionnelle: $\mathcal{DF}$.

Cette notion tend à capturer l'idée commune de "dépendance" entre des informations : *le salaire dépend de la qualification, le montant de la vignette automobile dépend de l'âge du véhicule et de sa puissance, le prix d'un article ne dépend ni du vendeur, ni de l'acheteur (dans la plupart des cas).*

Définition. Soit $R(\Delta)$ une relation, Δ ses attributs et X et Y des groupes d'attributs de R, il existe une **dépendance fonctionnelle** entre X et Y (on dit aussi que X détermine Y et l'on note $X \rightarrow Y$) si dans la relation R chaque valeur de X détermine une et une seule valeur de Y.
Plus formellement, si pour toute extension r de R et pour tous les tuples t_1 et t_2 de R on a $\Pi_X(t_1) = \Pi_X(t_2) \Rightarrow \Pi_Y(t_1) = \Pi_Y(t_2)$ [1].

Exemple. Ensemble de $\mathcal{DF}$ avec une extension de la relation associée conforme à ces $\mathcal{DF}$:

$$\begin{array}{ll} A & \rightarrow B \\ B, C & \rightarrow D \\ D & \rightarrow E \\ A, C & \rightarrow D \\ A, C & \rightarrow E \end{array}$$

A	B	C	D	E
a1	b1	c1	d3	e2
a1	b1	c3	d4	e3
a2	b2	c4	d2	e1
a3	b1	c1	d3	e2
a2	b2	c4	d2	e1

1. On rappelle que $\Pi_X(t)$ est la projection sur X du tuple t, autrement dit la valeur de t dans la ou les colonnes X

Il est important de noter qu'une dépendance fonctionnelle est une assertion sur toutes les extensions possibles de la relation et non pas uniquement sur les valeurs actuelles.

9.3.1 Propriétés des dépendances fonctionnelles.

Afin de manipuler les dépendances fonctionnelles il existe 3 règles qui permettent le cas échéant de déduire de nouvelles dépendances fonctionnelles et de permettre l'élaboration de preuves. Prenons comme référence une relation $R(\Delta)$ et les lettres majuscules W, X, Y, Z pour désigner des groupes d'attributs de cette relation.

- *La reflexivité.* Si X est un ensemble d'attributs tels que $Y \subseteq X$ alors on a $X \to Y$ (et comme dépendance triviale $X \to X$).
- *L'augmentation.* Si $X \to Y$ et W est un ensemble d'attributs alors $WX \to WY$ (WX est une notation simplifiée de $W \cup X$).
- *La transitivité.* Si $X \to Y$ et $Y \to Z$ alors $X \to Z$

Ces règles sont correctes car elles ne déduisent que des $\mathcal{DF}$ valides, et elles sont complètes car elles permettent de déduire toute dépendance valide à partir des axiomes de départ. On appelle ces règles les **axiomes d'Armstrong** en l'honneur de leur auteur.

Exemple. Démontrer à partir de l'ensemble de $\mathcal{DF}$ précédent que $A, D \to B, E$.
Par augmentation de $A \to B$ avec D on obtient $A, D \to B, D$.
Par augmentation de $D \to E$ avec B on obtient $B, D \to B, E$.
Par transitivité des deux précédentes on obtient $A, D \to B, E$.

Afin de faciliter les démonstrations, on est amené à construire à partir des axiomes d'Armstrong, d'autres règles parfois plus simples à manier. On utilisera avantageusement l'union, la décomposition et la pseudo-transitivité.

- *L'union.* Si $X \to Y$ et $X \to Z$ alors $X \to YZ$.
- *La décomposition.* Si $X \to YZ$ alors $X \to Y$ et $X \to Z$
- *La pseudo-transitivité.* Si $X \to Y$ et $WY \to Z$ alors $WX \to Z$

Ces règles de bon sens peuvent bien sûr se démontrer à partir des axiomes d'Armstrong. Démontrons par exemple la règle d'Union : L'hypothèse de départ est :$X \to Y$ et $X \to Z$.
Par augmentation de $X \to Y$ avec X, on obtient $X, X\,(i.eX) \to X, Y$.
Par augmentation de $X \to Z$ avec Y, on obtient $X, Y \to Y, Z$.
En utilisant finalement la transitivité on obtient : $X \to Y, Z$.

9.3.2 Notions de fermeture

Dans les travaux sur la normalisation il est souvent nécessaire de connaître tous les attributs d'une relation dont la valeur est forcée par celle des autres attributs. La solution consiste à calculer la fermeture de l'ensemble des $\mathcal{DF}$ dont on dispose.

Fermeture d'un ensemble de $\mathcal{DF}$. On appelle **fermeture** (ou clôture) d'un ensemble F de $\mathcal{DF}$, l'ensemble de toutes les $\mathcal{DF}$ qui sont conséquences de F. On note cet ensemble $F+$.

Cet ensemble peut être calculé à l'aide des axiomes d'Armstrong, mais ce calcul se révèle très vite fastidieux car le nombre de dépendances à calculer est souvent très important.

On notera néanmoins que cette notion est importante pour montrer l'équivalence de deux ensembles de $\mathcal{DF}$. Deux ensembles de $\mathcal{DF}$ sont dits **équivalents** lorsqu'ils ont la même fermeture.

Pour savoir si une dépendance fonctionnelle $X \to Y$ est conséquence d'un ensemble de $\mathcal{DF}$ F il existe un autre procédé bien plus pratique basé sur l'algorithme suivant :

Fermeture d'un ensemble d'attributs. Cet algorithme de saturation permet d'obtenir tous les attributs déterminés par un ensemble d'attributs X relativement à l'ensemble des dépendances fonctionnelles F.

$result = X$
`TQ (`*result* `évolue)`
`Pour chaque dépendance fonctionnelle` $Y \rightarrow Z$ `dans` F
`Si` $Y \subseteq result$ `Alors` $result = result \cup Z$
`FTQ`

Considérons par exemple l'ensemble F de dépendances:

$$A \rightarrow B, C$$
$$A \rightarrow H$$
$$B \rightarrow H$$
$$C, G \rightarrow H$$
$$C, G \rightarrow I$$
$$A, B \rightarrow I$$

On démontre facilement, soit par les axiomes d'Armstrong, soit par l'algorithme précédent que $A, G \rightarrow I$ car la clôture de $\{AG\}$ est $\{AGBCHI\}$.

Notion de clé. Une **clé candidate** d'une relation $R(\Delta)$ est un groupe d'attributs X tel que $X \rightarrow \Delta$ et $\forall X' \subset X \ \neg(X' \rightarrow \Delta)$. Une clé est donc un groupe minimal d'attributs qui détermine tous les autres.

Cette notion est fondamentale. Toutes les clés possibles d'une relation doivent être impérativement identifiées avant toute tentative de normalisation.

Lors des implémentations, on appelle parfois **clé primaire** la clé candidate ayant été privilégiée, mais cette notion n'a a priori aucune importance en ce qui concerne l'identification des redondances.

9.3.3 Notion de couverture irredondante.

Un ensemble de $\mathcal{DF}$ n'est pas forcément exprimé de la manière la plus claire et lisible par le demandeur. Certaines sont parfois redondantes et d'autres découlent de celles qui sont exprimées. Il est donc intéressant de définir un ensemble de $\mathcal{DF}$ équivalent (au sens de la clôture) à l'ensemble initial mais avec uniquement les dépendances les plus "pertinentes". Cet ensemble sera appelé couverture irredondante de l'ensemble des $\mathcal{DF}$ initiales.

Dépendance fonctionnelle élémentaire. Soit X un groupe d'attributs et A un attribut unique non inclus dans X, la dépendance fonctionnelle $X \rightarrow A$ est dite **élémentaire** si A ne dépend pas aussi d'un sous ensemble d'attributs de X. Plus formellement $\forall X' \subset X \ \neg(X' \rightarrow A)$.

Le fait que A soit atomique n'est pas une contrainte car par la règle de décomposition toute dépendance de la forme $X \rightarrow Y$ avec Y non atomique peut être mise immédiatement sous forme de plusieurs dépendances élémentaires. C'est le cas de la dépendance $A \rightarrow B, C$ de l'exemple précédent qui est trivialement décomposée en $A \rightarrow B$ et $A \rightarrow C$.

Les dépendances non élémentaires alourdissent inutilement l'ensemble des $\mathcal{DF}$. Il est donc intéressant dans un premier temps de simplifier les parties gauches de chaque dépendance en supprimant les attributs redondants.

Vérification de dépendances élémentaire. Grâce à l'algorithme de saturation précédent, il est très facile de démontrer qu'une dépendance fonctionnelle $X \to A$ est élémentaire. Si elle ne l'est pas, il existe alors un groupe d'attributs B qui est inutile pour cette dépendance. Il suffit donc de calculer $(X - \{B\})+$ à partir de F et vérifier que l'on obtient encore A. Si c'est le cas B est redondant. Si l'on reprend l'exemple de $\mathcal{DF}$ précédent, la dépendance $A, B \to H$ qui en découle n'est pas élémentaire, les autres par contre sont élémentaires. Cette dépendance peut être simplifiée en $A \to H$.

Le calcul de l'ensemble de toutes $\mathcal{DF}$ élémentaires à partir d'un ensemble de $\mathcal{DF}$ élémentaires initial se fait très facilement. Seule la règle de transitivité fournit de nouvelles $\mathcal{DF}$ élémentaires. Il suffit donc de calculer la **clôture transitive** d'un ensemble de $\mathcal{DF}$ élémentaires pour obtenir l'ensemble complet des $\mathcal{DF}$ élémentaires.

Dépendance fonctionnelle redondante. De manière similaire au calcul de dépendances élémentaires, il est aussi possible d'éliminer les dépendances élémentaires qui s'obtiennent pas transitivité à partir des autres qui sont appelées **dépendances redondantes** d'un ensemble de dépendances F. Pour savoir si $X \to A$ est redondant, il suffit de calculer $X+$ à partir de $F - \{X \to A\}$ et vérifier que l'on obtient encore A. Avec l'ensemble de $\mathcal{DF}$ précédent, la dépendance $A, G \to I$ qui en découle est une dépendance redondante tout comme $A \to H$ ou toute autre dépendance obtenue par transitivité.

On appelle **couverture irredondante** d'un ensemble de $\mathcal{DF}$, l'ensemble F de dépendances fonctionnelles élémentaires vérifiant les propriétés :

- Toute $\mathcal{DF}$ élémentaire est dans $F+$.
- Aucune dépendance de F n'est redondante.

On démontre aisément que tout ensemble de dépendances possède une couverture irredondante. Il existe en général plusieurs couvertures irredondantes possibles pour un ensemble de $\mathcal{DF}$ selon l'ordre par lesquelles les dépendances redondantes sont supprimées.

Pour obtenir une couverture irredondante d'un ensemble de $\mathcal{DF}$ il suffit donc de simplifier l'ensemble de $\mathcal{DF}$ initial afin de supprimer les attributs irredondants et ne plus avoir que des conclusions atomiques, puis de supprimer les dépendances élémentaires redondantes.

Exemple. Reprenons l'un des ensembles de $\mathcal{DF}$ précédemment utilisé:

$$\begin{array}{ll} A & \to B \\ B, C & \to D \\ D & \to E \\ A, C & \to D \\ A, C & \to E \end{array}$$

Ces dépendances sont élémentaires (aucune partie gauche n'est redondante et chaque conclusion est atomique). $A, C \to D$ est redondant car obtenu par $A \to B$ puis augmentation de C et enfin transitivité avec $B, C \to D$. De même $A, C \to E$ est redondant car obtenu par transitivité de $A, C \to D$ et $D \to E$.

La couverture irredondante de cet ensemble est donc :

$$\begin{array}{ll} A & \to B \\ B, C & \to D \\ D & \to E \end{array}$$

On notera que la couverture irredondante est parfois appelée *minimale* (alors qu'elle n'est pas forcément minimale au sens du nombre de $\mathcal{DF}$ qu'elle contient).

9.4 Décomposition d'une relation en sous-relations.

Quand une relation pose des problèmes de mise à jour, la théorie de la normalisation propose de décomposer la relation initiale en plusieurs sous-relations qui ne souffrent plus de ces anomalies. Bien sûr, la relation initiale R doit toujours pouvoir être reconstruite par opérations de jointure des sous-relations R_i.

$$R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$$

S'il est très facile de décomposer une relation en sous-relations, les décompositions qui nous intéressent doivent vérifier deux critères fondamentaux: être à jonction conservative et préserver les dépendances fonctionnelles.

9.4.1 Décomposition à jonction conservative.

Une décomposition est à jonction conservative si la jointure naturelle des sous-relations calcule exactement tous les tuples de la relation initiale. Dans un tel cas la décomposition est dite **sans perte**. Il est formellement possible de démontrer que la décomposition d'une relation R en deux sous-relations R_1 et R_2 est à jonction conservative. Il suffit pour cela que l'une des deux propriétés suivantes soit vérifiée:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

Exemple. Cet exemple montre que selon la décomposition d'une relation vérifiant un ensemble de $\mathcal{DF}$ en deux sous-relations, la jointure peut très bien ne pas fournir la relation initiale.

A	B	C	D
a	5	x	2
b	5	y	1
c	5	x	2

A $\rightarrow$ B,C
C $\rightarrow$ D
D $\rightarrow$ B

La relation précédente est décomposée en deux sous-relations $R1(A, B)$ et $R2(B, C, D)$ et pourtant la jointure de $R1$ et $R2$ ne fournit pas la relation initiale (la jointure fournit 6 tuples alors que la relation initiale n'en possédait que 3 !).

A	B
a	5
b	5
c	5

B	C	D
5	x	2
5	y	1

On vérifiera par la propriété précédente que d'autres décompositions comme R1(A,B,C) R2(A,D) ne posent par contre aucun problème.

Il existe aussi des algorithmes syntaxiques basés sur l'élaboration d'une extension de R, qui montrent que l'éclatement d'une relation en sous-relations est à jonction conservative. L'avantage de ces algorithmes est qu'ils fonctionnent sur un nombre quelconque de relations et qu'ils fournissent un contre-exemple immédiatement.

9.4.2 Décomposition avec préservation des dépendances fonctionnelles.

Après décomposition d'une relation R soumise aux dépendances fonctionnelles F en un ensemble de relations R_i il doit être possible, lorsque l'on effectue des mises à jour sur la base, de vérifier que l'on n'entre pas en contradiction avec les dépendances relatives à la base. Pour effectuer cette vérification il est souhaitable de concevoir des bases qui permettent cette vérification sans effectuer de jointure. Il faut vérifier notamment que si F_i est l'ensemble des dépendances de $F+$ qui ne concernent que R_i, toutes les dépendances de $F+$ se déduisent de $F' = (\bigcup F_i)+$. Autrement

dit, chacune des dépendances de F doit être impliquée par F'.
Cette propriété est appelée conservation des dépendances fonctionnelles.

Par exemple, si l'on reprend la décomposition à jonction conservative proposée précédemment sur la relation suivante:

R	A	B	C	D
	a	5	x	2
	b	5	y	1
	c	5	x	2

A → B,C
C → D
D → B

R1(A,B,C) R2(A,D) est une décomposition à jonction conservative mais avec perte de $\mathcal{DF}$. Ex: $D \rightarrow B$ ne peut pas se retrouver sur la projection des $\mathcal{DF}$ initiales sur la décomposition.

La jonction conservative et la préservation des $\mathcal{DF}$ sont deux critères qui doivent être vérifiés pour chaque découpage d'une relation en sous-relations. Ces critères sont indépendants et il se peut fort bien qu'une décomposition possède seulement l'un ou l'autre de ces deux critères ou même aucun des deux !

Le théorème suivant exprime la possibilité de décomposer "proprement" une relation dès lors que l'on a identifié une dépendance fonctionnelle inter-attributs.

Théorème de décomposition. Si $R(\Delta)$ est une relation avec $X \rightarrow Y$ comme dépendance fonctionnelle inter-attributs[2] alors $R(\Delta) = R(\Delta_1) \bowtie_X R(\Delta_2)$ avec $\Delta_1 = X \cup Y$ et $\Delta_2 = \Delta - \{Y\}$.

On démontrera aisément que toute décomposition obtenue par ce théorème de décomposition est à jonction conservative au prix parfois de perte de dépendances fonctionnelles.

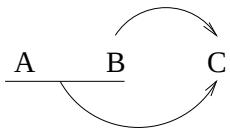
9.5 La Deuxième Forme Normale.

Une relation est dite en **deuxième forme normale - 2NF** lorsqu'elle est en 1NF et à la condition qu'un attribut n'appartenant à aucune clé soit en dépendance élémentaire avec toutes les clés (pas de dépendance partielle).

En d'autres termes, dès qu'un attribut n'appartenant pas à une clé dépend d'une partie d'une clé, la relation n'est pas 2NF. Dès lors qu'une relation n'est pas en 2NF, des redondances d'informations peuvent survenir.

Exemple abstrait Exemple de relation 1NF qui n'est pas 2NF illustrée par une extension avec redondances.

A,B → C
B → C



A	B	C
a1	b1	c1
a2	b1	c1

Exemple concret Soit la relation *PrêtK7(NumCli, NomCli, AdrCli, NumK7, TitK7, DatePrêt)* qui permet de gérer les prêts de cassettes vidéo aux clients.

Dressons la liste des dépendances fonctionnelles de base entre les différents attributs de cette relation:

$$\begin{aligned} NumCli &\rightarrow NomCli, AdrCli \\ NumK7 &\rightarrow TitK7 \\ NumCli, NumK7 &\rightarrow DatePret \end{aligned}$$

2. La notation $R(\Delta_i)$ étant bien sûr une notation abrégée de $\Pi_{\Delta_i}(R)$

et donc par application des propriétés des $\mathcal{DF}$ on obtient :

$$NumCli, NumK7 \rightarrow NomCli, AdrCli, TitK7, DatePret$$

$NumCli, NumK7$ est la seule clé de cette relation, or $NomCli$ et $AdrCli$ ne dépendent que de $NumCli$, de même $TitK7$ ne dépend que de $NumK7$; la relation *PrêtK7* n'est donc pas en 2NF. En effet, dès qu'un client loue plusieurs cassettes, son nom et son adresse sont inutilement reproduits dans la base; de même le titre d'une cassette est répété pour chacun de ses emprunts.

L'application du théorème de décomposition aux dépendances fonctionnelles fautives conduirait à l'éclatement de la relation *PrêtK7* en trois relations en 2NF:
Client(NumCli, NomCli, AdrCli)
K7(NumK7,TitK7),
Prêt(NumCli,NumK7,DatePrêt)

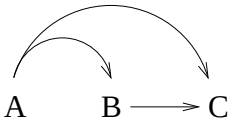
9.6 Troisième Forme Normale.

Une relation est dite en **troisième forme normale - 3NF** lorsqu'elle est en 2NF et à la condition que tout attribut n'appartenant pas à une clé ne dépend pas d'un attribut non clé.

En d'autres termes, dès qu'il existe une dépendance entre deux attributs n'appartenant pas à une clé, la relation n'est pas en 3NF. Dès lors qu'une relation n'est pas en 3NF, des redondances d'informations peuvent survenir.

Exemple abstrait Exemple de relation 2NF qui n'est pas 3NF illustrée par une extension avec redondances.

A $\rightarrow$ B,C
B $\rightarrow$ C



A	B	C
a1	b1	c1
a2	b1	c1

Exemple concret Soit la relation *Enseignant(Nom, Bureau, Bâtiment, Discipline, téléphone)* complétée par les règles de gestion (contrainte d'intégrité) suivantes : *Un bâtiment héberge des enseignants d'une même discipline; un bureau d'un bâtiment donné possède un numéro d'appel unique.*

Il est évident que l'attribut **Nom** est ici la clé de cette relation. Par ailleurs, les règles de gestion se traduisent par les $\mathcal{DF}$ suivantes:

$$\begin{aligned} Batiment &\rightarrow Discipline \\ Batiment, Bureau &\rightarrow Telephone \\ Telephone &\rightarrow Bureau, Batiment \end{aligned}$$

La dépendance $Batiment \rightarrow Discipline$ ne concerne pas la clé, la relation *Enseignant* n'est pas en 3NF. En exploitant le théorème de décomposition de façon à isoler les $\mathcal{DF}$ indirectes, on obtient alors les relations :

Siège(Nom,Bureau, Bâtiment)
Annuaire(Bureau, Bâtiment, Téléphone)
Affectation(Bâtiment,Discipline)

Théorème. Toute relation admet une décomposition en 3NF à jonction conservative (sans perte) et avec préservation des dépendances fonctionnelles.

Le processus de conception de schéma relationnel, de la qualité duquel dépendra la pertinence et l'efficacité de la base de donnée, devra prendre en compte ces critères (normalité, préservation des contenus, préservation des dépendances fonctionnelles). Or, à partir du recueil des $\mathcal{DF}$, il est possible de construire un "bon" schéma relationnel en exploitant des manipulations algorithmiques des dépendances (recherche de clé...), des algorithmes de conception de schéma relationnel en 3NF (algorithme de *synthèse* ou algorithme de *décomposition*).

9.6.1 Algorithme de Normalisation.

L' algorithme de conception de schéma relationnel en 3NF (appelé **Algorithme de synthèse**) peut alors s'énoncer comme suit:

- **En entrée**: un schéma relationnel $R(\Delta)$ avec F les $\mathcal{DF}$ associées à R.
- **En sortie**: un schèma relationnel $S = \{R_1, R_2, \ldots, R_p\}$ avec les $R_i(\Delta_i)$ en 3NF.
  - *étape 1.* Rechercher une couverture irredondante $IRR(F)$ de F
  - *étape 2.* Partitionner $IRR(F)$ en $F_1, F_2, ..., F_p$ tels que toutes les $\mathcal{DF}$ d'un même groupe aient la même partie gauche.
  - *étape 3.* Construire les $R_i(\Delta_i)$ avec Δ_i constitué de l'union des attributs de F_i.
  - *étape 4.* Si aucune des clés de la relation initiale n'est contenue dans une des relations obtenues il est nécessaire de rajouter une relation constituée d'une des clés candidates.

Intuitivement, les relations obtenues sont en 3NF de par les caractères élémentaires et irredondants des $\mathcal{DF}$ dont elles sont issues. Le résultat obtenu n'est pas toujours optimal. On note par ailleurs que par construction $\bigcup F_i = IRR(F)$ et donc qu'il n'y a pas de perte des dépendances fonctionnelles.

Exemple. Soit la relation $R(A, B, C, D, E)$ soumise aux dépendances fonctionnelles élémentaires suivantes:

$$\begin{aligned} A &\rightarrow B \\ A &\rightarrow C \\ C, D &\rightarrow E \\ B &\rightarrow D \end{aligned}$$

La seule clé candidate est (A). L'ensemble de $\mathcal{DF}$ fourni est une couverture irredondante de l'ensemble de toutes les $\mathcal{DF}$ de R. L'algorithme de synthèse fournit donc les trois relations

R1(A,B,C)
R2(C,D,E)
R3(B,D)

9.7 La Troisième Forme Normale de Boyce-Codd.

Les schémas en 3NF bien qu'épurés des situations les plus courantes d'anomalies n'en sont pas toujours exempts. Il convient donc de pousser plus loin l'analyse pour des cas relativement rares.

Il est en effet encore possible que des dépendances existent d'un attribut non clé vers un attribut clé. Ce schéma engendre à nouveau des problèmes de redondances. La forme normale de Boyce-Codd tente de prendre en compte ce critère supplémentaire pour assurer un schéma valide.

Nous dirons qu'une relation est en **Troisième Forme Normale de Boyce-Codd - BCNF** à la condition qu'elle soit en 3NF, et que les seules $\mathcal{DF}$ élémentaires sont celles dans lesquelles une clé candidate détermine un attribut.

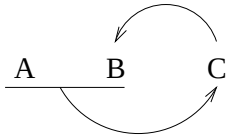
Contrairement à la 3NF il ne doit plus y avoir de dépendances d'un attribut vers une clé.

Théorèmes.

1. Toute relation admet une décomposition en BCNF mais ceci se fait parfois au prix de perte de dépendances fonctionnelles.
2. Toute relation sous forme BCNF est aussi sous 3NF (mais la réciproque n'est pas vraie).

Exemple abstrait Exemple de relation 3NF qui n'est pas BCNF illustrée par une extension avec redondances.

A,B → C
C → B



A	B	C
a1	b1	c1
a2	b1	c1

Exemple concret. Considérons la relation suivante: *Cours(M:Matière, C:classe, P:professeur)* complétée par les règles de gestion suivantes: *un professeur n'enseigne qu'une seule matière, une classe n'a qu'un seul enseignant par matière* desquelles on déduit les $\mathcal{DF}$ suivantes: $M, C \rightarrow P$; $P \rightarrow M$. Cette relation est en 3NF, néanmoins il est impossible d'enregistrer un professeur sans classe affectée, et la disparition d'une classe peut entraîner la disparition de professeur. Clairement, ceci est du au fait qu'une $\mathcal{DF}$ n'ait pas comme origine une clé de la relation.

1NF	Tous les attributs sont atomiques .
2NF	1NF + tout attribut non clé dépend entièrement de chaque clé.
3NF	2NF + pas de dépendance entre attributs non clé.
BCNF	Chaque partie gauche d'une dépendance est une clé.

FIG. 9.1 – *Formes normales*

Pour sortir de ce piège, on va là encore utiliser le théorème de décomposition. on obtient alors: *Compétence(P,M) et Affectation(P, C)*. Notons que l'on perd la $\mathcal{DF}$: $M, C \rightarrow P$. Ceci peut avoir comme conséquence, si l'on y prend pas garde, de venir en contradiction avec la règle de gestion dont la $\mathcal{DF}$ perdue est l'expression.

En effet rien ne s'oppose plus (d'un point de vue restreint à la relation *Affectation*: clé - règle de gestion associée) à l'ajout d'un couple $Affectation(Durand, 305)$ à la relation, même si, par ailleurs, il existe déjà dans la base les tuples $Compétence(Durand, Gestion)$, $Compétence(Dupond, Gestion)$ et $Affectation(Dupond, 305)$. Alors que, considerant la relation *Cours*, le statut de clé de M, C bloquerait cette tentative d'ajout. La mise en BCNF avec perte de $\mathcal{DF}$ se paye donc parfois au prix d'une complexification des mécanismes de contrôle des règles d'intégrité.

9.7.1 Algorithme de Décomposition.

C'est l'algorithme suggéré par le théorème de décomposition. En entrée et en sortie, on a les mêmes spécifications que pour l'algorithme de synthèse. On notera par ailleurs que c'est aussi un algorithme de décomposition en 3NF.

– **Initialisation** $Result = \{R\}$

- **Itération**. Si il existe un schéma R_i dans $Result$ qui n'est pas BCNF alors
  - Chercher une dépendance non triviale $X \to Y$ sur R_i telle que $X \to R_i$ n'est pas dans $F+$ (X non clé candidate).
  - Appliquer le théorème de décomposition à partir de cette $\mathcal{DF}$: $Result = (Result - R_i) \cup (R_i - Y) \cup (XY)$
- **élimination des relations imbriquées**. S'il existe $R_i(\Delta_i)$ et $R_j(\Delta_j)$ dans $Result$ avec $\Delta_i \subset \Delta_j$ alors supprimer R_i de $Result$.

Il est important de noter que la décomposition finale obtenue dépend de l'ordre dans lequel les dépendances seront traitées. Il existe en général plusieurs décompositions BCNF d'une relation.

Ce dernier algorithme a deux inconvénients majeurs. Le premier est que le résultat obtenu dépend de l'ordre dans lequel on a opéré les décompositions successives, le second est qu'il ne préserve pas toujours les dépendances fonctionnelles. Par contre, nous verrons que son utilisation peut être étendu à d'autres types de dépendances.

Exemple. Reprenons la relation $R(A, B, C, D, E)$ précédemment traitée. Ses dépendances élémentaires sont :

$$\begin{aligned} A &\to B \\ A &\to C \\ C, D &\to E \\ B &\to D \end{aligned}$$

La seule clé est (A) et l'algorithme de décomposition fournit deux possibilités selon que l'on traite $B \to D$ ou $C, D \to E$ en premier.

1. On décompose $B \to D$ en premier.
   R1(B,D)
   R2(A,B,C,E)
   Si on projette $F+$ sur $R2$ on constate que (A) est clé de $R2$ et que $R2$ n'est pas BCNF à cause de $B, C \to E$. On décompose donc R2 en sortant cette dépendance.
   R21(B,C,E)
   R22(A,B,C)
   Cette fois-ci les deux relations obtenues sont BCNF. L'algorithme s'arrête.
   On constate que la décomposition obtenue préserve les $\mathcal{DF}$ initiales.
2. On décompose $C, D \to E$ en premier.
   R1(C,D,E)
   R2(A,B,C,D)
   R1 est BCNF mais si on projette $F+$ sur $R2$ on constate que R2 n'est pas BCNF à cause de $B \to D$. On décompose donc R2 en sortant cette dépendance.
   R21(B,D)
   R22(A,B,C)
   Cette fois-ci les deux relations obtenues sont BCNF. L'algorithme s'arrête.
   On constate que la décomposition obtenue préserve les $\mathcal{DF}$ initiales et que cette solution est identique à celle fournie par l'algorithme de synthèse.

Pour finir nous dirons qu'une décomposition BCNF est préférable à la 3NF si elle conserve les $\mathcal{DF}$. Dans le cas contraire il est préférable de conserver la 3NF. Dans l'exemple précédent, on conserverait donc la décomposition 3NF au lieu des deux BCNF.

9.8 Méthodologie

Il est clair qu'une décomposition BCNF est préférable à une 3NF mais uniquement si elle conserve les $\mathcal{DF}$. Dans le cas contraire il est préférable de rester avec une 3NF.

L'exemple précédent suggère une bonne méthodologie pour obtenir un bon schéma.

- Appliquer l'algorithme de synthèse (qui assure d'être 3NF et de préserver les $\mathcal{DF}$).
- Pour toute relation obtenue qui n'est pas BCNF appliquer à cette relation l'algorithme de décomposition. Si celui-ci fournit une décomposition qui préserve les dépendances, on conserve cette décomposition, sinon on garde la relation 3NF initiale.

On notera cependant que, bien que très pratique, cette méthodologie ne fournit pas en général la décomposition la plus optimale en nombre de relations obtenues.

9.9 Les Dépendances Multivaluées.

Comme nous allons le mettre en évidence sur un exemple, les redondances et anomalies de mises à jour n'ont pas toujours pour origine une $\mathcal{DF}$ (un groupe d'attribut "fonction" [3] d'un autre) bien que résultant d'une forme de dépendance inter-attributs: *les dépendances multivaluées* - $\mathcal{DM}$.

Exemple. Soit la relation *Renseignements(nom, diplôme, enfant)*, attendu qu'une personne peut avoir plusieurs diplômes et plusieurs enfants et qu'un enfant a généralement deux parents, nous ne pouvons mettre à jour ici aucune $\mathcal{DF}$; la clé de cette relation est l'ensemble de ses attributs et elle est en BCNF. Or dès qu'une personne a plus d'un diplôme et plus d'un enfant, des redondances apparaîtront au sein de la relation. Ainsi pour rendre compte que Ma Dalton a quatre fils Joe, Jack, William et Awerell et un Master de Pédagogie et de Droit Pénal, cela nécessiterait 8 tuples:

Pédagogie	Ma Dalton	Joe
Pénal	Ma Dalton	Joe
Pédagogie	Ma Dalton	Jack
Pénal	Ma Dalton	Jack
...	Ma Dalton	....

Inutile d'insister lourdement sur les pénalisations induites par ces redondances lors des opérations de mises à jour : pour toute obtention, radiation ou modification de diplômes, l'opération sera répétée autant de fois qu'il y a d'enfants.

Intuitivement, on sent que ce n'est plus le Diplôme mais un ensemble de diplômes qui dépend du *Nom* (idem pour les enfants). D'où l'extension de la notion de $\mathcal{DF}$ à la notion de dépendances **multi**valuées.

Formalisation-Définitions. Nous donnerons ici plusieurs définitions plus ou moins formelles, toutes équivalentes afin de bien cerner la notion de $\mathcal{DM}$.

Prenons comme référence une relation $R(\Delta)$ et X, Y, Z une partition de Δ. Nous appellerons $Im_Y(x)$ où x est une valeur de l'attribut X, l'ensemble

$$\{y \in dom(Y) / \exists z \in Dom(Z) \quad R(x, y, z)\}$$

(i.e l'ensemble des valeurs de l'attribut Y associées à la valeur x de l'attribut X au travers de la relation R).

On dira qu'il existe une $\mathcal{DM}$ entre X et Y, que l'on notera $X \twoheadrightarrow Y$ à l'une des conditions suivantes:

- toute valeur de X détermine un ensemble unique de valeurs de Y indépendamment des autres attribuuts.
- $Im_Y(x)$ est indépendant de Z pour tout x, et Y et Z sont indépendants.

3. au sens mathématique du terme: à tout valeur x pour laquelle la fonction f est définie, on peut associer son unique image $f(x)$.

- pour toute valeur x appartenant au $dom(X), Im_Y(x) = Im_Y(x,z)$ pour tout $z \in Im_Z(x)$.
- $R(x,y_1,z_1) \wedge R(x,y_2,z_2) \rightarrow R(x,y_1,z_2) \wedge R(x,y_2,z_1)$.

Comme pour les $\mathcal{DF}$, nous pouvons formaliser les propriétés naturelles des $\mathcal{DM}$:

- **P1.** Les $\mathcal{DF}$ sont un cas particulier de $\mathcal{DM}$: $X \rightarrow Y \rightarrow X \twoheadrightarrow Y$
- **P2.** Les $\mathcal{DM}$ triviales: pour tout $Y \subseteq X$ on a $X \twoheadrightarrow Y$;
- **P3. La complémentarité:** si $X \twoheadrightarrow Y$ alors $X \twoheadrightarrow Z$ compte tenu de "l'indépendance" entre Y et Z (ou encore de la symetrie des définitions relativement à Y et Z.
- **P4.** $X \twoheadrightarrow \Delta - X$ et donc par complémentarité $X \twoheadrightarrow \emptyset$

Comme pour les $\mathcal{DF}$, toute $\mathcal{DM}$ peut donner lieu à une décomposition sans perte de la relation, ce qui peut se traduire par le théorème suivant:

Théorème de décomposition.

Soit $R(\Delta)$ une relation et $X, YetZ$ une partition de Δ avec $X \twoheadrightarrow Y$ alors:

$$R(\Delta) = R(\Delta_1) \bowtie_X R(\Delta_2)$$

où $\Delta_1 = X \cup YetDelta_2 = X \cup Z$ et $\bowtie_X$ désigne l'équi-jointure sur X.

Notons que l'application du théorème de décomposition aux $\mathcal{DM}$ relevant de la propriété **P4**, ne donnerait que la relation elle-même.

Les différents degré de normalité mesurant en quelques sorte les dépendances entre attributs, il convient d'en rajouter un permettant de spécifier que les dépendances fonctionnelles et multivaluées sont déterminées à partir des clés.

9.9.1 La Quatrième Forme Normale

On dira qu'une relation est en *quatrième forme normale - 4NF* lorsque toute dépendance multivaluée (donc y compris les $\mathcal{DF}$) non triviales ont comme déterminant une clé de la relation (ou un groupe d'attributs contenant une clé). Cette définition implique que toute relation en 4NF est en BCNF.

Exemple Imaginons que soient recensées les compositions des équipes lors de rencontres sportives au travers de la relation suivante:

Composition(I:Identifcation_Match,Ma: Membre_EquipeA,Mb: Membre_EquipeB), la clé de cette relation est I, Ma, Mb, on ne peut en effet mettre en évidence aucune $\mathcal{DF}$: *pour un match donné, les équipes A et B sont composés de plusieurs membres; les joueurs ont généralement participés à plusieurs matchs.* Par contre, pour un match, la composition des équipes est unique, et il n'y a aucun lien formel entre les deux; on a donc $I \twoheadrightarrow Ma$ et aussi, inévitablement $I \twoheadrightarrow Mb$. La relation *Composition* n'est donc pas en 4NF bien qu'étant en BCNF.

L'algorithme basé sur la décomposition des relations peut encore s'appliquer pour obtenir un schéma relationnel en 4NF. Cela donnerait pour l'exemple présenté les relations suivantes *CompositionA(I, Ma) et CompositionB(I, Mb).*

Exemple On considère le schéma relationnel $R =< \Delta, F >$ où les relations sont construites sur les attributs **S**éminaire, **T**ype, **I**nstructeur, **J**our, S**A**lle, **E**tudiant, **N**ote et **L**ivre. La relation R exprime les emploi du temps pour une série de séminaires. Nous supposerons que R est muni des dépendances suivantes:

. $d_1 : S \rightarrow T$
. $d_2 : T, J \rightarrow S$
. $d_3 : E, J \rightarrow S$
. $d_4 : A, J \rightarrow S$

. $d_5 : E, S \rightarrow N$
. $d_6 : S, J \rightarrow I$
. $d_7 : T \twoheadrightarrow L$
. $d_8 : S \twoheadrightarrow J, A$

L'interprétation des dépendances fonctionnelles est aisée. La dépendance multivaluée d_7 exprime qu'à un type de séminaire correspond un ensemble d'ouvrages de références; de même d_8 exprime que la connaissance d'un séminaire entraîne la connaissance des couples de valeur *(jour, salle)* où ce séminaire est programmé et ceci indépendamment des autres attributs.

On pourra vérifier que la clé de cette relation est A, J, E, L. Ce schéma relationnel n'est donc pas en 4NF puisque $T \twoheadrightarrow L$.
Par conséquent, on peut décomposer le schéma initial en

- $R_1 =< \{T, L\}, \{T \twoheadrightarrow L\} >$ en 4NF
- $< \{S, T, I, J, A, E, N\}, \{d_1, d_2, d_3, d_3, d_4, d_5, d_6, d_8\} >$ qui n'est pas en 4NF. En effet, la clé de cette relation est A, J, E et on a la $\mathcal{DM}$: $S \twoheadrightarrow J, A$.

On procède donc à une nouvelle décomposition sur cette dernière relation en

- $R_2 =< \{S, J, A\}, \{d_4, d_8\} >$ en 4NF
- $< \{S, T, E, N, I\}, \{d_1, d_5\}$ qui n'est pas en 2NF. En effet, la clé de cette relation est E, S, I et on a la $\mathcal{DF}$ $S \rightarrow T$. Notons ici, les dépendances d'ors et déjà perdues.

On décompose donc cette dernière relation pour obtenir:

- $R_3 =< \{S, T\}, \{d_1\} >$ en 4NF
- $< \{S, E, N, I\}, \{d_5\} >$ qui n'est pas en 2NF. En effet la clé en est E, S, I et on a la $\mathcal{DF}$ $E, S \rightarrow N$.

Une ultime décomposition donnera donc

- $R_4 =< \{E, S, N\}, \{d_5\} >$ en 4NF
- $R_5 =< \{E, S, I\}, \emptyset >$ en 4NF

Finalement, on obtient une décomposition en cinq relations, ce schéma relationnel est satisfaisant dans la mesure où chacune des relation à une signification claire.

Exercice. Considérons une variante de la relation précédente *Compositionbis(I,Ma,Mb, N:numero_de_place_joueur)*, cette relation indique que lors du match i_m, la place numéro n étant occupé par le joueur j_a pour l'équipe A et par le joueur j_b pour l'équipe B. Sous quelle forme normale est cette relation?

D'autres formes de dépendances sont très proches des $\mathcal{DM}$ traduisant des règles de gestion sémantiquement très voisines de celles pouvant être exprimées en terme de $\mathcal{DM}$. Néanmoins, elles ne pourront induire de décomposition sur la relation elle-même mais sur une projection de celle ci.

9.10 Les Dépendances Hiéarchiques.

Considérons la relation *Rencontres_par_équipes*(I: Identification_Match, Ma : Membre_EquipeA, Mb: Membre_EquipeB, R: Résultat) pour laquelle nous supposerons qu'il s'agit de rencontre de Judo par équipes, au cours desquelles tout membre d'une équipe rencontre chaque membre de l'équipe adverse, ce sont les résultats de ces combats individuels que représente l'attribut R.

On observe ici une $\mathcal{DF}$ relative à la signification de l'attribut R telle que nous venons de le préciser: $I, Ma, Mb \rightarrow R$. Comme précédemment, pour un match donné, les équipes A et B ont toujours une composition unique mais on ne peut parler ici de $\mathcal{DM}$.

En effet, pour un match donné, on ne peut séparer les couples de combattants puisqu'y sont attachés les résultats des combats: les ensembles d'attributs Ma et Mb, R ne sont pas indépendants.

Nous sommes en présence d'un nouveau type de contrainte d'intégrité qui possède la propriété d'être multivaluée sur **une projection** de la relation. On appelle ce type de contrainte d'intégrité une *dépendance hiérarchique*.

Formalisation. Soit une relation R pour laquelle on a pu partitionner l'ensemble Δ de ses attributs en $X, Y_1, Y_2, ..., Y_n, W$, pour laquelle on a la propriété suivante:

$$\forall X \in Dom(x) \quad Im_{Y_1,Y_2,...,Y_n}(x) = Im_{Y_1}(x) \times Im_{Y_2}(x) \times ... \times Im_{Y_n}(x)$$

(où le symbole $\times$ désigne le produit cartésien des ensembles)
alors on dira qu'il existe une *dépendances hiérarchique* entre X et $Y_1, Y_2, ..., Y_n$.

C'est bien le cas pour la relation *Rencontres_par_équipes* entre I et Ma, Mb, en effet:

pour tout rencontre x, $Im_{Ma,Mb}(x)$ est égal à l'ensemble des combats de la rencontre; et $Im_{Ma}(x) \times Im_{Mb}(x)$ est le produit cartésien des compositions des deux équipes opposées; et ces deux ensembles sont bien identiques.

Mais ces concepts sont encore insuffisants, en effet il existe des décompositions de relations qui ne sont pas conséquence de $\mathcal{DF}$ ou de $\mathcal{DM}$; nous verrons que ces décompositions ne sont plus binaires (produit deux relations) comme dans celles issues de $\mathcal{DF}$ ou $\mathcal{DM}$ mais d'ordre plus important. Toutes les dépendances desquelles résulte une possibilité de décomposition, sont unifiées par le concept de **dépendances produit**.

9.10.1 Dépendances Produit.

Etant donné une relation $R(\Delta)$ et un ensemble de parties de Δ: $X_1, X_2, ..., X_n$ (non nécessairement disjointes) telles que $\bigcup_{i=1}^{n} X_i = \Delta$. $X_1, X_2, ..., X_n$ est une dépendance produit dans R si et seulement si

$$R(\Delta) = \bowtie_{i=1}^{n} R(X_i)$$

Dans le cas de décompositions en trois relations, on parle de façon plus spécifique de *dépendances mutuelles*.

Il devient naturellement plus délicat d'exhiber un exemple "concret" de relation présentant une dépendance mutuelle ou pire encore une dépendance produit d'ordre supérieur à trois. L'énoncé de la contrainte d'intégrité dont elle serait l'expression doit faire intervenir au moins trois attributs liés par un savant lacis de dépendance. Tentons néanmoins l'aventure:

Exemple Considérons un groupe de magasins franchisés (disposant d'une certaine autonomie mais devant respecter certaines conventions). Supposons que le groupe a négocié auprès d'un certain nombre de marque l'accord suivant:

Si un franchisé propose un certain type d'article, que ce type d'article figure au catalogue d'une marque déjà représentée en magasin en alors il devra proposer à la vente les produis de ce type de cette marque. Les franchisés ne travaillent qu'avec les marques ayant passé ce type d'accord auprès du groupe.

Par exemple, si je vends des polos Laposte et des parfums Yves S'Implorant, je suis par contrat obligé de vendre les parfums Laposte

Soit la relation *Produit(F:identification_franchisé, T:type_produit, M:marque_produit)* stipulant qu'un franchisé f commercialise le produit de type t de la marque m.

Analyses des dépendances inter-attributs: il n'y a pas de $\mathcal{DF}$ (non triviales), la clé de la relation est F, T, M. Il n'y a pas non plus de $\mathcal{DM}$(non triviales), en effet si on avait

- $F \twoheadrightarrow T$ et par conséquent aussi $F \twoheadrightarrow M$, il n'y aurait plus de lien les types de produit et les marques. Ou encore, qu'un magasin distribuant un type de produit t et diffusant la marque m, dispose nécessairement d'article de type t de la marque m.

- $T \twoheadrightarrow F$ et par conséquent aussi $T \twoheadrightarrow M$. Cela signifierait qu'un franchisé commercialisant un type de produit t est **obligé** de travailler avec toutes les marques qui disposent de ce type de produit. Ceci n'est dans les clauses du contrat
- $M \twoheadrightarrow F$ et par conséquent aussi $M \twoheadrightarrow T$. Cela signifierait qu'un franchisé diffusant une marque est obligé de vendre tous les types de produits de cette marque. Ce n'est pas dans les clauses du contrat.

Par contre, on peut se convaincre que

$$Produit(F,T,M) = [R(F,T) \bowtie_F R(F,M)] \bowtie_{T,M} R(T,M)$$

En effet, dans la relation $(R(F,T) \bowtie_F R(F,M))$, sont gênants les triplets (f,t,m) tels qu'il n'existe pas de produit de type t de marque m, or ces triplets disparaîtront lors de l'équi-jointure avec la relation $R(T,M)$ qui précise par marque, les types de produit existant. On est donc ici en présence de la dépendance mutuelle F,T,M.

Ceci conduit à introduire une cinquième forme normale

La Cinquième Forme Normale. Une relation sera en *cinquième forme normale - 5NF* à la condition que toute dépendance produit soit induite par une clé candidate.

Notons qu'une relation en 5NF est nécessairement en 4NF. La notion de dépendance produit recouvrant celle de $\mathcal{DM}$.

9.11 Conclusion sur la Normalisation.

L'étude théorique menée sur les différentes formes de dépendances inter-attributs et les degrés de normalité d'une relation n'a pas pour ambition de substituer une approche purement formelle à une approche plus intuitive en phase avec l'interprétation naturelle des attributs manipulés. Sa principale vocation est d'identifier clairement les causes, les effets et les remèdes (et leurs effets secondaires) aux différentes formes de redondances d'information. Les méthodes algorithmiques validées par l'étude théorique n'ont pas d'intérêts pratiques dans la majorité des cas. Lorsque le modèle conceptuel de données a été élaboré soigneusement, le passage au modèle logique par les méthodes traditionnelles donne un résultat généralement satisfaisant. Cependant, une sensibilisation à des formes plus subtiles de dépendances, et les possibilités de recours à des méthodes automatiques peuvent permettre la prise en compte de cas plus exceptionnels.

De plus, d'autres considérations parfois prioritaires, peuvent amener l'administrateur de bases de données à une *dénormalisation*.

9.11.1 Dénormalisation.

On appelle *dénormalisation* l'introduction volontaire de redondance répondant principalement à des considérations d'efficacité.

Exemple Soit le schéma relationnel suivant:

. Article(<u>numéro_article</u>,désignatin,prix)
. Commande(<u>numéro_commande</u>, date_commande, numéro_client)
. Ligne_commande(<u>numéro_commande</u>,<u>n</u>uméro_article, quantité)
. Client(<u>numéro_client</u>, nom, adresse)

On vérifiera que toutes les relations de ce schéma sont en BCNF.

Supposons que le marketing (pour l'émission de mailing) soumet très fréquemment à la base des requêtes de la forme:

R69 *Quels sont les noms et adresses des clients ayant passés commande de tel(s) produit(s) ?*

```
SELECT nom, adresse
```

```
FROM article , client , commande , ligne_commande
WHERE  désignation = ??
   AND ligne_commande.numéro_article = produit.numéro_article
   AND ligne_commande.numéro_commande = commande.numéro_commande
   AND commande.numéro_client = client.numéro_client .
```

Chaque extraction nécessite donc trois jointures ce qui du point de vue des temps de réponse peut être jugé trop pénalisant voire incompatible avec les impératifs du service. La reprise de l'attribut *Désignation* au niveau de la relation *Ligne_commande* permettrait d'en éviter une. Il va de soi, qu'en contrepartie d'un gain de temps, cela réclamerait plus d'espace de stockage, et alourdirait les transactions de mise à jour des désignations des produits (ce qui peut être acceptable si celles-çi sont peu fréquentes).

De même, si pour des besoins de gestion financière, on désire avoir la possibilité d'obtenir l'évaluation des commandes. L'extraction de cette information nécessitera systématiquement une jointure; ce qui est le cas, par exemple, pour la requête SQL suivante:

R70 *Evaluation des commandes*

```
SELECT SUM(prix*quantité), commande.numéro_commande
FROM commande, article
WHERE date_commande = ??
  AND commande.numéro_article = article.numéro_article
GROUP BY commande.numéro_commande
```

Or cette information pourrait être enregistrée directement au niveau de la relation *commande*, cet attribut étant mis à jour lors de la transaction *"saisie de commande"*.

Parc(numero_série, type, caractéristiques_techniques, lieu_d'installation); $Enseignant(Nom, Bureau, Batiment, Disc$
et les contraintes d'intégrité suivantes: *Poste(V:ville, D:département, C:code_postal)*, de plus tout préposé de ce prestigieux service public pourra vous dire avec sa coutumière amabilité que: $V, D \rightarrow C$ et $C \rightarrow D$

Chapitre 10

Les bases de données de type réseau

Le modèle réseau a été proposé par le groupe CODASYL en 1971 à partir des travaux de BACHMAN et amélioré en 1978.

Le modèle réseau se présente sous forme d'un réseau d'entités reliées entre elles à l'aide de pointeurs logiques. L'élément fondamental dans le modèle réseau est le lien. Un lien est défini entre deux types d'enregistrement dans une direction donnée. Il offre le moyen d'accéder à partir d'un enregistrement de type A aux enregistrements de type B qui lui sont reliés (Fig 10.1.

La représentation est la suivante :

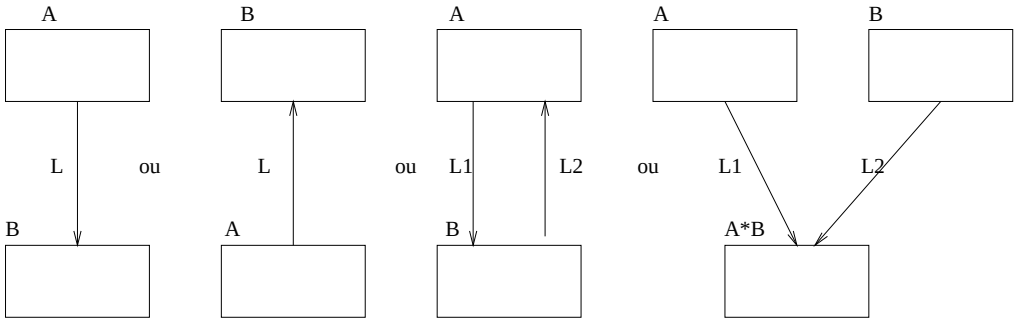


FIG. 10.1 – *Représentation des liens réseau*

Le lien CODASYL appelé SET pourra être construit si l'association est de type 1-1, 1-n ou n-1. Les liens de type n-m entre deux enregistrements de type A et B sont possibles en construisant un enregistrement de type AxB dont les éléments sont des couples d'éléments des enregistrements de type A et B.

La structure fondamentale du modèle réseau doit vérifier que le lien entre les types d'enregistrements est 1-n et qu'il associe un enregistrement propriétaire (OWNER) à un enregistrement membre (MEMBER). Cette structure spécifiée par CODASYL s'appelle un COSET.

Le diagramme de BACHMAN permet de représenter un COSET (Fig 10.2).

Le mécanisme d'accès qui permet de passer de l'enregistrement de type A aux enregistrements de type B est celui d'une liste circulaire où la tête de liste est constituée par l'article a de l'enregistrement A (OWNER) et les éléments de la liste sont constitués par les articles b1, b2,..bn de l'enregistrement de type B (Fig 10.3).

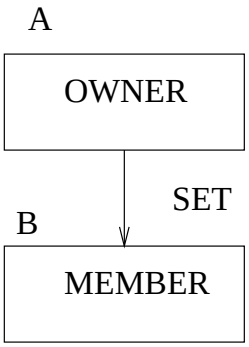


FIG. 10.2 – *Représentation d'un lien réseau*

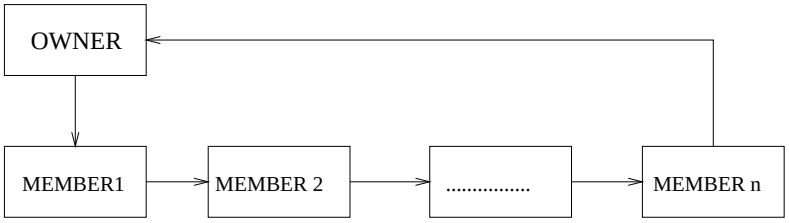


FIG. 10.3 – *Instanciation d'un lien*

10.1 Notions de base

- **Les items**
  Un item ou atome est la plus petite unité d'information portant un nom. Il correspond au champ des bases de données hiérarchiques.
- **Les articles**
  Un article ou enregistrement est une collection d'items regroupés constituant l'unité d'échange entre la base de données et les programmes qui y accèdent. Le concept d'article est analogue à celui d'enregistrement des fichiers et à celui de segment des bases de données hiérarchiques.
- **Les associations d'articles**
  Un article type, dit article père (ou propriétaire), peut être associé à plusieurs autres articles type dits articles fils (ou membre). Un article type peut être le fils de plusieurs articles type père. Ces associations de niveau purement hiérarchique utilisées à plusieurs niveaux peuvent former aussi bien des arbres que des réseaux.

  Attention : un type d'article ne peut être à la fois propriétaire et membre d'une association. Par contre les associations binaires et n-aires sont autorisées.
- **Les liens**
  Les liens autorisés par les SGBD réseaux sont de type :

  1 - 1 : à une occurrence d'un article père est associée une occurrence de son article fils
  1 - n : à une occurrence d'un article père sont associées plusieurs occurrences de son article fils
  n - 1 : à plusieurs occurrences d'un article père est associée une occurrence de son article fils.

- **Les schémas réseaux**
  Un schéma réseau est constitué d'un ensemble d'articles reliés par des liens de type père fils et est représenté par un graphe convexe. Les arcs sont orientés du propriétaire vers le

membre. L'arc porte le nom de l'association et chaque sommet porte le nom de l'article associé.

10.2 Le SGBD de type CODASYL IDS-II (Integrated Data Store)

10.2.1 La définition des données

Elle se fait par la description d'un schéma de la base. Le langage de description est proche du langage `COBOL`. Il permet la définition des articles, des items, des associations et la façon de les implanter.

Il y a plusieurs possibilités d'insertion d'un article dont le type est membre d'un ensemble.

- FIRST : l'insertion s'effectue en début des occurrences
- LAST : l'insertion s'effectue en fin des occurrences
- PRIOR : l'insertion s'effectue avant la dernière occurrence accédée
- NEXT : l'insertion s'effectue après la dernière occurrence accédée
- SORTED : l'insertion s'effectue suivant l'ordre croissant ou décroissant de la clé.

Exemple : Soit (7) l'occurrence à insérer.

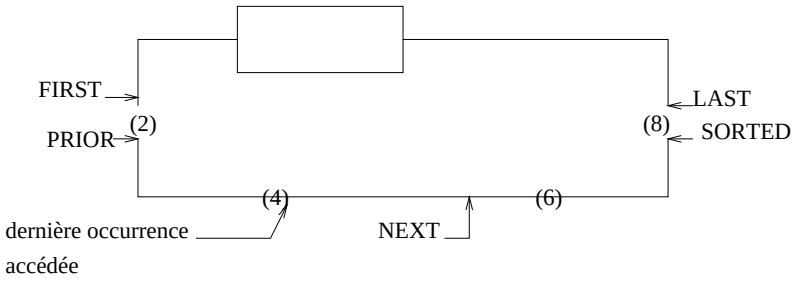


FIG. 10.4 – *exemple*

10.2.2 Le mode de rattachement des articles

Le mode d'appartenance d'un article membre est défini dans le schéma et servira lors de l'insertion.

- MANUAL L'opération de stockage dans la base doit être effectuée explicitement par l'utilisateur.
- AUTOMATIC L'opération de stockage dans la base s'effectue automatiquement par le système.
- OPTIONAL Une occurrence peut être créée dans la base sans être rattachée obligatoirement à une occurrence d'un propriétaire.
- MANDATORY Une occurrence ne peut être créée que si elle est rattachée à une occurrence d'un propriétaire.

L'appartenance d'un article membre doit toujours être spécifiée dans l'une des quatre combinaisons suivantes :
OPTIONAL_MANUAL, OPTIONAL_AUTOMATIC, MANDATORY_MANUAL, MANDATORY_AUTOMATIC.

10.2.3 Le placement des articles

Le mode de placement des articles est défini dans le schéma pour chaque type d'article selon trois procédés. Le SGBD doit utiliser l'un de ces trois procédés pour localiser ou stocker un article.

- DIRECT : une clé attachée à l'article est fournie par l'utilisateur
- CALC USING : le système calcule l'adresse de l'article à l'aide d'une procédure de hachage.
- VIA nom_association SET le système calcule l'adresse de l'article en fonction du type de placement du propriétaire et du membre de l'association.

10.2.4 Le langage de manipulation de données

Le langage de manipulation d'`IDS II` est intégré dans un langage hôte procédural `COBOL` ou `PL1`.

10.2.5 Recherche d'articles

- FIND : accède à la localisation d'une occurrence d'un article dans la base de données. La recherche s'effectue soit à partir de la clé, soit dans une association, soit à partir d'un propriétaire.

10.2.6 Echanges d'articles

- GET : permet le rangement de l'article, recherché par le FIND, dans une zone de travail de l'utilisateur.
- STORE : permet d'écrire une occurrence d'article dans la base de données.

10.2.7 La mise à jour

- ERASE : supprime l'occurrence de l'article dans la base de données. Si l'occurrence de l'article est propriétaire d'occurrences d'associations, tous ses descendants sont supprimés si la clause ALL est précisée.
- MODIFY : modifie les données de l'occurrence de l'article rangée dans la zone de travail avant réécriture dans la base de données.
- CONNECT : permet le rattachement d'une occurrence d'article membre à une occurrence d'article propriétaire.
- DISCONNECT : supprime le rattachement d'une occurrence d'article membre de celle d'un propriétaire.

10.2.8 Ouverture et fermeture

- READY : ouverture des fichiers de travail sur lesquels les traitements vont s'effectuer.
- FINISH : fermeture des fichiers de travail.

10.3 Utilisation

Une base de données de type réseau `CODASYL` peut être implémentée et manipulée à partir d'une base "CONVENTION" mise en place par les règles de gestion suivantes. Une société possède plusieurs unités de production situées sur la zone de compétence d'une Agence de l'Eau. Chaque unité signe avec l'Agence différents contrat pour lutter contre la pollution. Des mesures de surveillance de la pollution seront effectuées sur les rejets des unités pour vérifier si les contrats sont respectés. Ces mesures sont effectuées par des organismes mandatés par l'Agence.

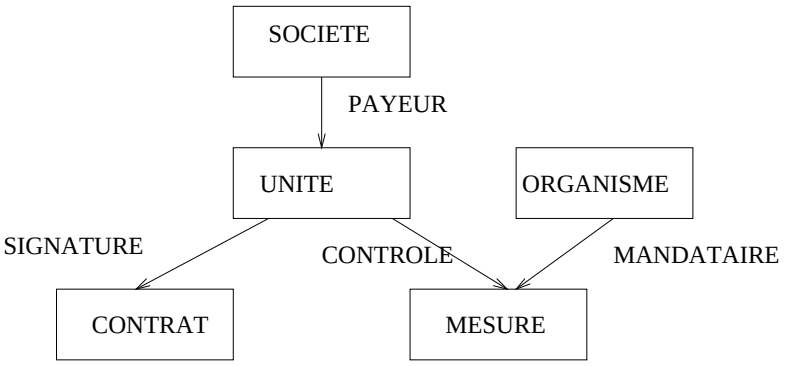


FIG. 10.5 – *Représentation du graphe de la base "CONVENTION"*

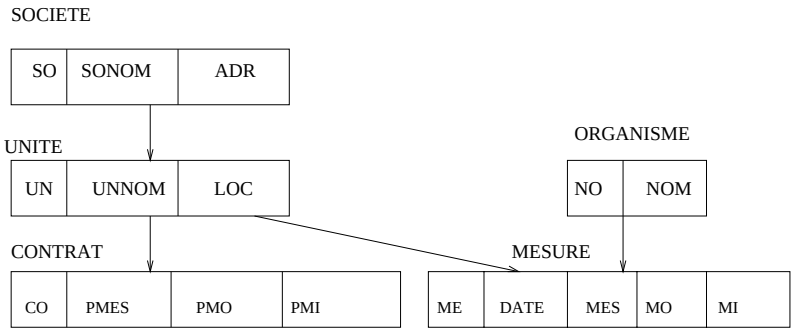


FIG. 10.6 – *Représentation partielle des articles*

10.3.1 Définition simplifiée du schéma

```
SCHEMA NAME IS CONVENTION
AREA NAME   IS  F-CONTRAT   /*SOCIETE+UNITE+CONTRAT */
AREA NAME   IS  F-MESURE     /*ORGANISME + MESURE */


RECORD NAME IS SOCIETE
LOCATION  MODE  IS  CALC  USING  SO
DUPLICATES  ARE  NOT  ALLOWED  WITHIN  F-CONTRAT
     02  SO   TYPE   IS  CHARACTER      6
     02  SONOM     TYPE  IS       CHARACTER      40
     02  ADR       TYPE  IS       CHARACTER 40


     RECORD NAME IS UNITE
     LOCATION  MODE  IS  CALC  USING  UN
     DUPLICATES  ARE  NOT  ALLOWED  WITHIN  F-CONTRAT
     02  UN   TYPE IS    CHARACTER        6
     02  UNNOM     TYPE  IS  CHARACTER  40
     02  LOC       TYPE  IS  CHARACTER  36


     RECORD  NAME  IS  CONTRAT
     LOCATION  MODE  IS  CALC  USING  CO
     DUPLICATES  ARE  NOT  ALLOWED  WITHIN  F-CONTRAT
     02  CO   TYPE IS    CHARACTER      10
     02  PMES TYPE IS    SIGNED  BINARY 15
     02  PMO       TYPE  IS  SIGNED  BINARY 15
     02  PMI       TYPE  IS  SIGNED  BINARY 15


     RECORD  NAME  IS  MESURE
     LOCATION  MODE  IS  VIA  CONTROLE  WITHIN  AREA  OF  OWNER
     02  ME   TYPE IS    CHARACTER           8
     02  DATE TYPE IS    CHARACTER           8
     02  MES       TYPE  IS  SIGNED  BINARY 15
     02  MO   TYPE IS    SIGNED  BINARY 15
     02  MI   TYPE IS    SIGNED  BINARY 15


     RECORD  NAME  IS  ORGANISME
     LOCATION  MODE  IS  CALC  USING  NO
     DUPLICATES  ARE  NOT  ALLOWED  WITHIN  F-MESURE
     02  NO   TYPE IS    CHARACTER           6
     02  NOM  TYPE IS    CHARACTER      40


SET  NAME  IS  PAYEUR
     OWNER  IS  SOCIETE
     ORDER  IS  PERMANENT  INSERTION  IS  LAST
     MEMBER  IS  UNITE
     INSERTION  IS  AUTOMATIC  RETENTION  IS  MANDATORY
     SET  SELECTION  FOR  PAYEUR  IS  THRU  PAYEUR
     OWNER  IDENTIFIED  BY  CALC KEY


SET  NAME  IS  SIGNATURE
     OWNER  IS  UNITE
```

```
      ORDER  IS  PERMANENT  INSERTION  IS  NEXT
      MEMBER  IS  CONTRAT
      INSERTION  IS  AUTOMATIC  RETENTION  IS  MANDATORY
      SET  SELECTION  FOR  SIGNATURE  IS  THRU  SIGNATURE
      OWNER  IDENTIFIED  BY  CALC  KEY


SET  NAME  IS  CONTROLE
      OWNER  IS  UNITE
      ORDER  IS  PERMANENT  INSERTION  IS  SORTED  BY  DEFINED  KEYS
      MEMBER  IS  MESURE
      KEY  IS  DESCENDING  DATE
      INSERTION  IS  AUTOMATIC  RETENTION  IS  OPTIONAL
      SET  SELECTION  FOR  CONTROLE  IS  THRU  CONTROLE
      OWNER  IDENTIFIED  BY  CALC  KEY


SET  NAME  IS  MANDATAIRE
      OWNER  IS  ORGANISME
      ORDER  IS  PERMANENT  INSERTION  IS  SORTED  BY  DEFINED  KEYS
      MEMBER  IS  MESURE
      KEY  IS  ASCENDING  ME
      INSERTION  IS  AUTOMATIC  RETENTION  IS  MANDATORY
      SET  SELECTION  IS  THRU  MANDATAIRE
      OWNER IDENTIFIED  BY  CALC KEY

 END  SCHEMA
```

10.3.2 Exemples de manipulation de la base

Pour accéder à la base, il est besoin de définir dans le langage hôte un sous-schéma. Le sous-schéma, sous-ensemble du schéma, est un schéma externe vu par le programme d'application.

Sous-schéma

```
      TITLE  DIVISION.
      SUB-SCHEMA  ENGAGEMENT  WITHIN CONVENTION.
      MAPPING  DIVISION.
      STRUCTURE  DIVISION.
      REALM  SECTION.
      RD  F-CONTRAT.
      SET  SECTION.
      SD  PAYEUR.
      SD  SIGNATURE.
      RECORD  SECTION.
      01  SOCIETE.
          02 SO           PIC X(6).
      01  UNITE.
          02 UN           PIC X(6).
          02 UNNOM        PIC X(40).
          02 LOC          PIC X(36).
      01  CONTRAT.
          02 CO           PIC X(10).
          02 PMES         PIC 9(8)V99.
          02 PMO          PIC 9(8)V99.
          02 PMI          PIC 9(8)99.
      END.
```

Liste des unités dont la société est demandée en paramètre

```
    PROCEDURE DIVISION.
          READY F-CONTRAT.
          ACCEPT SO.
          FIND ANY SOCIETE.
          FIND FIRST UNITE WITHIN PAYEUR.
          PERFORM SUITE UNTIL DB-STATUS 0 GOTO FIN.

    SUITE.
          GET UNITE.
          DISPLAY UN, UNNOM, LOC
          FIND NEXT UNITE WITHIN PAYEUR.
    FIN.
          FINISH F-CONTRAT.
```

Liste des contrats des unités localisées à DOUAI

```
    PROCEDURE DIVISION.
          READY F-CONTRAT.
          FIND FIRST UNITE WITHIN F-CONTRAT.
          PERFORM SUITE1 UNTIL DB-STATUS   0
          GO TO FIN.
    SUITE1.
          GET  UNITE.
          IF  LOC    'DOUAI'  THEN
          GO  TO  SUITE 3  ELSE
          DISPLAY  UN, UNNOM
          FIND  FIRST  CONTRAT  WITHIN  SIGNATURE
          PERFORM  SUITE2  UNTIL  DB-STATUS    0
          GO  TO  SUITE3.

    SUITE2.
          GET  CONTRAT.
          DISPLAY  CO.
          FIND  NEXT  CONTRAT  WITHIN  SIGNATURE.

    SUITE3.
          FIND  NEXT  UNITE  WITHIN  F-CONTRAT.
          GO  TO  SUITE1.

    FIN.
          FINISH F-CONTRAT.
```

Suppression de tous les contrats de l'unité 2703

```
PROCEDURE DIVISION.
     READY F-CONTRAT.
     MOVE '2703' TO UN.
     FIND ANY UNITE.
     ERASE ALL UNITE.
     FINISH F-CONTRAT.
```

Modification de la localité de l'unité demandée en paramètre

```
PROCEDURE DIVISION.
   READY  F-CONTRAT.
   ACCEPT UN.
```

```
 FIND  ANY UNITE.
GET UNITE MOVE 'LILLE' TO LOC MODIFY UNITE FINISH F-CONTRAT
```

10.4 Avantages et inconvénients des bases de données de type réseau

Avantages

- Il existe une bonne adéquation entre la représentation conceptuelle des données grâce à un modèle de type entités-associations et une implantation avec un SGBD de type réseau.
- La représentation sous forme de graphe et de liens maillés est claire.
- Les performances, et en particulier, les temps moyens d'accès aux données, sont généralement très correctes.
- L'accès à la base de données peut se faire à un point d'entrée quelconque.

Inconvénients

- Seules les associations binaires peuvent être définies. Les liens de type n-m ne peuvent être définis de manière naturelle.
- Toute modification de la structure des données impose une reconfiguration de la base de données. Il est impossible de rajouter et de supprimer des items, des articles ou des associations de manière dynamique.
- Les données sont peu indépendantes des traitements. La modification du schéma de la base nécessite de modifier les programmes dont les sous-schémas sont concernés par la modification.

Chapitre 11

Le mode Client-Serveur

Pour de très nombreuses applications, travailler seul sur une seule et même machine n'est pas satisfaisant. C'est notamment le cas pour la mise à jour et l'interrogation de bases de données dans lesquelles plusieurs personnes doivent simultanément pouvoir accéder aux données. Par exemple, les différentes agences d'une banque doivent pouvoir accéder à la base contenant le solde des comptes bancaires, les différents magasins d'une société de vente de vêtements doivent pouvoir accéder simultanément aux quantités en stock. Il est bien évident que dans ces exemples il n'y a qu'une seule et même base de données mais plusieurs personnes qui y accèdent à travers des logiciels qui tournent sur des machines différentes. Dans un tel cas, on parle d'architecture Client-Serveur. Le Serveur est un serveur de ressources communes et les logiciels qui y accèdent sont appelés Clients de ce serveur. D'une manière générale, en mode Client-Serveur, un programme client s'adresse à un programme serveur qui s'exécute sur une machine distante pour échanger des informations et des services (fig. 11.1). Il y a en général plusieurs clients qui accèdent simultanément à un même serveur. Cette branche d'application constitue l'un des axes les plus importants de l'informatique distribuée.

De nombreux domaines parfois très différents nécessitent ce type d'architecture :

- Le *serveur* de bases de données ; le serveur administre les données et les droits des utilisateurs d'une base de données que différents *clients* peuvent interroger. SQL-Server est un serveur de bases de données tandis que ACCESS ou FoxPro sont des clients potentiels.
- Le *serveur* WEB ; le serveur gère les connexions et renvoie les pages HTML que les différents Browsers *clients* souhaitent afficher. Apache ou IIS sont des serveurs WEB tandis que Netscape Navigator ou Internet Explorer sont des logiciels clients appelés Navigateurs [1].
- Le *serveur* de messagerie ; le serveur gère la gestion des files d'attentes et l'acheminement des messages emails qui sont envoyés par les *clients*. Sendmail est un serveur de messagerie tandis qu'Eudora ou Outlook sont des clients.
- Le *serveur* XWindow ; le serveur graphique s'occupe de l'affichage des fenêtres sur l'écran et gère les ressources graphiques que les applications graphiques *clientes* souhaitent utiliser. Sous XWindow, toute application graphique est potentiellement cliente du serveur X.
- Le *serveur* de fichiers ; dans un réseau de machines, le serveur de fichiers banalise le système de fichiers pour que les *clients* accèdent à leurs fichiers indépendamment de la machine sur laquelle ils sont connectés.
- Le *serveur* d'objets distribués ; le serveur (RMI, Corba) s'occupe de l'interrogation et de la persistance des objets ainsi que de l'envoi de la réponse aux programmes *clients* qui invoquent les méthodes à distance.

Chaque serveur, une fois lancé, scrute les éventuelles demandes des clients sur l'un des ports d'entrée-sortie de l'ordinateur sur lequel il est installé. Un certain nombre de numéros de ports

1. Browser

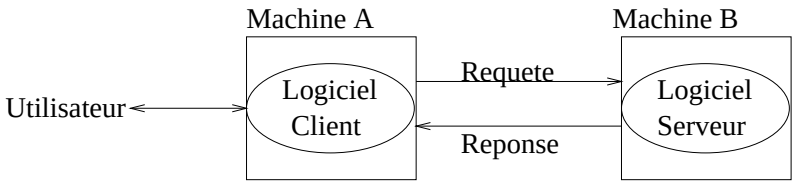


FIG. 11.1 – *le mode Client-Serveur*

sont réservés de manière standard (fig. 11.2). Par exemple, un serveur HTTP utilisera en généal le port 80.

Jusqu'à récemment, les applications Client-Serveur étaient réalisées par un constructeur qui fournissait à la fois la partie cliente et la partie serveur. L'inconvénient majeur de cette approche réside dans le fait qu'il devient impossible de changer son client ou son serveur indépendamment de l'autre car tous deux doivent impérativement utiliser un même protocole propriétaire.

service	port
Echo	7
FTP	21
Telnet	23
SMTP	25
Time	37
Name	42
HTTP	80
Gopher	70
POP3	110

FIG. 11.2 – *numéros de port standard*

Peu à peu le besoin de permettre la connexion de n'importe quel type de client avec n'importe quel type de serveur s'est fait sentir, permettant aux développeurs d'utiliser les outils les plus performants même s'ils n'appartiennent pas au même constructeur.

Ainsi des protocoles comme ODBC (Open DataBase Connectivity) ou JDBC (Java DataBase Connectivity) sont nés. Ces protocoles sont définis dans des pilotes (drivers) propres à chaque serveur qui permettent de traduire les requêtes du logiciel client en ordres exploitables par le serveur, puis à traduire le format des données renvoyées par le serveur en données manipulables par le client.

A l'époque des grands systèmes informatiques centralisés, le seul problème important était celui du choix du bon constructeur. A l'heure du Client-Serveur il est maintenant question de savoir répartir l'application entre serveur et client (fig. 11.3). Quelles fonctions se retrouvent dans le serveur et quelles autres dans le client. Ce choix dépend évidemment de nombreux paramètres comme le type d'application à réaliser, le coût du matériel mais aussi la puissance de calcul disponible, la puissance du serveur et les choix stratégiques de l'entreprise.

Dans un futur proche, chaque machine sera à la fois client et serveur de données et de services, permettant ainsi l'avènement d'une informatique nomade avec des "agents intelligents" qui géreront les négociations avec leurs homologues situés aux quatre coins de la planète. Chaque base de données sera distribuée et chacun pourra apporter sa contribution à la connaissance mondiale .... mais ceci est une autre histoire.

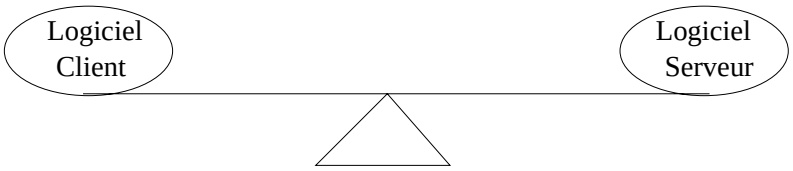


FIG. 11.3 – *Que mettre dans le Client et que mettre dans le serveur ?*

11.1 ODBC

(ou l'accès aux Bases de données dans le monde Microsoft)

Afin de faciliter la connexion entre les clients bases de données qui existent dans le monde Windows et les différents SGBD du marché, Microsoft a proposé un format de communication entre clients et serveurs de ce domaine. Ce format se nomme ODBC (Open DataBase Connectivity) et constitue un standard de fait du monde Windows. Il permet de faire communiquer à peu près tout serveur de bases de données avec des clients tournant sur système Windows. Il est par exemple très facile de faire communiquer un AS400 avec ACCESS. De cette manière il est possible d'utiliser la puissance de l'AS400 en ce qui concerne l'hébergement des données et la souplesse d'ACCESS pour la construction des formulaires et des états (fig. 11.4).

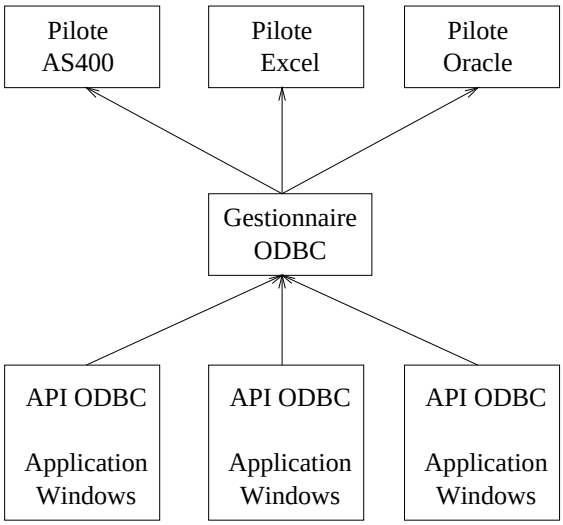


FIG. 11.4 – *Principe de fonctionnement d'ODBC*

Dès l'installation d'un système Microsoft (Windows 95,98,NT) un certain nombre de pilotes ODBC sont disponibles. Ce sont tous les pilotes nécessaires aux applications Microsoft [2]. On en trouve la liste dans le panneau de configuration, `ODBC 32bits` ou dans le fichier `ODBC.ini` du répertoire `windows`.

- Microsoft DBASE Driver
- Microsoft Excel Driver
- Microsoft FoxPro Driver
- Microsoft Access Driver

2. tous les produits Microsoft utilisent la même librairie dynamique `odbcjt32.dll` du répertoire `windows/system`

– Microsoft Text Driver

Chacun de ces pilotes gère les données comme il l'entend. Si cette gestion est classique dans le cas de "vrais" SGBD, cela devient beaucoup moins "standard" pour des liaisons avec Excel ou des fichiers Textes !! Par exemple, lors d'une liaison avec Excel, chaque table correspondra à une feuille de calcul figurant dans une seule et même enveloppe symbolisant la base tandis que dans le cas des fichiers textes, chaque table correspondra à un fichier texte indépendant (autant de fichiers que de tables).

Si vous souhaitez effectuer une liaison ODBC avec un autre logiciel que ceux de la liste précédente, c'est le constructeur de votre serveur de bases de données qui doit vous le fournir. C'est le cas si vous souhaitez, par exemple, vous connecter à une base de données des logiciels Sybase, Oracle ou même AS 400. Une procédure d'installation du nouveau pilote vous est en même temps fournie.

Les avantages d'un développement ODBC sont indéniables puisqu'il devient possible d'écrire des applications accédant à des données réparties entre plusieurs sources hétérogènes. On développe son application sans se soucier de la source de données qui sera utilisée en exploitation. La base de données utilisée côté serveur pourra donc être interchangée sans aucune modification du développement fait dans la partie cliente.

Il est très facile de créer une liaison ODBC entre un client et un serveur. L'exemple que nous allons détailler consistera à accéder à des données Excel à partir de tables ACCESS. Ceci se fait en trois parties : créer l'enveloppe Excel, définir le DSN (Data Source Name) ODBC, faire la liaison dans Access.

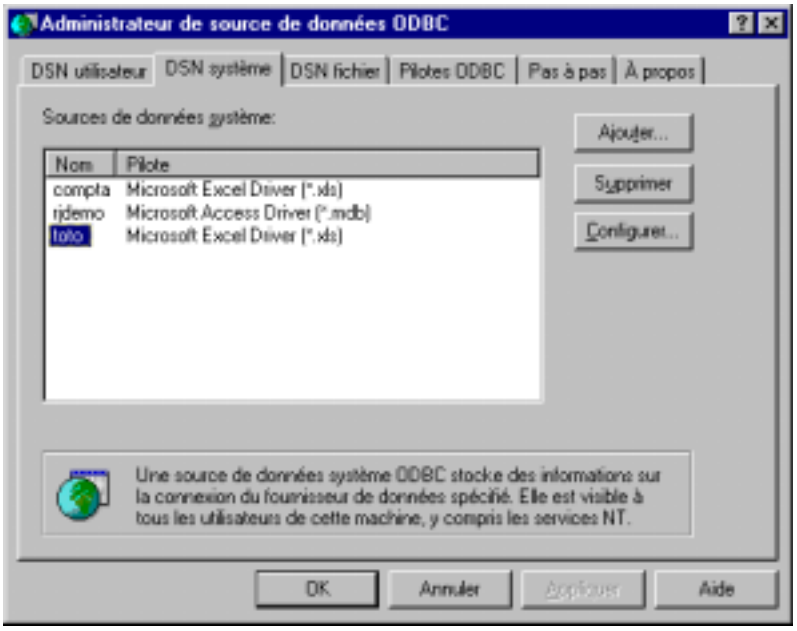


FIG. 11.5 – *Définition d'un nouveau DSN*

1. Tout d'abord, il est nécessaire de créer une enveloppe Excel sur laquelle pointera notre lien ODBC. Créez donc le classeur `toto.xls` (sans rien dedans) dans votre répertoire courant.
2. Il faut maintenant établir la liaison ODBC avec notre enveloppe Excel. Pour cela, lancer l'application `ODBC 32bits` dans le panneau de configuration, choisir `DSN System` puis `Ajouter`. Sélectionnez alors le pilote Excel et cliquer sur `Terminer`. Le nom de la source de données vous est alors demandé. C'est le nom symbolique qui sera ensuite utilisé dans ACCESS (il peut très bien être complètement différent du classeur Excel). Choisissez par exemple

`toto` (fig. 11.5). Cliquez sur `Classeur Sélectionner` et pointez sur votre enveloppe Excel `toto.xls`. Vous avez donc associé à un classeur physique, un nom logique, ce qui définit un DSN (Data Source Name) au sens Microsoft.

3. Lancez maintenant ACCESS, créer une nouvelle base puis une nouvelle table, mais au lieu de choisir le mode `Création`, choisissez le mode `Attacher la table`. Il vous reste à choisir le type de fichiers ODBC puis à pointer sur votre DSN fraîchement créé `toto` dans le panneau `source de données machine`. Toutes les actions et requêtes que vous allez effectuer maintenant sur cette table Access, seront mises à jour dans votre enveloppe Excel!

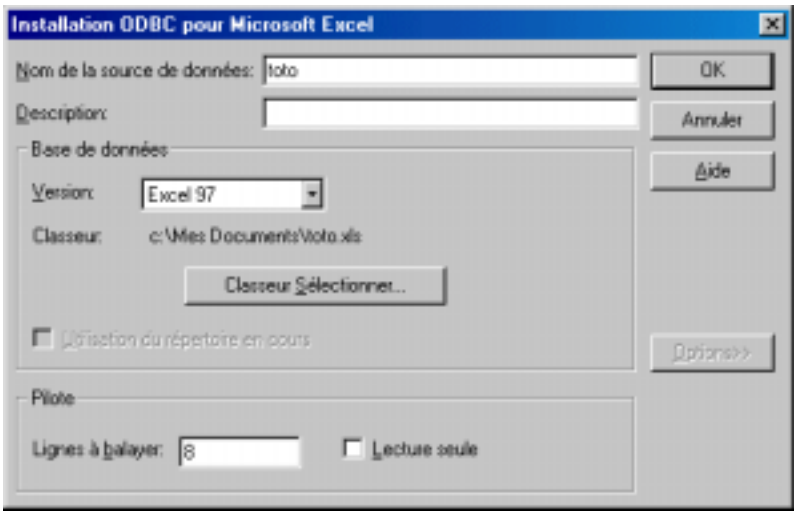


FIG. 11.6 – *Définition d'une source de données*

Le langage utilisé par ODBC pour communiquer avec les serveurs est rarement utilisé directement dans les applications. Il n'est intéressant que pour les personnes qui souhaitent développer leurs propres pilotes.

Dans la définition du lien ODBC il est possible de demander la trace dans un fichier des ordres ODBC qui transitent entre le client et le serveur.

11.2 JDBC

Le langage Java contient depuis la version 1.1 une API adaptée à la connexion avec les bases de données nommée JDBC. L'avantage de cette API est qu'elle a été construite indépendamment de toute base de données offrant ainsi une portabilité sans équivalent: Le langage Java qui est interprété peut s'exécuter sur toute architecture et l'API JDBC étant indépendante du SGBD s'utilise de la même manière pour toute base de données.

Si ODBC offre une couche d'abstraction universelle indépendante de la base de données, il n'est, par contre, pas indépendant de la plate-forme (MS Windows). JDBC par contre offre au développeur l'abstraction dans les deux directions: base de données et plate-forme système.

Ecrire une application Java de connexion à une base de données nécessite le JDK (Java Development Kit) développé par le constructeur SUN pour permettre la compilation des programmes Java, un serveur de bases de données pour répondre aux requêtes et un Driver particulier propre à chaque SGBD.

11.2.1 Qu'est ce que JDBC?

JDBC est une API fournie avec Java permettant l'accès à n'importe quelle base de données à travers un réseau. Le terme JDBC signifie Java DataBase Connectivity et permet d'écrire facilement des applications Java qui interagissent avec une base de données en respectant le principe de Java: "un seul code + une seule compilation = parfaite portabilité". JDBC permet notamment de résoudre le problème lié à tous les Intranet actuels: comment relier la base de données d'entreprise à des pages WEB et ainsi permettre aux clients de visualiser les informations qui les concernent à travers le réseau? Les avantages de JDBC par rapport aux autres approches (ODBC, appels systèmes etc...) sont avant tout des avantages liés à Java: portabilité sur de nombreux systèmes d'exploitation et de nombreuses bases de données, uniformité du langage de description des applications, des Applets et des accès bases de données et surtout liberté totale vis à vis des différents constructeurs.

Pour permettre l'accès aux bases de données, le JDK fournit le paquetage `java.SQL` qui est mis à la disposition des développeurs pour formuler et gérer les requêtes aux bases de données.

Ce paquetage offre plusieurs interfaces définissant les objets nécessaires à la connexion à une base éloignée et à la création et exécution de requêtes SQL ainsi que des classes utilitaires.

Interfaces	Classes	Exceptions
Array	**Date**	BatchUpdateException
Blob	DriverManager	DataTruncation
CallableStatement	DriverPropertyInfo	**SQLException**
Clob	Time	SQLWarning
Connection	Timestamp	
DatabaseMetaData	Types	
Driver		
PreparedStatement		
Ref		
ResultSet		
ResultSetMetaData		
SQLData		
SQLInput		
SQLOutput		
Statement		
Struct		

Les classes indiquées en gras sont celles utilisées dans ce livre. On pourrait sans doute considérer que ce sont les premières à connaître dans l'API `java.sql`.

11.2.2 Structure d'une application JDBC

Chaque programme Java souhaitant utiliser l'API JDBC doit tout d'abord inclure le paquetage java.sql: `import java.sql.*;`. Ensuite, comme pour toutes les entrées-sorties Java, des exceptions sont générées en cas d'erreur. Tous les ordres SQL doivent donc capter l'exception `SQLexception` qui est appelée dès qu'un ordre SQL ne se passe pas correctement. Cette classe contient notamment la méthode `getMessage()` qui renvoie le message en clair de l'erreur.

La philosophie de développement d'une application JDBC est alors la suivante:

1. Enregistrer le pilote JDBC.
   Chaque base de donnée utilise un pilote (driver) qui lui est propre et qui permet de convertir les requêtes JDBC dans le langage natif du SGBD. L'enregistrement de ce pilote doit être effectué dans le DriverManager Java au début de chaque application.
   Quand une classe de Driver est chargée, elle doit créer une instance d'elle même et s'enregistrer auprès du Driver Manager. Ceci se fait par l'Applet à la méthode suivante:
   `Class.forName("nom du driver");`

Par exemple, le nom du pilote fourni par le constructeur SUN pour effectuer un pont JDBC-ODBC est `sun.jdbc.odbc.JdbcOdbcDriver`. L'ordre d'enregistrement aura donc la forme :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

On notera que certains compilateurs refusent cette notation et demandent l'invocation explicite de la méthode `newInstance()` de la manière suivante :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
```

2. Etablir la connexion avec la base de données

   Une fois le pilote enregistré, il est alors possible d'établir la connexion avec la base de données. Ceci se fait par la méthode `getConnection` du Driver Manager qui renvoie un objet de type `Connection`. Il est nécessaire de passer 3 arguments à cette méthode : l'URL de la base de données, le nom de l'utilisateur de la base et enfin son mot de passe. Ces trois arguments doivent être fournis sous forme de chaînes de caractères.

   Par exemple, dans le cas du pont JDBC-ODBC, le nom de DSN à passer doit être de la forme `jdbc:odbc:monDSN`. Si votre DSN se nomme `compta` accessible par l'utilisateur `admin` et le mot de passe `ukvg` la connexion se fera par l'ordre :

```
java.sql.Connection con =
      DriverManager.getConnection("jdbc:odbc:compta","admin","ukvg");
```

3. Créer une zone de description de requête

   Pour définir une requête, il est ensuite nécessaire de créer un objet de type `Statement`. Cet objet offre non seulement une zone de description de la requête SQL par elle-même, mais aussi les ordres d'exécution de cette requête.

   La création de l'objet `Statement` se fait par la méthode `createStatement` de l'objet `Connection` précédemment créé :

```
java.sql.Statement stmt = con.createStatement();
```

4. Exécuter la requête

   Une fois la zone de description, créée, il est alors possible de la remplir avec une requête SQL puis d'exécuter cette requête.

   Deux méthodes de l'objet `Statement` permettent respectivement de définir et exécuter la requête. Il s'agit de la méthode `executeQuery` qui permet d'exécuter une requête du type SELECT et qui renvoie un objet de type `ResultSet` contenant les tuples résultant et de la méthode `executeUpdate` qui permet d'effectuer les requêtes du type CREATE, INSERT, UPDATE et DELETE et qui renvoie un entier (`int`) indiquant le nombre de tuples traités.

   Ces deux méthodes nécessitent un argument de type chaîne qui indique la requête SQL à exécuter. Par exemple ,

```
java.sql.ResultSet rs = stmt.executeQuery("SELECT * FROM clients");
int nb = stmt.executeUpdate("INSERT into clients values('Durand','Paul');
```

5. Traiter les données retournées

   L'objet `ResultSet` retourné après l'exécution de la méthode `executeQuery` contient plusieurs méthodes permettant de manipuler l'ensemble résultant de la requête :

   Pour parcourir l'ensemble résultat, il n'existe qu'une seule méthode : la méthode `next`. Initialement le pointeur est positionné avant le premier tuple, chaque appel à la méthode `next` fait avancer le pointeur sur le tuple suivant. `next` renvoie à chaque invocation un booléen permettant de savoir s'il y a encore des tuples disponibles.

```
while (rs.next())
{// traitement de chaque tuple
...
}
```

Il n'est donc pas possible de revenir aux tuples précédents ou de parcourir l'ensemble résultat dans un ordre aléatoire; on commence par le premier, on passe au suivant et ainsi de suite jusqu'au dernier, sans possibilité de retour en arrière. Si un retour arrière est souhaité il est donc nécessaire de refaire un `ExecuteQuery`.

Pour accéder aux éléments d'un tuple, l'objet `ResultSet` fournit toute une série de méthodes de la forme `getXXX()` permettant de lire le type de données `xxx` dans chaque colonne du tuple courant. On utilisera par exemple les méthodes `getInt`, `getFloat`, `getDate`, `getLong`, `getInt`, `getString` etc ... La colonne est identifiée soit par son nom que l'on passe en paramètre sous forme de chaîne de caractère soit par son numéro relatif dans l'ordre des colonnes.

```
int pds = getInt(3); // accède à la 3ème colonne
int prod= getString("PRODUIT"); // accède à la colonne PRODUIT
```

Type SQL	Type Java	fonction d'accès Java
CHAR	String	getString()
VARCHAR	String	getString()
NUMERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	Boolean	getBoolean()
INTEGER	Integer	getInt()
REAL	Float	getFloat()
DOUBLE	Double	getDouble()
BINARY	byte[]	getBytes()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

FIG. 11.7 – *Conversions principales SQL - Java*

Il faut cependant noter que Java ne fournit pas directement la possibilité de savoir si une colonne SQL est nulle ou pas. JDBC fournit pour cela la méthode `wasNull()`[3] de la classe `ResultSet` qui permet de savoir *a posteriori* si une colonne était nulle ou pas.

```
int poids = rs.getInt("poids");
if (!rs.wasNull()) System.out.println("le poids est connu");
```

6. Fermer les différents espaces

   Tout programme bien construit se doit de fermer les espaces ouverts durant son travail. La méthode `close` est prévue à cet effet. Elle est définie pour les objets `ResultSet`, `Statement` et `Connection`.

```
rs.close();
st.close();
con.close();
```

Nous en savons assez pour écrire nos premières applications JDBC, mais auparavant, il reste à rappeler que toutes ces méthodes déclenchent des exceptions qui doivent impérativement être testées (ce qui n'est pas sans alourdir le code) :

3. Il existe une autre solution qui consiste à toujours utiliser getObject() pour récupérer les données

```
String url = "jdbc:mySubprotocol:myDataSource";
Class.forName("myDriver.ClassName");
con = DriverManager.getConnection(url,"myLogin","myPassword");
stmt.executeUpdate(createString);
ResultSet rs = stmt.executeQuery(selectString);
while (rs.next()) ...
rs.close();
stmt.close();
con.close();
```

FIG. 11.8 – *Méthodes JDBC principales*

11.2.3 Quelques exemples d'applications

Afin d'illustrer la manière d'utiliser les objets et méthodes précédents pour écrire une application JDBC, nous présentons trois petites applications Java permettant respectivement la création d'une table CLIENTS(nom,prenom,age), l'insertion de tuples dans cette table et enfin l'interrogation de ces tuples. Les bases sont utilisées via le pilote `jdbc:odbc` qui permet d'accéder à toute source de données ODBC par un programme Java.

Création d'une table

```
import java.sql.*;

public class Create
{
  public static void main(String args[])
    {
      String url = "jdbc:odbc:compta";
      Connection con;
      Statement stmt;
      String createString;
      createString = "create table CLIENTS " +
        "(NOM varchar(10), PRENOM varchar(10), AGE int)";

      try // enregistrement du driver
        {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
      catch(java.lang.ClassNotFoundException e)
        {
          System.err.print("ClassNotFoundException: ");
          System.err.println(e.getMessage());
        }

      try // connexion et execution de la requete
        {
          con = DriverManager.getConnection(url, "admin", "ukvg");
          stmt = con.createStatement();
          stmt.executeUpdate(createString);
          stmt.close();
          con.close();
        }
      catch(SQLException ex)
        {
          System.err.println("SQLException: " + ex.getMessage());
        }
    }
}
```

Insertion de tuples

```
import java.sql.*;

public class Insert
{
  public static void main(String args[])
    {
      String url = "jdbc:odbc:compta";
      Connection con;
      Statement stmt;

      try
        {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
      catch(java.lang.ClassNotFoundException e)
        {
          System.err.print("ClassNotFoundException: ");
          System.err.println(e.getMessage());
        }

      try
        {

          con = DriverManager.getConnection(url,"admin", "ukvg");
          stmt = con.createStatement();
          stmt.executeUpdate("insert into CLIENTS " +
                             "values('Durand', 'paul',10)" );
          stmt.executeUpdate("insert into CLIENTS " +
                             "values('Dupont','luc',14)" );
          stmt.executeUpdate("insert into CLIENTS " +
                             "values('Lefebvre','henri',17)" );

          stmt.close();
          con.close();
        }
      catch(SQLException ex)
        {
          System.err.println("SQLException: " + ex.getMessage());
        }
    }
}
```

Sélection de tuples

```
import java.sql.*;

public class Select
{

  public static void main(String args[])
    {
      String url = "jdbc:odbc:compta";
      Connection con;
      Statement stmt;
      String query = "select NOM,PRENOM,AGE from CLIENTS";

      try
        {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
      catch(java.lang.ClassNotFoundException e)
        {
          System.err.print("ClassNotFoundException: ");
          System.err.println(e.getMessage());
        }

      try
        {
          con = DriverManager.getConnection(url,"admin", "ukvg");
          stmt = con.createStatement();
          ResultSet rs = stmt.executeQuery(query);

          System.out.println("Liste des clients:");
          while (rs.next())
            {
              String n = rs.getString(1); // nom
              String p = rs.getString(2); // prenom
              int a = rs.getInt(3);       // age
              System.out.println(n + " " + p + " " + a);
            }

          stmt.close();
          con.close();

        }
      catch(SQLException ex)
        {
          System.err.println("SQLException: " + ex.getMessage());
        }
    }
}
```

11.2.4 Différents pilotes

Sans rentrer dans les détails techniques nécessaires uniquement aux concepteurs de nouveaux pilotes, il est néanmoins important d'apporter quelques précisions sur les pilotes JDBC.

Il existe, selon la taxonomie de JavaSoft, 4 types de pilotes JDBC différents :

- **Type 1.** Ce sont les pilotes accédant à une base de données par l'intermédiaire de ponts. L'exemple typique est le pilote fourni par SUN permettant le pont entre JDBC et ODBC. C'est celui que nous avons utilisé dans nos différents exemples. Ce type de pilote ne peut être utilisé que par les applications Java. En effet le modèle classique de sécurité pour l'exécution des Applets (untrusted) interdit à une Applet de charger du code natif dans la mémoire vive de la plate-forme d'exécution.
- **Type 2.** Ce sont les pilotes d'API natifs. Ils sont fournis par les éditeurs de bases de données et gèrent des appels C/C++ directement avec la base. Ils sont en général payants. Comme précédemment, le modèle classique de sécurité pour l'exécution des Applets (untrusted) interdit à une Applet de charger du code natif dans la mémoire vive de la plate-forme d'exécution. Ce type de Pilote ne peut donc pas être utilisé dans une Applet.
- **Type 3.** Ce sont les pilotes qui interagissent avec une API réseau générique et qui communiquent avec une application intermédiaire (middleware) sur le serveur. C'est typiquement le cas de l'API RMI-JDBC fournie par l'INRIA ou du système DBAnywhere offert par Symantec Visual Café. Ils sont écrits en Java et peuvent donc être chargés sans aucun problème par la plate-forme d'exécution de l'Applet. Un seul de ces pilotes permet d'interagir avec plusieurs bases de données, ce qui rend cette approche très souple à l'utilisation.
- **Type 4.** Ce sont les pilotes qui interagissent avec la base de données via les sockets et encapsulent complètement l'interface cliente du SGBD. Ils sont généralement fournis par les éditeurs de SGBD. Là encore, il n'y a aucun problème d'exécution que ce soit dans une application ou dans une Applet.

Une dernière contrainte est à noter en ce qui concerne JDBC et les Applets : Le modèle classique de sécurité de l'exécution des Applets (untrusted) ne permet d'ouvrir une connexion réseau qu'avec la machine sur laquelle elle est hébergée. Le serveur de base de données (ou le serveur middleware) doit donc être installé sur la même machine que le serveur HTTP [4].

11.2.5 Les requêtes pré-compilées

Dans la majorité des applications d'accès aux bases de données, les requêtes SQL dépendent de paramètres du programme. Pour introduire une variable Java dans la requête SQL, deux méthodes sont possibles.

La première, très simple, consiste à remarquer que l'ordre SQL est décrit dans une chaîne de caractères, et que donc, il est possible de définir cette chaîne en utilisant des variables. Il est donc possible d'écrire :

```
String personnes[]={"Durand", "Dupond","Martin"};
String table=''Clients'';
String query;
    for (int i=0; i< personnes.length; i+)
    {
        query = "select * from " + table
                + " where nom = '" + personne[i] + "'";
    }
    ....
```

4. On prendra garde à ne pas confondre HTTP qui est un protocole d'expression de requêtes entre clients et serveurs et le WEB, abréviation de World Wide Web, qui correspond à la toile virtuelle reliant toutes les machines du monde. HTTP est l'un des protocoles du WEB comme FTP, POP3 ou SMTP. On dit souvent "serveur WEB" mais c'est un abus de langage.

La seconde méthode consiste à utiliser une requête précompilée à l'aide de l'objet `PreparedStatement`. Cet objet, qui hérite de l'objet `Statement` permet d'envoyer une requête sans paramètres à la base de données pour précompilation, puis à spécifier au moment voulu la valeur des paramètres.

La création d'un `PreparedStatement` s'obtient à l'aide de la méthode `prepareStatement` de l'objet `Connection`. Cette méthode nécessite un argument de type chaîne de caractères, décrivant la requête SQL à effectuer. Les arguments dynamiques doivent être marqués par un point d'interrogation.

```
PreparedStatement st = c.prepareStatement("SELECT * FROM ? WHERE nom = ?");
```

Chaque argument dynamique doit ensuite être positionné à l'aide des méthodes `setInt`, `setFloat`, `setDate`, `setLong`, `setString`, etc ... de l'objet `PreparedStatement`. Ces méthodes nécessitent deux arguments, le premier de type entier qui indique le numéro relatif de l'argument dans la requête, le second qui indique la valeur à positionner. Par exemple :

```
st.setString(1,"Clients");
st.setString(2,personne[1]);
```

Pour exécuter la requête ainsi construite il suffit alors d'exécuter les méthodes `executeQuery` ou `bexecuteUpdate` de l'objet `PreparedStatement` sans aucun argument. Comme pour la méthode classique la méthode `executeQuery` retourne un objet de type `ResultSet` contenant les tuples resultant du SELECT, tandis que la méthode `executeUpdate` retourne le nombre de tuples mis à jour.

```
java.sql.ResultSet rs = st.executeQuery();
```

Il est clair que si votre requête doit être exécutée plusieurs fois avec des arguments variables, la méthode `PreparedStatement` est bien plus rapide qu'avec un `Statement` classique. Assurez vous néanmoins que votre SGBD accepte les requêtes précompilées, ce qui n'est pas le cas de tous.

11.2.6 Commit et Rollback

Par défaut, tout objet `Connection` fonctionne en mode auto-commit. Ce qui veut dire, qu'un Commit est effectué automatiquement après chaque ordre SQL. Il est néanmoins possible de repasser en mode manuel par l'appel à la méthode `setAutoCommit` de l'objet `Connection`. Cette méthode admet un seul argument booléen indiquant si oui ou non le mode est automatique.

```
con.setAutoCommit(false)
```

Une fois mis en mode manuel, c'est à l'application de solliciter le commit ou le rollback par les méthodes `commit` et `rollback` de l'objet `Connection`. Ceci est notamment très intéressant lorsqu'il est nécessaire de valider tout un groupe d'instructions à la fois.

```
con.commit()
```

11.2.7 La méta-base

En plus des différents objets permettant d'exécuter les requêtes, JDBC offre aussi la possibilité d'obtenir des renseignements sur les données à travers deux classes : `DatabaseMetaData` et `ResultSetMetaData` . La première donne des renseignements très complets sur la méta-base en général tandis que la seconde indique les renseignements sur le `ResultSet` obtenu après une requête. Les méthodes de ces deux classes sont très nombreuses (plus de 130 pour la première). Voyons sur un exemple simple, la manière de récupérer les informations sur les colonnes d'une table :

```
import java.sql.*;

class Meta  {

  public static void main(String args[])
    {
      String url = "jdbc:odbc:compta";
      Connection con;
      String query = "select * from Clients";
      Statement stmt;

      try
        {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
      catch(java.lang.ClassNotFoundException e)
        {
          System.err.print("ClassNotFoundException: ");
          System.err.println(e.getMessage());
        }

      try
        {
          con = DriverManager.getConnection(url, "admin", "ukvg");
          stmt = con.createStatement();
          ResultSet rs = stmt.executeQuery(query);
          ResultSetMetaData rsmd = rs.getMetaData();

          int nbCols = rsmd.getColumnCount();
          System.out.println("Cette table contient "+ nbCols + " colonnes");

          for (int i = 1; i <= nbCols; i++)
            {
              System.out.println("Colonne "+ i);
              System.out.println("Nom  : " + rsmd.getColumnName(i));
              System.out.println("Type : " + rsmd.getColumnTypeName(i));
              System.out.println("Prec : " + rsmd.getPrecision(i));
              System.out.println("Read only : " + rsmd.isReadOnly(i));
              System.out.println("Auto num  : " + rsmd.isAutoIncrement(i));
              System.out.println("Null accepté : " + rsmd.isNullable(i));
              System.out.println("");
            }

          stmt.close();
          con.close();
        }
      catch(SQLException ex)
        {
          System.err.print("SQLException: ");
          System.err.println(ex.getMessage());
        }
    }
}
```

11.2.8 Un exemple graphique

Afin de mettre en application les méthodes JDBC que nous venons de présenter, voici le code d'une petite application Java fournissant une interface graphique d'accès à différentes bases de données. Cette application graphique est écrite à l'aide des JFC (Swing).

mettre le code ....

11.3 WEB et Bases de données

Diffuser ses informations sur le WEB est actuellement le cheval de bataille de beaucoup d'entreprises. Elles peuvent ainsi présenter leurs produits, prendre des commandes et afficher des informations à leurs clients en temps réel. Aux débuts d'Internet les choses étaient relativement simples : Les données fournies par les serveurs étaient uniquement statiques composées de textes et d'images (fig. 11.9). Peu à peu, le besoin s'est fait sentir de générer les pages en fonction du contenu du SGBD. Les pages sont alors générées dynamiquement par le serveur. Pour effectuer cela il est nécessaire d'avoir une architecture spécialisée.

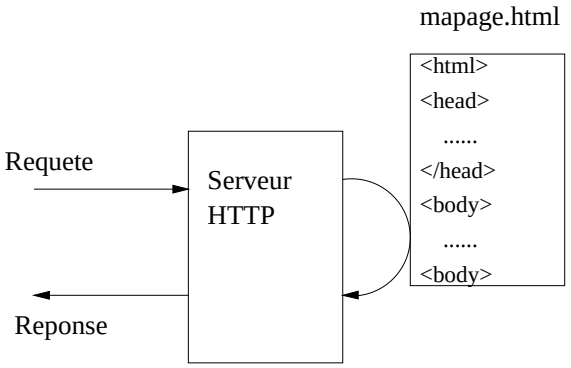


FIG. 11.9 – *Requête à un serveur HTTP*

Ce type d'architecture est peu onéreux puisqu'il ne nécessite que l'existence d'un Browser WEB du côté client. Du côté de l'entreprise il faut par contre un serveur HTTP pour gérer les connexions extérieures, un serveur de bases de données pour gérer le système d'information de l'entreprise et une API (Application Programming Interface) permettant de relier la base de données au WEB. Il existe pour cela différentes possibilités (CGI, scripting, Applets, Servlets) avec pour chacune des avantages et des inconvénients que nous détaillons sous forme de tableau.

Technologie	Principe	Avantages	Inconvénients
CGI	un processus par requête est lancé sur le serveur, renvoie du HTML.	gratuit, peut être écrit dans n'importe quel langage.	lent, difficile à développer, appels natifs à des procédures du SGBD.
Scripting	script propre au constructeur intégré dans les pages HTML, renvoie du HTML.	facile à développer et clair.	payant (cher), lié à un constructeur, langage propre au SGBD et au serveur HTTP.
Applets	Code Java exécuté sur le poste client, entièrement développé en Java avec AWT ou Swing.	gratuit, permet de gérer des applications complexes, portable (JDBC), indépendant des plateformes matérielles et logicielles, pas de code HTML	lent à charger, les serveurs HTTP et BDD doivent être sur la même machine.
Servlets	Code Java exécuté sur le serveur, renvoie du HTML	gratuit, portable (JDBC), indépendant des plateformes matérielles et logicielles, rapide, facile à développer.	limité à HTML.

Les CGI, du fait de leur lenteur d'exécution et de la difficulté à conserver les données d'une requête à l'autre (un process à chaque fois) sont maintenant dépassés. Le Scripting, bien que très souple est entièrement lié à un serveur HTTP et à une base de données (voir même un système d'exploitation). C'est par exemple le cas avec `Oracle Web Application Server` et `Oracle 8` ou avec `Internet Information Server` et `SQL Server` (technique ASP [5]). La solution Java apparait comme la solution d'avenir avec des Applets pour les applications clientes gourmandes en temps de calcul (graphiques etc ...) et les Servlets pour la génération de pages HTML dynamiques. L'architecture ouverte et indépendante des plateformes permise par le langage Java (Applications, Applets, Servlets) permet de conserver le même langage pour tous les développements de l'entreprise, ce qui n'est pas le moindre de ses avantages.

11.3.1 Applets ou Servlets?

Nous l'avons vu, l'architecture ouverte et indépendante des systèmes permettant d'accéder à une base de données via le WEB passe par le langage Java et l'API JDBC. Cette approche peut se faire de deux manières conceptuellement très différentes: Applets et Servlets. Rappelons les principes de ces deux approches:

– *Applets.* Elles sont entièrement écrites en Java, interface graphique comprise. On utilise pour cela les API AWT ou Swing. L'Applet est toujours exécutée sur le poste client. Celui-ci doit donc la charger à la première exécution, ce qui nécessite parfois le transfert de plusieurs centaines de K-octets de données. Le modèle de sécurité Java impose qu'une Applet ne peut se connecter qu'aux applications présentes sur la machine d'où elle a été chargée. Le serveur de bases de données et le serveur HTTP doivent donc se situer sur la même machine. Toujours pour des raisons de sécurité, Java interdit aux Applets de faire des appels système, ce qui par exemple interdit l'utilisation du pont ODBC-JDBC. D'un point de vue sécurité, il faut être conscient du fait que le client charge l'Applet chez lui, et que, même si cela semble complexe, il lui est possible de décrypter les requêtes faites par l'Applet, les noms des tables, des colonnes voire même les mots de passe des connexions inscrits dans cette Applet. En ce qui concerne le type d'application nécessitant des Applets, plus l'interface utilisateur

5. ASP est certainement la technologie la plus répandue actuellement grâce à la notoriété de Microsoft. Les ASP utilisent du code VBScript et sont interprétées à chaque appel. Elles ne fonctionnent qu'en environnement totalement Microsoft, avec SQL-Server et IIS par exemple.

est complexe, plus l'approche Applet devient intéressante. S'il s'agit de réaliser une base de données de figures constituée de vecteurs et de points puis de permettre aux clients d'afficher chaque figure et lui apliquer des rotations et homothéties etc, l'approche Applet est la seule possible.

– *Servlets*. Elles sont écrites en Java et génèrent en retour[6] une nouvelle page HTML. L'interface graphique est donc limitée à la puissance du langage HTML. Les Servlets s'exécutent toujours sur le serveur et ne sont donc pas astreintes aux règles de sécurité des Applets. Elles peuvent faire des appels système comme par exemple utiliser le pont ODBC-JDBC. D'un point de vue sécurité, comme la Servlet tourne uniquement sur le serveur, le client ne reçoit que les réponses aux requêtes et ne peut en aucun cas, voir le code de cette requête. L'approche Servlets est donc plus sécurisée que l'approche Applets. Moins il y a de calcul à faire côté client, plus l'approche Servlet est intéressante. Si l'application doit afficher des listes de valeurs, comme c'est le cas pour afficher les commandes en cours d'un client, les produits disponibles de la société ou effectuer une recherche dans l'annuaire de la société, cela ne posera aucun problème en HTML.

En résumé, les servlets sont aux serveurs ce que les applets sont aux browsers. La question fondamentale qu'il faut se poser est "Où doit se faire le calcul? sur le poste client ou sur le serveur?". La réponse à cette question dépend bien évidemment de l'application concernée. Précisons néanmoins que ces deux approches ne sont pas antinomiques et qu'il est bien évidemment possible d'avoir des Servlets qui génèrent des pages HTML avec des appels aux Applets. Le mélange des genres est tout à fait possible et même nécessaire dans le cadre d'applications complexes.

11.3.2 Principe d'écriture de Servlets

Une Servlet est un morceau de code Java exécuté dans des serveurs multi-threadés comme c'est le cas pour un serveur HTTP. Elle a pour objectif de recevoir et répondre aux requêtes des clients (fig. 11.10).

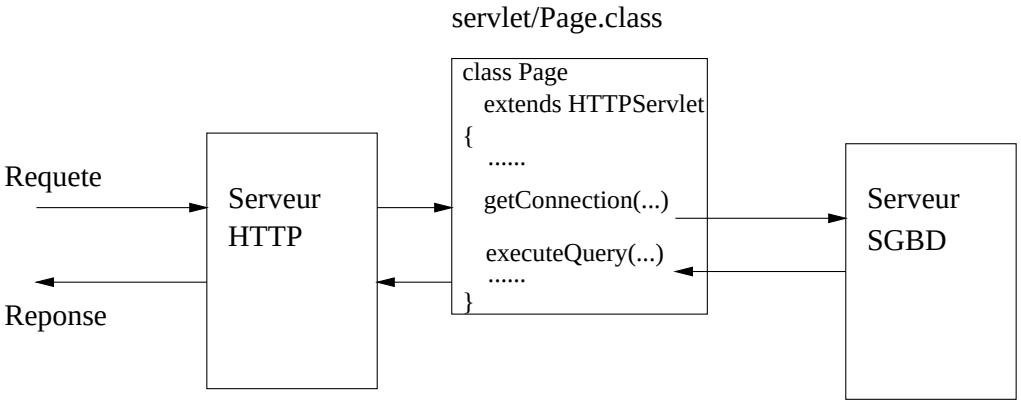


FIG. 11.10 – *Exécution d'une Servlet*

Les Servlets ne font à l'heure actuelle pas directement partie du JDK[7]. Sun fournit pour cela un paquetage à part nommé JSDK[8] (Java Servlet Development Kit) qui définit toutes les classes nécessaire à la réalisation de Servlets.

Une Servlet implémente nécessairement l'interface `javax.servlet.Servlet`. Cette interface définit les méthodes permettant d'initialiser la Servlet, de recevoir et répondres aux requêtes des

6. En réalité les Servlets peuvent retourner d'autres formats que du HTML, mais cela sort du cadre de ce livre
7. disponible sur `http://www.javasoft.com`
8. Disponible sur `http://jserv.javasoft.com`

clients et enfin de détruire la Servlet et ses ressources. Le cycle de vie d'une Servlet est donc le suivant :

1. la Servlet est créée puis initialisée (méthode `init`) :
   `void init(ServletConfig)`
2. les services aux clients sont exécutés (méthode `service` et ses dérivées)
3. la Servlet est détruite et les ressources libérées (méthode `destroy`) :
   `void destroy()`

Il est important de bien noter qu'une Servlet est chargée par le serveur HTPP lors de la première invocation et qu'elle reste active dans le serveur tant que celui-ci le juge utile[9]. A la seconde invocation de la Servlet, elle n'est pas recréée, seule la méthode de service est appelée. On en conclue immédiatement que la première invocation d'une Servlet prend toujours plus de temps que les suivantes. On a donc une forme de persistance[10].

Afin de faciliter le traitement particulier des serveurs HTTP, la classe `Servlet` est affinée en `javax.servlet.http.HttpServlet`. Cette classe contient notamment les méthodes `doPost` et `doGet` qui remplacent avantageusement la méthode `service`[11] de la classe mère.

Par défaut ces méthodes n'effectuent aucun calcul et renvoient un code d'erreur (`HTTP_BAD_REQUEST` pour `doPost` et `doGet`). C'est au développeur de la Servlet de décider de la redéfinition des méthodes qu'il souhaite réaliser.

Dans le cas d'un serveur HTTP, la Servlet a pour objectif de renvoyer une page HTML fonction de paramètres saisis par l'utilisateur (pages dynamiques). Contrairement aux Applets, la Servlet ne contient aucune interface graphique; ce sont les pages HTML qui saisissent les paramètres et présentent les données résultat. L'approche Servlet se fait donc en trois parties: La construction de la page HTML qui présente le formulaire d'interrogation des paramètres, la Servlet qui effectue la requête et la page résultat générée par la Servlet.

Pour écrire une Servlet il faut tout d'abord, comme dans tout programme Java qui utilise des classes externes, importer les paquetages utilisés. Au minimum il est nécessaire d'importer `javax.servlet.*` et `javax.servlet.http.*` ainsi que `java.io.*` pour les `Exceptions`.

Une Servlet à destination du WEB (qui est le cas qui nous intéresse le plus ici) hérite de la classe `HTTPServlet`. Elle doit obligatoirement contenir soit la méthode `doGet` soit la méthode `doPost` selon le mode d'envoi du formulaire HTML qui exécute la Servlet.

- `doGet`. C'est le mode d'appel par défaut à partir de la ligne d'URL du Browser. C'est aussi le mode d'appel à partir d'un hyperlien (tag `<A>`). Dans le cas de paramètres, les valeurs sont concaténées à l'URL sous la forme `?var="val"`. Par exemple,
  `http://127.0.0.1/servlet/listeComptes?nom="Dupond"?prenom="pierre"` permet d'appeler la méthode `doGet` de la Servlet `listeComptes` avec les paramètres nom et prénom respectivement mis aux valeurs "Dupond" et "pierre". Les serveurs de Servlets restreignent en général la longueur de l'URL à 240 caractères.
- `doPost`. Cette méthode permet d'envoyer de nombreuses données au serveur à travers les Sockets. Avec cette méthode les paramètres sont placés après l'entête et ne sont donc pas visibles de l'utilisateur. L'URL ne change pas du tout. C'est le mode d'appel généralement utilisé à partir d'un formulaire (tag `<Form>`) bien que `doGet` soit aussi possible dans ce cas. Il est à noter que la méthode `doGet` est plus rapide à l'exécution que la méthode `doPost`.

En résumé, le code à écrire dans les deux cas est en général le même. La différence vient du fait que la requête POST peut traiter plus de paramètres que la méthode GET, que GET est appelée en général à partir d'une URL tandis que POST est plutôt utilisée à partir d'une Form.

Deux objets sont passés en paramètre à ces méthodes : `HttpServletRequest` qui contient les renseignements sur le formulaire initial et `HttpServletResponse` qui contient le flux de sortie pour la génération de la page à générer.

9. Les serveurs détruisent en général les Servlets dès que la taille mémoire générale occupée devient trop importante. Dans ce cas, les Servlet les moins utilisées sont détruites en premier

10. La persistance totale d'une Servlet en vue d'un rechargement ultérieur doit être traité par le programmeur

11. Il existe d'autres méthodes implémentables dans les Servlets comme Head, Put, Trace, Options. Leur description sort du cadre de ce livre

C'est la méthode `getWriter()` sur l'objet `HttpServletResponse` qui fournit le flux de sortie. Tandis que la méthode `getParameter()` sur l'objet `HttpServletRequest` permet la récupération des paramètres d'entrée.

Sous sa forme la plus simple (sans accès à une BDD) une Servlet est donc décrite de la manière suivante:

```
// First.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class First extends HttpServlet
{
  public void doGet( HttpServletRequest req, HttpServletResponse res )
       throws ServletException, IOException
  {
    ServletOutputStream out             = res.getOutputStream();

    res.setContentType( "text/html" );

    out.println( "<head><title>servlet First</title></head><body><center>" );
    out.println( "<h1>Test de ma Servlet</h1>" );
    out.println( "<h1>Super ! ça marche</h2>" );
    out.println( "</center>" );
    out.println( "</body>" );
  }
}
```

Il est important de noter que le poste client ne reçoit que le résultat de la Servlet (fig. 11.11) et non pas son code qui n'est présent que sur le serveur. Notamment, si le client demande à afficher la page générée il ne verra qu'une page HTML classique correspondant au résultat de la Servlet (fig. 11.12).

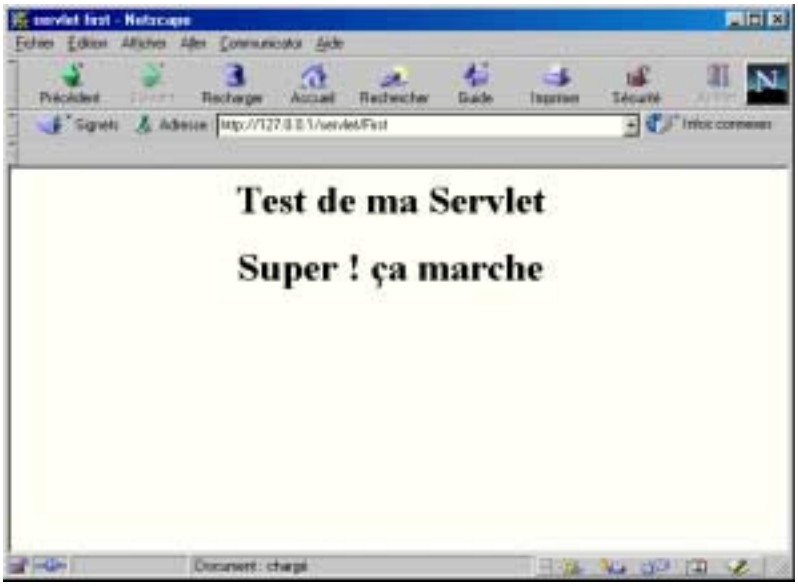


FIG. 11.11 – *Exécution de la Servlet*

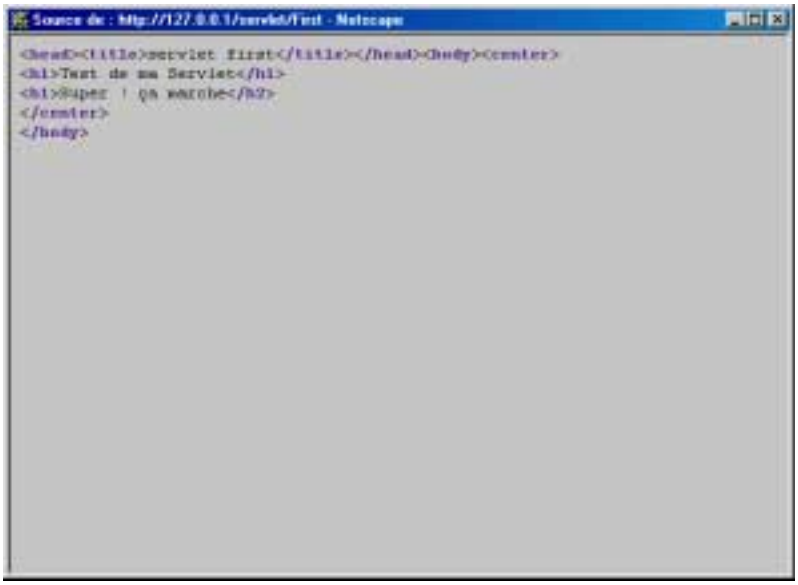


FIG. 11.12 – *Affichage du source sur le poste client*

Parfois, une même Servlet doit pouvoir être utilisée avec `doGet` comme avec `doPost`. La technique couramment utilisée dans ce cas consiste à implémenter l'une des deux méthodes et à faire en sorte que la seconde soit implémentée comme un simple appel à la première. Notre première Servlet peut donc être écrite plus élégamment de la manière suivante :

```
// FirstBis.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FirstBis extends HttpServlet
{
  public void doGet( HttpServletRequest req, HttpServletResponse res )
       throws ServletException, IOException
  {
    ServletOutputStream out             = res.getOutputStream();

    res.setContentType( "text/html" );

    out.println( "<head><title>servlet FirstBis</title></head><body><center>" );
    out.println( "<h1>Test de ma Servlet</h1>" );
    out.println( "<h2>Super ! ça marche</h2>" );
    out.println( "</center>" );
    out.println( "</body>" );
  }

  public void doPost( HttpServletRequest req, HttpServletResponse res )
       throws ServletException, IOException
  {
    doGet(req,res);
  }
}
```

11.3.3 La récupération des paramètres

Traditionnellement, le premier programme écrit dans un langage informatique est un programme qui génère “Hello World”. Soyons donc tout de suite beaucoup plus ambitieux en écrivant une Servlet qui pourra saluer l'utilisateur en précisant son nom.

Ceci se fait en deux temps: l'écriture de la page HTML de lancement de la Servlet, puis , l'écriture de la servlet proprement dite.

```
<!--   second.html  -->
<html>
<head>
<title> Hello world évolué !<title>
<body>
<form method=post action="/servlet/Second">
Quel est votre nom ?
<input type=text name="nom"><P>
<input type=submit>
</form>
</body>
</html>
```

```
// Second.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Second extends HttpServlet
{
  public void doPost( HttpServletRequest req, HttpServletResponse res )
       throws ServletException, IOException
  {
    ServletOutputStream out = res.getOutputStream();

    res.setContentType( "text/html" );

    out.println( "<head><title>servlet Second</title></head><body><center>" );
    out.println( "<h1>Salut " + getParameter("name") + "</h1>" );
    out.println( "</center>" );
    out.println( "</body>" );
  }

  public void doGet( HttpServletRequest req, HttpServletResponse res )
       throws ServletException, IOException
  {
    doGet(req,res);
  }
}
```

L'appel de cette Servlet se fait donc par le formulaire précédent en cliquant sur le bouton Submit. Dans ce cas, c'est une requête POST qui est envoyée. Comme la requête GET a aussi été implémentée, la Servlet peut donc être appelée à l'URL[12]
`http://127.0.0.1/servlet/Second?nom=Philippe+Mathieu`

11.3.4 Configuration de l'environnement d'exécution

Les serveurs HTTP actuels n'intègrent pas encore tous la possibilité d'exécuter des Servlets directement. La plupart nécessitent un patch additionnel. Pour faciliter les tests le JSDK fournit un mini serveur de Servlets nommé `servletrunner` qui permet de tester en local les Servlets que l'on souhaite développer.

JDK
JSDK
Serveur SGBD
Serveur HTTP
Patch extension Servlets (ou servletrunner)
Driver JDBC

FIG. 11.13 – *Outils nécessaires au développement de Servlets*

Pour mettre en place en situation réelle une Servlet il faut bien sûr en plus du JDK et du JSDK pour la compilation et l'exécution, un serveur HTTP qui gère les Servlets (éventuellement à l'aide d'un patch supplémentaire) et un driver JDBC si l'on souhaite accéder à des bases de données (voir fig.11.13 et fig.11.14).

12. Une URL ne peut contenir d'espace. Les browsers acceptent le caractère + en remplacement.

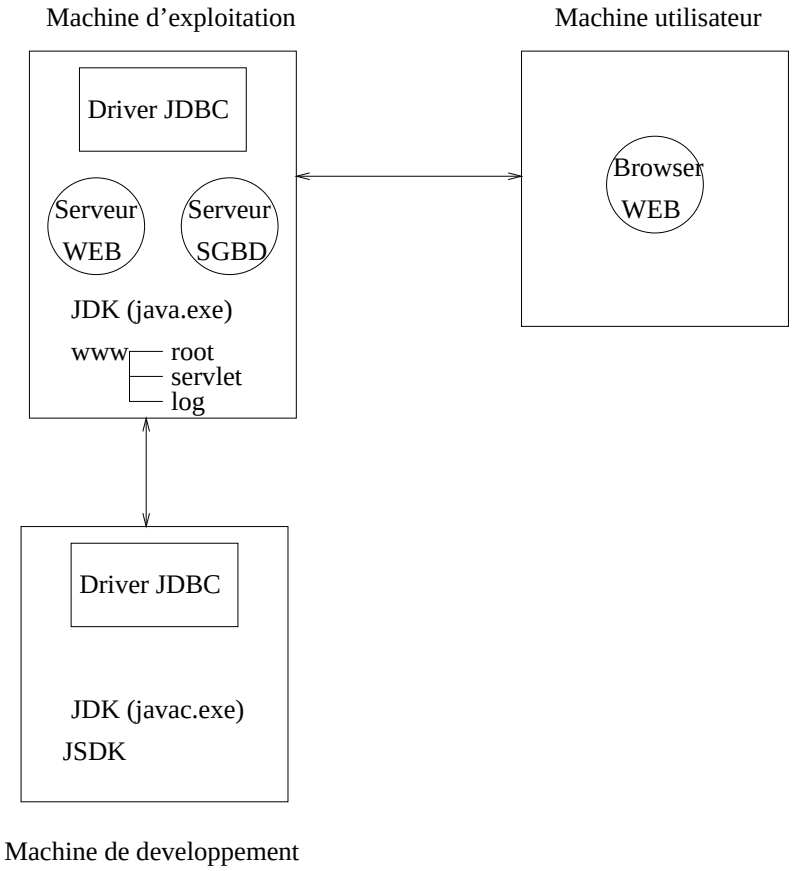


FIG. 11.14 – *Architecture minimale de gestion des Servlets*

11.3.5 Le JSDK

La version sur laquelle nous nous basons est le JSDK 2.1 bien quela majorité des exemples de ce livre se contentent d'une version 1.0

Le JSDK est constitué de 2 packages, l'un décrivant les fonctionnalités des Servlets en général, l'autre décrivant les classes et méthodes des Servlet HTTP.

– package `javax.servlet`

Interfaces	Classes	Exceptions
RequestDispatcher	GenericServlet	ServletException
Servlet	ServletInputStream	UnavailableException
ServletConfig	ServletOutputStream	
ServletContext		
ServletRequest		
ServletResponse		
SingleThreadModel		

– package `javax.servlet.http`

Interfaces	Classes	Exceptions
HttpServletRequest	Cookie	
HttpServletResponse	**HttpServlet**	
HttpSession	HttpSessionBindingEvent	
HttpSessionBindingListener	HttpUtils	
HttpSessionContext		

Les classes indiquées en gras sont celles utilisées dans ce livre. On pourrait sans doute considérer que ce sont les premières à connaître dans le `JSDK`.

11.3.6 Publier sur le WEB l'annuaire de sa société

Afin d'illustrer la manière de développer un lien WEB - SGBD avec des Servlets, nous traitons ici sous une forme rudimentaire, le problème de la publication de l'annuaire d'une société avec recherche par nom de personne. Nous nous limiterons à une table contenant les champs : `nom`, `prenom`, `sexe`, `fonction`, `bureau`, `adresse`, `téléphone`. Le client doit pouvoir rechercher les coordonnées d'une personne par son nom (ou les premières lettres du nom). On remarquera que le code JDBC nécessaire aux deux approches est strictement identique.

Comme nous l'avons dit précédemment, l'approche Servlet se fait en trois parties : La page HTML qui présente le formulaire d'interrogation, la Servlet qui effectue la requête et la page résultat générée par la Servlet.

```
<!--   annuaire.html  -->
<HTML>
<HEAD>
   <TITLE>Annuaire Tartampion</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF" <CENTER>

<CENTER><H1>Annuaire de la société Tartampion</H1></CENTER>

<HR WIDTH="75%">
<CENTER>
<H2>recherche de coordonnées</H2></CENTER>

<P>Tapez les premières lettres de la personne recherchée:

<P><FORM METHOD=POST ACTION=http://127.0.0.1/servlet/annuaire method=POST>
<INPUT TYPE=TEXT NAME="nom" SIZE=10 MAXLENGHT=20 VALUE="">

<P><INPUT TYPE=SUBMIT NAME="go" VALUE="Rechercher">
<INPUT TYPE=RESET   NAME="reset" VALUE="Reset"></FORM>
</BODY>
</HTML>
```

FIG. 11.15 – *formulaire annuaire.html*

```
/* servlet de recherche : annuaire.java */

import java.io.*;
import java.sql.*;           // pour JDBC
import javax.servlet.*;      // pour les servlets
import javax.servlet.http.*;

public class annuaire extends HttpServlet
{
  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
      res.setContentType("text/html");

      PrintWriter out = res.getWriter();
      out.println("<HEAD><TITLE>Réponse annuaire </TITLE></HEAD><BODY>");
      out.println("<CENTER><h1>Voici la réponse à votre requête </h1></CENTER>");
      out.println("<HR WIDTH=75%>");
      try
        {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
          String url = "jdbc:odbc:rjdemo";
          java.sql.Connection c = DriverManager.getConnection(url,
                                                  "MATHIEUP", "ukvg");

          java.sql.Statement st = c.createStatement();

          java.sql.ResultSet rs = st.executeQuery(
                    "select * from table1 where nom like '" +
                                  req.getParameter("nom") + "%'" );

          rs.next();

          if (rs.getString("sexe").equals("M")) out.print("<P><B>Monsieur </B>");
          else out.println("<P><B>Madame </B>");
          out.println(rs.getString("prenom") + " " + rs.getString("nom") );

          out.println("<P><P><B>Fonction  : </B>" + rs.getString(5) );
          out.println("<P><P><B>Adresse   : </B>" + rs.getString(6) );
          out.println("<P><P><B>Téléphone : </B>" + rs.getString(7) );
        }
      catch (SQLException e)
        {
          out.println("Visiblement cette personne n'existe pas !");
        }

      out.println("<P><P>Cliquer <A href=annuaire.html>ici</A> " +
                        "pour revenir à la page précédente");

      out.println("</BODY>");
      out.close();
    }
}
```

On peut voir le résultat de cette approche sur la trace d'exécution du formulaire HTML (fig 11.15) et de la Servlet Java (fig 11.16).

Quand peu de requêtes sont faites simultanément, la Servlet précédente ne pose aucun problème. Si on l'étudie plus finement, on se rend rapidement compte que seule la connexion au SGBD prend

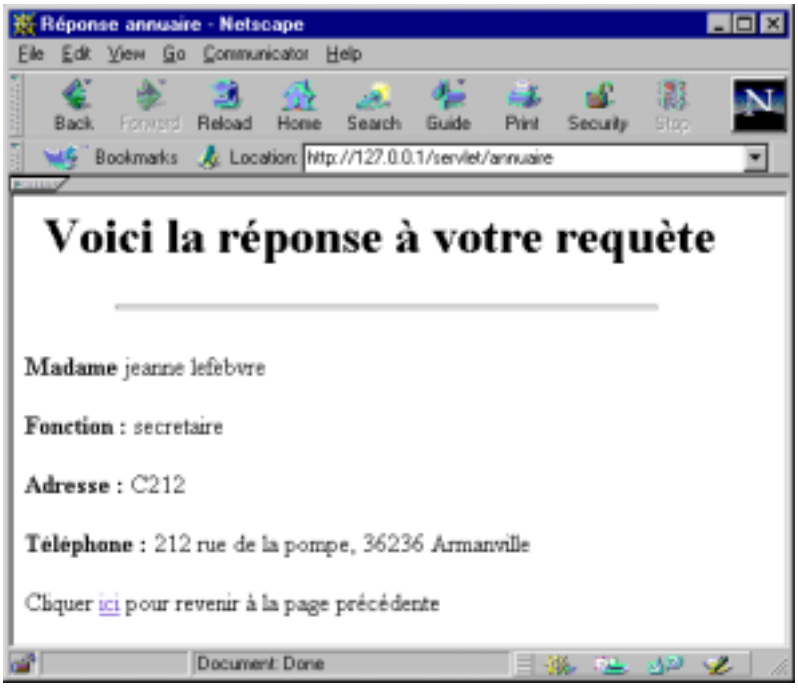


FIG. 11.16 – *Servlet annuaire.java*

du temps. Comme la connexion est placée dans la méthode de service, elles est donc effectuée à chaque rechargement de la Servlet. Si cette Servlet est destinée à être appelée de nombreuses fois simultanément, cette approche est alors pénalisante. On a vu précédemment que le cycle de vie de la Servlet consistait à appeler d'abord la méthode `init` puis une ou plusieurs fois les méthodes de service (`doGet` ou `doPost`) et enfin la méthode `destroy`. Il est alors judiscieux de placer la connexion au SGBD dans la méthode `init`. De cette manière la connexion ne sera réalisée qu'une seule fois tout au long de la durée de vie de la Servlet. Son exécution sera grandement accélérée lors d'accès multiples. Notre Servlet `annuaire` devient alors :

```
/* servlet de recherche : annuaire.java amélioré */

import java.io.*;
import java.sql.*;           // pour JDBC
import javax.servlet.*;      // pour les servlets
import javax.servlet.http.*;

public class annuaire extends HttpServlet
{ private java.sql.Connection c=null;

  public void init(ServletConfig config) throws ServletException
  {
    super.init(config);
    try
    {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
      String url = "jdbc:odbc:rjdemo";
      c = DriverManager.getConnection(url,"MATHIEUP", "ukvg");
    }
    catch (ClassNotFoundException e)
    { throw new BaseIndisponibleException(this, "driver non trouvé");
    catch (SQLException e)
    { throw new BaseIndisponibleException(this, "connexion impossible");
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
      res.setContentType("text/html");

      PrintWriter out = res.getWriter();
      out.println("<HEAD><TITLE>Réponse annuaire </TITLE></HEAD><BODY>");
      out.println("<CENTER><h1>Voici la réponse à votre requête </h1></CENTER>");
      out.println("<HR WIDTH=75%>");
      try
        {
          java.sql.Statement st = c.createStatement();

          java.sql.ResultSet rs = st.executeQuery(
                    "select * from table1 where nom like '" +
                                  req.getParameter("nom") + "%'" );

          rs.next();

          if (rs.getString("sexe").equals("M")) out.print("<P><B>Monsieur </B>");
          else out.println("<P><B>Madame </B>");
          out.println(rs.getString("prenom") + " " + rs.getString("nom") );

          out.println("<P><P><B>Fonction  : </B>" + rs.getString(5) );
          out.println("<P><P><B>Adresse   : </B>" + rs.getString(6) );
          out.println("<P><P><B>Téléphone : </B>" + rs.getString(7) );
        }
      catch (SQLException e)
        {
          out.println("Visiblement cette personne n'existe pas !");
        }

      out.println("<P><P>Cliquer <A href=annuaire.html>ici</A> " +
                        "pour revenir à la page précédente");

      out.println("</BODY>");
      out.close();
    }
}
```

Chapitre 12

Les serveurs d'applications

12.1 Le suivi de session

Imaginons que nous souhaitions construire une page qui indique à l'aide d'un compteur le nombre de fois où elle a été chargée dans le browser du client. La Servlet nécessaire à ce traitement pourrait s'écrire de la manière suivante :

```
// Cpt1.java

import java.io.*;
import javax.servlet.*;      // pour les servlets
import javax.servlet.http.*;

public class Cpt1 extends HttpServlet
{ int cpt=0;

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        res.setContentType("text/html");

        PrintWriter out = res.getWriter();

        cpt++;
        out.println("<head> <title>Implémenter un compteur</title> </head>");
        out.println("<body>");
        out.println("<h1> La valeur du compteur est : "+ cpt + "</h1>");
        out.println("</body>");
    }
}
```

Le problème vient du fait qu'une Servlet est un Thread et qu'elle est persistante dans le serveur WEB après l'exécution de la requête. L'attribut `cpt` est donc unique dans le serveur. Il indique donc le nombre total d'accès à cette page quels que soient les clients.

Si l'on souhaite maintenant avoir un compteur pour chaque client, il faut un dispositif permettant de créer un attribut particulier pour chaque client. Ce dispositif c'est le suivi de session. Le JSDK contient un objet `HTTPSession` qui permet d'automatiser cette tâche. La session est un dictionnaire rangé dans le serveur dans lequel on peut ajouter et accéder à des objets par des clés. Les méthodes principales pour la manipulation de cet objet sont :

`getSession`, méthode de l'objet `HTTPServletRequest` qui renvoie l'objet `HttpSession` du serveur. S'il n'esxistait pas, il est alors créé.

`getValue`, méthode de l'objet `HttpSession` qui permet de ranger un objet identifié par une clé dans la session.
`putValue`, méthode de l'objet `HttpSession` qui permet de récupérer un objet identifié par une clé dans la session.
`invalidate`, méthode de l'objet `HttpSession` qui permet de détruire la session.

La Servlet d'implémentation d'un compteur propre à chaque client peut donc s'écrire de la manière suivante:

```
// Cpt2.java

import java.io.*;
import javax.servlet.*;      // pour les servlets
import javax.servlet.http.*;

public class Cpt2 extends HttpServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        HttpSession session = req.getSession( true );
        Integer cpt = (Integer)session.getValue( "compteur" );
        cpt = new Integer( cpt == null ? 1 : cpt.intValue() + 1 );
        session.putValue( "compteur", cpt );


        res.setContentType("text/html");

        PrintWriter out = res.getWriter();

        out.println("<head> <title>Implémenter un compteur</title> </head>");
        out.println("<body>");
        out.println("<h1> La valeur du compteur est : "+ cpt + "</h1>");
        out.println("</body>");
    }
}
```

Il est à noter que les sessions sont en général implémentées à l'aide de Cookies créés sur chaque poste client et cotenant un identifiant pour l'utilisateur. Quand le client accède à la page, le serveur lit l'identifiant du client rangé dans son Cookie, et lit ensuite l'attribut propre à ce client grâce à l'identifiant. Rappelons que HTTP est un protocole non connecté et que donc il n'y a aucun lien permanent entre le serveur et le client. Le seveur n'a donc pas d'autre moyen que les Cookies pour savoir si le client est déjà venu sur le serveur ou pas.

12.2 Les Java Server Pages: JSP

Les Java Server Pages, plus communément appelées JSP ont été introduites récemment (la version 1.0 est sortie le 29 avril 1999) par SUN pour fournir une alternative Java aux Active Server Pages (ASP) de Microsoft. L'analyseur de JSP doit être installé sur un serveur HTTP. Le principe est simple: si dans les Servlets on écrit du code Java qui contient du HTML, avec les JSP c'est le contraire: on écrit du HTML avec des tags spéciaux pour des appels à des méthodes Java. Comme vous pouvez ne pas utiliser ces tags spéciaux, toute page HTML classique est donc une JSP à l'extension près. Une page JSP se reconnaît avant tout à son extension `.jsp`.

Exemple: Une page HTML classique est aussi une page JSP

```
<!-- trivial.jsp  -->
<html>
<head> </head>
<body>
   <h1>Test de page JSP</h1>
</body>
</html>
```

Afin de décrire le code à intégrer dans la page, les JSP offrent des éléments de script qui permettent de préciser les actions Java à effectuer. Il y a 4 types d'éléments : les directives qui permettent d'envoyer des messages à l'analyseur de pages, les déclarations qui permettent de définir des attributs et méthodes de la Servlet, les scriptlets qui sont des fragments de code Java à exécuter et enfin les expressions qui sont évaluées et remplacées par une chaîne de caractère émise dans le flux de sortie.

La syntaxe à utiliser est différente pour chacun de ces éléments

```
<%@ pour les directives   %>+
<%= pour les expressions  %>
<%  pour le scriptlets    %>
<%! pour les déclarations %>
```

12.2.1 Les JSP directives

Les directives JSP sont des messages à destination de l'analyseur de la page. Elles s'écrivent avec la syntaxe suivante : `<%@ directive... %>`

Les différentes directives disponibles sont : page (infos sur la page), include (librairies à ajouter pour l'analyse de la page), taglib (pour nommer de nouveaux tags).

12.2.2 Les JSP expressions

Contrairement au code Java habituel, l'expression ne doit pas être terminée par un point-virgule. Les expressions sont évaluées et remplacées par leur résultat. Ce sont en général des méthodes qui renvoient un type String.

Voici un exemple très simple de JSP qui fait appel à la date et heure du moment de l'exécution :

```
<!--  date.jsp  -->
<HTML>
<HEAD>
        <TITLE>Affichage de la date avec une JSP</TITLE>
        <%@ import="java.util.Date" %>
</HEAD>
<BODY>

<H1>Accès à la date à l'aide de JSP</H1>

La date du jour est <%= new Date() %>.

</BODY>
</HTML>
```

Il ne faut néanmoins pas confondre écriture et exécution. Si les JSP semblent différents des Servlets il n'en est rien. Une JSP est tout d'abord prétraitée par le serveur qui construit une Servlet équivalente si elle n'existait pas déjà. Cette Servlet est ensuite compilée puis exécutée (fig. 12.1). contrairement à ce que l'on pourrait parfois croire, une JSP est une Servlet à l'exécution !

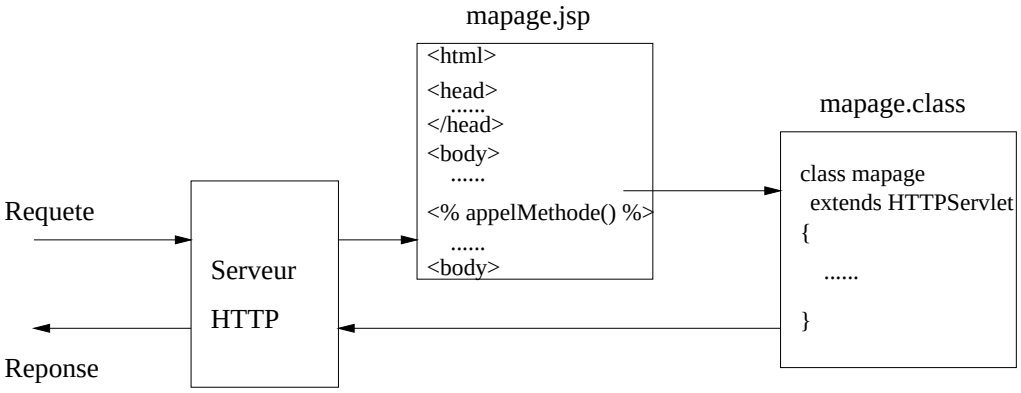


FIG. 12.1 – *Exécution d'une JSP*

A l'appel de la page `essai.jsp` celle ci est transformée par le serveur en une Servlet `essai.java` qui est ensuite compilée pour fournir `essai.class` qui est ensuite exécutée pa le serveur pour fournir la page générée sur le poste client. Ensuite, la Servlet reste en vie en dehors de la requête[1] et cela tant que le serveur le souhaite. C'est pourquoi on dit parfois qu'une Servlet est persistante.

12.2.3 Les JSP scriptlets

Les scriptlets permettent de générer du code Java dans la Servlet. Dès qu'une structure de contrôle (If, For,While) est nécessaire, une scriptlet doit être écrite dans la JSP entre les tags <% et %>[2]

Les scriptlets ont accès aux variables suivantes :

`request` pour l'objet `javax.servlet.http.HttpServletRequest` de la Servlet.

`response` pour l'objet `javax.servlet.http.HttpServletResponse` de la Servlet.

`out` pour l'objet `java.io.PrintWriter` de la Servlet.

`in` pour l'objet `java.io.BufferedReader` de la Servlet.

On pourra écrire par exemple :

```
<%
String sexe = request.getParameter( "Sexe" );
String nom  = request.getParameter( "Nom" );
if ( sexe.equals("M"))
   out.println( "Bonjour Monsieur " + nom );
else
   out.println( "Bonjour Madame " + nom );
%>
```

Il est possible d'effectuer des déclarations de variables dans une scriptlet. Dans ce cas, la variable est locale à la méthode de service de la Servlet. Elle sera donc réallouée à chaque invocation de cette méthode (par exemple si le client appuie sur le bouton `Reload` de son Browser).

1. Nous rappelons que HTTP est un mode non connecté
2. Certains serveurs d'applications utilisent encore la notation avant normalisation qui utilisait le tag ¡SCRIPT¿.

12.2.4 Les JSP déclarations

Les déclarations permettent de définir les attributs et méthodes de la Servlet qui sera générée. Attention, les objets déclarés sont partagés par tous les utilisateurs.

Contrairement à une déclaration dans une scriptlet, une variable déclarée ici est globale à la Servlet. Elle n'est donc allouée qu'une seule fois, à la création de cette Servlet.

```
<html>
<head>
    <%@ import="java.util.*" %>
</head>
<body>
<h1>Test de page JSP</h1>

<%!
// déclaration du vecteur et d'une méthode d'ajout
private Vector v=new Vector();
private void ajouter(String s){v.addElement(s);}
%>

<% // remplissage du vecteur
ajouter("pierre");
ajouter("paul");
// ou encore
v.addElement("jean");
%>

<%
// affichage du vecteur
for (int i=0;i<v.size();i++)
        out.println((String)v.elementAt(i));
%>

</body>
</html>
```

Si nous reprenons la création d'une page d'affichage de compteur, une solution au problème peut s'écrire de la manière suivante :

```
<!--   cpt.jsp   -->
<html>
<head>
<title>Implémenter un compteur avec une JSP</title>
        <%@ import="java.util.* %>
</head>
<body>

<%!
    private int val=0;
    public String getCpt() {return "" + val; }
    public void incr() {val++;}
%>

<h1> La valeur du compteur est  <%= getCpt() %>  </h1>
<% incr(); %>
</body>
</html>
```

Attention, l'attribut `val` comme nous l'avons précisé, est global à la Servlet qui est elle-même unique dans le serveur. Il n'y a donc qu'un compteur pour tous les clients.

12.2.5 Les JSP Beans

Bien sûr, pour que l'approche JSP soit intéressante, le code Java inclus dans la page HTML ne doit pas être trop important; juste quelques appels de méthodes. De plus, il existe bien souvent plusieurs applications dans l'entreprise qui ont les mêmes besoins d'accès à la base. Chacune risque de réécrire dans ses Servlets le même code Java. Extraire ce code Java de la Servlet et le placer dans un objet à part permet donc de simplifier l'écriture de plusieurs applications puisqu'il n'y a alors plus qu'à écrire des appels de méthodes adéquates pour réaliser une Servlet, une Applet ou tout autre type d'application. Ces composants logiciels qui contiennent la logique de l'entreprise et permettent de simplifier le développement d'applications sont appelés Enterprise Java Beans ou plus communément EJB.

Le Bean contient la logique de l'entreprise. Il peut être utilisé par d'autres Beans, des applications centralisées classiques, des applications Client-Serveur spécialisées ou comme ici dans des Servlets à destination d'un serveur HTTP.

Dans le cas de Servlets le Bean peut par exemple être utilisé pour stocker des informations nécessaires à la session HTTP, pour passer des informations d'une page à une autre ou des informations sur la requête en cours de calcul, ou encore pour masquer des phases complexes de calcul.

Le problème principal des Beans est la persistance. Un composant logiciel doit pouvoir exister durant toute la session de l'utilisateur et pas seulement durant le chargement d'une page. Il lui faut donc un serveur particulier pour l'héberger et gérer sa persistance. Les autres applications invoquent alors les méthodes de ces objets à distance à travers des protocoles comme RMI (Remote Protocol Invocation) ou Corba. A la première invocation, le Bean est lancé sur le serveur d'objets, les applications qui en ont ensuite besoin peuvent exécuter ses méthodes par simple requête à ce serveur.

Dans un tel cas, l'architecture logicielle ne relie plus simplement un serveur HTTP à un serveur de bases de données dans une architecture 2-Tier (Fig. 12.2) mais un serveur HTTP à un serveur d'objets distribués qui est lui même relié à un serveur de bases de données. On parle alors dans ce cas d'architecture 3-Tier[3] (Fig. 12.3). Comme rien n'empêche d'avoir plusieurs serveurs d'objets

3. Nous reprenons ici le terme Anglais Tier qui signifie Partie et qui est maintenant couramment utilisé.

distribués dans l'entreprise (un par spécialité ou secteur de l'entreprise par exemple) on parlera même parfois d'architecture n-Tiers.

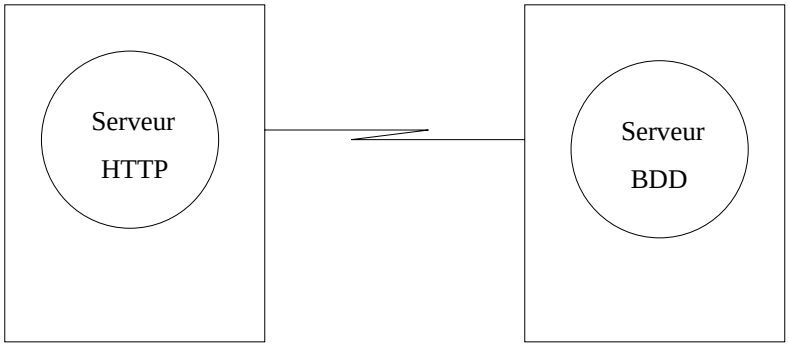


FIG. 12.2 – *Architecture 2-Tier*

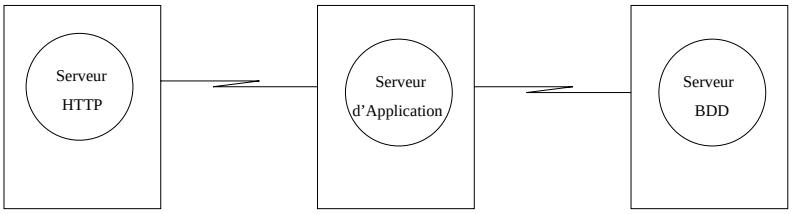


FIG. 12.3 – *Architecture 3-Tier*

S'il n'est pas très difficile de réaliser soi-même un serveur d'objets distribués avec RMI, des solutions logicielles complètes prenant en compte ce type de serveur apparaissent de plus en plus. Elles sont regroupées sur le vocable "serveurs d'applications". La réalisation de serveurs d'objets distribués sort du cadre de ce livre, revenons donc à l'utilisation des Beans dans une page JSP.

La syntaxe à utiliser pour le chargement des Beans est la suivante :

```
<BEAN name="lookup name" varname="alternate variable name"
type="class" introspect="yes or no" beanName="file name"
create="yes or no" scope="request or session">
<PARAM name="name" value="value">
.
.
.
</BEAN>
```

La signification des différents paramètres est la suivante :

- `name` nom de l'instance du Bean dans la Servlet, obligatoire.
- `varname` synonyme du nom du Bean dans la JSP, optionnel (par défaut identique à `name`).
- `type` nom de la classe définissant le Bean, obligatoire.
- `introspect` indique si l'introspection doit être exécutée sur ce Bean, optionnel (par défaut yes).
- `create` indique si le Bean doit être créé s'il n'existe pas déjà dans la cession, optionnel (par défaut yes). Si `create="yes"` est indiqué dans la JSP il est inutile de faire un `new` pour crér l'objet. C'est le serveur qui s'en charge.

- `scope` précise la durée de vie du Bean. Si c'est "request" le Bean meurt à la fin de la requête, si c'est session il meurt à la fin de la cession, optionnel (par défaut request)
- `beanName` spécifie l'accès au fichier de définition du Bean. Optionnel, par défaut il est recherché dans le classpath).

Pour accéder à l'une des méthodes d'un Bean, on invoque simplement cette méthode à partir d'une JSP expression ou une JSP scriptlet. Par exemple ,

```
<BEAN name="t"
   type="test.Foo" introspect="no"
   create="yes" scope="request">
</BEAN>

<%
   // utilisation du bean
   out.println(t.toString());
%>
```

Attention, bien que le nom fixé dans l'attribut `name` semble propre à la page JSP il n'en est rien! Ce nom est uilisé comme clé pour ranger l'identifiant du Bean dans la session. La session étant unique pour le serveur, il n'est pas possible d'avoir plusieurs clés. On ne peut donc pas utiliser dans une autre JSP page le même nom interne d'objet pour une classe différente.

Si l'on reprends l'implémentation d'une page affichant un compteur qui indique le nombre de fois où le client a accédé à la page. Nous pouvons cette fois l'écrire à l'aide de Beans avec les deux programmes suivants. Le premier qui implémente l'objet `Compteur` en Java et le second qui décrit une JSP utilisant ce Bean.

```
// Compteur.java
package test;

public class Compteur
{
    private int val=0;
    public String getCpt() {return "" + val; }
    public void incr() {val++;}
}
```

```
<!--   compteur.jsp   -->
<html>
<head>
<title>Implémenter un compteur</title>
</head>
<body>

<bean name="cpt" type="test.Compteur" create="yes"
scope="session" introspect="yes">
<param name="word" value="servlets">
</bean>

<h1> La valeur du compteur est  <%= cpt.getCpt() %>  </h1>
<% cpt.incr() %>
</body>
</html>
```

12.2.6 L'annuaire revisité

Cmme dans le chapitre précédent, reprenons l'application Annuaire mais cette fois-ci ave Beans et JSP. La page HTML de départ reste identique. Nous créons ensuite un Bean capable d'aller rechercher une personne et offrant des méthodes d'interrogation. Nous terminons enfin par la JSP qui utilise ce Bean.

```
<!--   annuaire2.html  -->
<HTML>
<HEAD>
   <TITLE>Annuaire Tartampion</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF" <CENTER>

<CENTER><H1>Annuaire de la société Tartampion</H1></CENTER>

<HR WIDTH="75%">
<CENTER>
<H2>recherche de coordonnées</H2></CENTER>

<P>Tapez les premières lettres de la personne recherchée:

<P><FORM METHOD=POST ACTION=http://127.0.0.1/annuaire2.jsp method=POST>
<INPUT TYPE=TEXT NAME="nom" SIZE=10 MAXLENGHT=20 VALUE="">

<P><INPUT TYPE=SUBMIT NAME="go" VALUE="Rechercher">
<INPUT TYPE=RESET   NAME="reset" VALUE="Reset"></FORM>
</BODY>
</HTML>
```

```
// Personne.java
package test;

import java.sql.*;           // pour JDBC


public class Personne
{
    private String nom, prenom, sexe, fonction, adr, tel;

    public boolean rechPersonne(String nomPers)
    {
        try
            {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
                String url = "jdbc:odbc:annuaire";
                java.sql.Connection c = DriverManager.getConnection(url,
                                                    "MATHIEUP", "ukvg");

                java.sql.Statement st = c.createStatement();

                java.sql.ResultSet rs = st.executeQuery(
                   "select * from annuaire where nom like '" + nomPers + "%'" );

                rs.next();
                nom      = rs.getString("nom");
                prenom   = rs.getString("prenom");
                sexe     = rs.getString("sexe");
                fonction = rs.getString(4);
                adr      = rs.getString(5);
                tel      = ""+rs.getInt(6);
                return true;
            }
        catch (Exception e)
            {
                System.out.println(e.getMessage());
                return false;
            }

    }

    public String getIntitule()
    {
        if (sexe.equals("M")) return "Monsieur";
        else return "Madame";
    }
    public String getNom()     {return nom;}
    public String getPrenom()  {return prenom;}
    public String getFonction(){return fonction;}
    public String getAdresse() {return adr;}
    public String getTel()     {return ""+tel;}
}
```

```
<!-- annuaire2.jsp -->

      <HEAD><TITLE>Réponse annuaire </TITLE></HEAD><BODY>

<bean name="p" type="test.Personne" create="yes" scope="session"
introspect="yes">
<param name="word" value="servlets">
</bean>

      <CENTER><h1>Voici la réponse à votre requête </h1></CENTER>
      <HR WIDTH=75%>
      <%  if (p.rechPersonne(request.getParameter("nom"))) { %>
          <P><P><B><%= p.getIntitule()%></B>
          <%= p.getPrenom()%> <%= p.getNom()%>
          <P><P><B>Fonction   : </B> <%= p.getFonction()%>
          <P><P><B>Adresse    : </B> <%= p.getAdresse()%>
          <P><P><B>Téléphone  : </B> <%= p.getTel()%>
      <% } else {%>
          Visiblement cette personne n'existe pas !
      <% } %>
          <P><P>Cliquer <A href=annuaire2.html>ici</A>
          pour revenir à la page précédente
      </BODY></html>
```

12.3 Les serveurs d'applications

Le serveur d'application intègre dans un ensemble de développement l'ensemble des 3 parties d'une architecture 3-tier : la partie pour communiquer avec le système de l'entreprise (back-end system), la partie pour communiquer avec le système client (front-end system) qui est généralement mais pas nécessairement un client WEB, la partie qui intègre les composants logiciels contenant la logique du système d'information. Le résultat est un ensemble logiciel suffisamment adaptable, modulaire, robuste et dynamique permettant de répondre aux problèmes actuels des entreprises.

La technologie Java, du fait de sa simplicité, sa puissance et sa portabilité prend une place de plus en plus importante dans ce genre d'architecture. Si les premiers serveurs d'applications étaient tournés vers C ou Perl, la plupart d'entre-eux se tournent actuellement vers le langage Java.

Tout au long de ces pages nous avons décrit plusieurs outils Java permettant de connecter efficacement une base de données à un serveur HTTP : Les Servlets, les JSP, les EJB. Tout ceci nécessite plusieurs serveurs qui interagissent ensemble pour répondre à la requête du client : un serveur HTTP pour les pages HTML, un serveur de Servlets, un analyseur de JSP et un serveur d'objets distribués. Les ensembles logiciels qui offrent une solution logicielle globale à ces différents besoins sont appelés "serveurs d'applications".

Actuellement de plus en plus de sociétés proposent des serveurs d'applications :

- Forte WebEnterprise
  `http://www.forte.com/product/enterprise/index.htm`
- GemStone Systems GemStone/J
  `http://www.gemstone.com/products/j/main.html`
- IBM WebSphere Application Server
  `http://www.software.ibm.com/webservers/appserv/`
- Netscape Application Server
  `http://www.netscape.com/appserver/v2.1/index.html`
- New Atlanta ServletExec 2.0
  `http://www.newatlanta.com/products.html`

- SilverStream SilverStream Version 2.0
  `http://www.silverstream.com/website/silverstream/pages/products_f.html`

Annexe A

Installation d'un environnement pour les Servlets

A.1 Installation pour la compilation

Avant toutes choses, vous devez avoir installé le JDK (Java Development Kit) et le JSDK (Java Servlet Development Kit) puis avoir configuré correctement les variables d'environnement.

Par exemple, si le JDK est installé dans le répertoire `c:\jdk1.2` vous devez ajouter dans l'`autoexec.bat` la ligne `set PATH = %PATH%;c:\jdk1.2\bin;`.

Ouvrez une fenêtre de commande (Démarrer/Executer/command) et tapez `java -version` Si tout est bien installé, vous devez voir apparaître le numéro de version de votre JDK.

maintenant, il faut configurer l'accès au JSDK pour permettre l'accès aux classes Java propres aux Servlets. Si il est installé dans le répertoire `c:\jsdk2.0`, vous devez ajouter dans l'`autoexec.bat` la ligne `set CLASSPATH = %CLASSPATH%;c:\jsdk2.0\lib\jsdk.jar`.

A.2 Installation des serveurs

Une hiérarchie pour serveur HTTP contient au minimum un répertoire pour stocker les pages HTML, un répertoire pour stocker les Servlets et un répertoire pour stocker les informations sur les anciennes connexions. Nous les nommerons respectivement `root`, `servlet` et `log`. Créez par exemple à la racine de votre répertoire la hiérarchie suivante :

```
www---- root
    |
     -- servlet
    |
     -- log
```

Actuellement, très peu de serveurs HTTP sont capables de gérer directement des Servlets. Il faut en général ajouter un patch d'extension à cet effet. Pour notre part, nous utiliserons les outils les plus simples que l'on puisse trouver : un mini serveur HTTP, `TinyWeb` (43K) qui ne comprend rien aux Servlets, et le serveur `servletrunner` fourni par SUN avec le JSDK qui ne comprend que les Servlets (voir fig.11.13 et fig.11.14).

Créez un raccourci sur `tiny.exe`. TinyWeb accepte plusieurs paramètres au lancement. le premier précise le nom du répertoire contenant les pages WEB et le second précise le numéro de port du serveur. Avec le bouton droit de la souris accédez au menu `propriétés` du raccourci et spécifiez les paramètres adéquats :

```
Cible:        c:\tiny.exe c:\www\root 8080
démarrer en:  c:\www\log
```

Créez un raccourci sur `servletrunner`. l'option `-d` permet de préciser le nom du répertoire contenant les Servlets, l'option `-p` précise le numéro de port de ce serveur. Avec le bouton droit de la souris accédez au menu `propriétés` du raccourci et spécifiez les paramètres adéquats:

```
Cible:  c:\jsdk2.0\bin\servletrunner -d c:\www\servlet -p 8090
```

Avec notre configuration, l'accès à une page HTML se fera à l'URL

`http://127.0.0.1:8080/page.html`

et l'accès à une Servlet se fera à l'URL

`http://127.0.0.1:8090/servlet/maservlet`

Remarque: Le numéro de port par défaut d'un serveur HTTP est 80. dans ce cas, l'usage du numéro dans l'URL est facultatif. De même, lorsque les pages appelées sont au même endroit que le serveur il est inutile de préciser l'adresse IP. En situation réelle, tout ceci ne serait donc pas indiqué.

L'usage de deux serveurs, l'un pour le WEB, l'autre pour les Servlets n'est pas utile en général. les serveurs HTTP intègrent (par l'intermédiaire de patchs) l'accès aux Servlets sans passer par `servletrunner`. c'est le cas par exemple avec le serveur `Apache` et le patch additionnel `ServletExec`. A terme, les serveurs HTTP intègreront directement cette technologie comme c'est déjà le cas pour le serveur `Jigsaw`. L'avantage de notre solution est qu'elle est très petite et entièrement gratuite!

A.3 Test de la configuration

Créez la page `first.html` suivante et placez la dans le répertoire `\www\root`.

```
<HTML>
<HEAD>
   <TITLE>first</TITLE>
</HEAD>
<BODY>
Voici ma page
<P>et l'appel de ma <A HREF="http://127.0.0.1:8090/servlet/first">servlet</A>
<P>c'est fini !
<BR>
</BODY>
</HTML>
```

Créez le programme `first.java` suivant et placez le fichier `first.class` dans le répertoire `\www\servlet`.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class first extends HttpServlet
{
  public void service( HttpServletRequest req, HttpServletResponse res )
       throws ServletException, IOException
  {
    ServletOutputStream out = res.getOutputStream();

    res.setContentType( "text/html" );
```

```
    out.println( "<head><title>servlet first</title></head><body><center>" );
    out.println( "<h1>Test de ma Servlet</h1>" );
    out.println( "</center>" );
    out.println( "<P> Retour vers  ma ");
    out.println( "<A HREF="http://127.0.0.1:8080/first.html">page</A>");
    out.println( "</body>" );
  }
}
```

Si tout a bien fonctionné, vous êtes fin prêts à tester les Servlets d'accès aux bases de données comme la Servlet `annuaire.java`

Attention : `servletrunner` doit impérativement être arrêté puis relancé si l'on souhaite prendre en compte des modifications effectuées sur des Servlets.

A.4 Quelques adresses

- JDK, `www.javasoft.com`, indispensable pour compiler du code java. Contient JDBC.
- JSDK, `www.jserv.com`, indispensable pour compiler des Servlets.
- TinyWeb, `http://www.ritlabs.com/tinyweb/` le plus petit des serveurs HTTP sur l'Internet. Gratuit.
- Jigsaw, `www.w3c.org`, un serveur HTTP entièrement écrit en Java, avec accès direct aux Servlets. Nécessite une mémoire centrale de 48Mo minimum.
- Apache, `www.apache.org` , le plus connu, le plus utilisé, le plus robuste, mais nécessite un patch comme ServletExec pour exécuter des Servlets.
- PWS, `www.microsoft.com`, serveur HTTP personnel de microsoft. Très facile à installer mais ne fonctionne qu'en 16 bits et necessite un patch comme ServletExec pour exécuter des Servlets.
- ServletExec, `www.newatlanta.com` patch d'extension aux Servlets pour Apache et PWS avec une version d'évaluation gratuite.

Annexe B

Rappels HTML

Le terme HTML a très vite pénétré le marché du grand public tant et si bien que peu de monde connaît maintenant la signification de ce signe. HTML signifie donc *Hyper Text Markup Language*. C'est un langage, au même titre que n'importe quel langage informatique, à ceci prêt qu'il a pour objectif de décrire des pages de textes contenant des liens vers d'autres pages, images ou sons. C'est ce que l'on appelle un hypertexte multimédia. HTML est un langage de balises, tout enrichissement du texte étant inscrit dans le document sous forme de balises elles aussi écrites à l'aide de caractères. Ce langage n'est pas le premier du genre puisqu'il est issu du langage SGML très utilisé dans le monde l'édition dont le langage LaTeX(utilisé pour rédiger ce document) est aussi tiré.

Ce chapitre n'a pas pour objectif de décrire l'ensemble des caractéristiques du langage HTML mais de donner au lecteur les connaissances minimum lui permettant de rédiger une page de lancement de Servlet d'accès à une base de données.

- Objectif : permettre de décrire des présentations multimédia (texte, images, sons, vidéo) avec des liens hypertexte pour l'Internet (Version actuelle HTML 4.0). HTML est en constante évolution.
- Une page HTML s'édite en clair dans un fichier ASCII. Seuls les 128 premiers caractères (7bits) sont autorisés, les caractères spéciaux doivent être écrits avec un code.
- Les retour de lignes se font à l'aide d'un marqueur de paragraphe
- Les caractères espace consécutifs sont ramenés à 1 seul espace
- Editeurs spécialisés : Netscape Composer, Microsoft FrontPage etc ...

B.1 Le langage

HTML est un langage de Tags (balises) qui sont des délimiteurs de début et fin d'action. On trouvera par exemple le tag HTML `<html> </html>`, le tag HEAD `<head> </head>` et le tag BODY `<body> </body>` dans la plupart des pages.

La page Minimum permettant d'afficher des informations est réduite aux tags suivants:

```
<html>
<head>
   <title> ... </title>
</head>
<body>
</body>
</html>
```

Le texte mis dans la section `<title>` n'est pas affiché directement sur la page HTML sur le poste client. Il est utilisé pour nommer la page dans les bookmarks ou la barre d'état des différents Browsers.

B.2 Formatage de texte

Afin de gérer les titres et sous-titres il existe 6 niveaux prédéfinis de titres `<h1>...</h1>` à `<h6>...</h6>`.

De nombreux tags servent à enrichir le texte:
italique `<i>...</i>`
gras `<b>...</b>`
clignotant `<blink>...</blink>`
centré `<center>...</center>`
renforcé `<em>...</em>`
taille d'une police `<font size=n> ... </font>`
couleur d'une police `<font color=red> ... </font>`

D'autres tags permettent d'indenter le texte affiché:
paragraphe `<p>`
retour ligne `<br>`
filet `<hr>`
epaisseur du filet `<hr size=n>`
longueur du filet `<hr align=center width=50%>`

B.3 Exemple de page simple

Voici une page HTML mettant en œuvre quelques uns de ces tags:

```
<html>
<head>
   <title> ma premiere page </title>
</head>
<body>
   <center>
      <h1> <b> Page de cours d'HTML </b> </h1> </center>
   <br>
   <hr width="50%">
   <h2>Le formatage</h2>
       <br> <b>gras</b>
       <br> <i>italique</i>
       <br> <u>souligne</u>
</body>
</html>
```

On peut voir le résultat sur la fig.B.1.

B.4 Listes et énumerations

Listes standard:

```
 <ul>
    <li> item1
    <li> item2
```

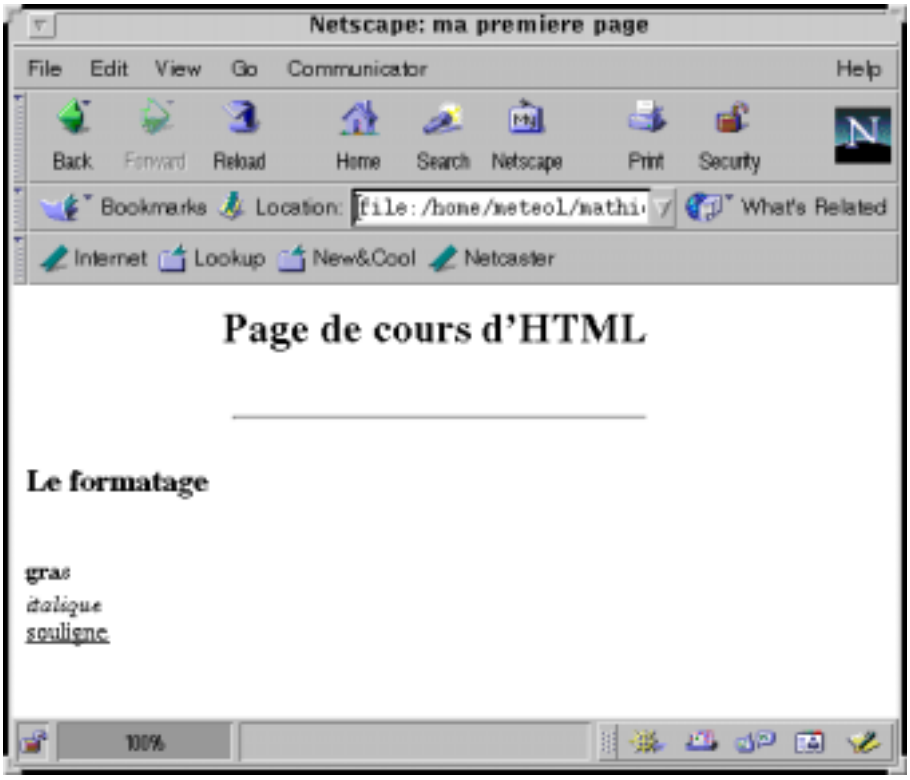


FIG. B.1 – *Page HTML rudimentaire*

```
</ul>
```

Listes numérotées :

```
<ol>
   <li> item1
   <li> item2
</ol>
```

les sous-listes sont obtenues par imbrication de tags `<ul>...</ul>`

B.5 Les accents

Par défaut HTML est un langage qui s'exprime uniquement avec des caractères ASCII. Les accents sont donc prohibés si on souhaite écrire du HTML pur. Dans ce cas, un codage spécial permet de les exprimer. Voici une liste des principales lettres accentuées.

à `&agrave;`
â `&acirc;`
é `&eacute;`
è `&egrave;`
ê `&ecirc;`
î `&icirc;`
ô `&ocirc;`
ù `&ugrave;`

Néanmoins les browsers WEB actuels interprètent directement les caractères latins accentués et il est donc possible de taper directement des lettres accentuées dans les pages HTML. Nous

utiliserons cette politique par la suite.

B.6 Les liens hypertextes

Un lien représente un saut vers un autre endroit. Il est caractérisé par un texte visible par l'utilisateur et une adresse de destination appelée quand on clique sur le lien.

```
<a href="URL complète">texte affiché</a>
```

On peut créer un lien externe vers une autre page du domaine local :

```
<a href="toto.html"> cliquer ici </a>
```

ou même sur un autre serveur :

```
<a href="http://www.amazon.com"> chez Amazon </a>
```

Le lien peut être un simple texte ou même une photo (tag `<img>`).
On peut voir le résultat sur la fig.B.2.

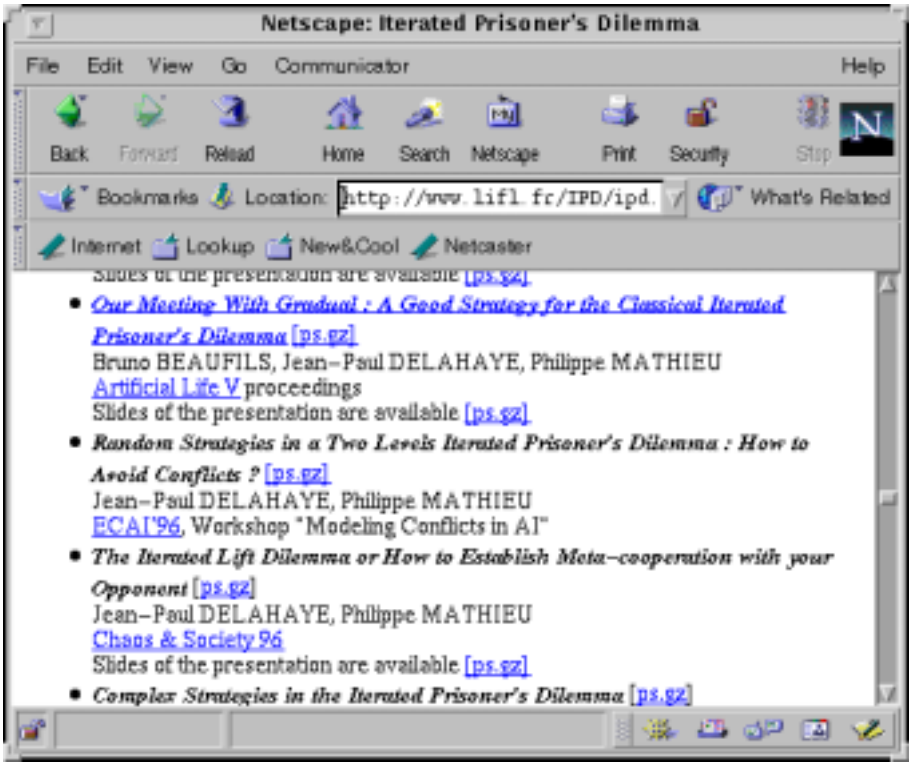


FIG. B.2 – *Page HTML avec liens*

L'attribut `target` dans le lien permet de spécifier la fenêtre de destination de la page à charger. Ceci est notamment utile dans les sites qui contiennent des frames puisque cet attribut permet de préciser dans quelle sous-fenêtre de la frame la page sera chargée. On notera que si la chaîne spécifiée dans l'attribut `target` n'est pas connue dans les pages, le Browser créera une nouvelle instance du Browser pour afficher la nouvelle page.

B.7 Les ancres

Le lien peut pointer sur une autre page HTML mais aussi sur un endroit précis de la même page (une ancre).

définition d'une ancre `<a name="nomAncre">`
lien interne vers l'ancre de la même page `<a href="#nomAncre"> cliquer ici </a>`
Un lien externe peut aussi contenir une ancre vers un endroit précis de l'autre document:

```
<a href="http://www.amazon.com/#nomAncre">cliquer ici</a>
```

B.8 Rappel URL

Une URL permet d'identifier précisément une ressource sur l'internet
forme: `ressource://host.domaine:port/pathname`
ressource: file, http, news, gopher, telnet, wais, mailto
Un lien peut donc pointer sur le composeur de mail `<a href=mailto:mathieu@lifl.fr>courrier PM</a>`
ou sur la récupération d'un fichier par ftp `<a href=ftp://ftp.lifl.fr/soft/prison.zip>un soft</a>`
Par défaut, le serveur HTTP est placé sur le port 80, la machine locale se nomme `localhost` et est accessible via l'adresse IP `127.0.0.1`

B.9 Les images

Formats standard reconnus: GIF ou JPEG

```
<img src="nomImage">

<img alt="texte" src="nomImage">

<img src="nomImage" width=100 height=100>
```

Attention: Les images sont en général de grosse taille et donc lentes à charger par l'utilisateur

Autres formats: AVI pour les vidéo, WAV pour le son

B.10 Les tableaux

le tableau est délimité par `<table>...</table>` les lignes sont délimitées par `<tr>...</tr>` les colonnes sont délimitées par `<td>`
La taille des cases est fixée par leur contenu
Exemple

```
<table border=1 cols=3>
    <tr> <td> l1c1 <td> l1c2 <td> l1c3 </tr>
    <tr> <td> l2c1 <td> l2c2 <td> l2c3 </tr>
</table>
```

On peut voir le résultat sur la fig.B.3.

B.11 Les frames

Les Frames permettent de diviser une page HTML en plusieurs fenêtres indépendantes. Généralement l'un des Frames contient l'index avec les liens sur les pages concernées, tandis que l'autre présente la page en cours d'affichage.
Page principale de définition des Frames (sans tag `<body>`)

```
<html>
   <head>
      <title>fenetre divisée en 3 morceaux</title>
```

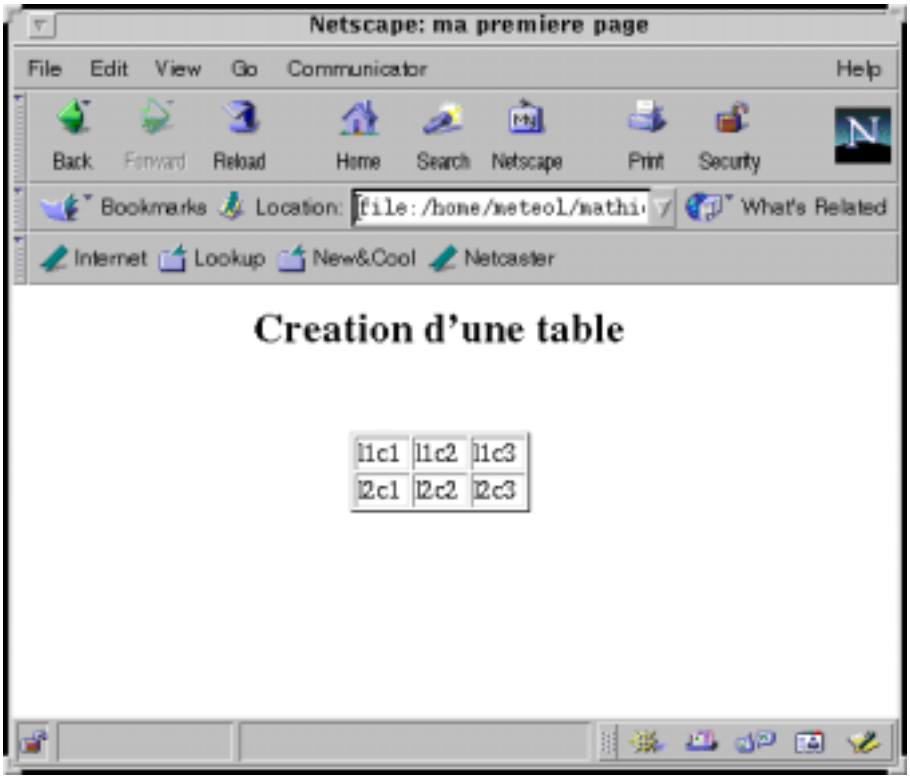


FIG. B.3 – *Page HTML contenant un tableau*

```
   </head>
   <frameset cols = "25%, 50%, *">
      <frame src = "pageIndex.html" name = "index">
      <frame src = "pageMain.html"  name = "main">
      <frame src = "pageNotes.html" name = "notes">
   </frameset>
</html>
```

On peut voir le résultat sur la fig.B.4.

L'attribut `name` est très important puisque c'est lui qui permet d'identifier la sous-fenêtre de la frame dans laquelle les futures pages sont chargées. C'est l'attribut `target` d'un lien hypertexte qui permet de spécifier le nom de la sous-fenêtre de destination. Par exemple le lien

`<a href="http:detail.html" target="notes">affichage du détail</a>`

permettra d'afficher la page `detail.html` dans la colonne `notes` de la frame précédente.

B.12 Les Applets

Une Applet est un morceau de code java qui s'exécute sur le poste Client.

```
<applet code="toto.class" width=100 height=100>
   <param name="param1> value="taratata">
   <param name="param2> value="tirititi">
</applet>
```

C'est la méthode `getParameter` de l'Applet qui permet de récupérer les paramètres passés.

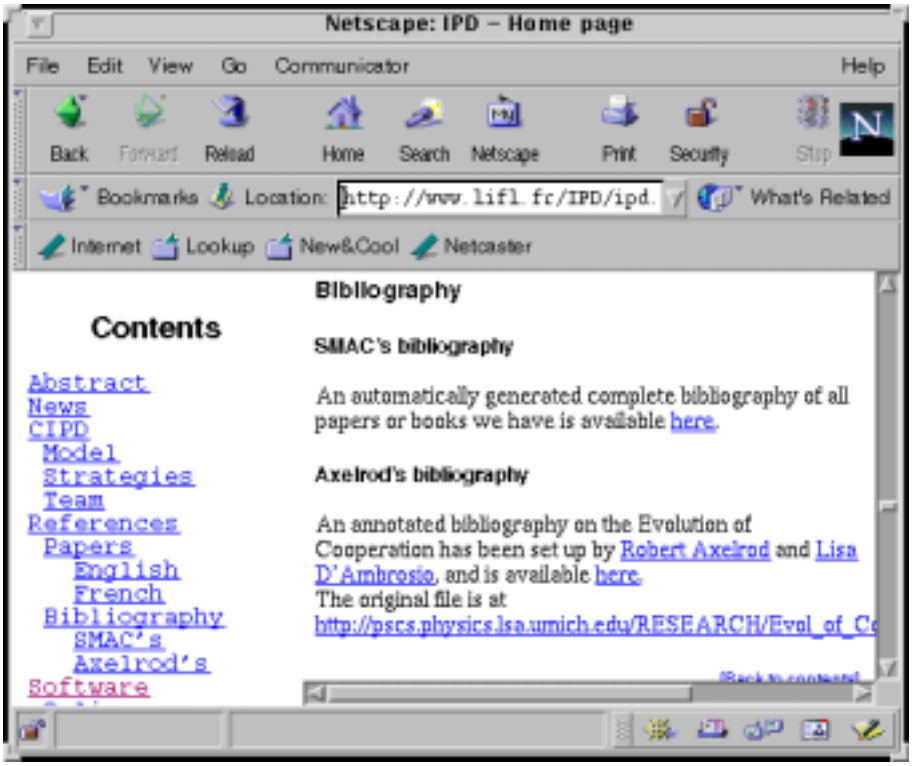


FIG. B.4 – *Page HTML contenant une frame*

B.13 Les Forms

Les Forms permettent de définir des interfaces d'échange d'informations entre le serveur HTTP et l'utilisateur (CGI, Servlets)

```
<form action="url" method="post">
entrez votre nom <INPUT TYPE="text" SIZE="20" NAME=entree>
<br> <INPUT TYPE="submit"><INPUT TYPE="reset">
</form>
```

Le champs `URL` contient l'adresse à laquelle le formulaire sera envoyé. Le champs `method` contient le type de méthode utilisé pour envoyer les paramètres. Il en existe 2: `GET` et dans ce cas le contenu de la Form est concaténé à l'URL et `POST` et dans ce cas le contenu est passé par un message de requête. Le champs `TYPE` contient le type d'objet graphique que l'on souhaite afficher. Il peut contenir notamment les valeurs text, hidden, password, checkbox, radio, submit et reset en fonction de l'objet que l'on souhaite afficher.

L'attribut `name` est lui aussi très important puisque c'est lui qui permettra d'identifier l'objet dans une Servlet ou un cgi. C'est la méthode `getParameter("entree")` d'une Servlet qui permet par exemple de récupérer la valeur saisie dans le champs texte nommé `entree`.

B.14 Exemple de Form

```
<form action=http://127.0.0.1:8080/servlet/survey method=POST>
  <input type=hidden name=survey value=Survey01Results>
  <BR><BR>How Many Employees in your Company?<BR>
    <BR>1-100<input type=radio name=employee value=1-100>
```

```
    <BR>100-200<input type=radio name=employee value=100-200>
    <BR>200-300<input type=radio name=employee value=200-300>
    <BR>300-400<input type=radio name=employee value=300-400>
    <BR>500-more<input type=radio name=employee value=500-more>
  <BR><BR>General Comments?<BR>
    <BR><input type=text name=comment>
  <BR><BR>What IDEs do you use?<BR>
    <BR>JavaWorkShop<input type=checkbox name=ide value=WorkShop>
    <BR>J++<input type=checkbox name=ide value=J++>
    <BR>Cafe<input type=checkbox name=ide value=Cafe>
  <BR><BR><input type=submit><input type=reset>
</form>
```

On peut voir le résultat sur la fig.B.5.

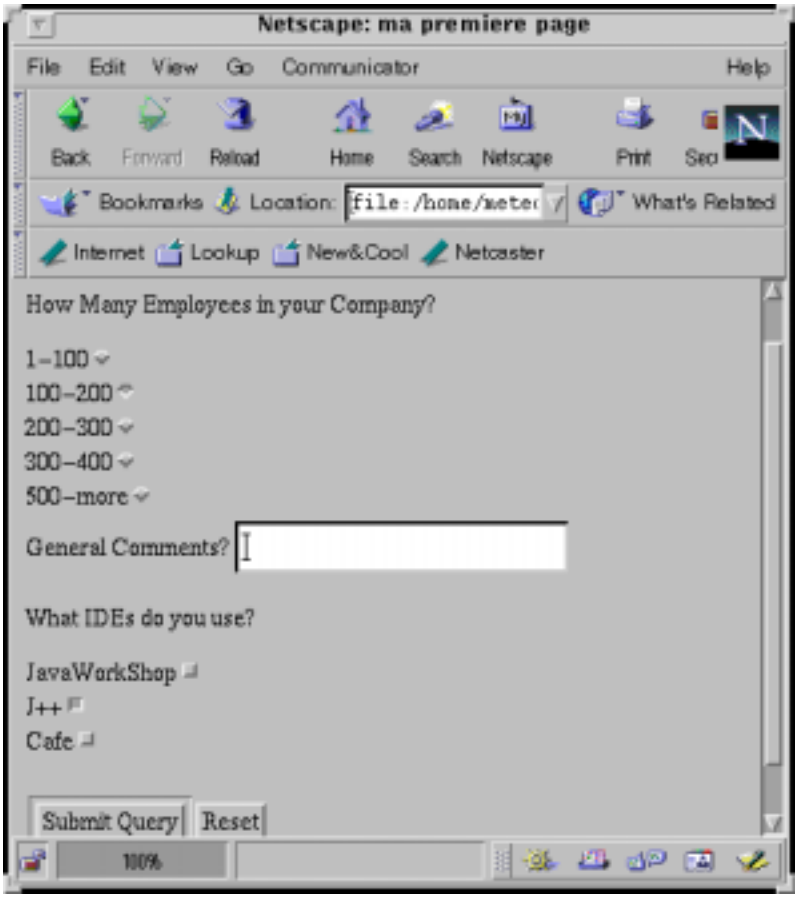


FIG. B.5 – *Page HTML contenant une forme*

B.15 Quelques conseils

- Dans chaque navigateur il est possible de voir le code source de la page chargée. Ne pas s'en priver !
- Les constructeurs de pages Wysiwyg ne dispensent pas d'aller voir le code généré.
- S'inspirer des autres pages pour améliorer les siennes
- Tout ce qui est affiché sur votre page est chargé dans votre mémoire donc sauvegardable.

- Ne pas surcharger les pages. les images et les Applets ou les Gif animés sont longs à charger
- Toujours penser à l'utilisateur final !

Annexe C

Glossaire

2-tier : Type d'architecture de connexion BDD-WWW dans lequel le serveur WEB communique directement avec le serveur de données. Cette architecture est acceptable tant que le système d'information n'est pas trop volumineux.

3-tier : Type d'architecture de connexion BDD-WWW dans lequel le serveur WEB communique avec le serveur de données par l'intermédiaire d'un serveur d'objets type RMI ou serveur d'applications. Permet de séparer la logique de l'entreprise dans des EJB et l'affichage de pages pour les clients.

API : Application Programming Interface ; ensemble de procédures définissant un mode de programmation dans un langage donné. JDBC est une API Java.

Applet : Code Java chargé sur un serveur WEB et qui s'exécute sur le Browser client dans une architecture Client-Serveur.

ASP : Active Server Page ; technologie propriétaire Microsoft permettant d'interfacer le serveur WEB IIS avec une base de données (généralement SQL Server) à travers des pages HTML dynamiques.

Association : lien entre deux entités représentant une action reliant ces entités.

Atomicité : Propriété d'une transaction qui assure que soit tous les ordres de la transaction sont passés soit aucun ne l'est.

Cardinalités : nombre d'occurrences d'un élément d'une entité dans une association.

CGI : Common Gateway Interface ; Code C ou Perl chargé sur un serveur Web et permettant notamment l'interface entre le WEB et un SGBD. Peu à peu remplacé par des ASP, JSP ou des Servlets.

Clé primaire : identifiant permettant de sélectionner un tuple unique dans une table.

Clé étrangère : clé primaire d'une autre table recopiée pour des besoins de référence. Généralement une contrainte d'intégrité référentielle doit être vérifiée entre la clé étrangère et la clé primaire correspondante.

Cluster : objet physique qui contient plusieurs tables avec une ou plusieurs colonnes en commun. Ces colonnes communes sont stockées physiquement ensemble dans la base de données. Les clusters permettent d'optimiser à la fois la place mémoire et le temps d'exécution des requêtes sur des tables.

Cohérence : propriété d'une base de données fidèle à toutes les contraintes d'intégrité.

Commit : ordre demandant au SGBD de prendre en compte effectivement les modifications effectuées. Après un commit, aucun retour arrière n'est possible.

Contraintes d'intégrité : ensemble de règles à respecter pour les données qui seront saisies dans une base de données.

Curseur : Mécanisme permettant d'inclure des sélections de tables SQL dans un langage impératif.

DBA : Database Administrator. L'administrateur de base de données. Il a la charge de la définition des schémas, du système de gestion des données et de la gestion des utilisateurs.

DBMS : Database Management System; le système de gestion de bases de données.

DCL : Date Control Language; ensemble des ordres d'un langage de manipulation de bases de données concernant la gestion des droits et des utilisateurs.

DDL : Data Definition Language; ensemble des ordres d'un langage de manipulation de bases de données concernant la gestion des schémas.

DML : Data Manipulation Language; ensemble des ordres d'un langage de manipulation de bases de données concernant la gestion des données.

EJB : Enterprise Java Bean; Code Java qui s'exécute côté serveur permettant de faire l'interface entre le SGBD et le serveur WEB. Lié en général à des JSP.

Entité : objet ayant une existence propre dans l'entreprise.

HTML : HyperText Markup Language; langage de création de pages Hypertexte utilisé sur l'Internet.

HTTP : HyperText Transfer Protocol; protocole réseau de transport de données utilisé sur l'Internet

Index : Objet physique permettant un accès rapide aux données d'une table par la ou les colonnes indexées. Les index accélèrent les requêtes de sélection mais pénalisent les requêtes de mise à jour.

Isolation : Propriété des transactions qui assure que même en accès concurrent elles perçoivent toujours la base de manière cohérente.

JDBC : Java Database Connectivity; API Java permettant d'interfacer les bases de données au langage Java.

JDK : Java Development Kit ; paquetage de développement Java fourni par Sun. Il contient notamment toutes les API standard comme JDBC ainsi que le compilateur Java.

JSDK : Java Servlet Development Kit ; paquetage de développement de Servlets fourni par Sun. Il contient notamment la classe HTTPServlet ainsi qu'un mini serveur de Servlets.

JSP : Java Server Pages; technologie proposée par SUN permettant de décrire des pages HTML dynamiques. En général la page est compilée par le serveur et génère une Servlet

Jointure : requête reliant deux tables en vue de construire une seule table résultat.

Journal : Outil d'enregistrement des mises à jour dans la base pour assurer l'atomicité des transactions et le redémarrage en cas de panne.

Lien hiérarchique : lien de type 1:n dans le modèle conceptuel de données.

Lien maillé : lien de type n:m dans le modèle conceptuel de données.

Lien réflexif : lien partant d'une entité vers elle même dans le modèle conceptuel de données.

MCD : Modèle conceptuel de données (méthode Merise); description des données du système d'informations dans un langage proche de l'utilisateur. Parfois appelé modèle Entités-Associations.

MLD : Modèle logique de données (méthode Merise); Modèle logique de données ; description des données du système d'informations sous forme d'objets logiques du SGBD (tables en relationnel, prédicats en déductif ...)

ODBC : Object DataBase Connection ; interface permettant à différents SGBD de communiquer entre eux en mode client-serveur. Les clients doivent tourner sous système Windows.

Propriété : élément de base du dictionnaire de données.

QBE : Query by Example ; langage graphique d'interrogation de bases de données relationnelles à l'aide d'exemples.

Requête : ordre donné au SGBD permettant de retrouver ou de mettre à jour des données de la base.

Rollback : ordre demandant au SGBD de revenir dans le dernier état cohérent (après le dernier commit), et donc, de ne pas prendre en compte les dernières modifications effectuées.

Réplication : mécanisme permettant de travailler sur des copies de la base qui seront réintégrées par la suite.

Servlet : Code Java chargé sur un serveur WEB et qui s'exécute sur ce même serveur permettant ainsi d'interfacer facilement WEB et SGBD dans une architecture Client-Serveur pour l'Internet.

SGBD : Système de gestion de base de données. Les systèmes actuels se basent sur les modèles hiérarchiques, réseaux, relationnel, déductif, objets.

SGBDOO : Système de gestion de base de données orienté objets.

SGBDR : Système de gestion de base de données relationnel.

SQL : Structured Query Language; langage d'interrogation de bases de données relationnelles basé sur le langage naturel.

Sérialisable : Se dit d'une exécution de transactions de manière concurrente qui donne le même résultat que l'une de leurs exécutions séquentielles.

Transaction : unité de traitement qui , appliquée à une base de données cohérente, restitue une base de données cohérente. Une transaction se termine par un Commit ou un Rollback.

Trigger : procédures utilisateur permettant de décrire des contraintes d'intégrité complexes.

tuple : ligne d'une table relationnelle.

URL : Uniform Resource Locator; adresse Internet permettant d'identifier une ressource sur le Net.

VSAM : système de gestion de fichiers indexés sur machines IBM.

Vue : Objet SQL permettant de conserver le script d'une requête. Une vue est réinterprétée à chaque accès, elle n'a pas d'existence physique. La vue permet d'obtenir une indépendance logique des requêtes de l'utilisateur par rapport aux tables réelles.

WWW : World Wide Web ; L'Internet !

Bibliographie

[Car90] Christian Carrez. *Des structures aux bases de données.* Dunod, Paris, France, 1990. ISBN : 2-04-019855-5.

[CBS96] T Conolly, C Begg, and A Strachan. *Database systems. A practical approach to design, implementation and management.* Addison Wesley, Harlow, England, 1996. ISBN : 0-201-42277-8.

[Cla78] K. L. Clark. Negation as failure. In Gallaire and Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New-York, 1978.

[Cod70] E. F. Codd. A relational model for large shared data banks. *CACM*, 13(6):377–387, 1970.

[Cod72] E.F. Codd. Further normalization of the data base relational model. In R. Rustin, editor, *Data Bases systems*, pages 33–64, Englewood Cliffs, New Jersey, 1972. Prentice Hall.

[Cod74] E. F. Codd. Recent investigations in relational data base systems. In *Proc. IFIP Congress*, 1974.

[Dat89] Chris Date. *Introduction au standard SQL.* InterEditions, Paris, France, 1989. Traduction de A Guide to the SQL standard, Addison Wesley 1987; ISBN : 2-7296-0261-5.

[Del95] Pierre Delmal. *SQL 2. De la théorie à la pratique.* De Boeck Université, Bruxelles, Belgique, 1995. ISBN : 2-8041-2179-8.

[Fag77] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. on Database systems*, 2(3):262–278, 1977.

[Gab89] Joseph Gabbay. *Apprendre et pratiquer Merise.* MIPS. Masson, Paris, France, 1989. ISBN: 2-225-81687-5.

[Gar96] Georges Gardarin. *Bases de données. Les systèmes et leurs langages.* Eyrolles, Paris, France, 1996. ISBN: 2-212-07500-6.

[HC98] Jason Hunter and William Crawford. *Java Servlet Programming.* O'Reilly, Paris, France, 1998. ISBN: 1-56592-391-X.

[Kou92] Michel Koutchouk. *SQL et DB2. Le relationnel et sa pratique.* Masson, Paris, France, 1992. ISBN : 2-225-82799-0.

[KS88] Henry F Korth and Abraham Silberschatz. *Systèmes de gestion des bases de données.* McGraw Hill, Paris, 1988. ISBN : 2-7042-1170-1.

[MB90] Serge Miranda and José-Maria Busta. *L'art des bases de données*, volume 1 et 2. Eyrolles, Paris, 1990.

[Mic92] Microsoft. *Introduction à la programmation de MS Access.* Microsoft Corp., 1992.

[ML94] Christian Marée and Guy Ledant. *SQL 2 Initiation Programmation.* Armand Colin / Masson, Paris, 1994. ISBN : 2-200-21411-1.

[Ree97] George Reese. *JDBC et Java, guide du programmeur.* O'Reilly, Paris, France, 1997. ISBN : 2-84177-042-7.

[Ull88] Jeffey D. Ullman. *Database and knowledge-base systems*, volume 1 et 2. Computer Science Press, Rockville, Maryland, 1988. ISBN : 0-7167-8158-1.

[Zlo77] M. M. Zloff. Query-by-example: a database language. *IBM systems J*, 16(4):324–343, 1977.