GEORGES GARDARIN

Bases de données

EYROLLES

EYROLLES

Bases de données

La référence en langue française sur les bases de données

Les bases de données jouent un rôle sans cesse croissant dans les systèmes d'information d'entreprise, qu'il s'agisse d'applications de gestion traditionnelles (comptabilité, ventes, décisionnel…) ou d'applications intranet, e-commerce ou de gestion de la relation client.
Comprendre les principes des bases de données, les langages d'interrogation et de mise à jour, les techniques d'optimisation et de contrôle des requêtes, les méthodes de conception et la gestion des transactions devient une nécessité pour tous les professionnels et futurs professionnels de l'informatique.
Complet et didactique, l'ouvrage se caractérise par des définitions précises des concepts, une approche éclairante des algorithmes et méthodes, de nombreux exemples d'application, une bibliographie commentée en fin de chaque chapitre et un recueil d'exercices en fin d'ouvrage. Il traite aussi bien des bases de données relationnelles, que des bases de données objet et objet-relationnelles.

Georges Gardarin
Chercheur renommé dans le domaine des bases de données et professeur à l'université Paris 6 puis à l'université de Versailles Saint-Quentin, Georges Gardarin a créé et dirigé successivement un projet de recherche INRIA sur les BD relationnelles parallèles (1980-89), le laboratoire PRiSM de Versailles (1990-99), qui regroupe une centaine de spécialistes en réseaux, bases de données et parallélisme, et enfin la société e-XMLmedia (2000-2002), éditeur de composants XML. Il est aujourd'hui professeur à l'université de Versailles et participe à des projets de recherche européens en médiation de données hétérogènes.

Au sommaire

Conception **Nord Compo**

Code éditeur : G11281
ISBN : 2-212-11281-5

EYROLLES

Bases de données

Bases de données

Georges Gardarin

5e tirage 2003

EYROLLES

EDITIONS EYROLLES
61, bd, Saint-Germain
75240 Paris cedex 05
www.editions-eyrolles.com

Cet ouvrage a fait l'objet d'un reconditionnement à l'occasion de son 5e tirage (format semi-poche et nouvelle couverture). Le texte de l'ouvrage reste inchangé par rapport aux tirages précédents.

À ma petite fille, Marie, née à la fin de la gestation de ce livre,
avec l'espoir que les Bases de Données l'aideront à mieux vivre et comprendre.

REMERCIEMENTS

Je tiens à exprimer mes vifs remerciements à tous ceux qui, par leurs travaux, leurs idées, leurs présentations, leurs collaborations ou leurs relectures, ont participé de près ou de loin à la réalisation de cet ouvrage, en particulier :

Catherine Blirando, Christophe Bobineau, Luc Bouganim, Mokrane Bouzeghoub, Tatiana Chan, Jean-Luc Darroux, Thierry Delot, Françoise Fabret, Béatrice Finance, Dana Florescu, Élisabeth Métais, Philippe Pucheral, Fei Sha, Éric Simon, Tuyet Tram Dang Ngoc, Patrick Valduriez, Yann Viémont, Fei Wu, Karine Zeitouni.

Une mention particulière à Hélène qui m'a supporté pendant tous ces nombreux week-ends et vacances passés à rédiger.

AVANT-PROPOS

J'ai commencé à travailler dans le domaine des bases de données en 1968 à l'université de Paris VI (non, pas en lançant des pavés), alors que le modèle réseau pointait derrière les fichiers séquentiels puis indexés. Sous la direction de Michel Rocher qui fut plus tard directeur d'Oracle France, avec Mireille Jouve, Christine Parent, Richard Gomez, Stefano Spaccapietra et quelques autres, nous avions développé une famille de Systèmes de Fichiers pour Apprendre Les Autres : les SFALA. Nous enseignions essentiellement les méthodes d'accès séquentielles, indexées, hachées, et surtout le système. Bientôt (en 1972), nous avons introduit les Systèmes de Données pour Apprendre Les Autres (SDALA). Il y eut un SDALA basé sur le modèle réseau, puis bientôt un basé sur le modèle relationnel. Aujourd'hui, on pourrait faire un SDALA objet, multimédia, et bientôt semi-structuré...

Le premier système de gestion de données que j'ai construit à l'Hopital Necker gérait un disque SAGEM à têtes fixes de 256 K octets ! Ceci me semblait énorme par rapport au 8 K de mémoire. Le système gérait les dossiers des malades avec des fichiers hachés. C'était en 1969 alors que j'étais encore élève à l'ENSET et stagiaire à TITN. Le deuxième le fut à OrdoProcesseurs, une société française qui vendait des mini-ordinateurs français. C'était en 1974 et les disques atteignaient déjà 10 méga-octets. Le troisième le fut à l'INRIA au début des années 1980 : c'était l'un des trop rares systèmes relationnels français commercialisés, le SGBD SABRE. Les disques dépassaient déjà 100 M octets ! Aujourd'hui, les disques contiennent plusieurs giga-octets et l'on parle de pétabases (10 puissance 15 octets). Demain, et demain est déjà là, avec l'intégration des réseaux et des bases de données, tous les serveurs de données du monde seront interconnectés et l'on gérera des volumes inimaginables de données en ligne par des techniques plus ou moins issues des bases de données...

Alors que les bases de données semblaient au début réservées à quelques applications sophistiquées de gestion, toute application moderne utilise aujourd'hui une base de données sous une forme ou sous une autre. Certes, il y a encore beaucoup de données dans des fichiers, mais l'équilibre – où plutôt le déséquilibre – se déplace. Toute

application de gestion non vieillotte utilise une BD relationnelle, les BD objets percent dans les applications à données complexes, et les serveurs Web s'appuient de plus en plus sur des bases de données. Pourquoi ? Car les BD offrent le partage, la fiabilité, les facilités de recherche et bientôt la souplesse et l'intelligence avec le support de données multimédia et semi-structurées, et de techniques venues de l'intelligence artificielle, telles le *data mining*. Les BD sont de plus en plus distribuées, intégrées avec les réseaux Intranet et Internet. D'où leur généralisation.

Voilà donc un domaine que j'ai eu la chance de traverser depuis sa naissance qui a créé beaucoup d'emplois et qui va continuer à en créer dans le millénaire qui vient. Le vingt et unième siècle devrait être celui des sciences et techniques de l'information, au moins à son début. Certains m'objecteront que l'on a créé beaucoup de formulaires, alourdi la gestion et plus généralement la société, et aussi détruit les petits emplois. Peut-être, mais ce n'est pas l'objectif, et ceci devrait être corrigé (notamment avec les formulaires en ligne). D'autres objecteront que nous créons (avec Internet notamment) une civilisation à deux vitesses : je le crains malheureusement, et voilà pourquoi il est très nécessaire de simplifier et démystifier, par exemple en écrivant des livres essayant de mettre ces techniques à la portée de tous.

Dans le domaine des bases de données, comme dans beaucoup d'autres, la France a gâché beaucoup de chances, mais reste encore très compétitive, au moins en compétences sinon en produits. En fait, depuis septembre 1997, l'industrie française ne possède plus de SGBD important. Auparavant, elle avait possédé SOCRATE, IDS II, SABRE (un SGBD important pour l'auteur), et enfin O2. À vrai dire, le seul bien vendu fut IDS.II, un produit issu des États-Unis. Mais enfin, nous avions la maîtrise de la technologie...

Ce livre présente une synthèse des principes et des techniques actuelles en matière de base de données. Il traite des bases de données relationnelles et des bases de données objet. Ces paradigmes sont au cœur des systèmes d'information d'aujourd'hui. Ils sont essentiels pour les entreprises et méritent d'être connus de tout étudiant à l'Université ou en École d'ingénieur.

Ce livre est accompagné d'un compagnon plus petit traitant des nouvelles techniques : *data warehouse*, *data mining*, BD *Web* et BD *multimédia*. Il s'agit là des nouveaux thèmes en vogue du domaine, qui vont sans doute profondément révolutionner l'informatique de demain. Nous avons choisi de ne pas intégrer ces sujets à ce livre mais à un volume séparé, car ils ne sont pas encore stabilisés, alors que le relationnel et l'objet – ainsi que le mélange des deux, connu sous le nom objet-relationnel – le sont beaucoup plus.

En résumé, cet ouvrage est le fruit d'une expérience de trente ans d'enseignement, de formation et de conseil à l'université et dans l'industrie. Il aborde tous les sujets au cœur des systèmes d'information modernes. Chaque chapitre traite un thème particulier. Á l'aide de notions précisément définies – une technique d'enseignement inven-

tée par Michel Rocher dans les années 1970, procédant par définitions informelles –, nous clarifions des concepts souvent difficiles.

En annexe de cet ouvrage, vous trouverez une cinquantaine de textes d'exercices qui dérivent de sujets d'examens proposés aux étudiants à Paris VI ou à Versailles depuis 1980, adaptés et modernisés.

Avec cet ouvrage, nous espérons mettre entre les mains des générations actuelles et futures d'enseignants, d'ingénieurs et de chercheurs une expérience pratique et théorique exceptionnelle en matière de bases de données.

NOTATIONS

Afin de présenter la syntaxe de certains langages, nous utiliserons les notations suivantes :

[*a*] signifie que l'élément *a* est optionnel ;

[*a*]... signifie que l'élément *a* peut-être répété 0 à *n* fois (n entier positif) ;

{*a b*} signifie que les éléments *a* et *b* sont considérés comme un élément unique ;

{*a* | *b*} signifie un choix possible entre l'alternative *a* ou *b* ;

< *a* > signifie que *a* est un paramètre qui doit être remplacé par une valeur effective.

SOMMAIRE

PARTIE 1 – LES BASES

PARTIE 3 – L'OBJET ET L'OBJET-RELATIONNEL

PARTIE 4 – AU-DELÀ DU SGBD

Partie 1

LES BASES

I – Introduction (*Introduction*)

II – Objectifs et architecture des SGBD
(*DBMS Objectives and Architecture*)

III – Fichiers, hachage et indexation
(*File management*)

IV – Bases de données réseaux et hiérarchiques
(*Legacy databases*)

V – Logique et bases de données (*Logic and databases*)

Chapitre I

INTRODUCTION

1. QU'EST-CE QU'UNE BASE DE DONNÉES ?

Les bases de données ont pris aujourd'hui une place essentielle dans l'informatique, plus particulièrement en gestion. Au cours des trente dernières années, des concepts, méthodes et algorithmes ont été développés pour gérer des données sur mémoires secondaires ; ils constituent aujourd'hui l'essentiel de la discipline « Bases de Données » (BD). Cette discipline est utilisée dans de nombreuses applications. Il existe un grand nombre de Systèmes de Gestion de Bases de Données (SGBD) qui permettent de gérer efficacement de grandes bases de données. De plus, une théorie fondamentale sur les techniques de modélisation des données et les algorithmes de traitement a vu le jour. Les bases de données constituent donc une discipline s'appuyant sur une théorie solide et offrant de nombreux débouchés pratiques.

Vous avez sans doute une idée intuitive des bases de données. Prenez garde cependant, car ce mot est souvent utilisé pour désigner n'importe quel ensemble de données ; il s'agit là d'un abus de langage qu'il faut éviter. Une base de données est un ensemble de données modélisant les objets d'une partie du monde réel et servant de support à une application informatique. Pour mériter le terme de base de données, un ensemble de données non indépendantes doit être interrogeable par le contenu, c'est-à-dire que l'on doit pouvoir retrouver tous les objets qui satisfont à un certain critère, par exemple tous les produits qui coûtent moins de 100 francs. Les données doivent être interrogeables

selon n'importe quel critère. Il doit être possible aussi de retrouver leur structure, par exemple le fait qu'un produit possède un nom, un prix et une quantité.

Plutôt que de disserter longuement sur le concept de bases de données, précisons ce qu'est un SGBD. Un SGBD peut être perçu comme un ensemble de logiciels systèmes permettant aux utilisateurs d'insérer, de modifier et de rechercher efficacement des données spécifiques dans une grande masse d'informations (pouvant atteindre quelques milliards d'octets) partagée par de multiples utilisateurs. Les informations sont stockées sur mémoires secondaires, en général des disques magnétiques. Les recherches peuvent être exécutée à partir de la valeur d'une donnée désignée par un nom dans un ensemble d'objets (par exemple, les produits de prix inférieur à 100 francs), mais aussi à partir de relations entre objets (par exemple, les produits commandés par un client habitant Paris). Les données sont partagées, aussi bien en interrogation qu'en mise à jour. Le SGBD rend transparent le partage, à savoir donne l'illusion à chaque utilisateur qu'il est seul à travailler avec les données.

En résumé, un SGBD peut donc apparaître comme un outil informatique permettant la sauvegarde, l'interrogation, la recherche et la mise en forme de données stockées sur mémoires secondaires. Ce sont là les fonctions premières, complétées par des fonctions souvent plus complexes, destinées par exemple à assurer le partage des données mais aussi à protéger les données contre tout incident et à obtenir des performances acceptables. Les SGBD se distinguent clairement des systèmes de fichiers par le fait qu'ils permettent la description des données (définition des types par des noms, des formats, des caractéristiques et parfois des opérations) de manière séparée de leur utilisation (mise à jour et recherche). Ils permettent aussi de retrouver les caractéristiques d'un type de données à partir de son nom (par exemple, comment est décrit un produit). Le système de fichiers est un composant de plus bas niveau ne prenant pas en compte la structure des données. La tendance est aujourd'hui à intégrer le système de fichiers dans le SGBD, construit au-dessus.

En conséquence, un SGBD se compose en première approximation de trois couches emboîtées de fonctions, depuis les mémoires secondaires vers les utilisateurs (voir figure I.1) :

– La gestion des récipients de données sur les mémoires secondaires constitue traditionnellement la première couche ; c'est le **gestionnaire de fichiers**, encore appelé système de gestion de fichiers. Celui-ci fournit aux couches supérieures des mémoires secondaires idéales adressables par objets et capables de recherches par le contenu des objets (mécanismes d'indexation notamment).
– La gestion des données stockées dans les fichiers, l'assemblage de ces données en objets, le placement de ces objets dans les fichiers, la gestion des liens entre objets et des structures permettant d'accélérer les accès aux objets constituent la deuxième couche ; c'est le système d'accès aux données ou **SGBD interne**. Celui-ci repose généralement sur un modèle de données internes, par exemple des tables reliées par des pointeurs.

– La fonction essentielle de la troisième couche consiste dans la mise en forme et la présentation des données aux programmes d'applications et aux utilisateurs interactifs. Ceux-ci expriment leurs critères de recherches à l'aide de langages basés sur des procédures de recherche progressives ou sur des assertions de logiques, en référençant des données dérivées de la base ; c'est le **SGBD externe** qui assure d'une part l'analyse et l'interprétation des requêtes utilisateurs en primitives internes, d'autre part la transformation des données extraites de la base en données échangées avec le monde extérieur.

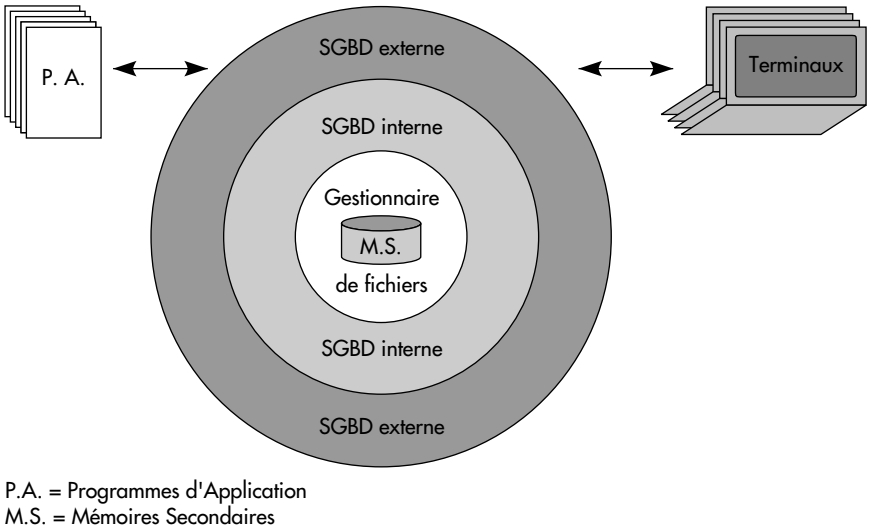


Figure I.1 : Première vue d'un SGBD

Ces couches de fonctions constituent sans doute seulement la moitié environ du code d'un SGBD. En effet, au-delà de ses fonctions de recherche, de rangement et de présentation, un SGBD gère des problèmes difficiles de partage et de cohérence de données. Il protège aussi les données contre les accès non autorisés. Ces fonctions qui peuvent paraître annexes sont souvent les plus difficiles à réaliser et nécessitent beaucoup de code.

Pour être complet, signalons qu'au-dessus des SGBD les systèmes d'informations intègrent aujourd'hui de plus en plus souvent des ateliers de génie logiciel permettant de modéliser les données d'une base de données et de représenter les traitements associés à l'aide de graphiques et de langages de spécifications. Ces outils d'aide à la conception, bien que non intégrés dans le SGBD, permettent de spécifier les descriptions des données. Ils s'appuient pour cela sur les modèles de données décrits dans cet ouvrage et supportés par les SGBD.

2. HISTORIQUE DES SGBD

Les SGBD ont bientôt quarante ans d'histoire. Les années 60 ont connu un premier développement des bases de données sous forme de fichiers reliés par des pointeurs. Les fichiers sont composés d'articles stockés les uns à la suite des autres et accessibles par des valeurs de données appelées clés. Les systèmes IDS.I et IMS.I développés respectivement à Honeywell et à IBM vers 1965 pour les programmes de conquête spatiale, notamment pour le programme APOLLO qui a permis d'envoyer un homme sur la lune, sont les précurseurs des SGBD modernes. Ils permettent de constituer des chaînes d'articles entre fichiers et de parcourir ces chaînes.

Les premiers SGBD sont réellement apparus à la fin des années 60. La première génération de SGBD est marquée par la séparation de la description des données et de la manipulation par les programmes d'application. Elle coïncide aussi avec l'avènement des langages d'accès navigationnels, c'est-à-dire permettant de se déplacer dans des structures de type graphe et d'obtenir, un par un, des articles de fichiers. Cette première génération, dont l'aboutissement est marqué par les recommandations du CODASYL, est basée sur les modèles réseau ou hiérarchique, c'est-à-dire des modèles de données organisés autour de types d'articles constituant les nœuds d'un graphe, reliés par des types de pointeurs composant les arcs du graphe. Cette génération a été dominée par les SGBD TOTAL, IDMS, IDS 2 et IMS 2. Elle traite encore aujourd'hui une partie importante du volume de données gérées par des SGBD.

La deuxième génération de SGBD a grandi dans les laboratoires depuis 1970, à partir du modèle relationnel. Elle vise à enrichir mais aussi à simplifier le SGBD externe afin de faciliter l'accès aux données pour les utilisateurs. En effet, les données sont présentées aux utilisateurs sous forme de relations entre domaines de valeurs, simplement représentées par des tables. Les recherches et mises à jour sont effectuées à l'aide d'un langage non procédural standardisé appelé SQL (*Structured Query Language*). Celui-ci permet d'exprimer des requêtes traduisant directement des phrases simples du langage naturel et de spécifier les données que l'on souhaite obtenir sans dire comment les accéder. C'est le SGBD qui doit déterminer le meilleur plan d'accès possible pour évaluer une requête. Cette deuxième génération reprend, après les avoir faits évoluer et rendus plus souples, certains modèles d'accès de la première génération au niveau du SGBD interne, afin de mieux optimiser les accès. Les systèmes de deuxième génération sont commercialisés depuis 1980. Ils représentent aujourd'hui l'essentiel du marché des bases de données. Les principaux systèmes sont ORACLE, INGRES, SYBASE, INFORMIX, DB2 et SQL SERVER. Ils supportent en général une architecture répartie, au moins avec des stations clients transmettant leurs requêtes à de puissants serveurs gérant les bases.

La troisième génération a été développée dans les laboratoires depuis le début des années 80. Elle commence à apparaître fortement dans l'industrie avec les extensions objet des systèmes relationnels. Elle supporte des modèles de données extensibles

intégrant le relationnel et l'objet**,** ainsi que des architectures mieux réparties, permettant une meilleure collaboration entre des utilisateurs concurrents. Cette troisième génération est donc influencée par les modèles à objets, intégrant une structuration conjointe des programmes et des données en types, avec des possibilités de définir des sous-types par héritage. Cependant, elle conserve les acquis du relationnel en permettant une vision tabulaire des objets et une interrogation via le langage SQL étendu aux objets. Elle intègre aussi le support de règles actives plus ou moins dérivées de la logique. Ces règles permettent de mieux maintenir la cohérence des données en répercutant des mises à jour d'un objet sur d'autres objets dépendants. Les systèmes objet-relationnels tels Oracle 8, DB2 Universal Database ou Informix Universal Server, ce dernier issu du système de recherche Illustra, sont les premiers représentants des systèmes de 3e génération. Les systèmes à objets tels ObjectStore ou O2 constituent une voie plus novatrice vers la troisième génération. Tous ces systèmes tentent de répondre aux besoins des nouvelles applications (multimédia, Web, CAO, bureautique, environnement, télécommunications, etc.).

Quant à la quatrième génération, elle est déjà en marche et devrait mieux supporter Internet et le Web, les informations mal structurées, les objets multimédias, l'aide à la prise de décisions et l'extraction de connaissances à partir des données. Certes, il devient de plus en plus dur de développer un nouvel SGBD. On peut donc penser que les recherches actuelles, par exemple sur l'interrogation par le contenu des objets multimédias distribués et sur l'extraction de connaissances (*data mining*) conduiront à une évolution des SGBD de 3e génération plutôt qu'à une nouvelle révolution. Ce fut déjà le cas lors du passage de la 2e à la 3e génération, la révolution conduite par l'objet ayant en quelque sorte échouée : elle n'a pas réussi à renverser le relationnel, certes bousculé et adapté à l'objet. Finalement, l'évolution des SGBD peut être perçue comme celle d'un arbre, des branches nouvelles naissant mais se faisant généralement absorber par le tronc, qui grossit toujours d'avantage.

3. PLAN DE CET OUVRAGE

Ce livre traite donc de tous les aspects des bases de données relationnelles et objet, mais aussi objet-relationnel. Il est découpé en quatre parties autonomes, elles-mêmes divisées en chapitres indépendants, en principe de difficulté croissante.

La **première partie** comporte, après cette introduction, quatre chapitres fournissant les bases indispensables à une étude approfondie des SGBD.

Le chapitre II ébauche le cadre général de l'étude. Les techniques de modélisation de données sont tout d'abord introduites. Puis les objectifs et les fonctions des SGBD sont développés. Finalement, les architectures fonctionnelles puis opérationnelles des

SGBD modernes sont discutées. L'ensemble du chapitre est une introduction aux techniques et problèmes essentiels de la gestion des bases de données, illustrées à partir d'un langage adapté aux entités et associations.

Le chapitre III se concentre sur la gestion des fichiers et les langages d'accès aux fichiers. Certains peuvent penser que la gestion de fichiers est aujourd'hui dépassée. Il n'en est rien, car un bon SGBD s'appuie avant tout sur de bonnes techniques d'accès par hachage et par index. Nous étudions en détail ces techniques, des plus anciennes aux plus modernes, basées sur les indexes multiples et les hachages dynamiques multi-attributs ou des *bitmaps*.

Le chapitre IV traite des modèles légués par les SGBD de première génération. Le modèle réseau tel qu'il est défini par le CODASYL et implanté dans le système IDS.II de Bull est développé. Des exemples sont donnés. Le modèle hiérarchique d'IMS est plus succinctement introduit.

Le chapitre V introduit les fondements logiques des bases de données, notamment relationnelles. Après un rappel succinct de la logique du premier ordre, la notion de bases de données logique est présentée et les calculs de tuples et domaines, à la base des langages relationnels, sont introduits.

La **deuxième partie** est consacrée au relationnel. Le modèle et les techniques de contrôle et d'optimisation associées sont approfondis.

Le chapitre VI introduit le modèle relationnel de Codd et l'algèbre relationnelle associée. Les concepts essentiels pour décrire les données tels qu'ils sont aujourd'hui supportés par de nombreux SGBD sont tout d'abord décrits. Les types de contraintes d'intégrité qui permettent d'assurer une meilleure cohérence des données entre elles sont précisés. Ensuite, les opérateurs de l'algèbre sont définis et illustrés par de nombreux exemples. Enfin, les extensions de l'algèbre sont résumées et illustrées.

Le chapitre VII est consacré à l'étude du langage standardisé des SGBD relationnels, le fameux langage SQL. Les différents aspects du standard, accepté en 1986 puis étendu en 1989 et 1992, sont tout d'abord présentés et illustrés par de nombreux exemples. La version actuelle du standard acceptée en 1992, connue sous le nom de SQL2, est décrite avec concision mais précision. Il s'agit là du langage aujourd'hui offert, avec quelques variantes, par tous les SGBD industriels.

Le chapitre VIII traite des règles d'intégrité et des bases de données actives. Le langage d'expression des contraintes d'intégrité et des déclencheurs intégré à SQL est étudié. Puis, les différentes méthodes pour contrôler l'intégrité sont présentées. Enfin, les notions de base de données active et les mécanismes d'exécution des déclencheurs sont analysés.

Le chapitre IX expose plus formellement le concept de vue, détaille le langage de définition et présente quelques exemples simples de vues. Sont successivement abordés : les mécanismes d'interrogation de vues, le problème de la mise à jour des vues, l'utilisation des vues concrètes notamment pour les applications décisionnelles et quelques autres extensions possibles du mécanisme de gestion des vues.

Le chapitre X présente d'abord plus précisément les objectifs de l'optimisation de requêtes et introduit les éléments de base. Une large part est consacrée à l'étude des principales méthodes d'optimisation logique puis physique. Les premières restructurent les requêtes alors que les secondes déterminent le meilleur plan d'exécution pour une requête donnée. L'optimisation physique nécessite un modèle de coût pour estimer le coût de chaque plan d'exécution afin de choisir le meilleur. Un tel modèle est décrit, puis les stratégies essentielles permettant de retrouver un plan d'exécution proche de l'optimal sont introduites.

La **troisième partie** développe les approches objet et objet-relationnel. Les problèmes fondamentaux posés par l'objet sont analysés en détail.

Le chapitre XI développe l'approche objet. Les principes de la modélisation de données orientée objet sont tout d'abord esquissés. Puis, les techniques plus spécifiques aux bases de données à objets, permettant d'assurer la persistance et le partage des objets, sont développées. Enfin, ce chapitre propose une extension de l'algèbre relationnelle pour manipuler des objets complexes.

Le chapitre XII présente le standard de l'ODMG, en l'illustrant par des exemples. Sont successivement étudiés : le contexte et l'architecture d'un SGBDO conforme à l'ODMG, le modèle abstrait et le langage ODL, un exemple de base et de schéma en ODL, le langage OQL à travers des exemples et des syntaxes types de requêtes, l'intégration dans un langage de programmation comme Java et les limites du standard de l'ODMG.

Le chapitre XIII présente le modèle objet-relationnel, et son langage SQL3. Il définit les notions de base dérivées de l'objet et introduites pour étendre le relationnel. Il détaille le support des objets en SQL3 avec de nombreux exemples. Il résume les caractéristiques essentielles du langage de programmation de procédures et fonctions SQL/PSM, appoint essentiel à SQL pour assurer la complétude en tant que langage de programmation. Il souligne les points obscurs du modèle et du langage SQL3.

Le chapitre XIV présente une synthèse des techniques essentielles de l'optimisation des requêtes dans les bases de données objet, au-delà du relationnel. Les nombreuses techniques présentées sont issues de la recherche ; elles commencent aujourd'hui à être intégrées dans les SGBD objet-relationnel et objet. Une bonne compréhension des techniques introduites de parcours de chemins, d'évaluation de coût de requêtes, de placement par groupes, de prise en compte des règles sémantiques, permettra sans nul doute une meilleure optimisation des nouvelles applications.

La **dernière partie** traite trois aspects indépendants importants des bases de données : extensions pour la déduction, gestion de transactions et techniques de conception.

Le chapitre XV décrit les approches aux bases de données déductives. Plus précisément, il est montré comment une interprétation logique des bases de données permet de les étendre vers la déduction. Le langage de règles Datalog est présenté avec ses diverses extensions. Les techniques d'optimisation de règles récursives sont approfondies. L'intégration de règles aux objets est exemplifiée à l'aide de langages concrets implémentés dans des systèmes opérationnels.

Le chapitre XVI essaie de faire le point sur tous les aspects de la gestion de transactions dans les SGBD centralisés. Après quelques rappels de base, nous traitons d'abord les problèmes de concurrence. Nous étudions ensuite les principes de la gestion de transactions. Comme exemple de méthode de reprise intégrée, nous décrivons la méthode ARIES implémentée à IBM, la référence en matière de reprise. Nous terminons la partie sur les transactions proprement dites en présentant les principaux modèles de transactions étendus introduits dans la littérature. Pour terminer ce chapitre, nous traitons du problème un peu orthogonal de confidentialité.

Le chapitre XVII traite le problème de la conception des bases de données objet-relationnel. C'est l'occasion de présenter le langage de modélisation UML, plus précisément les constructions nécessaires à la modélisation de BD. Nous discutons aussi des techniques d'intégration de schémas. Le chapitre développe en outre les règles pour passer d'un schéma conceptuel UML à un schéma relationnel ou objet-relationnel. La théorie de la normalisation est intégrée pour affiner le processus de conception. Les principales techniques d'optimisation du schéma physique sont introduites.

Enfin, le chapitre XVIII couvre les directions nouvelles d'évolution des SGBD : data-warehouse, data mining, Web et multimédia. Ces directions nouvelles, sujets de nombreuses recherches actuellement, font l'objet d'un livre complémentaire du même auteur chez le même éditeur.

4. BIBLIOGRAPHIE

De nombreux ouvrages traitent des problèmes soulevés par les bases de données. Malheureusement, beaucoup sont en anglais. Vous trouverez à la fin de chaque chapitre du présent livre les références et une rapide caractérisation des articles qui nous ont semblé essentiels. Voici quelques références d'autres livres traitant de problèmes généraux des bases de données que nous avons pu consulter. Des livres plus spécialisés sont référencés dans le chapitre traitant du problème correspondant.

[Date90] Date C.J., *An Introduction to Database Systems*, 5e edition, The Systems Programming Series, volumes I (854 pages) et II (383 pages), Addison Wesley, 1990.

Ce livre écrit par un des inventeurs du relationnel est tourné vers l'utilisateur. Le volume I traite des principaux aspects des bases de données relationnelles, sans oublier les systèmes basés sur les modèles réseau et hiérarchique. Ce volume est divisé en six parties avec des appendices traitant de cas de systèmes. La partie I introduit les concepts de base. La partie II présente un système relationnel type, en fait une vue simplifiée de DB2, le SGBD d'IBM. La partie III approfondit le modèle et les langages de manipulation associés. La partie IV traite de l'environnement du SGBD. La partie V est consacrée à la conception

des bases de données. La partie VI traite des nouvelles perspectives : répartition, déduction et systèmes à objets. Le volume II traite des problèmes d'intégrité, de concurrence et de sécurité. Il présente aussi les extensions du modèle relationnel proposées par Codd (et Date), ainsi qu'une vue d'ensemble des bases de données réparties et des machines bases de données.

[Delobel91] Delobel C., Lécluse Ch., Richard Ph., *Bases de Données : Des Systèmes Relationnels aux Systèmes à Objets*, 460 pages, InterÉditions, Paris, 1991.

Une étude de l'évolution des SGBD, des systèmes relationnels aux systèmes objets, en passant par les systèmes extensibles. Un intérêt particulier est porté sur les langages de programmation de bases de données et le typage des données. Le livre décrit également en détail le système O2, son langage CO2 et les techniques d'implémentation sous-jacentes. Un livre en français.

[Gardarin97] Gardarin G., Gardarin O., *Le Client-Serveur*, 470 pages, Éditions Eyrolles, 1997.

Ce livre traite des architectures client-serveur, des middlewares et des bases de données réparties. Les notions importantes du client-serveur sont dégagées et expliquées. Une part importante de l'ouvrage est consacrée aux middlewares et outils de développement objet. Les middlewares à objets distribués CORBA et DCOM sont analysés. Ce livre est un complément souhaitable au présent ouvrage, notamment sur les middlewares, les bases de données réparties et les techniques du client-serveur.

[Gray91] Gray J. Ed., *The Benchmark Handbook*, Morgan & Kaufman Pub., San Mateo, 1991.

Le livre de base sur les mesures de performances des SGBD. Composé de différents articles, il présente les principaux benchmarks de SGBD, en particulier le fameux benchmark TPC qui permet d'échantillonner les performances des SGBD en transactions par seconde. Les conditions exactes du benchmark définies par le « Transaction Processing Council » sont précisées. Les benchmarks de l'université du Madisson, AS3AP et Catell pour les bases de données objets sont aussi présentés.

[Korth97] Silberschatz A., Korth H., Sudarshan S., *Database System Concepts*, 819 pages, Mc Graw-Hill Editions, 3e edition, 1997.

Un livre orienté système et plutôt complet. Partant du modèle entité-association, les auteurs introduisent le modèle relationnel puis les langages des systèmes commercialisés. Ils se concentrent ensuite sur les contraintes et sur les techniques de conception de bases de données. Les deux chapitres qui suivent sont consacrés aux organisations et méthodes d'accès de fichiers. Les techniques des SGBD relationnels (reprises après pannes, contrôle de concurrence, gestion de transaction) sont ensuite exposées. Enfin, les extensions vers les systèmes objets, extensibles et distribués sont étudiées. Le dernier chapitre pré-

sente des études de cas de systèmes et deux annexes traitent des modèles réseaux et hiérarchiques. La nouvelle bible des SGBD en anglais.

[Maier83] Maier D., *The Theory of Relational Databases*, Computer Science Press, 1983.

Le livre synthétisant tous les développements théoriques sur les bases de données relationnelles menés au début des années 80. En 600 pages assez formelles, Maier fait le tour de la théorie des opérateurs relationnels, des dépendances fonctionnelles, multivaluées, algébriques et de la théorie de la normalisation.

[Parsaye89] Parsaye K., Chignell M., Khoshafian S., Wong H., *Intelligent Databases*, 478 pages, Wiley Editions, 1989.

Un livre sur les techniques avancées à la limite des SGBD et de l'intelligence artificielle : SGBD objets, systèmes experts, hypermédia, systèmes textuels, bases de données intelligentes. Le SGBD intelligent est à la convergence de toutes ces techniques et intégre règles et objets.

[Ullman88] Ullman J.D., *Principles of Database and Knowledge-base Systems*, volumes I (631 pages) et II (400 pages), Computer Science Press, 1988.

Deux volumes très complets sur les bases de données, avec une approche plutôt fondamentale. Jeffrey Ullman détaille tous les aspects des bases de données, des méthodes d'accès aux modèles objets en passant par le modèle logique. Les livres sont finalement très centrés sur une approche par la logique des bases de données. Les principaux algorithmes d'accès, d'optimisation de requêtes, de concurrence, de normalisation, etc. sont détaillés. À noter l'auteur traite dans un même chapitre les systèmes en réseau et les systèmes objets, qu'il considère de même nature.

[Valduriez99] Valduriez P., Ozsu T., *Principles of Distributed Database Systems*, 562 pages, Prentice Hall, 2e édition,1999.

Le livre fondamental sur les bases de données réparties. Après un rappel sur les SGBD et les réseaux, les auteurs présentent l'architecture type d'un SGBD réparti. Ils abordent ensuite en détail les différents problèmes de conception d'un SGBD réparti : distribution des données, contrôle sémantique des données, évaluation de questions réparties, gestion de transactions réparties, liens avec les systèmes opératoires et multibases. La nouvelle édition aborde aussi le parallélisme et les middlewares. Les nouvelles perspectives sont enfin évoquées.

Chapitre II

OBJECTIFS ET ARCHITECTURE DES SGBD

1. INTRODUCTION

Même si vous n'avez jamais utilisé de système de gestion de bases de données (SGBD), vous avez certainement une idée de ce qu'est une base de données (BD) et par là même un SGBD. Une BD est peut-être pour certains une collection de fichiers reliés par des pointeurs multiples, aussi cohérents entre eux que possible, organisés de manière à répondre efficacement à une grande variété de questions. Pour d'autres, une BD peut apparaître comme une collection d'informations modélisant une entreprise du monde réel. Ainsi, un SGBD peut donc être défini comme un ensemble de logiciels systèmes permettant de stocker et d'interroger un ensemble de fichiers interdépendants, mais aussi comme un outil permettant de modéliser et de gérer les données d'une entreprise.

Les données stockées dans des bases de données modélisent des objets du monde réel, ou des associations entre objets. Les objets sont en général représentés par des articles de fichiers, alors que les associations correspondent naturellement à des liens entre articles. Les données peuvent donc être vues comme un ensemble de fichiers reliés par des pointeurs ; elles sont interrogées et mises à jour par des programmes d'applications écrits par les utilisateurs ou par des programmes utilitaires fournis avec le

SGBD (logiciels d'interrogation interactifs, éditeurs de rapports, etc.). Les programmes sont écrits dans un langage de programmation traditionnel appelé langage de 3e génération (C, COBOL, FORTRAN, etc.) ou dans un langage plus avancé intégrant des facilités de gestion d'écrans et d'édition de rapports appelé langage de 4e génération (Visual BASIC, SQL/FORMS, MANTIS, etc.). Dans tous les cas, ils accèdent à la base à l'aide d'un langage unifié de description et manipulation de données permettant les recherches et les mises à jour (par exemple, le langage SQL). Cette vision simplifiée d'un SGBD et de son environnement est représentée figure II.1.

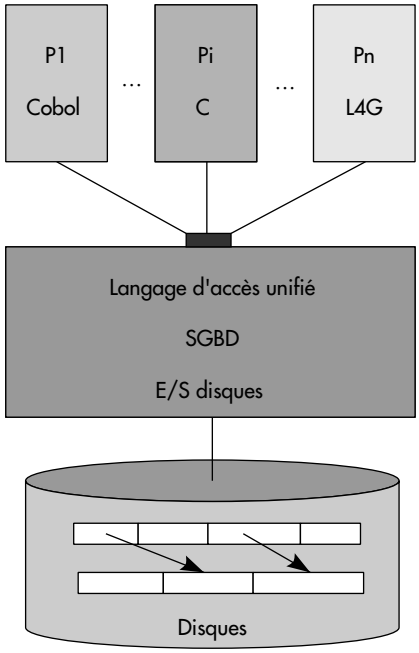


Figure II.1 : Composants d'un environnement base de données

L'objectif de ce chapitre est d'essayer de clarifier la notion plus ou moins floue de SGBD. Pour cela nous présentons d'abord les objectifs que ces systèmes cherchent à atteindre. Bien sûr, peu de SGBD satisfont pleinement tous ces objectifs, mais ils y tendent tous plus ou moins. Ensuite, nous exposerons les méthodes et concepts de base nécessaires à la compréhension générale du fonctionnement des SGBD, puis l'architecture fonctionnelle de référence proposée par le groupe de normalisation ANSI/X3/SPARC. Une bonne compréhension de cette architecture est essentielle à la compréhension des SGBD proposés jusqu'à ce jour. Pour conclure le chapitre, nous étudierons diverses architectures opérationnelles proposées par des groupes de normalisation ou des constructeurs de SGBD, telles l'architecture client-serveur à deux ou trois states (*2-tiers* ou *3-tiers*) implantée aujourd'hui par beaucoup de constructeurs.

2. MODELISATION DES DONNEES

Une idée centrale des bases de données est de séparer la description des données effectuée par les administrateurs de la manipulation effectuée par les programmes d'application. La description permet de spécifier les structures et les types de données de l'application alors que la manipulation consiste à effectuer des interrogations, des insertions et des mises à jour. Dès 1965, l'idée de décrire les données des applications de manière indépendante des traitements fut proposée. Aujourd'hui, plusieurs niveaux de description gérés par un SGBD permettent de réaliser des abstractions progressives des données stockées sur disques, de façon à s'approcher de la vision particulière de chaque utilisateur.

2.1. INSTANCES ET SCHÉMAS

Toute description de données consiste à définir les propriétés d'ensembles d'objets modélisés dans la base de données, et non pas d'objets particuliers. Les objets particuliers sont définis par les programmes d'applications lors des insertions et mises à jour de données. Ils doivent alors vérifier les propriétés des ensembles auxquels ils appartiennent. On distingue ainsi deux notions essentielles :

– le **type d'objet** permet de spécifier les propriétés communes à un ensemble d'objets en termes de structures de données visible et d'opérations d'accès,
– l'**instance d'objet** correspond à un objet particulier identifiable parmi les objets d'un type.

Bien qu'**occurrence** soit aussi employé, on préfère aujourd'hui le terme d'**instance**. Nous précisons ci-dessous ces notions en les illustrant par des exemples.

Notion II.1 : Type d'objet (*Object type*)

Ensemble d'objets possédant des caractéristiques similaires et manipulables par des opérations identiques.

EXEMPLES

1. Le type d'objet Entier = {0, ± 1… ± N… ± ∞} muni des opérations standards de l'arithmétique {+, *, /, –} est un type d'objet élémentaire, supporté par tous les systèmes.
2. Le type d'objet Vin possédant les propriétés Cru, Millésime, Qualité, Quantité peut être muni des opérations Produire et Boire, qui permettent respectivement d'accroître et de décroître la quantité. C'est un type d'objet composé pouvant être utilisé dans une application particulière, gérant par exemple des coopératives vinicoles.

3. Le type d'objet Entité possédant les propriétés P1, P2…Pn et muni des opérations Créer, Consulter, Modifier, Détruire est un type d'objet générique qui permet de modéliser de manière très générale la plupart des objets du monde réel. ■

Notion II.2 : Instance d'objet (*Object instance*)

Élément particulier d'un type d'objets, caractérisé par un identifiant et des valeurs de propriétés.

EXEMPLES

1. L'entier 10 est une instance du type Entier.
2. Le vin (Volnay, 1992, Excellent, 1000) est une instance du type Vin.
3. L'entité e(a1, a2, …an), où a1, a2…,an désignent des valeurs particulières, est une instance du type Entité. ■

Toute description de données s'effectue donc au niveau du type, à l'aide d'un ensemble d'éléments descriptifs permettant d'exprimer les propriétés d'ensembles d'objets et composant un **modèle de description de données**. Ce dernier est souvent représenté par un formalisme graphique. Il est mis en œuvre à l'aide d'un **langage de description de données (LDD)**. La description d'un ensemble de données particulier, correspondant par exemple à une application, à l'aide d'un langage de description, donne naissance à un **schéma de données.** On distingue généralement le schéma source spécifié par les **administrateurs de données** et le schéma objet résultant de la compilation du précédent par une machine. Le schéma objet est directement utilisable par le système de gestion de bases de données afin de retrouver et de vérifier les propriétés des instances d'objets manipulées par les programmes d'applications.

Notion II.3 : Modèle de description de données (*Data model*)

Ensemble de concepts et de règles de composition de ces concepts permettant de décrire des données.

Notion II.4 : Langage de description de données (*Data description language*)

Langage supportant un modèle et permettant de décrire les données d'une base d'une manière assimilable par une machine.

Notion II.5 : Schéma (*Schema*)

Description au moyen d'un langage déterminé d'un ensemble de données particulier.

2.2. NIVEAUX D'ABSTRACTION

Un objectif majeur des SGBD est d'assurer une abstraction des données stockées sur

disques pour simplifier la vision des utilisateurs. Pour cela, trois niveaux de description de données ont été distingués par le groupe ANSI/X3/SPARC [ANSI78, Tsichritzis78]. Ces niveaux ne sont pas clairement distingués par tous les SGBD : ils sont mélangés en deux niveaux dans beaucoup de systèmes existants. Cependant, la conception d'une base de données nécessite de considérer et de spécifier ces trois niveaux, certains pouvant être pris en compte par les outils de génie logiciel aidant à la construction des applications autour du SGBD.

2.2.1. Le niveau conceptuel

Le niveau central est le niveau conceptuel. Il correspond à la structure canonique des données qui existent dans l'entreprise, c'est-à-dire leur structure sémantique inhérente sans souci d'implantation en machine, représentant la vue intégrée de tous les utilisateurs. La définition du **schéma conceptuel** d'une entreprise ou d'une application n'est pas un travail évident. Ceci nécessite un accord sur les concepts de base que modélisent les données. Par exemple, le schéma conceptuel permettra de définir :

1. les types de données élémentaires qui définissent les propriétés élémentaires des objets de l'entreprise (cru d'un vin, millésime, qualité, etc.) ;
2. les types de données composés qui permettent de regrouper les attributs afin de décrire les objets du monde réel ou les relations entre objets (vin, personne, buveur, etc.) ;
3. les types de données composés qui permettent de regrouper les attributs afin de décrire les associations du monde réel (abus de vin par un buveur, production d'un vin par un producteur, etc.) ;
4. éventuellement, des règles que devront suivre les données au cours de leur vie dans l'entreprise (l'âge d'une personne est compris entre 0 et 150, tout vin doit avoir un producteur, etc.).

Un exemple de schéma conceptuel défini en termes de types d'objets composés (souvent appelés entités) et d'associations entre ces objets est représenté figure II.2. Le type d'objet Buveur spécifie les propriétés d'un ensemble de personnes qui consomment des vins. Le type d'objet Vin a été introduit ci-dessous. Une consommation de vin (abusivement appelée ABUS) associe un vin et un buveur. La consommation s'effectue à une date donnée en une quantité précisée.

```
Type d'objets :
BUVEUR(Nom, Prénom, Adresse)
VIN(Cru, Millésime, Qualité, Quantité)
Type d'associations :
ABUS(BUVEUR, VIN, Date, Quantité)
```

Figure II.2 : Exemple de schéma conceptuel

2.2.2. Le niveau interne

Le niveau interne correspond à la structure de stockage supportant les données. La définition du **schéma interne** nécessite au préalable le choix d'un SGBD. Elle permet donc de décrire les données telles qu'elles sont stockées dans la machine, par exemple :

– les fichiers qui les contiennent (nom, organisation, localisation…) ;
– les articles de ces fichiers (longueur, champs composants, modes de placement en fichiers…) ;
– les chemins d'accès à ces articles (index, chaînages, fichiers inversés…).

Une implémentation possible des données représentées figure II.2 est illustrée figure II.3. Il existe un fichier indexé sur le couple (Cru, Millésime) dont chaque article contient successivement le cru, le millésime, la qualité, la quantité stockée de vin. Il existe un autre fichier décrivant les buveurs et leurs abus. Chaque article de ce deuxième fichier contient le nom, le prénom et l'adresse d'un buveur suivi d'un groupe répétitif correspondant aux abus comprenant le nombre d'abus et pour chacun d'eux un pointeur sur le vin bu, la date et la quantité. Un index sur le nom du buveur permet d'accéder directement aux articles de ce fichier. Un autre index permet aussi d'y accéder par la date des abus.

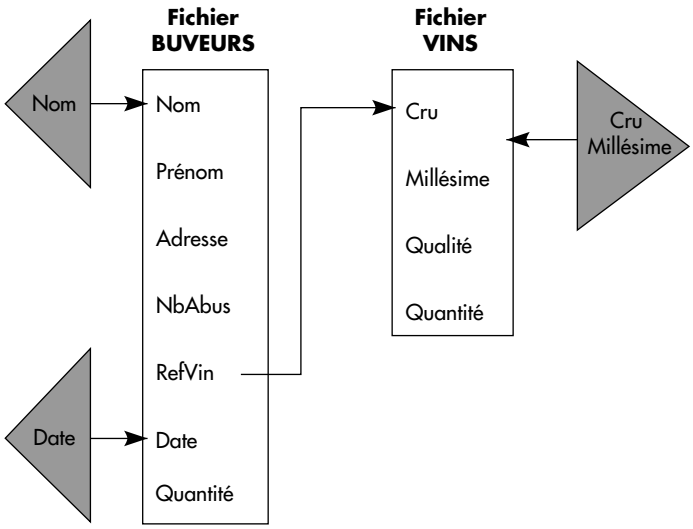


Figure II.3 : Exemple de schéma interne

2.2.3. Le niveau externe

Au niveau externe, chaque groupe de travail utilisant des données possède une description des données perçues, appelée **schéma externe**. Cette description est effectuée selon la manière dont le groupe voit la base dans ses programmes d'application. Alors

qu'au niveau conceptuel et interne les schémas décrivent toute une base de données, au niveau externe ils décrivent simplement la partie des données présentant un intérêt pour un utilisateur ou un groupe d'utilisateurs. En conséquence, un schéma externe est souvent qualifié de vue externe. Le modèle externe utilisé est dépendant du langage de manipulation de la base de données utilisé. La figure II.4 donne deux exemples de schéma externe pour la base de données dont le schéma conceptuel est représenté figure II.2. Il est à souligner que la notion de schéma externe permet d'assurer une certaine sécurité des données. Un groupe de travail ne peut en effet accéder qu'aux données décrites dans son schéma externe. Les autres données sont ainsi protégées contre les accès non autorisés ou mal intentionnés de la part de ce groupe de travail.

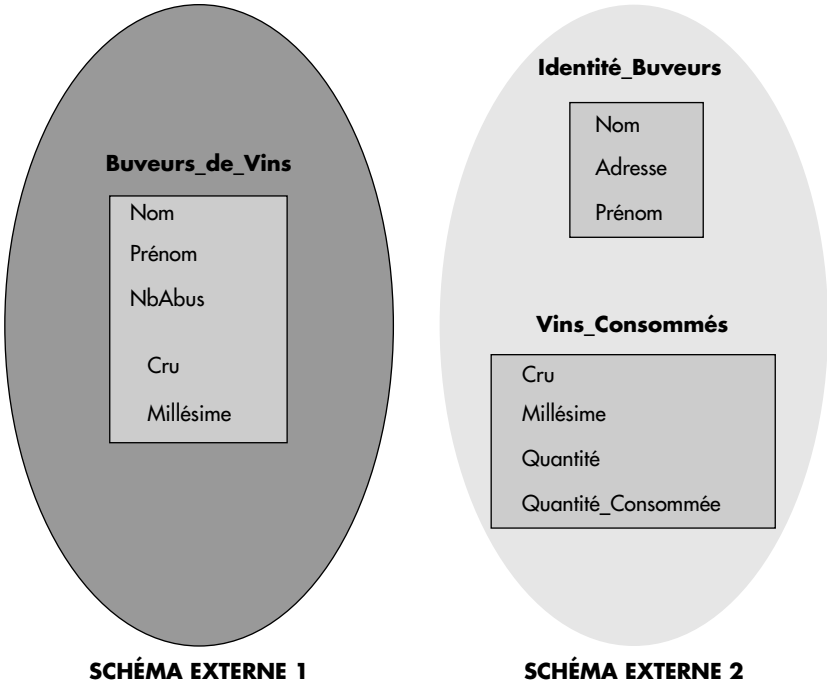


Figure II.4 : Exemples de schémas externes

2.2.4. Synthèse des niveaux de schémas

Retenez que, pour une base particulière, il existe un seul schéma interne et un seul schéma conceptuel. En revanche, il existe en général plusieurs schémas externes. Un schéma externe peut être défini par un groupe d'utilisateurs. À partir de là, il est possible de construire des schémas externes pour des sous-groupes du groupe d'utilisateurs considéré. Ainsi, certains schémas externes peuvent être déduits les uns des autres. La figure II.5 illustre les différents schémas d'une base de données centralisée.

Nous rappelons ci-dessous ce que représente chacun des niveaux de schéma à l'aide d'une notion.

Notion II.6 : Schéma conceptuel (*Conceptual Schema*)

Description des données d'une entreprise ou d'une partie d'une entreprise en termes de types d'objets et de liens logiques indépendants de toute représentation en machine, correspondant à une vue canonique globale de la portion d'entreprise modélisée.

Notion II.7 : Schéma interne (*Internal Schema*)

Description des données d'une base en termes de représentation physique en machine, correspondant à une spécification des structures de mémorisation et des méthodes de stockage et d'accès utilisées pour ranger et retrouver les données sur disques.

Notion II.8 : Schéma externe (*External Schema*)

Description d'une partie de la base de données extraite ou calculée à partir de la base physique, correspondant à la vision d'un programme ou d'un utilisateur, donc à un arrangement particulier de certaines données.

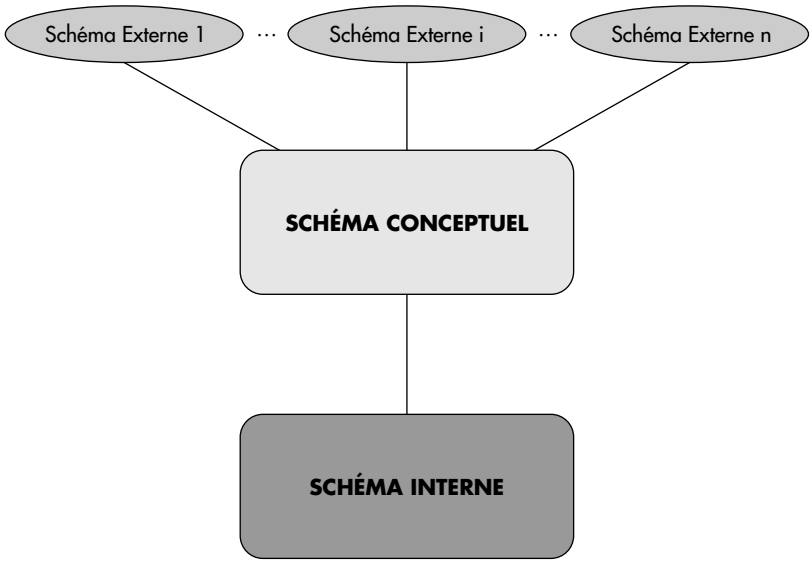


Figure II.5 : Les trois niveaux de schémas

2.3. LE MODÈLE ENTITÉ-ASSOCIATION

Les données élémentaires décrivent des événements atomiques du monde réel. Par exemple, la donnée « 10 000 francs » peut correspondre à une instance de salaire ou

de prix. Dupont Jules est un nom. Dans les bases de données, des instances de types élémentaires sont groupées ensemble pour constituer un objet composé. L'abstraction qui concatène des données élémentaires (et plus généralement des objets) est appelée l'**agrégation**. La figure II.6 représente par un graphe l'agrégation des données (Volnay, 1978, Excellente, 100) pour constituer un objet composé décrivant le vin identifié par Volnay78.

Notion II.9 : Agrégation (*Aggregation*)

Abstraction consistant à grouper des objets pour constituer des objets composés d'une concaténation d'objets composants.

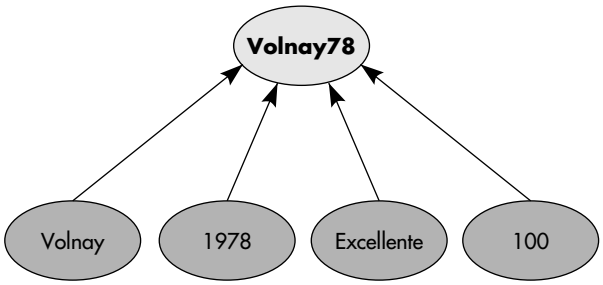


Figure II. 6 : Exemple d'agrégation de valeurs

Le modèle entité-association [Chen76] est basé sur une perception du monde réel qui consiste à distinguer des agrégations de données élémentaires appelées **entités** et des liaisons entre entités appelées **associations**. Intuitivement, une entité correspond à un objet du monde réel généralement défini par un nom, par exemple un vin, un buveur, une voiture, une commande, etc. Une entité est une agrégation de données élémentaires. Un type d'entité définit un ensemble d'entités constitué par des données de même type. Les types de données agrégées sont appelés les attributs de l'entité ; ils définissent ses propriétés.

Notion II.10 : Entité (*Entity*)

Modèle d'objet identifié du monde réel dont le type est défini par un nom et une liste de propriétés.

Une association correspond à un lien logique entre deux entités ou plus. Elle est souvent définie par un verbe du langage naturel. Une association peut avoir des propriétés particulières définies par des attributs spécifiques.

Notion II.11 : Association (*Relationship*)

Lien logique entre entités dont le type est défini par un verbe et une liste éventuelle de propriétés.

Notion II.12 : Attribut (*Attribute*)

Propriété d'une entité ou d'une association caractérisée par un nom et un type élémentaire.

Le modèle entité-association, qui se résume aux trois concepts précédents, permet de modéliser simplement des situations décrites en langage naturel : les noms correspondent aux entités, les verbes aux associations et les adjectifs ou compléments aux propriétés. Il s'agit là bien sûr d'une abstraction très schématique d'un sous-ensemble réduit du langage naturel que nous illustrons par un exemple simple.

EXEMPLE

Les buveurs abusent de vins en certaines quantités à des dates données. Tout buveur a un nom, un prénom, une adresse et un type. Un vin est caractérisé par un cru, un millésime, une qualité, une quantité et un degré.

Il est possible de se mettre d'accord au niveau conceptuel pour représenter une telle situation par le schéma entité-association suivant :

- entités = {BUVEUR, VIN};
- association = {ABUS};
- attributs = {Nom, Prénom, Adresse et Type pour BUVEUR; Cru, Millésime, Qualité, Quantité et Degré pour VIN; Quantité et Date pour ABUS}. ■

Un des mérites essentiels du modèle entité-association est de permettre une représentation graphique élégante des schémas de bases de données [Chen76]. Un rectangle représente une entité ; un losange représente une association entre entités ; une ellipse représente un attribut. Les losanges sont connectés aux entités qu'ils associent par des lignes. Les attributs sont aussi connectés aux losanges ou rectangles qu'ils caractérisent. La figure II.7 représente le diagramme entité-association correspondant à la situation décrite dans l'exemple ci-dessus.

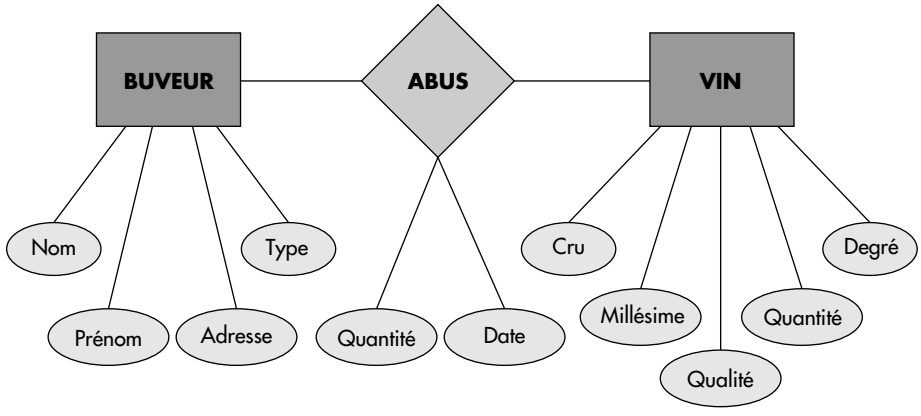


Figure II.7 : Exemple de diagramme entité-association

3. OBJECTIFS DES SGBD

Le principal objectif d'un SGBD est d'assurer l'indépendance des programmes aux données, c'est-à-dire la possibilité de modifier les schémas conceptuel et interne des données sans modifier les programmes d'applications, et donc les schémas externes vus par ces programmes. Cet objectif est justifié afin d'éviter une maintenance coûteuse des programmes lors des modifications des structures logiques (le découpage en champs et articles) et physiques (le mode de stockage) des données. Plus précisément, on distingue l'indépendance physique qui permet de changer les schémas internes sans changer les programmes d'applications, et l'indépendance logique qui permet de modifier les schémas conceptuels (par exemple, ajouter un type d'objet) sans changer les programmes d'applications.

Afin d'assurer encore une meilleure indépendance des programmes aux données est rapidement apparue la nécessité de manipuler (c'est-à-dire d'interroger et de mettre à jour) les données par des langages de haut niveau spécifiant celles que l'on veut traiter (le quoi) et non pas comment y accéder. Ainsi, les procédures d'accès aux données restent invisibles aux programmes d'application qui utilisent donc des langages non procéduraux. Ces langages référencent des descriptions logiques des données (les schémas externes) stockées dans le dictionnaire de données. Les descriptions de données, qui existent à plusieurs niveaux introduits ci-dessus, sont établies par les administrateurs de données. Un SGBD se doit donc de faciliter l'administration (c'est-à-dire la création et la modification de la description) des données. En résumé, voici les objectifs premiers d'un SGBD :

– Indépendance physique des programmes aux données
– Indépendance logique des programmes aux données
– Manipulation des données par des langages non procéduraux
– Administration facilitée des données.

Les SGBD conduisent à mettre en commun les données d'une entreprise, ou au moins d'une application dans une base de données décrite par un dictionnaire de données. Cette mise en commun ne va pas sans problèmes d'efficacité: de nombreux utilisateurs accèdent simultanément aux données souvent situées sur un même disque. La base de données devient ainsi un goulot d'étranglement. Il faut assurer globalement l'efficacité des accès. Il faut aussi garantir les utilisateurs contre les mises à jour concurrentes, et donc assurer le partage des données. L'environnement multi-usager nécessite de protéger la base de données contre les mises à jour erronées ou non autorisées: il faut assurer la cohérence des données. Notamment, des données redondantes doivent rester égales. Enfin, en cas de panne système, ou plus simplement d'erreurs de programmes, il faut assurer la sécurité des données, en permettant par exemple de repartir sur des versions correctes. En résumé, voici les objectifs additionnels des SGBD, qui sont en fait des conséquences des objectifs premiers :

– Efficacité des accès aux données
– Partage des données
– Cohérence des données
– Redondance contrôlée des données
– Sécurité des données.

Dans la pratique, ces objectifs ne sont que très partiellement atteints. Ci-dessous nous analysons plus précisément chacun d'eux.

3.1. INDÉPENDANCE PHYSIQUE

Bien souvent, les données élémentaires du monde réel sont assemblées pour décrire les objets et les associations entre objets directement perceptibles dans le monde réel. Bien que souvent deux groupes de travail assemblent différemment des données élémentaires, il est possible au sein d'une entreprise bien organisée de définir une structure canonique des données, c'est-à-dire un partitionnement en ensembles et sous-ensembles ayant des propriétés bien définies et cohérentes avec les vues particulières. Cet assemblage peut être considéré comme l'intégration des vues particulières de chaque groupe de travail. Il obéit à des règles qui traduisent l'essentiel des propriétés des données élémentaires dans le monde réel. Il correspond au schéma conceptuel d'une base de données.

Par opposition, la structure de stockage des données appartient au monde des informaticiens et n'a donc un sens que dans l'univers du système informatique. Le schéma interne décrit un assemblage physique des données en articles, fichiers, chemins d'accès (organisation et méthode d'accès des fichiers, modes de placement des articles dans les fichiers, critères de tri, chaînages…) sur des mémoires secondaires. Cet assemblage propre au monde informatique doit être basé sur des considérations de performances et de souplesse d'accès.

Un des objectifs essentiels des SGBD est donc de permettre de réaliser l'indépendance des structures de stockage aux structures de données du monde réel [Stonebraker74], c'est-à-dire entre le schéma interne et le schéma conceptuel. Bien sûr, ces deux schémas décrivent les mêmes données, mais à des niveaux différents. Il s'agit donc de pouvoir modifier le schéma interne sans avoir à modifier le schéma conceptuel, en tenant compte seulement des critères de performance et de flexibilité d'accès. On pourra par exemple ajouter un index, regrouper deux fichiers en un, changer l'ordre ou le codage des données dans un article, sans mettre en cause les entités et associations définies au niveau conceptuel.

Les avantages de l'indépendance physique peuvent être facilement compris si l'on considère les inconvénients de la non-indépendance physique. Celle-ci impliquerait que la manière dont les données sont organisées sur mémoire secondaire soit directe-

ment l'image de l'organisation canonique de données dans le monde réel. Pour permettre de conserver les possibilités d'optimisation de performances vitales aux systèmes informatiques, les notions de méthodes d'accès, modes de placement, critères de tri, chaînages et codages de données devraient directement apparaître dans le monde réel et donc dans les applications. Tout changement informatique devrait alors être répercuté dans la vie d'une entreprise et par conséquent impliquerait une reconstruction des applications. Cela est bien sûr impraticable, d'où la nécessité d'indépendance des structures de stockages aux données du monde réel.

3.2. INDÉPENDANCE LOGIQUE

Nous avons admis ci-dessus l'existence d'un schéma conceptuel modélisant les objets et associations entre objets dans le monde réel. Ce schéma résulte d'une synthèse des vues particulières de chaque groupe de travail utilisant la base de données, c'est-à-dire d'une intégration de schémas externes. En conséquence, chaque groupe de travail réalisant une application doit pouvoir assembler différemment les données pour former par exemple les entités et les associations de son schéma externe, ou plus simplement des tables qu'il souhaite visualiser. Ainsi, chacun doit pouvoir se concentrer sur les éléments constituant son centre d'intérêt, c'est-à-dire qu'un utilisateur doit pouvoir ne connaître qu'une partie des données de la base au travers de son schéma externe, encore appelé **vue**.

Il est donc souhaitable de permettre une certaine indépendance des données vues par les applications à la structure canonique des données de l'entreprise décrite dans le schéma conceptuel. L'indépendance logique est donc la possibilité de modifier un schéma externe sans modifier le schéma conceptuel. Elle assure aussi l'indépendance entre les différents utilisateurs, chacun percevant une partie de la base via son schéma externe, selon une structuration voire un modèle particulier.

Les avantages de l'indépendance logique [Date71] sont les suivants :

– permettre à chaque groupe de travail de voir les données comme il le souhaite ;
– permettre l'évolution de la vue d'un groupe de travail (d'un schéma externe) sans remettre en cause, au moins dans une certaine mesure, le schéma conceptuel de l'entreprise ;
– permettre l'évolution d'un schéma externe sans remettre en cause les autres schémas externes.

En résumé, il doit être possible d'ajouter des attributs, d'en supprimer d'autres, d'ajouter et de supprimer des associations, d'ajouter ou de supprimer des entités, etc., dans des schémas externes mais aussi dans le schéma conceptuel sans modifier la plus grande partie des applications.

3.3. MANIPULATION DES DONNÉES PAR DES LANGAGES NON PROCÉDURAUX

Les utilisateurs, parfois non professionnels de l'informatique, doivent pouvoir manipuler simplement les données, c'est-à-dire les interroger et les mettre à jour sans préciser les algorithmes d'accès. Plus généralement, si les objectifs d'indépendance sont atteints, les utilisateurs voient les données indépendamment de leur implantation en machine. De ce fait, ils doivent pouvoir manipuler les données au moyen de langages non procéduraux, c'est-à-dire en décrivant les données qu'ils souhaitent retrouver (ou mettre à jour) sans décrire la manière de les retrouver (ou de les mettre à jour) qui est propre à la machine. Les langages non procéduraux sont basés sur des assertions de logique du premier ordre. Ils permettent de définir les objets désirés au moyen de relations entre objets et de propriétés de ces objets.

Deux sortes d'utilisateurs manipulent en fait les bases de données : les utilisateurs interactifs et les programmeurs. Les utilisateurs interactifs peuvent interroger voire mettre à jour la base de données. Ils sont parfois non informaticiens et réclament des langages simples. De tels langages peuvent être formels (logique du premier ordre) ou informels (menus). Une large variété de langages interactifs doivent être supportés par un SGBD, depuis les langages de commandes semi-formels jusqu'aux langages graphiques, en passant par l'interrogation par menus ou par formes. La limite supérieure de tels langages est le langage naturel, qui reste cependant en général trop complexe et lourd pour interroger les bases de données.

Les programmeurs écrivent des programmes en utilisant des langages traditionnels dits de 3^{e} génération (C, COBOL, PL1, etc.), des langages plus récents orientés objet tels C++ ou Java ou des langages de 4^{e} génération (VB, PL/SQL, FORTÉ, etc.). Ces derniers regroupent des instructions de programmation structurée (WHILE, IF, CASE, etc.), des expressions arithmétiques, des commandes d'accès à la base de données et des commandes d'édition et entrée de messages (menus déroulants, gestion de fenêtres, rapports imprimés, etc.). Ils sont de plus en plus souvent orientés objets. Dans tous les cas, il est important que le SGBD fournisse les commandes nécessaires de recherche et mise à jour de données pour pouvoir accéder aux bases. Une intégration harmonieuse avec le langage de programmation, qui traite en général un objet à la fois, est souhaitable.

3.4. ADMINISTRATION FACILITÉE DES DONNÉES

Un SGBD doit fournir des outils pour décrire les données, à la fois leurs structures de stockage et leurs présentations externes. Il doit permettre le suivi de l'adéquation de ces structures aux besoins des applications et autoriser leur évolution aisée. Les fonc-

tions qui permettent de définir les données et de changer leur définition sont appelées outils d'administration des données. Afin de permettre un contrôle efficace des données, de résoudre les conflits entre divers points de vue pas toujours cohérents, de pouvoir optimiser les accès aux données et l'utilisation des moyens informatiques, on a pensé à centraliser ces fonctions entre les mains d'un petit groupe de personnes hautement qualifiées, appelées administrateurs de données.

En fait, la centralisation des descriptions de données entre les mains d'un groupe spécialisé a souvent conduit à des difficultés d'organisation. Aussi, l'évolution des SGBD modernes tend à fournir des outils permettant de décentraliser la description de données, tout en assurant une cohérence entre les diverses descriptions partielles. Un dictionnaire de données dynamique pourra ainsi aider les concepteurs de bases de données. Pour permettre une évolution rapide, les descriptions de données devront être faciles à consulter et à modifier. L'évolution va donc vers le développement d'outils intégrés capables de faciliter l'administration des données et d'assurer la cohérence des descriptions.

3.5. EFFICACITÉ DES ACCÈS AUX DONNÉES

Les performances en termes de débit (nombre de transactions types exécutées par seconde) et de temps de réponse (temps d'attente moyen pour une requête type) sont un problème clé des SGBD. L'objectif de débit élevé nécessite un *overhead* minimal dans la gestion des tâches accomplie par le système. L'objectif de bons temps de réponse implique qu'une requête courte d'un utilisateur n'attende pas une requête longue d'un autre utilisateur. Il faut donc partager les ressources (unités centrales, unités d'entrées-sorties) entre les utilisateurs en optimisant l'utilisation globale et en évitant les pertes en commutation de contextes.

Le goulot d'étranglement essentiel dans les systèmes de bases de données reste les E/S disques. Une E/S disque coûte en effet quelques dizaines de millisecondes. Afin de les éviter, on utilisera une gestion de tampons en mémoire centrale dans de véritables mémoires caches des disques, afin qu'un grand nombre d'accès aux données se fasse en mémoire. Un autre facteur limitatif est dû à l'utilisation de langages non procéduraux très puissants afin d'interroger et mettre à jour la base de données. Ainsi, il devient possible de demander en une requête le tri d'un grand volume de données. Il devient donc aussi nécessaire d'optimiser l'activité de l'unité centrale pour traiter les opérations en mémoire. En résumé, un SGBD devra chercher à optimiser une fonction de coût de la forme C(Q) = a * ES(Q) + b * UC(Q) pour un ensemble typique de requêtes (recherches et mises à jour) Q ; ES(Q) est le nombre d'entrées-sorties réalisées pour la requête Q et UC(Q) est le temps unité centrale dépensé ; a et b sont des facteurs convertissant entrées-sorties et temps d'unité centrale en coûts.

3.6. REDONDANCE CONTRÔLÉE DES DONNÉES

Dans les systèmes classiques à fichiers non intégrés, chaque application possède ses données propres. Cela conduit généralement à de nombreuses duplications de données avec, outre la perte en mémoire secondaire associée, un gâchis important en moyens humains pour saisir et maintenir à jour plusieurs fois les mêmes données. Avec une approche base de données, les fichiers plus ou moins redondants seront intégrés en un seul fichier partagé par les diverses applications. L'administration centralisée des données conduisait donc naturellement à la non-duplication physique des données afin d'éviter les mises à jour multiples.

En fait, avec les bases de données réparties sur plusieurs calculateurs interconnectés, il est apparu souhaitable de faire gérer par le système des copies multiples de données. Cela optimise les performances en interrogation, en évitant les transferts sur le réseau et en permettant le parallélisme des accès. On considère donc aujourd'hui que la redondance gérée par le SGBD au niveau physique des données n'est pas forcément mauvaise. Il faudra par contre éviter la redondance anarchique, non connue du système, qui conduirait les programmes utilisateurs à devoir mettre à jour plusieurs fois une même donnée. Il s'agit donc de bien contrôler la redondance, qui permet d'optimiser les performances, en la gérant de manière invisible pour les utilisateurs.

3.7. COHÉRENCE DES DONNÉES

Bien que les redondances anarchiques entre données soient évitées par l'objectif précédent, les données vues par l'utilisateur ne sont pas indépendantes. Au niveau d'ensemble de données, il peut exister certaines dépendances entre données. Par exemple, une donnée représentant le nombre de commandes d'un client doit correspondre au nombre de commandes dans la base. Plus simplement, une donnée élémentaire doit respecter un format et ne peut souvent prendre une valeur quelconque. Par exemple, un salaire mensuel doit être supérieur à 4 700 F et doit raisonnablement rester inférieur à 700 000 F. Un système de gestion de bases de données doit veiller à ce que les applications respectent ces règles lors des modifications des données et ainsi assurer la cohérence des données. Les règles que doivent explicitement ou implicitement suivre les données au cours de leur évolution sont appelées contraintes d'intégrité.

3.8. PARTAGE DES DONNÉES

L'objectif est ici de permettre aux applications de partager les données de la base dans le temps mais aussi simultanément. Une application doit pouvoir accéder aux données comme si elle était seule à les utiliser, sans attendre mais aussi sans savoir qu'une autre application peut les modifier concurremment.

En pratique, un utilisateur exécute des programmes généralement courts qui mettent à jour et consultent la base de données. Un tel programme interactif, appelé transaction, correspond par exemple à l'entrée d'un produit en stock ou à une réservation de place d'avion. Il est important que deux transactions concurrentes (par exemple, deux réservations sur le même avion) ne s'emmêlent pas dans leurs accès à la base de données (par exemple, réservent le même siège pour deux passagers différents). On cherchera donc à assurer que le résultat d'une exécution simultanée de transactions reste le même que celui d'une exécution séquentielle dans un ordre quelconque des transactions.

3.9. SÉCURITÉ DES DONNÉES

Cet objectif a deux aspects. Tout d'abord, les données doivent être protégées contre les accès non autorisés ou mal intentionnés. Il doit exister des mécanismes adéquats pour autoriser, contrôler ou enlever les droits d'accès de n'importe quel usager à tout ensemble de données. Les droits d'accès peuvent également dépendre de la valeur des données ou des accès précédemment effectués par l'usager. Par exemple, un employé pourra connaître les salaires des personnes qu'il dirige mais pas des autres employés de l'entreprise.

D'un autre côté, la sécurité des données doit aussi être assurée en cas de panne d'un programme ou du système, voire de la machine. Un bon SGBD doit être capable de restaurer des données cohérentes après une panne disque, bien sûr à partir de sauvegardes. Aussi, si une transaction commence une mise à jour (par exemple un transfert depuis votre compte en banque sur celui de l'auteur) et est interrompue par une panne en cours de mise à jour (par exemple après avoir débité votre compte en banque), le SGBD doit assurer l'intégrité de la base (c'est-à-dire que la somme d'argent gérée doit rester constante) et par suite défaire la transaction qui a échoué. Une transaction doit donc être totalement exécutée, ou pas du tout : il faut assurer l'atomicité des transactions, et ainsi garantir l'intégrité physique de la base de données.

4. FONCTIONS DES SGBD

Cette section présente les fonctions essentielles d'un SGBD. Un SGBD permet de décrire les données des bases, de les interroger, de les mettre à jour, de transformer des représentations de données, d'assurer les contrôles d'intégrité, de concurrence et de sécurité. Il supporte de plus en plus fréquemment des fonctions avancées pour la gestion de procédures et d'événements. Toutes ces fonctionnalités sont illustrées par des exemples simples.

4.1. DESCRIPTION DES DONNÉES

Un SGBD offre donc des interfaces pour décrire les données. La définition des différents schémas est effectuée par les **administrateurs** de données ou par les personnes jouant le rôle d'administrateur.

Notion II.13 : Administrateur de données (*Data Administrator*)

Personne responsable de la définition de schémas de bases de données.

Dans un SGBD ou un environnement de développement de bases de données supportant trois niveaux de schémas, les administrateurs de données ont trois rôles :

– **Administrateur de bases de données**. L'exécutant de ce rôle est chargé de la définition du schéma interne et des règles de correspondance entre les schémas interne à conceptuel.
– **Administrateur d'entreprise**. Le porteur de ce rôle est chargé de la définition du schéma conceptuel.
– **Administrateur d'application.** L'attributaire est chargé de la définition des schémas externes et des règles de correspondance entre les schémas externe et conceptuel.

Ces trois rôles peuvent être accomplis par les mêmes personnes ou par des personnes différentes. Un rôle essentiel est celui d'administrateur d'entreprise, qui inclut la définition des informations que contient la base de données au niveau sémantique, par exemple avec des diagrammes entité-association. La plupart des SGBD modernes supportent seulement un schéma interne et plusieurs schémas externes. Le schéma conceptuel est défini en utilisant un outil d'aide à la conception (par exemple au sein d'un atelier de génie logiciel) s'appuyant généralement sur des interfaces graphiques permettant d'élaborer des diagrammes de type entité-association.

Quoi qu'il en soit, les différents schémas et procédures pour passer de l'un à l'autre sont stockés dans le **dictionnaire des données**. Celui-ci peut être divisé en deux dictionnaires : le dictionnaire d'entreprise qui contient le schéma conceptuel et les procédures et commentaires s'appliquant sur ce schéma, et le dictionnaire des bases qui contient les schémas internes et externes, ainsi que les procédures de passage d'un niveau à l'autre. Tout dictionnaire contient en général des descriptions en langage naturel permettant de préciser la signification des données. Un dictionnaire de données peut contenir des informations non strictement bases de données, telles que des masques d'écrans ou des programmes. Les informations sont souvent stockées en format source, mais aussi en format compilé. Un dictionnaire de données organisé sous forme de base de données est appelé **métabase**.

Notion II.14 : Dictionnaire des données (*Data Dictionnary*)

Ensemble des schémas et des règles de passage entre les schémas associés à une base de données, combinés à une description de la signification des données.

Notion II.15 : Métabase (*Metabase*)

Dictionnaire de données organisé sous forme de base de données qui décrit donc les autres bases.

Un SGBD fournit des commandes permettant de définir les schémas interne, conceptuel et externe. Afin d'illustrer concrètement cette fonctionnalité, voici les commandes essentielles permettant de créer un schéma avec le seul modèle présenté jusque-là, c'est-à-dire le modèle entité-association. La syntaxe dérive d'une extension du langage QUEL [Zook77] au modèle entité-association.

Voici donc les commandes minimales nécessaires :

– pour créer une base de données :

```
CREATDB <nom-de-base>
```

– pour créer une entité :

```
CREATE ENTITY <nom-d'entité> (<nom-d'attribut> <type>
[{,<nom-d'attribut> <type>}…])
```

– pour créer une association :

```
CREATE RELATIONSHIP <nom-d'association> (<nom-d'entité>
[{,<nom d'entité>}…], [{,<nom-d'attribut> <type>}…])
```

– pour détruire une entité ou une association :

```
DESTROY {<nom-de-relation> | <nom-d'association>}
```

– pour détruire une base :

```
DESTROYDB <nom-de-base>.
```

Ces commandes permettent de créer un schéma conceptuel entité-association. Elles sont utilisées dans la figure II.8 pour créer la base de données correspondant au schéma conceptuel de la figure II.7.

```
CREATE ENTITY Buveur
(Nom Char(16), Prénom Char(16), Adresse Text, Type Char(4));
CREATE ENTITY Vin
(Cru Char(10), Millésime Int, Qualité Char(10), Quantité Int, Degré Real)
CREATE RELATIONSHIP Abus
(Buveur, Vin, Date Date, Quantité Int)
```

Figure II.8 : Exemple de description de données

D'autres commandes sont nécessaires, par exemple pour créer le schéma interne. Un exemple typique à ce niveau est une commande de création d'index sur un ou plusieurs attributs d'une entité :

```
CREATE INDEX ON <nom-d'entité>
USING <nom-d'attribut> [{,<nom-d'attribut>}…]
```

Nous verrons plus loin des commandes de création de vues (schémas externes).

4.2. RECHERCHE DE DONNÉES

Tout SGBD fournit des commandes de recherche de données. Les SGBD modernes offrent un langage d'interrogation assertionnel permettant de retrouver les données par le contenu sans préciser la procédure d'accès. Les SGBD de première génération offraient des langages procéduraux permettant de rechercher un objet dans la base de données par déplacements successifs. Afin d'illustrer un langage de requête non procédural, nous introduisons informellement un langage dérivé du langage QUEL adapté au modèle entité-association.

Le langage QUEL [Zook77] est le langage de manipulation de données du système INGRES [Stonebraker76], un des premiers systèmes relationnels développé à l'université de Californie Berkeley et commercialisé sur de nombreuses machines. Ce langage est aujourd'hui peu utilisé car il a été remplacé par SQL, langage plus commercial. Il a cependant le mérite d'être à la fois simple et didactique, car dérivé de la logique du premier ordre. Nous proposons une variante simplifiée étendue au modèle entité-association [Zaniolo83].

Afin d'exprimer des questions, QUEL permet tout d'abord la définition de variables représentant un objet quelconque d'une entité ou d'une association. Une définition de variables s'effectue à l'aide de la clause :

```
RANGE OF variable IS {nom-d'entité | nom d'association}.
```

La variable est associée avec l'entité ou l'association spécifiée. Plusieurs variables associées à plusieurs entités ou associations peuvent être déclarées par une clause RANGE. Les variables déclarées demeurent en principe connues jusqu'à une nouvelle déclaration ou la fin de session.

Dans le cas de la base de données créée figure II.8, on peut par exemple définir trois variables :

```
RANGE OF B IS Buveur;
RANGE OF V IS Vin;
RANGE OF A IS Abus.
```

Une commande de recherche permet de retrouver les données de la base répondant à un critère plus ou moins complexe, appelé **qualification**. Une qualification est une expression logique (ET de OU par exemple) de critères simples, chaque critère permettant soit de comparer un attribut à une valeur, soit de parcourir une association. En QUEL, un attribut est spécifié par X.Attribut, où X est une variable et Attribut un nom d'attribut de l'entité ou de l'association représentée par la variable. Un critère simple sera donc de la forme X.Attribut = valeur pour une recherche sur valeur, ou X.Entité = Y pour un parcours d'association. D'autres fonctionnalités sont possibles, comme nous le verrons plus loin dans cet ouvrage. Une qualification est une expression logique de critères simples.

Notion II.16 : Qualification de question (*Query Qualification*)

Expression logique construite avec des OU (OR), ET (AND), NON (NOT) de critères simples permettant d'exprimer une condition que doit satisfaire les résultats d'une question.

Une recherche s'exprime alors à l'aide de requête du type suivant, où la liste résultat est une suite d'attributs de variables ou de fonctions appliquées à ces attributs :

```
RETRIEVE (liste résultat)
[WHERE qualification] ;
```

Voici quelques questions simples afin d'illustrer les capacités minimales d'un SGBD.

(Q1) Rechercher les noms et adresses des buveurs :

```
RETRIEVE B.Nom, B.Adresse;
```

(Q2) Rechercher les crus et millésimes des vins de qualité excellente :

```
RETRIEVE V.Cru, V.Millésime
WHERE V.Qualité = "Excellente" ;
```

(Q3) Rechercher les noms des gros buveurs ainsi que les crus, dates et quantités de vins qu'ils ont consommé :

```
RETRIEVE B.Nom, V.Cru, A.Date, A.Quantité
WHERE B.Type = "Gros" AND A.Buveur = B AND A.Vin = V ;
```

A priori, un SGBD doit offrir un **langage complet**, c'est-à-dire un langage permettant de poser toutes les questions possibles sur la base de données. On limite cependant aux langages du premier ordre la notion de complétude. Les questions peuvent faire intervenir un grand nombre d'entités et d'associations, des calculs, des quantificateurs (quel que soit, il existe), des restructurations de données, etc. En restant au premier ordre, il n'est pas possible de quantifier des ensembles d'objets.

Notion II.17 : Langage complet (*Complete Language*)

Langage de requêtes permettant d'exprimer toutes les questions que l'on peut poser en logique du premier ordre à une base de données.

4.3. MISE À JOUR DES DONNÉES

Le concept de mise à jour intègre à la fois l'insertion de données dans la base, la modification de données et la suppression de données. Nous illustrons ces aspects par une variation du langage QUEL adapté au modèle entité association.

Le langage présenté comporte une commande pour insérer des instances dans la base dont la syntaxe est la suivante :

```
APPEND [TO] <nom-d'entité> [(<liste-d'attributs>)]
(<liste-de-valeurs>) [{,(<liste-de-valeurs>)}]… ;
```

Cette commande permet d'ajouter les instances définies par les listes de valeurs à l'entité de nom <nom-d'entité>. Plusieurs instances d'entités peuvent ainsi être ajoutées dans la base. La liste d'attributs permet de spécifier les seuls attributs que l'on désire documenter, les autres étant a priori remplacés par une valeur nulle signifiant

inconnue. Par exemple, l'ajout du vin <Volnay, 1978, Excellente, 100> dans la base s'effectuera par la commande :

```
(U1)  APPEND TO Vin (Cru, Millésime, Qualité, Quantité)
      (Volnay, 1978, Excellente, 100) ;
```

L'insertion d'instances d'association est plus délicate car il faut insérer un enregistrement référençant des entités de la base. À titre indicatif, cela peut être fait par une commande APPEND TO dans laquelle les références aux entités connectées sont des variables calculées par une qualification. On aboutit alors à une insertion qualifiée du type :

```
APPEND [TO] <nom-d'association> [(<liste-d'attributs>)]
<liste-de-valeurs> [{,<liste-de-valeurs>}]…
WHERE <qualification> ;
```

Une liste de valeurs peut alors comprendre des attributs extraits de la base par la qualification. Par exemple, la commande suivante insère un abus de Volnay 78 au buveur Dupont :

```
(U2)  APPEND TO abus(buveur, vin, date, quantité)
      (V, B, 10-02-92, 100)
      WHERE B.Nom = "Dupont" AND V.Cru = "Volnay"
      AND V. Millésime = 1978 ;
```

La modification de données s'effectue en général par recherche des données à modifier à l'aide d'une qualification, puis par renvoi dans la base des données modifiées. La commande peut être du style suivant :

```
REPLACE  <variable><attribut> = <valeur>
         [{,<attribut> = <valeur>}…]
[WHERE qualification]
```

Elle permet de changer la valeur des attributs figurant dans la liste pour tous les tuples de la variable satisfaisant la qualification. Par exemple, l'ajout de 1 000 litres aux stocks de Volnay s'effectuera par la commande (V est supposée une variable sur l'entité Vin) :

```
(U3)  REPLACE V (Quantité = Quantité + 1.000)
      WHERE V.Cru = "Volnay" ;
```

Finalement, il est aussi possible de supprimer des tuples d'une base de données par la commande très simple :

```
DELETE variable
WHERE <qualification>
```

Par exemple, la suppression de tous les vins de millésime 1992 s'effectue par :

```
(U4)  DELETE V
      WHERE V.Millésime = 1992 ;
```

4.4. TRANSFORMATION DES DONNÉES

Comme il peut exister plusieurs niveaux de schémas gérés par système pour décrire un même ensemble de données, un système de gestion de base de données doit pouvoir assurer le passage des données depuis le format correspondant à un niveau dans le format correspondant à un autre niveau. Cette fonction est appelée **transformation de données**.

Notion II.18 : Transformation de données (*Data mapping*)

Fonction effectuant la restructuration d'instances de données conformes à un schéma en instances de données conformes à un autre schéma.

Dans un SGBD à trois niveaux de schémas, il existera donc deux niveaux de transformation :

– la transformation conceptuelle – interne permettant de faire passer des instances de données depuis le format conceptuel au format interne et réciproquement ;

– la transformation externe – conceptuelle permettant de faire passer des instances de données depuis le format conceptuel au format externe et réciproquement.

À titre d'exemple, la figure II.9 (page suivante) représente la transformation d'un ensemble d'occurrences de données depuis le format conceptuel indiqué au format externe indiqué.

Pour être capable d'effectuer automatiquement la transformation des données d'un niveau à un autre, un SGBD doit connaître les correspondances existant entre les niveaux. Pour cela, lors de la définition des schémas, le groupe d'administration des données doit expliciter comment les schémas se déduisent les uns des autres au moyen de règles de correspondance. Ces règles sont souvent exprimer sous la forme de questions.

Notion II.19 : Règles de correspondance (*Mapping rules*)

Questions définissant les procédures de transformation des données depuis un niveau de schéma dans un autre niveau.

Dans les systèmes de gestion de base de données classiques, les règles de correspondance sont bien souvent mélangées avec les schémas. Il y a cependant intérêt à distinguer ces deux notions. Par exemple, le langage QUEL permet de définir une vue de la base (schéma externe) par une commande du type suivant :

```
DEFINE VIEW <nom d'entité> (<liste-d'attributs>) AS
RETRIEVE (liste résultat)
[WHERE qualification] ;
```

La règle de correspondance entre l'entité de la vue (une table en QUEL) et le schéma de la base est clairement exprimée par une question. On pourra par exemple définir le schéma externe Gros_Buveurs comme suit, B désignant une variable sur Buveur :

```
(V1)  DEFINE VIEW Gros_Buveurs (Nom, Prénom, Adresse) AS
      RETRIEVE B.Nom, B.Prénom, B.Adresse
      WHERE B.Type = "Gros";
```

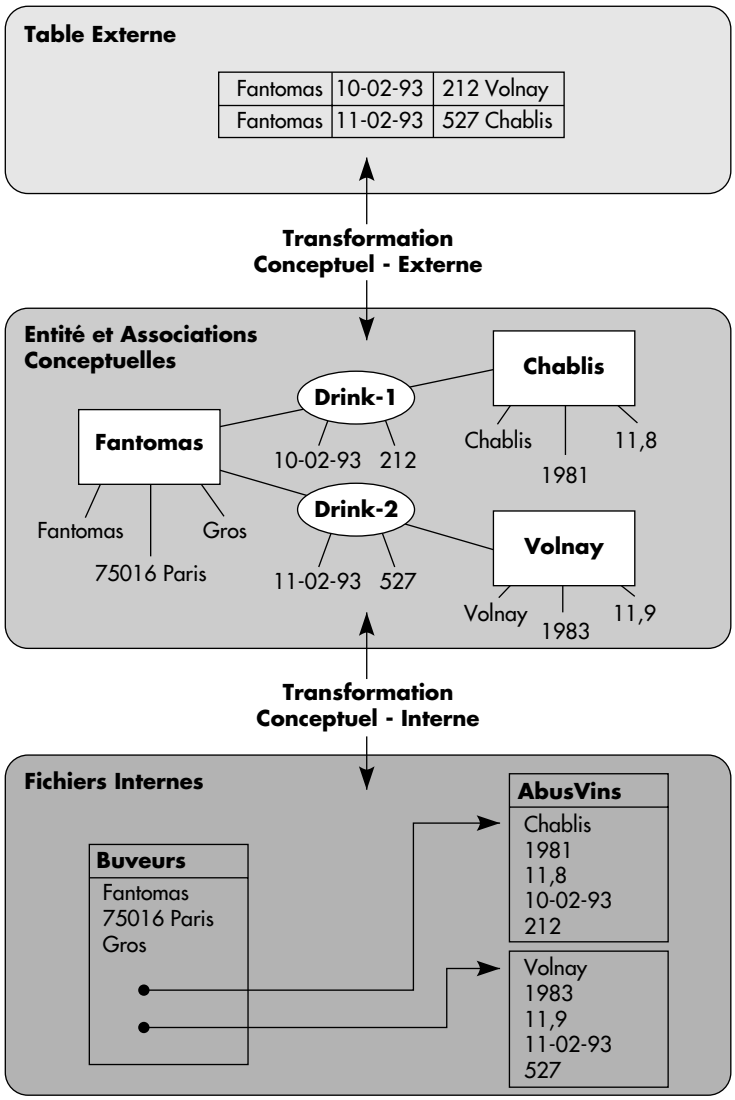


Figure II.9 : Transformation de données

4.5. CONTRÔLE DE L'INTÉGRITÉ DES DONNÉES

Comme on l'a vu au niveau des objectifs, un SGBD doit assurer le maintien de la cohérence des données par rapport aux schémas (contrôles de type), mais aussi entre elles (contrôle de redondance). On appelle **contrainte d'intégrité** toute règle implicite ou explicite que doivent suivre les données. Par exemple, si le SGBD supporte un modèle entité-association, les contraintes suivantes sont possibles :

1. Toute entité doit posséder un identifiant unique attribué par l'utilisateur. Pour les vins, nous avons supposé jusque-là que cru et millésime constituaient un identifiant. Il pourra être plus sage de numéroter les vins par un attribut numéro de vin (noté NV). Cet attribut devra être un identifiant unique, donc toujours documenté (non nul). Une telle contrainte est souvent appelé **contrainte d'unicité de clé.**
2. Certaines associations doivent associer des instances d'entité obligatoirement décrites dans la base. Ainsi, un abus ne peut être enregistré que pour un buveur et un vin existants dans la base. Une telle contrainte est souvent appelée **contrainte référentielle.**
3. Tout attribut d'entité ou d'association doit posséder une valeur qui appartient à son type. Par exemple, une quantité doit être un nombre entier. Il est même possible de préciser le domaine de variation permis pour un attribut ; par exemple, une quantité de vin peut varier entre 0 et 10 000. Une telle contrainte est souvent appelée **contrainte de domaine**.

Notion II. 20 : Contrainte d'intégrité (*Integrity Constraint*)

Règle spécifiant les valeurs permises pour certaines données, éventuellement en fonction d'autres données, et permettant d'assurer une certaine cohérence de la base de données.

En résumé, un grand nombre de type de contraintes d'intégrité est possible. Celles-ci gagnent à être déclarées au SGBD par une commande spécifique DEFINE INTEGRITY. Le SGBD doit alors les vérifier lors des mises à jour de la base.

4.6. GESTION DE TRANSACTIONS ET SÉCURITÉ

La gestion de transactions permet d'assurer qu'un groupe de mises à jour est totalement exécuté ou pas du tout. Cette propriété est connue sous le nom d'**atomicité** des transactions. Elle est garantie par le SGBD qui connaît l'existence de transactions à l'aide de deux commandes : BEGIN_TRANSACTION et END_TRANSACTION. Ces commandes permettent d'assurer que toutes les mises à jour qu'elles encadrent sont exécutées ou qu'aucune ne l'est.

Notion II.21 : Atomicité des transactions (*Transaction Atomicity*)

Propriété d'une transaction consistant à être totalement exécutée ou pas du tout.

Une transaction est donc un groupe de mises à jour qui fait passer la base d'un état à un autre état. Les états successifs doivent être cohérents et donc respecter les contraintes d'intégrité. Cette responsabilité incombe au programmeur qui code la transaction. Cette propriété est connue sous le nom de **correction des transactions**.

Notion II.22 : Correction des transactions (*Transaction Correctness*)

Propriété d'une transaction consistant à respecter la cohérence de la base de données en fin d'exécution.

Lorsqu'une transaction est partiellement exécutée, les données peuvent passer par des états incohérents transitoires, qui seront corrigés par les mises à jour suivantes de la transaction. Pendant cette période d'activité, les effets de la transaction ne doivent pas être visibles aux autres transactions. Cette propriété est connue sous le nom d'**isolation des transactions** ; l'isolation doit être assurée par le SGBD.

Notion II.23 : Isolation des transactions (*Transaction Isolation*)

Propriété d'une transaction consistant à ne pas laisser visible à l'extérieur les données modifiées avant la fin de la transaction.

En résumé, un bon SGBD doit donc assurer les trois propriétés précédentes pour les transactions qu'il gère : Atomicité, Correction, Isolation. Ces propriétés sont parfois résumées par le sigle ACID, le D signifiant que l'on doit aussi pouvoir conserver durablement les mises à jour des transactions (en anglais, *durability*). En plus, le SGBD doit garantir la sécurité des données. Rappelons que la sécurité permet d'éviter les accès non autorisés aux données par des mécanismes de contrôle de droits d'accès, mais aussi de restaurer des données correctes en cas de pannes ou d'erreurs.

4.7. AUTRES FONCTIONS

De nombreuses autres fonctions se sont progressivement intégrées aux SGBD. Par exemple, beaucoup savent aujourd'hui déclencher des procédures cataloguées par l'utilisateur lors de l'apparition de certaines conditions sur les données ou lors de l'exécution de certaines opérations sur certaines entités ou associations. Cette fonctionnalité est connue sous le nom de **déclencheur**, encore appelé réflexe dans le contexte des architectures client-serveur en relationnel. Les déclencheurs permettent de rendre les bases de données actives, par exemple en déclenchant des procédures de correction lors de l'apparition de certains événements. Il s'agit là d'une fonctionnalité nouvelle qui prend de plus en plus d'importance.

Notion II.24 : Déclencheur (*Trigger*)

Mécanisme permettant d'activer une procédure cataloguée lors de l'apparition de conditions particulières dans la base de données.

De manière plus générale, les SGBD sont amenés à supporter des règles permettant d'inférer (c'est-à-dire de calculer par des raisonnements logiques) de nouvelles données à partir des données de la base, lors des mises à jour ou des interrogations. Cela conduit à la notion de SGBD déductif, capable de déduire des informations à partir de celles connues et de règles de déduction.

Enfin, les SGBD sont aussi amenés à gérer des objets complexes, tels des dessins d'architecture ou des cartes de géographie, en capturant finement le découpage de ces gros objets en sous-objets composants. Ces objets pourront être atteints via des procédures elles-mêmes intégrées au SGBD. Cela conduit à la notion de SGBD à objets, capable de gérer des objets multiples manipulés par des fonctions utilisateurs.

5. ARCHITECTURE FONCTIONNELLE DES SGBD

5.1. L'ARCHITECTURE À TROIS NIVEAUX DE L'ANSI/X3/SPARC

Les groupes de normalisation se sont penchés depuis fort longtemps sur les architectures de SGBD. À la fin des années 70, le groupe ANSI/X3/SPARC DBTG a proposé une architecture intégrant les trois niveaux de schémas : externe, conceptuel et interne. Bien qu'ancienne [ANSI78], cette architecture permet de bien comprendre les niveaux de description et transformation de données possible dans un SGBD.

L'architecture est articulée autour du dictionnaire de données et comporte deux parties :

1. un ensemble de modules (appelés processeurs) permettant d'assurer la description de données et donc la constitution du dictionnaire de données ;
2. une partie permettant d'assurer la manipulation des données, c'est-à-dire l'interrogation et la mise à jour des bases.

Dans chacune des parties, on retrouve les trois niveaux interne, conceptuel et externe. L'architecture proposée est représentée figure II.10.

Les fonctions de chacun des processeurs indiqués sont les suivantes. Le processeur de schéma conceptuel compile le schéma conceptuel et, dans le cas où il n'y a pas d'erreur, range ce schéma compilé dans le dictionnaire des données. Le processeur de schéma externe compile les schémas externes et les règles de correspondance externe à conceptuel et, après une compilation sans erreur, range le schéma compilé et les

règles de correspondance dans le dictionnaire des données. Le processeur de schéma interne a un rôle symétrique pour le schéma interne.

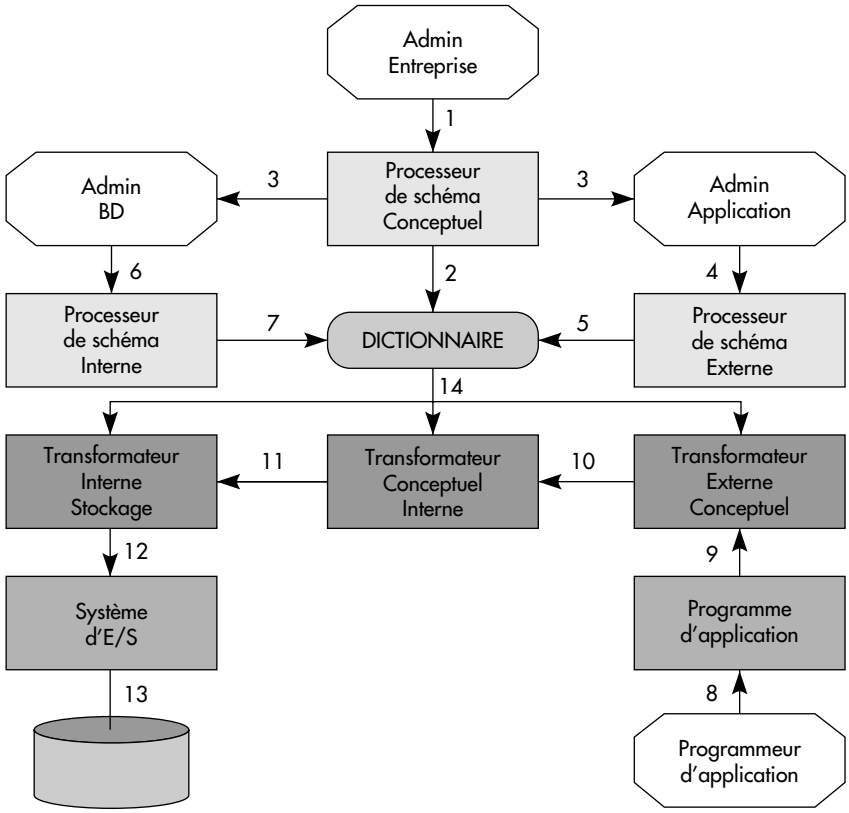


Figure II.10 : L'Architecture ANSI/X3/SPARC

Le processeur de transformation externe à conceptuel traduit les manipulations externes en manipulations conceptuelles et dans l'autre sens les données conceptuelles en données externes. Une requête externe peut donner naissance à plusieurs requêtes au niveau conceptuel. Le processeur de transformation conceptuel à interne traduit les manipulations conceptuelles en manipulations internes et dans l'autre sens les données internes en données conceptuelles. Finalement, le processeur de transformation interne à stockage traduit les manipulations internes en programmes d'accès au système de stockage et délivre les données stockées en format correspondant au schéma interne.

Les diverses interfaces indiquées correspondent successivement à (les numéros se rapportent à la figure II.10) :

(1) Langage de description de données conceptuel, format source ; il permet à l'administrateur d'entreprise de définir le schéma conceptuel en format source. Il

correspond par exemple aux commandes CREATE ENTITY et CREATE RELATIONSHIP vues ci-dessus (paragraphe II.4.1).

(2) Langage de description de données conceptuel, format objet ; il résulte de la compilation du précédent et permet de ranger le schéma objet dans le dictionnaire des données.

(3) Description de données conceptuel, format d'édition ; cette interface permet aux administrateurs d'applications et de bases de consulter le schéma conceptuel afin de définir les règles de correspondance. Il pourrait s'agir par exemple d'une visualisation graphique des diagrammes entité-association.

(4) Langages de description de données externes, format source ; ils peuvent être multiples si le SGBD supporte plusieurs modèles de données. Ils permettent aux administrateurs d'applications de définir les schémas externes et les règles de correspondance avec le schéma conceptuel. Par exemple, la commande DEFINE VIEW introduite ci-dessus illustre ce type de langage, qui permet donc de définir des schémas externes encore appelés vues.

(5) Langages de description de données externes, format objet ; ils correspondent à la compilation des précédents et permettent de ranger les schémas externes objets dans le dictionnaire de données.

(6) Langages de description de données internes, format source ; il permet à l'administrateur de bases de données de définir le schéma interne et les données de correspondance avec le schéma conceptuel. Par exemple, la commande CREATE INDEX vue ci-dessus (paragraphe II.4.1) se situe à ce niveau d'interface.

(7) Langage de description de données internes, format objet ; il correspond à la compilation du précédent et permet de ranger le schéma interne objet dans le dictionnaire des données.

(8) Langages de manipulation de données externes, format source ; ils permettent aux programmeurs d'applications, voire aux non-informaticiens, de manipuler les données décrites dans un schéma externe. Ils comportent des commandes du type RETRIEVE, APPEND, MODIFY et DELETE référençant les objets décrits dans un schéma externe.

(9) Langages de manipulation de données externes, format objet ; ils correspondent aux schémas compilés des précédents.

(10) Langage de manipulation de données conceptuelle, format objet ; il est généré par les processeurs de transformation externe à conceptuel afin de manipuler les données logiques décrites dans le schéma conceptuel. Ils comportent des primitives correspondant à la compilation des commandes du type RETRIEVE, APPEND, MODIFY et DELETE référençant cette fois les objets décrits dans le schéma conceptuel.

(11) Langage de manipulation de données interne, format objet ; il est généré par le processeur de transformation conceptuel à interne afin de manipuler les données internes. Il permet par exemple d'accéder aux articles de fichiers via des index.

(12) Langage de stockage de données, format objet ; il correspond à l'interface du système de stockage de données. Il permet par exemple de lire ou d'écrire une page dans un fichier.

(13) Interface mémoires secondaires ; elle permet d'effectuer les entrées-sorties sur les disques.

(14) Interface d'accès au dictionnaire des données ; elle permet aux divers processeurs de transformation d'accéder aux schémas objets et aux règles de correspondance.

5.2. UNE ARCHITECTURE FONCTIONNELLE DE RÉFÉRENCE

L'architecture à trois niveaux de schémas présentée ci-dessus permet de bien comprendre les niveaux de description et de manipulation de données. Cependant, elle n'est que peu suivie, la plupart des systèmes intégrant le niveau interne et le niveau conceptuel dans un seul niveau. En fait, le véritable niveau conceptuel est pris en charge par les outils d'aide à la conception à l'extérieur du SGBD, lors de la conception de l'application. Nous proposons une architecture de référence (voir figure II.11) plus proche de celles des SGBD actuels, basée sur seulement deux niveaux de schéma : le schéma et les vues. Le schéma correspond à une intégration des schémas interne et conceptuel, une vue est un schéma externe.

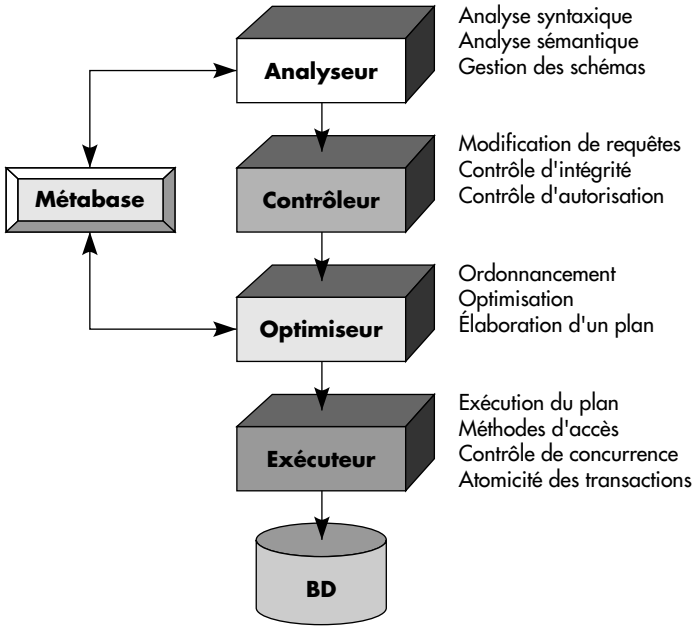


Figure II.11 : Architecture typique d'un SGBD

Du point de vue de la description de données, un SGBD gère un dictionnaire de données, encore appelé **métabase** car souvent organisé comme une base de données qui décrit les autres bases. Ce dictionnaire est alimenté par les commandes de définition du schéma (par exemple CREATE ENTITY, CREATE RELATIONSHIP, CREATE INDEX) et de définition des vues (par exemple DEFINE VIEW). Ces commandes sont analysées et traitées par le processeur d'analyse (ANALYSEUR), plus spécifiquement par la partie traitant le langage de description de données de ce processeur. Celle-ci fait souvent appel aux fonctions plus internes du SGBD pour gérer le dictionnaire comme une véritable base de données.

Du point de vue de la manipulation des données, les requêtes (par exemple, RETRIEVE, APPEND, MODIFY, DELETE) sont tout d'abord prises en compte par l'**analyseur de requêtes**. Celui-ci réalise l'analyse syntaxique (conformité à la grammaire) et sémantique (conformité à la vue référencée ou au schéma) de la requête. Celle-ci est alors traduite en format interne, les noms étant remplacés par des références internes.

Une requête en format interne référençant une vue doit tout d'abord être traduite en une (ou plusieurs) requête(s) référençant des objets existant dans la base, c'est-à-dire des objets décrits au niveau du schéma. Cette fonctionnalité, accomplie au niveau du **contrôleur de requêtes** figure II.11, est souvent appelée **modification de requêtes**, car elle consiste à changer la requête en remplaçant les références aux objets de la vue par leur définition en termes d'objets du schéma. C'est aussi au niveau du contrôleur que sont pris en compte les problèmes de contrôle de droits d'accès (autorisation de lire ou d'écrire un objet) et de contrôle d'intégrité lors des mises à jour. Le contrôle d'intégrité consiste à vérifier que la base n'est pas polluée lors des mises à jour, c'est-à-dire que les règles de cohérence des données restent vérifiées après mise à jour.

L'**optimiseur de requêtes** est un composant clé du SGBD. Son rôle essentiel est d'élaborer un plan d'accès optimisé pour traiter la requête. Pour se faire, il décompose en général la requête en opérations d'accès élémentaires (e.g., sélection d'index, lecture d'article, etc.) et choisit un ordre d'exécution optimal ou proche de l'optimum pour ces opérations. Il choisit aussi les méthodes d'accès à utiliser. Pour effectuer les meilleurs choix, l'optimiseur s'appuie souvent sur un modèle de coût qui permet d'évaluer le coût d'un plan d'accès avant son exécution. Le résultat de l'optimisation (le plan d'accès optimisé) peut être sauvegardé en mémoire pour des exécutions multiples ultérieures ou exécuté directement puis détruit.

L'**exécuteur de plans** a enfin pour rôle d'exécuter le plan d'accès choisi et élaboré par l'optimiseur. Pour cela, il s'appuie sur les méthodes d'accès qui permettent d'accéder aux fichiers via des index et/ou des liens. C'est aussi à ce niveau que sont gérés les problèmes de concurrence d'accès et d'atomicité de transactions. Les techniques utilisées dépendent beaucoup de l'architecture opérationnelle du SGBD qui s'exprime en termes de processus et de tâches.

5.3. L'ARCHITECTURE DU DBTG CODASYL

Le groupe de travail *Data Base Task Group* du comité CODASYL responsable du développement de COBOL a proposé depuis 1971 des recommandations pour construire un système de bases de données [Codasyl71]. Ces recommandations comportent essentiellement des langages de description de données et de manipulation de données orientés COBOL que nous étudierons plus loin, mais aussi une recommandation d'architecture.

L'architecture d'un système obéissant aux recommandations CODASYL s'articule autour du schéma qui permet de définir les articles, leurs données et les liens entre ces articles. Ce schéma peut être assimilé au schéma conceptuel de l'architecture ANSI/X3/SPARC, bien que comportant, à notre avis, pas mal de notions du niveau interne. La structure de stockage des données est définie par le schéma de stockage qui correspond plus ou moins au schéma interne ANSI. La notion de schéma externe est définie pour COBOL et est appelée sous-schéma. Un groupe de travail a également proposé des langages de description de sous-schémas pour FORTRAN ainsi qu'un langage de manipulation associé. L'architecture globale est représentée figure II.12.

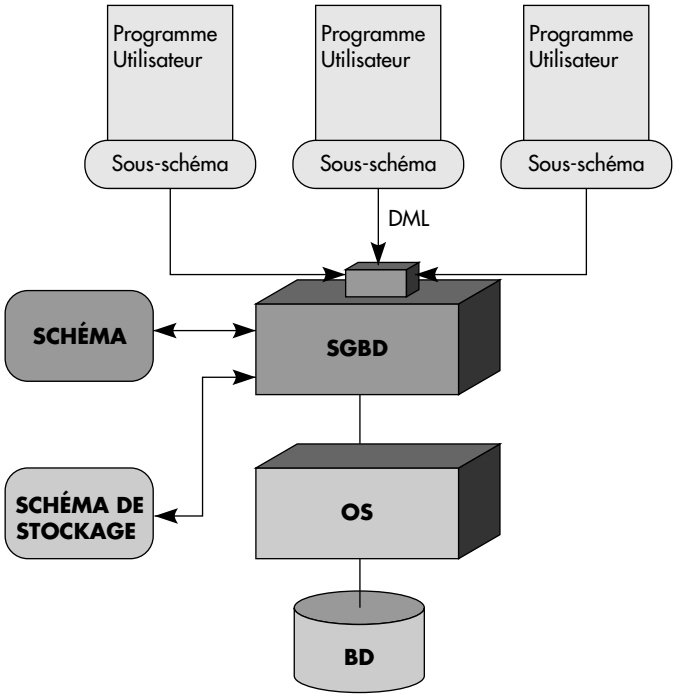


Figure II.12 : Architecture du DBTG CODASYL

6. ARCHITECTURES OPÉRATIONNELLES DES SGBD

Depuis le milieu des années 80, les SGBD fonctionnent selon l'architecture client-serveur. Nous introduisons ces architectures brièvement ci-dessous.

6.1. LES ARCHITECTURES CLIENT-SERVEUR

D'un point de vue opérationnel, un SGBD est un ensemble de processus et de tâches qui supportent l'exécution du code du SGBD pour satisfaire les commandes des utilisateurs. Depuis l'avènement des architectures distribuées autour d'un réseau local, les systèmes sont organisés selon l'architecture client-serveur. Cette architecture a été ébauchée dans un rapport du sous-groupe de l'ANSI/X3/SPARC appelé DAFTG (*Database Architecture Framework Task Group*) [ANSI86] et mise à la mode à la fin des années 80 par plusieurs constructeurs de SGBD.

L'**architecture client-serveur** inclut le noyau d'un SGBD tel que décrit ci-dessus, appelé DMCS (Description Manipulation and Control Sub-system), qui fonctionne en mode serveur. Autour de ce serveur s'articulent des processus attachés aux utilisateurs supportant les outils et les interfaces externes. Le DMCS est construit sur le gestionnaire de fichiers ou de disques virtuels du système opératoire. La figure II.13 (page suivante) illustre cette architecture.

Notion II.25 : Architecture client-serveur (*Client-server architecture*)

Architecture hiérarchisée mettant en jeu d'une part un serveur de données gérant les données partagées en exécutant le code du SGBD avec d'éventuelles procédures applicatives, d'autre part des clients pouvant être organisés en différents niveaux supportant les applications et la présentation, et dans laquelle les clients dialoguent avec les serveurs via un réseau en utilisant des requêtes de type question-réponse.

Le langage DL (*Data Language*) est le langage standard d'accès au SGBD, supporté par un protocole de niveau application pour le fonctionnement en mode réparti, appelé **protocole d'accès aux données distantes** (*Remote Data Access RDA*). Ce protocole est aujourd'hui en bonne voie de standardisation. Il est d'ailleurs complété par un protocole de gestion de transactions réparties.

Il existe différentes variantes de l'architecture client-serveur, selon qu'un processus serveur est associé à chaque utilisateur, ou que plusieurs utilisateurs partagent un même processus serveur. Dans le premier cas, le serveur est monotâche. Chaque processus client a un processus serveur associé. La machine supportant les serveurs doit partager son temps entre eux. Les commutations de processus peuvent être lourdes (quelques millisecondes sur UNIX). De plus, les processus serveurs partageant les

mêmes données, il est nécessaire de les synchroniser, par exemple par des sémaphores, afin d'éviter les problèmes de concurrence d'accès. Cela peut entraîner des pertes de temps importantes, et donc de mauvaises performances en présence d'usagers multiples.

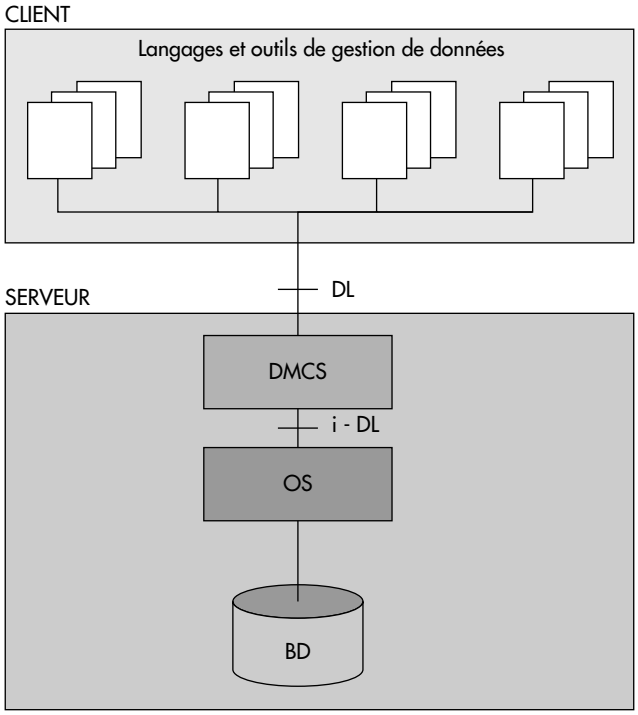


Figure II.13 : L'architecture client-serveur

Aujourd'hui, plusieurs systèmes proposent un serveur multitâche, capable de traiter plusieurs requêtes d'utilisateurs différents en parallèle. Cela est réalisé grâce à un multiplexage du serveur en tâches, les commutations de tâches étant assurées par le serveur lui-même, au niveau d'un gestionnaire de tâches optimisé pour les bases de données. Une telle architecture multitâche (en anglais, *multi-thread*) permet de meilleures performances en présence d'un nombre important d'utilisateurs.

Au-delà du *multi-thread*, le besoin en performance a conduit à partionner les traitements applicatifs de façon à réduire les communications entre client et serveur. Ainsi, le client peut invoquer des procédures applicatives qui manipulent la base directement sur le serveur. Ces procédures applicatives liées à la base sont appelées des **procédures stockées**. Elles évitent de multiples commandes et transferts de données sur le réseau. Ceux-ci sont remplacés par l'invocation de procédures stockées avec quelques paramètres et la transmission des paramètres retour. L'architecture obtenue permettant

deux couches de traitement applicatifs est appelée **architecture à deux strates** (*two-tiered architecture*).

Notion II.26 : Architecture client-serveur à deux strates (*Two-tiered client-server architecture*)

Architecture client-serveur composée : (i) d'un serveur exécutant le SGBD et éventuellement des pro-

La figure II.14 propose une vue plus détaillée d'une architecture client-serveur à deux strates. L'application est écrite à l'aide d'un outil applicatif, souvent un L4G. Elle soumet ses demandes de services (requêtes au SGBD ou invocation de procédures stockées au *middleware* qui les transfert au serveur. Le middleware comporte un composant client et un composant serveur qui permettent les échanges de commandes via le protocole réseau. Sur la figure, il est appelé outil de connectabilité. Si vous souhaitez en savoir plus sur le *middleware*, reportez-vous à [Gardarin97].

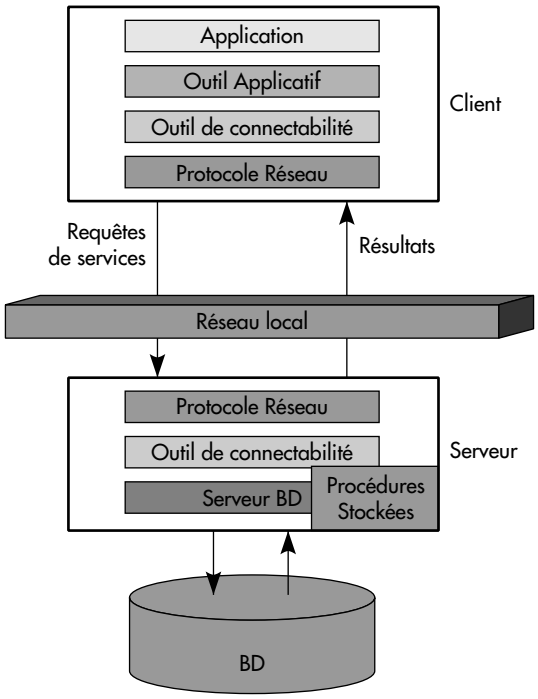


Figure II.14 : L'architecture client-serveur à deux strates

Avec l'apparition d'Internet et du Web, les architectures client-serveur ont évolué vers des **architectures à trois strates** (*three-tiered architecture*). Le client est responsable de la présentation. Il utilise pour cela des *browsers* Web. Le serveur d'application exé-

cute le code applicatif essentiel. Le serveur de données supporte le SGBD et gère éventuellement des procédures stockées.

Notion II.27 : Architecture client-serveur à trois strates (*Three-tiered client-server architecture*)

Architecture client-serveur composée : (i) d'un serveur de données exécutant le SGBD et éventuellement des procédures applicatives ; (ii) d'un serveur d'application exécutant le corps des applications ; (iii) de clients responsables des dialogues et de la présentation des données selon les standards du Web.

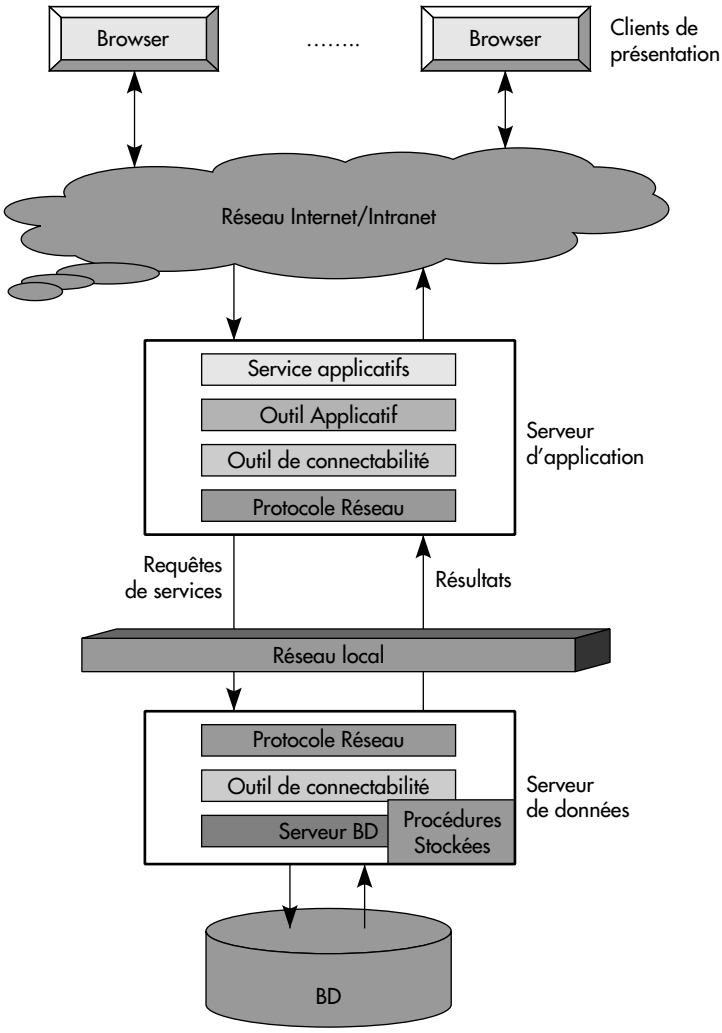


Figure II.15 : L'architecture client-serveur à trois strates

La figure II.15 illustre une telle architecture. Les *middlewares* mis en jeu sont beaucoup plus complexes.

L'architecture client-serveur est aujourd'hui bien adaptée aux systèmes répartis autour d'un réseau local et/ou d'Internet. Elle permet à de multiples postes ou stations de travail distribués sur la planète de partager les mêmes données. Celles-ci sont gérées de manière fiable et avec de bons contrôles de concurrence au niveau du serveur. Un processus client sur la station de travail ou l'ordinateur personnel gère les applications de l'utilisateur qui émettent des requêtes au serveur. Un client peut même invoquer plusieurs serveurs : on parle alors d'architecture client-multiserveur. L'inconvénient mais aussi l'avantage du client-serveur est de centraliser la gestion des données au niveau du serveur.

6.2. LES ARCHITECTURES RÉPARTIES

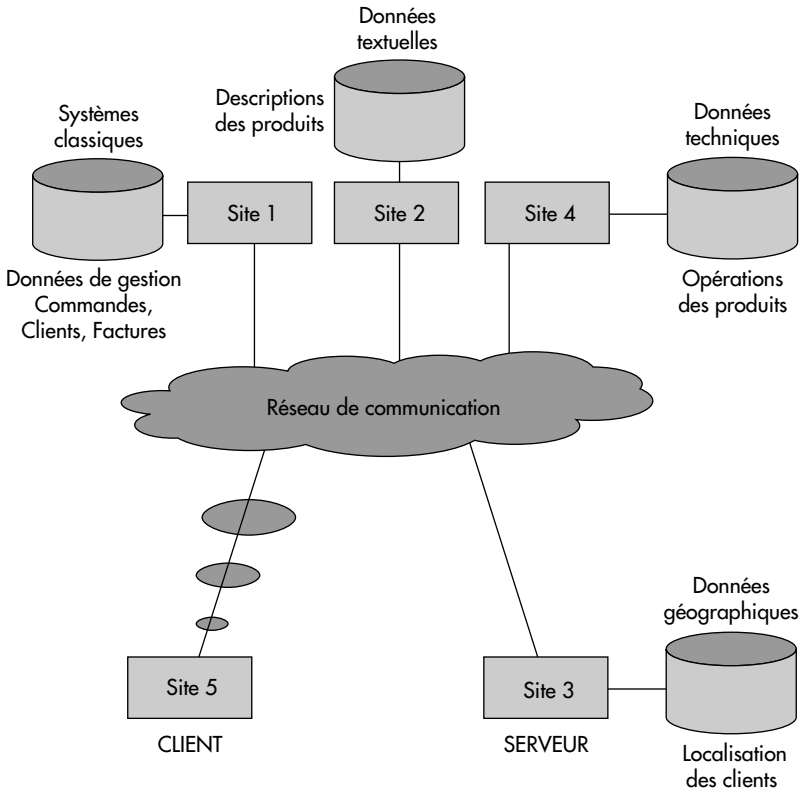


Figure II.16 : Exemple d'architecture répartie

Afin de répondre à la tendance centralisatrice de l'approche client-serveur, certains SGBD préconisent une **architecture répartie**. Une architecture répartie fait interagir plusieurs serveurs gérant un ensemble de bases perçu comme une seule base par les utilisateurs.

Notion : Architecture BD répartie *(Distributed database architecture)*

Architecture composée de plusieurs serveurs coopérant à la gestion de bases de données composées de plusieurs sous-bases gérées par un seul serveur, mais apparaissant comme des bases uniques centralisées pour l'utilisateur.

La figure II.16 illustre une architecture répartie. Celle-ci est composée de différents serveurs munis de SGBD différents et spécialisés. C'est un exemple de base de données répartie hétérogène, encore appelée base de données fédérée. Nous n'étudions pas les bases de données réparties dans cet ouvrage. L'auteur pourra se reporter à [Gardarin97] et à [Valduriez99] pour une étude plus complète des systèmes de bases de données réparties.

7. CONCLUSION

Dans ce chapitre, nous avons présenté les objectifs des systèmes de gestion de bases de données, puis les concepts et méthodes essentiels de ces systèmes. Cela nous a conduit à étudier les architectures fonctionnelles, puis les architectures opérationnelles. Afin de concrétiser nos propos, nous avons introduit une variante du langage QUEL pour décrire et manipuler les données. L'architecture proposée en 1978 par le groupe ANSI/X3/SPARC permet de bien comprendre les niveaux de schémas possible. L'architecture fonctionnelle de référence est voisine de celle retenue dans le système SABRINA [Gardarin86] ou encore de celle des systèmes INGRES ou ORACLE. Les architectures opérationnelles sont aujourd'hui client-serveur, mais évoluent de plus en plus souvent vers le réparti.

Il est possible de résumer ce chapitre en rappelant qu'un SGBD offre une interface de description de données qui permet de documenter le dictionnaire de données. Le compilateur du langage de description gère cette métabase. Un SGBD offre aussi une interface de manipulation de données (recherches et mises à jour) qui permet de modifier ou de retrouver des données dans la base. Le compilateur-optimiseur du langage de manipulation génère des plans d'accès optimisés. Ceux-ci sont exécutés par le processus serveur, qui gère aussi la concurrence et la fiabilité. Les requêtes peuvent être émises par des programmes d'applications écrits dans des langages plus ou moins traditionnels, ou par des utilisateurs travaillant en interactif à partir d'utilitaires.

Les SGBD tels que présentés dans ce chapitre s'appuient au niveau interne sur une gestion de fichiers. Celle-ci peut être une partie intégrante du système opératoire. Elle peut être plus ou moins sophistiquée. Dans le chapitre suivant, nous allons étudier la gestion de fichiers, de la plus simple à la plus élaborée. Selon le niveau de sophistication du gestionnaire de fichiers sur lequel il est construit, un SGBD devra ou non intégrer des fonctionnalités de type gestion de fichiers au niveau interne. Aujourd'hui, la plupart des SGBD intègrent leurs propres méthodes d'accès aux fichiers (parfois appelés segments), du type de celles que nous allons étudier dans le chapitre suivant.

8. BIBLIOGRAPHIE

[ANSI75] ANSI/X3/SPARC Study Group on Data Base Management Systems, « Interim Report », *ACM SIGMOD Bulletin*, vol. 7, n° 2, ACM Ed., 1975.

Ce document présente l'architecture ANSI/X3/SPARC et ses trois niveaux de schémas.

[ANSI78] ANSI/X3/SPARC Study Group on Data Base Management Systems, « Framework Report on Database Management Systems », *Information Systems*, vol. 3, n° 3 1978.

Il s'agit du document final présentant l'architecture à trois niveaux de schémas de l'ANSI.

[ANSI86] ANSI/X3/SPARC Database Architecture Framework Task Group, « Reference Model for DBMS Standardization », *ACM SIGMOD Record*, vol. 15, n° 1, mars 1986.

Il s'agit du document final présentant l'étude sur les architectures de SGBD du groupe DAFTG. Celui-ci conseille de ne pas chercher à standardiser davantage les architectures, mais plutôt de s'efforcer de standardiser les langages.

[Astrahan76] Astrahan M., Blasgen M., Chamberlin D., Eswaran K., Gray. J., Griffiths P., King W., Lorie R., McJones P., Mehl J., Putzolu G., Traiger I., Wade B., Watson V., « System R: Relational Approach to Database Management », *ACM Transactions on Database Systems*, vol. 1, n° 2, juin 1976.

Cet article présente System R, le premier prototype de SGBD relationnel réalisé par IBM dans son centre de recherches de San José. System R comporte une architecture à deux niveaux de schéma et deux processeurs essentiels : le RDS (Relational Data System) qui comprend analyseur, traducteur et optimiseur, et le RSS qui correspond à l'exécuteur. System R a donné naissance à SQL/DS et à DB2.

[Brodie86] Brodie M. Ed., *On Knowledge Base Management Systems*, Springer Verlag Ed., Berlin, 1986.

Ce livre discute les problèmes liés à l'introduction des techniques de l'intelligence artificielle au sein des SGBD. Les conséquences en termes d'objectifs et d'architecture sont analysées tout au long des multiples articles composant le livre.

[Carey85] Carey M.J., Dewitt D.J., « Extensible Database Systems », *Islamodrada Workshop*, 1985, dans [Brodie86].

Cet article plaide pour une architecture extensible des SGBD, notamment au niveau de l'optimiseur. Il propose de réaliser un optimiseur dirigé par une bibliothèque de règles spécifiant les techniques d'optimisation.

[Codasyl71] CODASYL DBTG, *Codasyl Data Base Task Group Report*, ACM Ed., New-York, avril 1971.

Ce document présente la synthèse des travaux du CODASYL en termes d'architecture, de langage de description et de langage de manipulation de données pour SGBD supportant le modèle réseau. Le groupe CODASYL était une émanation du groupe de standardisation de COBOL. Il a tenté de standardiser la première génération de SGBD.

[Chen76] Chen P.P., « The Entity-Relationship Model – Towards a Unified View of Data », *ACM Transactions on Database Systems*, vol. 1, n° 1, Mars 1976.

Cet article introduit le modèle entité-association pour décrire la vue des données d'une entreprise. En particulier, les diagrammes de Chen sont présentés. Il est montré que le modèle permet d'unifier les différents points de vue.

[Date71] Date C.J., Hopewell P., « File Definition and Logical Data Independance », *ACM SIGFIDET Workshop on Data Description, Access and Control*, ACM Ed., New-York, 1971.

L'indépendance physique et logique est discutée clairement pour la première fois. En particulier, les modifications de structures de fichiers qui doivent être possibles sans changer les programmes sont analysées.

[Gardarin86] Gardarin G., Kerhervé B., Jean-Noël M., Pasquer F., Pastre D., Simon E., Valduriez P., Verlaine L., Viémont Y., « SABRINA: un système de gestion de bases de données relationnel issu de la recherche », *Techniques et Sciences Informatique (TSI)*, Dunod Ed., vol. 5, n° 6, 1986.

Cet article présente le SGBD relationnel SABRINA réalisé dans le projet SABRE à l'INRIA, puis en collaboration avec plusieurs industriels, de 1979 à 1989. Ce système avancé supporte une architecture extensible voisine de l'architecture de référence présentée ci-dessus. Il est construit selon l'approche client-serveur.

[Gardarin97] Gardarin G., Gardarin O., *Le Client-Serveur*, 470 pages, Éditions Eyrolles, 1997.

Ce livre traite des architectures client-serveur, des middlewares et des bases de données réparties. Les notions importantes du client-serveur sont clairement expliquées. Une part importante de l'ouvrage est consacrée aux middlewares et outils de développement objet. CORBA et DCOM sont analysés. Ce livre est un complément souhaitable au présent ouvrage, notamment sur les bases de données réparties et les techniques du client-serveur.

[Stonebraker74] Stonebraker M.R., « A Functional View of Data Independance », *ACM SIGMOD Workshop on Data Description, Access and Control*, ACM Ed., mai 1974.

Un des premiers articles de Mike Stonebraker, l'un des pères du système INGRES. Il plaide pour l'introduction de vues assurant l'indépendance logique.

[Stonebraker76] Stonebraker M., Wong E., Kreps P., Held G.D., « The Design and Implementation of Ingres », *ACM Transactions on Database Systems*, vol. 1, n° 3, septembre 1976.

Cet article décrit le prototype INGRES réalisé à l'université de Berkeley. Celui-ci supporte le langage QUEL. Il est composé de 4 processus correspondant grossièrement à l'analyseur, au traducteur, à l'optimiseur et à l'exécuteur présenté ci-dessus. Ce prototype est l'ancêtre du système INGRES aujourd'hui très populaire.

[Stonebraker80] Stonebraker M., « Retrospection on a Database system », *ACM Transactions on Database Systems*, vol. 5, n° 2, mars 1980.

Cet article fait le bilan de la réalisation du système INGRES. Il souligne notamment l'importance du support génie logiciel pour la réalisation d'un SGBD. Par exemple, lorsqu'un module change de responsable, le nouveau responsable s'empresse de le réécrire sous pretexte de non propreté, etc.

[Stonebraker87] Stonebraker M., « The Design of the POSTGRES Storage System », *Int. Conf. on Very Large Databases*, Morgan & Kauffman Ed., Brighton, Angleterre, 1987.

M. Stonebraker présente la conception du noyau de stockage du système POSTGRES, successeur d'INGRES. Celui-ci était entièrement basé sur le contrôle de concurrence et le support de déclencheurs.

[Tsichritzis78] Tsichritzis D.C., Klug A. (Éditeurs), *The ANSI/X3/SPARC Framework*, AFIPS Press, Montvale, NJ, 1978.

Une autre version du rapport final de l'ANSI/X3/SPARC avec ses trois niveaux de schémas.

[Valduriez99] Valduriez P., Ozsu T., *Principles of Distributed Database Systems*, 562 pages, Prentice Hall, 2e édition,1999.

Le livre fondamental sur les bases de données réparties en anglais. Après un rappel sur les SGBD et les réseaux, les auteurs présentent l'architecture type d'un SGBD réparti. Ils abordent ensuite en détails les différents problèmes de conception d'un SGBD réparti : distribution des données, contrôle sémantique des données, évaluation de questions réparties, gestion de transactions réparties, liens avec les systèmes opératoires et multibases. La nouvelle édition aborde aussi le parallèlisme et les middlewares. Les nouvelles perspectives sont enfin évoquées.

[Weldon79] Weldon J.L., « The Practice of Data Base Administration », *National Computer Conference*, AFIPS Ed., V.48, New York, 1979.

Cet article résume les résultats d'une étude du rôle des administrateurs de données à travers une enquête réalisée auprès de 25 d'entre eux. Un appendice permet plus particulièrement de définir leurs tâches : établissement du schéma directeur, conception des bases de données, tests, contrôles et support opérationnel.

[Zaniolo83] Zaniolo C., « The Database Language GEM », *ACM SIGMOD Int. Conf. on Management of Data*, San José, CA, 1983.

Cet article présente une extension très complète du langage QUEL pour le modèle entité-association. Cette extension a été implantée au-dessus d'une machine bases de données exécutant un langage proche de QUEL.

[Zook77] Zook W. et. al., INGRES Reference Manual, Dept. of EECS, University of California, Berkeley, CA, 1977.

Ce document décrit les interfaces externes de la première version d'INGRES et plus particulièrement le langage QUEL.

Chapitre III

FICHIERS, HACHAGE ET INDEXATION

1. INTRODUCTION

Un SGBD inclut en son cœur une gestion de fichiers. Ce chapitre est consacré à l'étude des fichiers et de leurs méthodes d'accès. Historiquement, la notion de fichier a été introduite en informatique dans les années 50, afin de simplifier l'utilisation des mémoires secondaires des ordinateurs et de fournir des récipients de données plus manipulables aux programmes. Au début, un fichier était bien souvent l'image d'une portion nommée de bande magnétique. Puis, au fur et à mesure de la sophistication des systèmes informatiques, les fichiers sont devenus plus structurés. Des méthodes d'accès de plus en plus performantes ont été élaborées. Aujourd'hui, les fichiers sont à la base des grands systèmes d'information ; la gestion de fichiers est le premier niveau d'un SGBD.

Idéalement, un gestionnaire de fichiers doit permettre le traitement par l'informatique de la gestion complète d'une entreprise. Les données descriptives des objets gérés par l'entreprise, ainsi que les programmes spécifiant les traitements appliqués aux données, doivent pouvoir être stockés dans des fichiers gérés par le système informatique. Les traitements de ces données peuvent être exécutés par lots, en fin de mois par exemple, ou en fin de semaine, mais aussi (et c'est en général plus difficile) à l'unité

dès que survient un événement dans le monde réel : on parle alors de traitement transactionnel, mode de traitement privilégié par les bases de données.

Un gestionnaire de fichiers devrait par exemple permettre de traiter l'application comptabilité d'une société de livraison. Cette application est représentée figure III.1. Elle gère les comptes des clients et édite les factures correspondant aux livraisons. Pour calculer les montants à facturer et à déduire des comptes clients, un catalogue des prix est utilisé. Les traitements peuvent être effectués en traitement par lots, en fin de mois ; dans ce cas, les bordereaux de livraison du mois sont traités ensemble, par exemple à partir d'un fichier de saisie. Le traitement d'une livraison peut être effectué en transactionnel ; dans ce cas, la description de la livraison est tapée en direct sur un terminal et les traitements associés sont immédiatement exécutés.

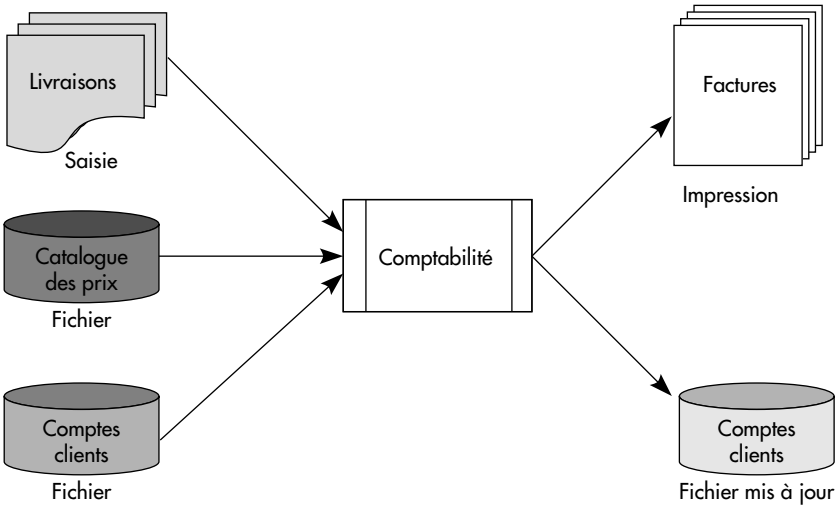


Figure III.1 : Un exemple d'application

Nous allons, dans ce chapitre, étudier plus spécifiquement la gestion des fichiers au cœur des SGBD. Un fichier est un récipient de données identifié par un nom et contenant des informations système ou utilisateur. La gestion des fichiers est une des fonctions essentielles offertes par les systèmes opératoires. C'est en effet grâce à cette fonction qu'il est possible de traiter et de conserver des quantités importantes de données, et de les partager entre plusieurs programmes. De plus, elle sert de base au niveau interne des Systèmes de Gestion de Bases de Données (SGBD) qui la complètent par des méthodes d'accès spécifiques ou la reprennent totalement.

Ce chapitre introduit tout d'abord les objectifs de la gestion des fichiers et les concepts de base associés, puis étudie les fonctions accomplies par le noyau d'un gestionnaire de fichiers. Enfin, et c'est son objet essentiel, il présente les principales organisations et méthodes d'accès de fichiers. Celles-ci sont groupées en deux

classes : (i) les méthodes par hachage qui appliquent une fonction de hachage à l'identifiant des articles d'un fichier, appelé clé, afin de retrouver son emplacement dans le fichier ; (ii) les méthodes indexées qui gèrent une table des matières du fichier afin d'accéder aux articles.

2. OBJECTIFS ET NOTIONS DE BASE

Nous rappelons ici les notions fondamentales d'architecture d'ordinateur et de gestion de fichiers. Le lecteur informaticien pourra sauter cette section, voire la suivante.

2.1. GESTION DES DISQUES MAGNÉTIQUES

L'informatisation de la gestion d'une grande entreprise nécessite le stockage de volumes de données importants – bien souvent plusieurs milliards d'octets. Il n'est pas possible de stocker de tels volumes de données en mémoire centrale. On est ainsi conduit à introduire des **mémoires secondaires**.

Notion III.1 : Mémoire secondaire (*External Storage*)

Mémoire non directement adressable par les instructions du processeur central, mais par des instructions d'entrées-sorties spécialisées et dont les temps d'accès sont très supérieurs à ceux de la mémoire centrale.

Il existe différents types de mémoires secondaires : disques magnétiques, disques optiques numériques, bandes magnétiques, cassettes, etc. Parmi les disques magnétiques, on distingue les disques à têtes fixes des disques à têtes mobiles. Les premiers ont des capacités variant de 1 à 100 millions d'octets et des temps d'accès de quelques millisecondes. Au contraire, les disques à têtes mobiles sont plus lents, mais supportent en général des capacités plus importantes. Les mini-disques des micro-ordinateurs sont dans cette catégorie, mais restent cependant de faible capacité (400 kilos à quelques méga-octets). À l'opposé, les disques optiques numériques constituent une technologie à bas prix permettant d'archiver plusieurs giga-octets sur un même disque. Ils ne sont malheureusement inscriptibles qu'une seule fois, bien que des disques optiques réinscriptibles commencent à apparaître.

Les disques magnétiques les plus utilisés ont des diamètres de 3 à 14 pouces et sont organisés en piles. IBM commence même à produire des disques de un pouce. Leur capacité s'est accrue depuis les années 1955 pour atteindre plusieurs milliards d'octets aujourd'hui. Les temps d'accès se sont également améliorés pour avoisiner 10 à 20 ms

en moyenne. Notons qu'ils restent encore de l'ordre de 20 000 fois plus élevés que ceux des mémoires centrales. Nous appellerons une pile de disques un **volume**.

Notion III.2 : Volume (*Disk Pack*)

Pile de disques constituant une unité de mémoire secondaire utilisable.

Un volume disque est associé à un tourne-disque. Les volumes peuvent être fixes ou amovibles. Si le volume est amovible, il est monté sur un tourne-disque pour utilisation puis démonté après. Les volumes amovibles sont montés et démontés par les opérateurs, en général sur un ordre du système. Un contrôleur de disque contrôle en général plusieurs tourne-disques. La notion de volume s'applique également aux bandes magnétiques où un volume est une bande. Une unité peut alors comporter un ou plusieurs dérouleurs de bande.

Un volume et l'équipement de lecture-écriture associé à un tourne-disque sont représentés figure III.2. Un volume se compose de p disques (par exemple 9), ce qui correspond à 2p – 2 surfaces magnétisées, car les deux faces externes ne le sont pas. Les disques à têtes mobiles comportent une tête de lecture-écriture par surface. Cette tête est accrochée à un bras qui se déplace horizontalement de manière à couvrir toute la surface du disque. Un disque est divisé en pistes concentriques numérotées de 0 à n (par exemple n = 1024). Les bras permettant de lire/écrire les pistes d'un volume sont solidaires, ce qui force leur déplacement simultané. Les disques tournent continûment à une vitesse de quelques dizaines de tours par seconde. L'ensemble des pistes, décrit quand les bras sont positionnés, est appelé cylindre.

Chaque piste d'un disque supporte plusieurs enregistrements physiques appelés secteurs, de taille généralement constante, mais pouvant être variable pour certains types de disque. Le temps d'accès à un groupe de secteurs consécutifs est une des caractéristiques essentielles des disques. Il se compose :

1. du temps nécessaire au mouvement de bras pour sélectionner le bon cylindre (quelques millisecondes à des dizaines de millisecondes selon l'amplitude du déplacement du bras et le type de disque) ;
2. du temps de rotation du disque nécessaire pour que l'enregistrement physique désiré passe devant les têtes de lecture/écriture ; ce temps est appelé temps de latence ; il est de quelques millisecondes, selon la position de l'enregistrement par rapport aux têtes et selon la vitesse de rotation des disques ;
3. du temps de lecture/écriture du groupe de secteurs, appelé temps de transfert ; ce temps est égal au temps de rotation multiplié par la fraction de pistes lue, par exemple 1/16 de 10 ms pour lire un secteur sur une piste de 16 secteurs avec un disque tournant à 100 tours par seconde.

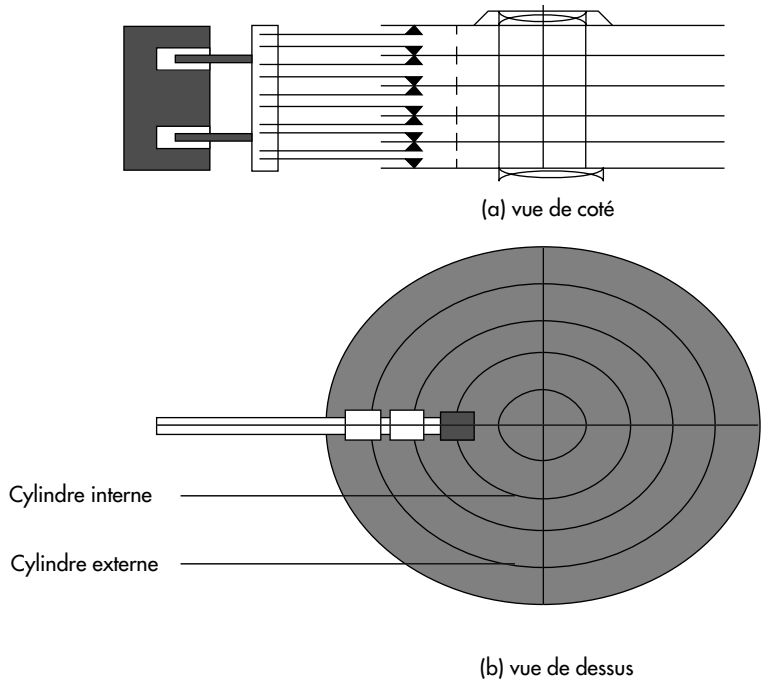


Figure III.2 : Volume amovible à têtes mobiles et équipement de lecture/écriture

Au total, les disques ont un temps d'accès variable selon la distance qui sépare l'enregistrement auquel accéder de la position des têtes de lecture/écriture. La tendance aujourd'hui est de réduire cette variance avec des moteurs à accélération constante pour commander les mouvements de bras, et avec des quantités de données enregistrées par cylindre de plus en plus importantes.

De plus en plus souvent, des volumes multiples organisés en **tableaux de disques** sont utilisés pour composer des unités fiables et de grande capacité (des dizaines de milliards d'octets). Ces systèmes de mémorisation portent le nom de RAID (*Redundant Array of Inexpensive disques*). On distingue le RAID 0 sans redondance des RAID 1 à 6 qui gèrent des redondances. Avec ces derniers, différentes techniques de redondance sont utilisées lors des écritures et lectures afin d'assurer une grande fiabilité. Le RAID 1 groupe les disques du tableau par deux et effectuent les écritures sur les deux disques, l'un apparaissant donc comme le miroir de l'autre. En cas de panne d'un disque, le disque miroir peut toujours être utilisé. Les RAID 2, 3 et 4 gèrent un disque de parité, respectivement au niveau bit, octet ou bloc. En cas de panne d'un disque, celui-ci peut être reconstitué grâce à la parité. Les RAID 5 et 6 sont basés sur des redondances plus fortes, avec des codes cycliques (CRC). Ils permettent de résister à des doubles pannes. Les disques RAID permettent des performances en lecture et écriture élevées car de multiples contrôleurs d'entrées-sorties fonctionnent en parallèle.

2.2. INDÉPENDANCE DES PROGRAMMES PAR RAPPORT AUX MÉMOIRES SECONDAIRES

Les disques magnétiques et plus généralement les mémoires secondaires doivent pouvoir être utilisés par plusieurs programmes d'application. En conséquence, il faut pouvoir partager l'espace mémoire secondaire entre les données des diverses applications. Une gestion directe de cet espace par les programmes n'est pas souhaitable car elle interdirait de modifier l'emplacement des données sans modifier les programmes d'application. Les adresses utilisées par les programmes doivent être indépendantes de l'emplacement des données sur disques. Il faut donc introduire des couches systèmes intermédiaires permettant d'assurer l'indépendance des programmes vis-à-vis de l'emplacement des données sur les mémoires secondaires. Autrement dit, l'allocation de la mémoire secondaire doit être gérée par le système.

D'un autre côté, les progrès technologiques ne cessent d'améliorer le rapport performance/prix des mémoires secondaires. La densité des disques magnétiques (nombre de bits enregistrés/cm^2) double approximativement tous les deux ans, ce qui permet d'accroître les capacités de stockage et les performances. Un bon système doit permettre aux utilisateurs de profiter des avancées technologiques, par exemple en achetant des disques plus performants, et cela sans avoir à modifier les programmes. En effet, la reprogrammation coûte très cher en moyens humains. En résumé, il faut assurer l'**indépendance des programmes d'application par rapport aux mémoires secondaires**. Cette indépendance peut être définie comme suit :

Notion III.3 : Indépendance des programmes par rapport aux mémoires secondaires (*Program-Storage device independence*)

Possibilité de changer les données de localité sur les mémoires secondaires sans changer les programmes.

Afin de réaliser cette indépendance, on introduit des objets intermédiaires entre les programmes d'application et la mémoire secondaire. Ces objets sont appelés **fichiers**. Ainsi, les programmes d'application ne connaissent pas les mémoires secondaires, mais les fichiers qui peuvent être implantés sur diverses mémoires secondaires. Un fichier peut être défini comme suit :

Notion III.4 : Fichier (*File*)

Récipient d'information caractérisé par un nom, constituant une mémoire secondaire idéale, permettant d'écrire des programmes d'application indépendants des mémoires secondaires.

Un programme d'application ne manipule pas globalement un fichier mais lit/écrit et traite des portions successives de celui-ci, correspondant en général à un objet du monde réel, par exemple un client, un compte, une facture. Une telle portion est appelée **article**.

Notion III.5 : Article (*Record*)

Élément composant d'un fichier correspondant à l'unité de traitement par les programmes d'application.

Les articles sont stockés dans les récipients d'information que constituent les fichiers. Ils ne sont pas stockés n'importe comment, mais sont physiquement reliés entre eux pour composer le contenu d'un fichier. Les structures des liaisons constituent l'**organisation du fichier**.

Notion III.6 : Organisation de fichier (*File organization*)

Nature des liaisons entre les articles contenus dans un fichier.

Les programmes d'application peuvent choisir les articles dans un fichier de différentes manières, par exemple l'un après l'autre à partir du premier, ou en attribuant un nom à chaque article, selon la **méthode d'accès** choisie.

Notion III.7 : Méthode d'accès (*Acces Method*)

Méthode d'exploitation du fichier utilisée par les programmes d'application pour sélectionner des articles.

2.3. UTILISATION DE LANGAGES HÔTES

Le système de gestion de fichiers doit être utilisable par un programme dont le code objet résulte d'une compilation d'un langage de haut niveau (COBOL, PL/1, PASCAL, C, etc.). De plus, il doit être possible d'utiliser les fichiers dans le langage choisi d'une manière aussi intégrée que possible, par exemple en décrivant les données du fichier comme des types dans le langage. On appelle **langage hôte** le langage de programmation qui intègre les verbes de manipulation de fichiers.

Notion III.8 : Langage hôte (*Host language*)

Langage de programmation accueillant les verbes de manipulation de fichiers et la définition des données des fichiers.

Il est utile de rappeler le cheminement d'un programme en machine. Celui-ci est donc écrit à l'aide d'un langage de programmation hôte et de verbes de manipulation de fichiers. Il est pris en charge par le compilateur du langage. Ce dernier génère du code machine translatable incluant des appels au système, en particulier au gestionnaire de fichiers. Le chargeur-éditeur de liens doit ensuite amener en bonne place en mémoire les différents modules composants du programme ; en particulier, les translations

d'adresse doivent être effectuées à ce niveau. On obtient alors du code exécutable. La figure III.3 illustre ces étapes.

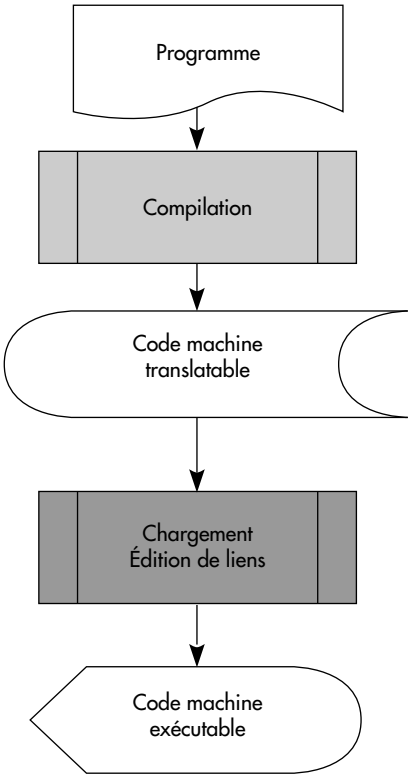


Figure III.3 : Cheminement d'un programme

2.4. POSSIBILITÉS D'ACCÈS SÉQUENTIEL ET SÉLECTIF

Afin de caractériser le comportement des programmes d'application vis-à-vis des fichiers, il est possible d'utiliser deux mesures. Le taux de consultation (TC) est le quotient du nombre d'articles utilement lus par un programme sur le nombre d'articles total du fichier. Le taux de mouvement (TM) est le quotient du nombre d'articles modifiés par un programme sur le nombre d'articles total du fichier. On est ainsi conduit à introduire deux types de méthodes d'accès.

Notion III.9 : Méthode d'accès séquentielle (*Sequential Acces Method*)

Méthode d'accès consistant à lire successivement tous les articles d'un fichier, depuis le premier jusqu'à l'article désiré.

Notion III.10 : Méthodes d'accès sélectives (*Key-based access methods*)

Ensemble de méthodes d'accès permettant de lire/écrire tout article au moyen de quelques accès disques (moins de 5, idéalement 1), y compris pour de très gros fichiers.

Une méthode d'accès séquentielle est adaptée si TC ou TC+TM sont de l'ordre de 1. Elle sera essentiellement utilisée en traitement par lots. Au contraire, une méthode d'accès sélective sera utilisée quand TC, ou TC+TM pour un programme modifiant, sera petit. Les méthodes d'accès sélectives sont donc particulièrement adaptées au transactionnel. Un gestionnaire de fichiers doit supporter des méthodes d'accès séquentielle et sélectives, cela afin de permettre à la fois les traitements par lots et le travail en transactionnel.

Afin de mettre en œuvre une méthode d'accès sélective, il faut pouvoir identifier de manière unique un article. En effet, une méthode d'accès sélective permet à partir de l'identifiant d'un article de déterminer l'adresse d'un article (adresse début) et de lire l'article en moins de 5 E/S. L'identifiant d'un article est appelé **clé** (que l'on qualifie parfois de primaire). La clé peut ou non figurer comme une donnée de l'article.

Notion III.11 : Clé d'article (Record Key)

Identifiant d'un article permettant de sélectionner un article unique dans un fichier.

Soit l'exemple d'un fichier décrivant des étudiants. Les articles comportent les champs suivants : numéro d'étudiant, nom, prénom, ville, date d'inscription et résultats. La clé est le numéro d'étudiant ; c'est une donnée de l'article. À partir de cette clé, c'est-à-dire d'un numéro d'étudiant, une méthode d'accès sélective doit permettre de déterminer l'adresse de l'article dans le fichier et d'accéder à l'article en principe en moins de 5 entrées/sorties disques.

Comme cela a été déjà dit, il existe différents types d'organisations sélectives (et méthodes d'accès associées). Nous allons ci-dessous étudier les principales. Elles peuvent être divisées en deux classes :

– Les méthodes d'accès par hachage utilisent des fonctions de calcul pour déterminer l'adresse d'un article dans un fichier à partir de sa clé ;
– Les méthodes d'accès par index utilisent des tables généralement stockées sur disques pour mémoriser l'association clé article-adresse article.

2.5. POSSIBILITÉ D'UTILISATEURS MULTIPLES

Dans les machines modernes, l'exécution d'une instruction d'entrée-sortie ne bloque pas le processeur central : celui-ci peut continuer à exécuter des instructions machine en parallèle à l'exécution de l'entrée-sortie. Afin d'utiliser ces possibilités, le système exécute plusieurs programmes usagers simultanément, ce qui conduit à une **simultanéité inter-usagers**, c'est-à-dire entre différents programmes utilisateurs.

Notion III.12 : Simultanéité inter-usagers (*Inter-user parallelism*)

Type de simultanéité consistant à exécuter un programme d'application en processeur central pendant qu'un autre programme effectue des entrées-sorties.

Un bon système de fichiers doit permettre le partage des fichiers par différents programmes d'application sans que ceux-ci s'en aperçoivent. Le partage peut être simultané, mais aussi plus simplement réparti dans le temps. Nous étudierons plus en détail les problèmes de partage dans le contexte des bases de données où ils se posent avec plus d'acuité encore.

2.6. SÉCURITÉ ET PROTECTION DES FICHIERS

L'objectif sécurité des fichiers contre les accès mal intentionnés, ou plus simplement non autorisés, découle directement du besoin de partager les fichiers. En effet, lorsque les fichiers sont partagés, le propriétaire désire contrôler les accès, les autoriser à certains, les interdire à d'autres. C'est là une des fonctions que doit rendre un bon système de gestion de fichiers. Ces mécanismes sont généralement réalisés à l'aide de noms hiérarchiques et de clés de protection associés à chaque fichier, voire à chaque article. L'usager doit fournir ces noms et ces clés pour accéder au fichier ou à l'article. Nous étudierons les solutions dans le cadre des bases de données où le problème devient plus aigu.

Le gestionnaire de fichiers doit aussi garantir la conservation des fichiers en cas de panne du matériel ou du logiciel. En conséquence, il doit être prévu de pouvoir repartir après panne avec des fichiers corrects. On considère en général deux types de pannes : les pannes simples avec seulement perte du contenu de la mémoire secondaire, les pannes catastrophiques où le contenu de la mémoire secondaire peut être détruit. Ainsi, il est nécessaire d'incorporer des procédures de reprise après pannes simples et catastrophiques. Nous étudierons ces procédures dans le cadre des bases de données où elles sont généralement plus complètes, bien que souvent prises en compte au niveau de la gestion de fichiers.

3. FONCTIONS D'UN GÉRANT DE FICHIERS

3.1. ARCHITECTURE D'UN GESTIONNAIRE DE FICHIERS

Un gestionnaire de fichiers est généralement structuré autour d'un noyau, appelé ici analyseur, qui assure les fonctions de base, à savoir la création/destruction des fichiers, l'allocation de la mémoire secondaire, la localisation et la recherche des

fichiers sur les volumes et la gestion des zones de mémoires intermédiaires appelées tampons. Les méthodes d'accès sont des modules spécifiques qui constituent une couche plus externe et qui utilisent les fonctions du noyau. La figure III.4 représente les différents modules d'un gestionnaire de fichiers typique. Notons que ces modules sont organisés en trois couches de logiciel : les méthodes d'accès, le noyau et le gestionnaire d'entrées-sorties composé de modules plus ou moins spécifiques à chaque périphérique. Chaque couche constitue une machine abstraite qui accomplit un certain nombre de fonctions accessibles aux couches supérieures par des primitives (par exemple, lire ou écrire un article, ouvrir ou fermer un fichier) constituant l'interface de la couche.

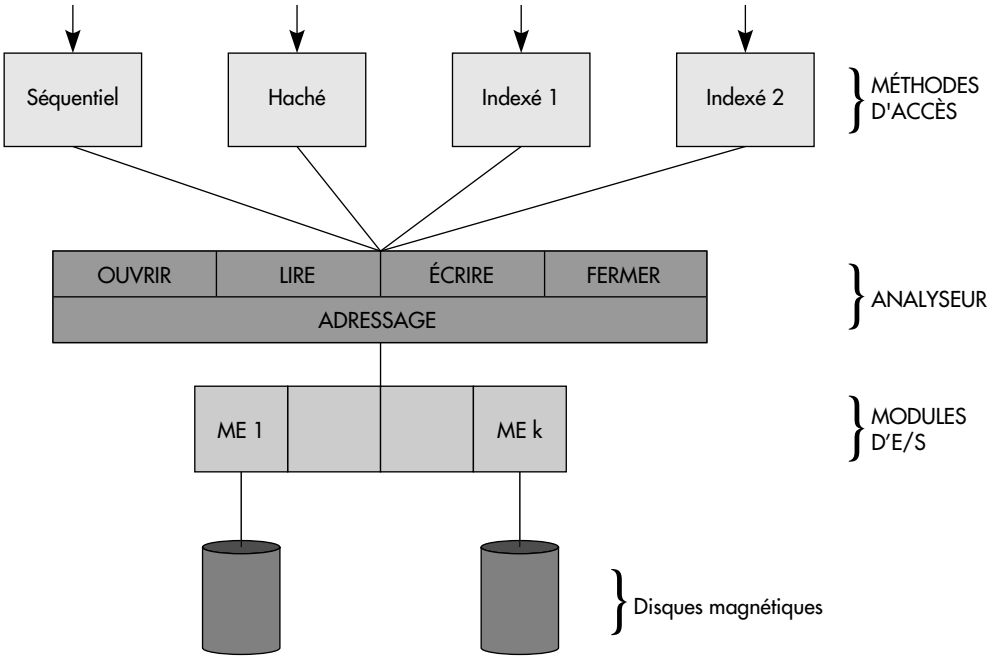


Figure III.4 : Architecture d'un gestionnaire de fichiers

Le noyau ou analyseur d'un gestionnaire de fichiers est chargé d'assurer la gestion des fichiers en temps que récipients non structurés. Il permet en général d'établir le lien entre un fichier et un programme (ouverture du fichier), de supprimer ce lien (fermeture), de lire ou écrire une suite d'octets à un certain emplacement dans un fichier. Il accède aux mémoires secondaires par l'intermédiaire du gestionnaire d'entrées-sorties. Celui-ci, qui n'appartient pas à proprement parler au gestionnaire de fichiers mais bien au système opératoire, gère les entrées-sorties physiques : il permet la lecture et l'écriture de blocs physiques de données sur tous les périphériques, gère les particularités de chacun, ainsi que les files d'attente d'entrées-sorties.

Chaque module méthode d'accès est à la fois chargé de l'organisation des articles dans le fichier et de la recherche des articles à partir de la clé. Un bon système de fichiers doit offrir une grande variété de méthodes d'accès. Il offrira bien sûr une méthode d'accès séquentielle, mais surtout plusieurs méthodes d'accès sélectives.

3.2. FONCTIONS DU NOYAU D'UN GESTIONNAIRE DE FICHIERS

3.2.1. Manipulation des fichiers

Le programmeur travaillant au niveau du langage machine, ainsi que les modules méthodes d'accès accèdent au noyau du gestionnaire de fichiers à l'aide d'un ensemble d'instructions de manipulation de fichiers. Tout d'abord, deux instructions permettent de **créer** et de **détruire** un fichier. Puis, avant de **lire** ou d'**écrire** des données dans un fichier, il faut contrôler son identité et ouvrir un chemin pour les données entre le programme effectuant les lectures-écritures et la mémoire secondaire. Cette opération est généralement effectuée par une instruction d'ouverture : **ouvrir**. L'opération inverse est exécutée lorsque le programme se désintéresse du fichier par une instruction de fermeture : **fermer**.

3.2.2. Adressage relatif

Un fichier étant généralement discontinu sur mémoire secondaire, il est utile de pouvoir adresser son contenu à l'aide d'une adresse continue de 0 à n appelée **adresse relative**. Cela présente l'intérêt de disposer d'un repérage indépendant de la localisation du fichier sur mémoire secondaire (en cas de recopie du fichier, l'adresse relative ne change pas) et de pouvoir assurer que l'on travaille bien à l'intérieur d'un fichier sans risque d'atteindre un autre fichier (il suffit de contrôler que l'adresse relative ne dépasse pas la taille du fichier).

Notion III.13 : Adresse relative (*Relative address*)

Numéro d'unité d'adressage dans un fichier (autrement dit déplacement par rapport au début du fichier).

Pour réaliser l'adressage relatif, on divise généralement le fichier en **pages** (on trouve également selon les implantations les termes de bloc, groupe, intervalle) : une adresse relative octet se compose alors d'un numéro de page suivi d'un numéro d'octet dans la page. Pour éviter un nombre trop important d'entrées-sorties, la taille de la page est choisie de façon à contenir plusieurs enregistrements physiques et des tampons d'une page sont utilisés. La taille de la page, fixée dans le système ou lors de la création du

fichier, est le plus souvent de l'ordre de quelques kilos (K) octets (par exemple, 4K). Elle dépend parfois de l'organisation du fichier. Celle d'un enregistrement physique dépasse rarement quelques centaines d'octets.

Ainsi, l'analyseur d'un gestionnaire de fichiers offre généralement la possibilité d'accéder à une ou plusieurs pages d'adresse relative (numéro de page) donnée dans un fichier. Il peut aussi permettre d'accéder directement aux octets, donc de lire une suite d'octets à partir d'une adresse relative en octets dans le fichier. L'analyseur se compose essentiellement d'algorithmes d'allocation de mémoire secondaire et de conversion d'adresse relative page en adresse réelle de l'enregistrement physique (secteur sur disque), et réciproquement. Finalement, il permet de banaliser toutes les mémoires secondaires en offrant un accès par adresse relative uniforme, avec quelques restrictions cependant : il est par exemple interdit d'accéder autrement qu'en séquentiel à une bande magnétique.

Les articles de l'usager sont alors implantés dans les pages par les méthodes d'accès selon l'organisation choisie. Si plusieurs articles sont implantés dans une même page, on dit qu'il y a **blocage**. Dans le cas où les articles sont implantés consécutivement sans trou à la suite les uns des autres, on dit qu'il y a **compactage** : aucune place n'est alors perdue sur la mémoire secondaire, mais des articles peuvent être à cheval sur plusieurs pages.

3.2.3. Allocation de la place sur mémoires secondaires

La taille d'un fichier est fixée soit de manière statique lors de sa création, soit de manière dynamique au fur et à mesure des besoins. Des solutions intermédiaires sont possibles avec une taille initiale extensible par paliers. Dans tous les cas, il est nécessaire de réserver des zones de mémoires secondaires continues pour le fichier. Ces zones sont appelées **régions**.

Notion III.14 : Région (*Allocation area*)

Ensemble de zones de mémoires secondaires (pistes) adjacentes allouées en une seule fois à un fichier.

Comme les fichiers vivent et sont de tailles différentes, les régions successivement allouées à un fichier ne sont généralement pas contiguës sur une mémoire secondaire. Le gestionnaire de fichiers doit alors pouvoir retrouver les régions composant un fichier. Pour cela, il peut soit garder la liste des régions allouées à un fichier dans une table, soit les chaîner, c'est-à-dire mettre dans une entrée d'une table correspondant à chaque région l'adresse de la région suivante.

La taille d'une région peut varier à partir d'un seuil minimal appelé **granule** (par exemple une piste, etc.). Il devient alors possible d'allouer des régions de taille variable à un fichier, mais toujours composées d'un nombre entier de granules consé-

cutifs. Un granule est souvent choisi de taille égale à une piste où à une fraction de piste.

Notion III.15 : Granule d'allocation (*Allocation granule*)

Unité de mémoire secondaire allouable à un fichier.

Lorsqu'un fichier est détruit ou rétréci, les régions qui lui étaient allouées et qui ne sont plus utilisées sont libérées. Les granules composants sont alors libérés et introduits dans la liste des granules libres. Afin de maximiser la proximité des granules alloués à un fichier, plusieurs méthodes d'allocations ont été proposées. Nous allons étudier ci-dessous quelques algorithmes d'allocation des régions aux fichiers.

3.2.4. Localisation des fichiers sur les volumes

Il est nécessaire de pouvoir identifier un volume. En effet, lorsqu'un volume amovible n'est pas actif, il peut être enlevé. Il faut donc une intervention manuelle pour monter/démonter un volume sur un tourne-disque. Cette opération est exécutée par un opérateur. Celui-ci reconnaît un volume grâce à un numéro (ou nom) attribué à chaque volume. Pour pouvoir contrôler les opérateurs lors des montages/démontages de volume et éviter les erreurs, ce numéro est écrit sur le volume dans le **label** du volume.

Notion III.16 : Label de volume (*Label*)

Premier secteur d'un volume permettant d'identifier ce volume et contenant en particulier son numéro.

Lorsqu'on a retrouvé et monté le volume supportant un fichier, il faut retrouver le fichier sur le volume. Pour cela, chaque fichier possède un ensemble de données descriptives (nom, adresse début, localisation, etc.) regroupées dans un **descripteur du fichier**.

Notion III.17 : Descripteur de fichier (*Directory entry*)

Ensemble des informations permettant de retrouver les caractéristiques d'un fichier, contenant en particulier son nom, sa localisation sur disque, etc.

Il est important que l'ensemble des descripteurs de fichiers contenu sur un volume définisse tous les fichiers d'un volume. On obtient ainsi des volumes autodocumentés, donc portables d'une installation à une autre. Pour cela, les descripteurs de fichiers d'un volume sont regroupés dans une table des matières du volume appelée **catalogue**.

Notion III.18 : Catalogue (*Directory*)

Table (ou fichier) située sur un volume et contenant les descripteurs des fichiers du volume.

Le catalogue est soit localisé en un point conventionnel du volume (par exemple, à partir du secteur 1), soit écrit dans un fichier spécial de nom standard. Le descripteur de ce premier fichier peut alors être contenu dans le label du volume. En résumé, la figure III.5 illustre l'organisation des informations étudiées jusqu'à présent sur un volume.

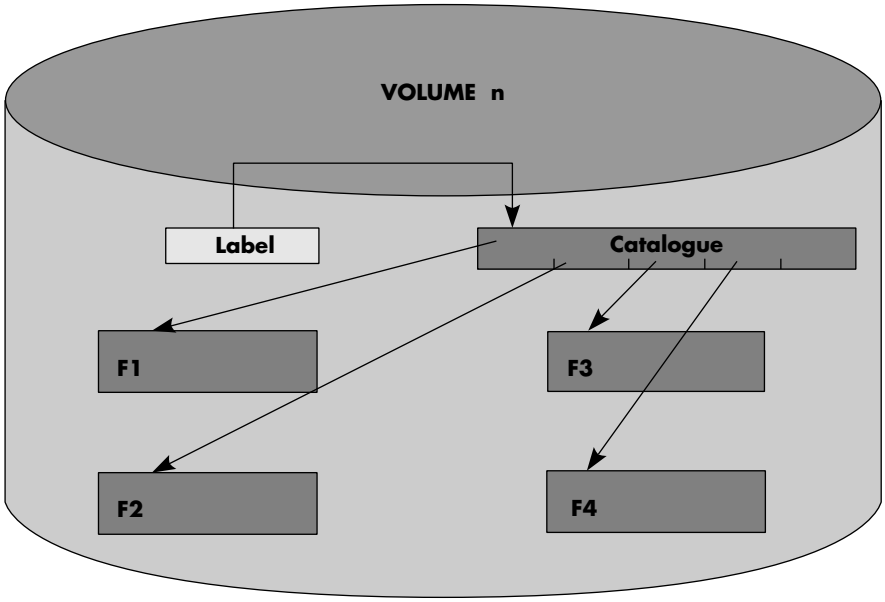


Figure III.5 : Schéma représentant l'organisation d'un volume

3.2.5. Classification des fichiers en hiérarchie

Quand le nombre de fichiers d'une installation devient élevé, on est conduit à classer les descripteurs de fichiers dans plusieurs catalogues, par exemple un par usager. Les descripteurs des fichiers catalogues peuvent alors être maintenus dans un catalogue de niveau plus élevé. On aboutit ainsi à des **catalogues hiérarchisés** qui sont implantés dans de nombreux systèmes [Daley65].

Notion III.19 : Catalogue hiérarchisé (Hierarchical directory)

Catalogue constitué d'une hiérarchie de fichiers, chaque fichier contenant les descripteurs des fichiers immédiatement inférieurs dans la hiérarchie.

Dans un système à catalogues hiérarchisés, chaque niveau de catalogue est généralement spécialisé. Par exemple, le niveau 1 contient un descripteur de fichier catalogue par usager. Pour chaque usager, le niveau 2 peut contenir un descripteur de fichier par application. Enfin, pour chaque couple <usager-application>, le niveau 3 peut contenir la liste des descripteurs de fichiers de données. La figure III.6 illustre un exemple de catalogue hiérarchisé. Le descripteur du catalogue de niveau 1 est appelé racine.

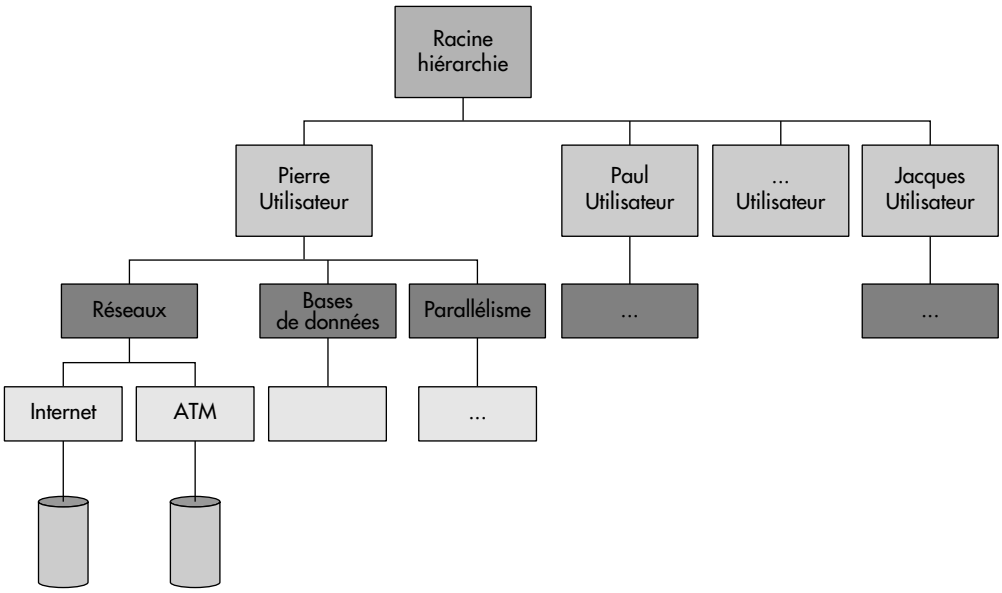


Figure III.6 : Exemple de catalogue hiérarchisé

La présence de catalogue hiérarchisé conduit à introduire des noms de fichiers composés. Pour atteindre le descripteur d'un fichier, il faut en effet indiquer le nom du chemin qui mène à ce descripteur. Voici des noms de fichiers possibles avec le catalogue représenté figure III.6 :

– PIERRE
– PIERRE > BASES-DE-DONNEES
– PIERRE > BASES-DE-DONNES > MODELES

Afin d'éviter un cloisonnement trop strict des fichiers, les systèmes à catalogues hiérarchisés permettent en général l'introduction de *liens*. Un lien est simplement un descripteur qui contient un pointeur logique sur un autre descripteur, éventuellement dans une autre branche de la hiérarchie. Par exemple, le descripteur de nom > LIONEL > BASES-DE-DONNEES > LANGAGES pourra être un descripteur de type lien indiquant qu'il est un synonyme du descripteur > PIERRE > BASES-DE-DONNEES > LANGAGES. Les noms d'usagers étant généralement fournis par le système, des descripteurs de type lien permettent le partage des fichiers entre usagers.

Pour simplifier la nomination des fichiers, les systèmes à catalogues hiérarchisés gèrent bien souvent la notion de **catalogue de base**, encore appelé catalogue courant. Lors du début de session (login), le système positionne le catalogue courant par exemple sur les applications de l'usager. Celui-ci peut alors se déplacer par rapport à ce catalogue courant. Par exemple, Pierre passera au catalogue bases-de-données par le nom > Bases-de-données. Celui-ci deviendra alors son catalogue courant. L'accès au fichier modèles se fera alors par le nom > Modèles. Le retour au catalogue initial de Pierre s'effectuera par < < ; en effet, les noms de répertoires sont déterminés de manière unique quand on remonte la hiérarchie.

3.2.6. Contrôle des fichiers

Le noyau d'un gestionnaire de fichiers inclut également des fonctions de contrôle des fichiers : partage des fichiers, résistances aux pannes, sécurité et confidentialité des données. Nous n'étudions pas ici ces problèmes, qui font l'objet de nombreux développements dans le contexte des bases de données, donc dans les chapitres suivants.

3.3. STRATÉGIE D'ALLOCATION DE LA MÉMOIRE SECONDAIRE

3.3.1. Objectifs d'une stratégie

Il existe différentes stratégies d'allocation de la mémoire secondaire aux fichiers. Une bonne stratégie doit chercher :

1. à minimiser le nombre de régions à allouer à un fichier pour réduire d'une part les déplacements des bras des disques lors des lectures en séquentiel, d'autre part le nombre de descripteurs de régions associés à un fichier ;
2. à minimiser la distance qui sépare les régions successives d'un fichier, pour réduire les déplacements de bras en amplitude.

Les stratégies peuvent être plus ou moins complexes. La classe la plus simple de stratégies alloue des régions de taille fixe égale à un granule, si bien que les notions de granule et région sont confondues. Une classe plus performante alloue des régions de tailles variables, composées de plusieurs granules successifs. Nous allons approfondir ces deux classes ci-dessous.

Auparavant, il est nécessaire de préciser que toutes les méthodes conservent une table des granules libres ; une copie de cette table doit être stockée sur disque pour des raisons de fiabilité. La table elle-même peut être organisée selon différentes méthodes :

– liste des granules ou régions libres, ordonnée ou non ; l'allocation d'une région consiste alors à enlever la région choisie de cette liste pour l'associer au descripteur du fichier ;

– table de bits dans laquelle chaque bit correspond à un granule; l'allocation d'un granule consiste alors à trouver un bit à 0, le positionner à 1 et à adjoindre l'adresse du granule alloué au descripteur du fichier.

3.3.2. Stratégie par granule (à région fixe)

Ces stratégies confondent donc les notions de région et de granule. Elles sont simples et généralement implantées sur les petits systèmes. On peut distinguer :

– La **stratégie du premier trouvé :** le granule correspondant à la tête de liste de la liste des granules libres, ou au premier bit à 0 dans la table des granules libres, est choisi ;

– La **stratégie du meilleur choix :** le granule le plus proche (du point de vue déplacement de bras) du dernier granule alloué au fichier est retenu.

3.3.3. Stratégie par région (à région variable)

Ces stratégies permettent d'allouer des régions composées de plusieurs granules consécutifs, selon les besoins des fichiers. Le noyau du gestionnaire de fichiers reçoit alors des demandes d'allocation et libération de régions de tailles variables. Dans le cas où aucune région de la taille demandée ne peut être constituée par des granules consécutifs, la demande peut éventuellement être satisfaite par plusieurs régions. Parmi les stratégies par région, on peut distinguer :

– La **stratégie du plus proche choix**. Deux régions libres consécutives sont fusionnées en une seule. Lors d'une demande d'allocation, la liste des régions libres est parcourue jusqu'à trouver une région de la taille demandée ; si aucune région de la taille demandée n'est libre, la première région de taille supérieure est découpée en une région de la taille cherchée qui est allouée au fichier et une nouvelle région libre correspondant au reste.

– La **stratégie des frères siamois**. Elle offre la possibilité d'allouer des régions de 1, 2, 4, 8… 2^K granules. Des listes séparées sont maintenues pour les régions libres de dimensions 2^0, 2^1… 2^K granules. Lors d'une demande d'allocation, une région libre peut être extraite de la liste des régions libres de taille 2^{i+1} pour constituer deux régions libres de taille 2^i. Lors d'une libération, deux régions libres consécutives (deux siamoises) de taille 2^i sont fusionnées afin de constituer une région libre de taille 2^{i+1}. L'algorithme de recherche d'une région libre de taille 2^i consiste à chercher cette région dans la liste des régions de taille 2^i. Si cette liste est vide, on recherche alors une région de taille 2^{i+1} que l'on divise en deux. S'il n'y en a pas, on passe alors au niveau suivant 2^{i+2}, etc., jusqu'à atteindre le niveau k; c'est seulement dans le cas où les listes 2^i à 2^K sont vides que l'on ne peut satisfaire la demande par une seule région. Cet algorithme, qui est emprunté aux mécanismes d'allocation de segments dans les systèmes paginés [Lister84], est sans doute le plus efficace ; il est aussi très bien adapté à certaines méthodes d'accès.

En résumé, les stratégies par région de tailles variables sont en général plus efficaces du point de vue déplacement de bras et taille de la table des régions d'un fichier. Un problème commun à ces stratégies peut cependant survenir après des découpages trop nombreux s'il n'existe plus de régions de taille supérieure à un granule. Dans ce cas, l'espace disque doit être réorganisé (ramassage des miettes). Ce dernier point fait que les stratégies par granule restent les plus utilisées.

4. ORGANISATIONS ET MÉTHODES D'ACCÈS PAR HACHAGE

Les organisations et méthodes d'accès par hachage sont basées sur l'utilisation d'une fonction de calcul qui, appliquée à la clé, détermine l'adresse relative d'une zone appelée paquet (*bucket* en anglais) dans laquelle est placé l'article. On distingue les méthodes d'accès par hachage statique, dans lesquelles la taille du fichier est fixe, des méthodes par hachage dynamique, où le fichier peut grandir.

4.1. ORGANISATION HACHÉE STATIQUE

C'est la méthode la plus ancienne et la plus simple. Le fichier est de taille constante, fixée lors de sa création. Une fois pour toute, il est donc divisé en p paquets de taille fixe L. La clé permet de déterminer un numéro de paquet N dont l'adresse relative est obtenue par la formule AR = N * L. Nous définirons un **fichier haché statique** comme suit.

Notion III.20 : Fichier haché statique (*Static hashed file*)

Fichier de taille fixe dans lequel les articles sont placés dans des paquets dont l'adresse est calculée à l'aide d'une fonction de hachage fixe appliquée à la clé.

À l'intérieur d'un paquet, les articles sont rangés à la suite dans l'ordre d'arrivée. Ils sont retrouvés grâce à la donnée contenant la clé. La figure III.7 illustre un exemple de structure interne d'un paquet. En tête du paquet, on trouve l'adresse du premier octet libre dans le paquet. Ensuite, les articles successifs du paquet sont rangés avec leur longueur en tête, par exemple sur deux octets. À l'intérieur d'un tel paquet, on accède à un article par balayage séquentiel. Des structures de paquet plus sophistiquées permettent l'accès direct à un article de clé donnée à l'intérieur d'un paquet. De telles structures sont plus efficaces que la structure simple représentée figure III.7.

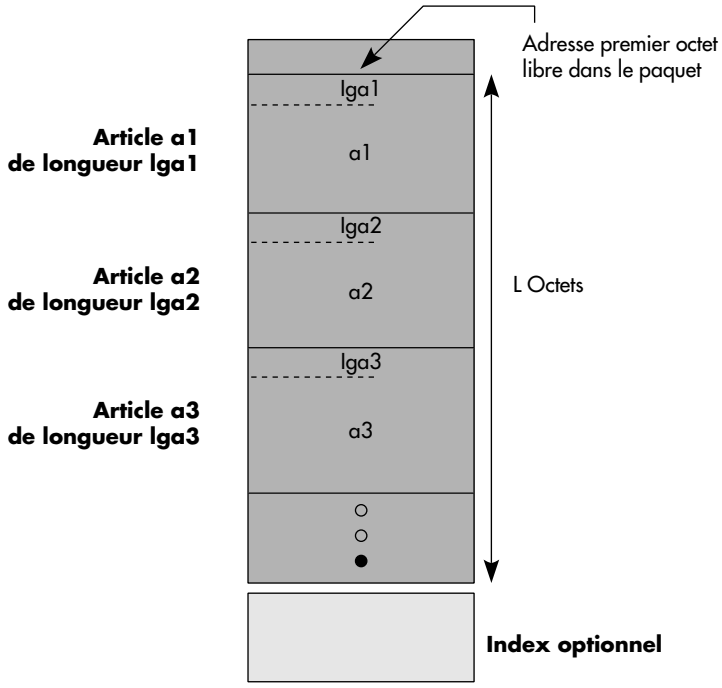


Figure III.7 : Structure interne d'un paquet

Lorsqu'un nouvel article est inséré dans un fichier, il est logé à la première place libre dans le paquet. S'il n'y a pas de place libre, on dit qu'il y a débordement. Il faut évidemment contrôler l'unicité de la clé d'un article lors des insertions. Cela nécessite de balayer tous les articles du paquet.

À partir de la clé d'un article, on calcule le numéro de paquet dans lequel l'article est placé à l'aide d'une fonction appelée **fonction de hachage** (Fig. III.8). Une fonction de hachage doit être choisie de façon à distribuer uniformément les articles dans les paquets. Plusieurs techniques sont possibles :

– le pliage, qui consiste à choisir et combiner des bits de la clé (par exemple par « ou exclusif ») ;
– les conversions de la clé en nombre entier ou flottant avec utilisation de la mantisse permettants d'obtenir également un numéro de paquet ;
– le modulo, sans doute la fonction la plus utilisée, qui consiste à prendre pour numéro de paquet le reste de la division de la clé par le nombre de paquets.

Ces techniques peuvent avantageusement être combinées.

Soit un fichier de 47 paquets et des articles de clé numérique. Le modulo 47 pourra alors être choisi comme fonction de hachage. Ainsi, l'article de clé 100 sera placé

dans le paquet 6, l'article de clé 47 dans le paquet 0, celui de clé 123 dans le paquet 29, etc.

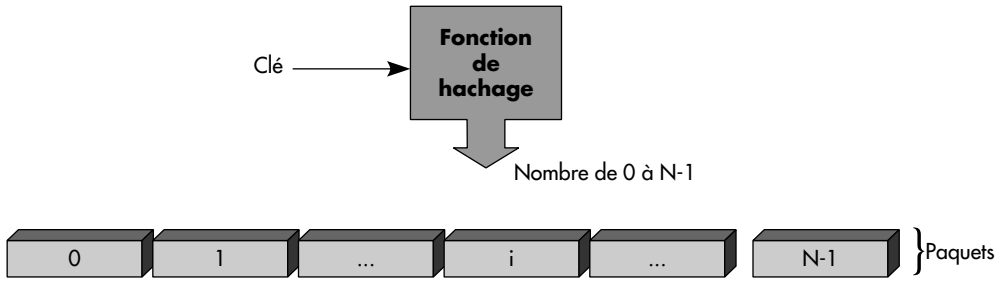


Figure III.8 : Illustration d'un fichier haché statique

Le problème de débordement se pose lorsqu'un paquet est plein. Une première solution simple consiste à ne pas gérer de débordements et à répondre fichier saturé à l'utilisateur. Cela implique une mauvaise utilisation de la place occupée par le fichier, surtout si la distribution des articles dans les paquets est mauvaise. Des solutions plus satisfaisantes consistent à utiliser une technique de débordement parmi l'une des suivantes [Knuth73] :

– l'**adressage ouvert** consiste à placer l'article qui devrait aller dans un paquet plein dans le premier paquet suivant ayant de la place libre ; il faut alors mémoriser tous les paquets dans lequel un paquet plein a débordé ;
– le **chaînage** consiste à constituer un paquet logique par chaînage d'un paquet de débordement à un paquet plein ;
– le **rehachage** consiste à appliquer une deuxième fonction de hachage lorsqu'un paquet est plein; cette deuxième fonction conduit généralement à placer les articles dans des paquets de débordements.

Dans tous les cas, la gestion de débordements dégrade les performances et complique la gestion des fichiers hachés.

La méthode d'accès basée sur l'organisation hachée statique a plusieurs avantages. En particulier, elle s'adapte à des fichiers de clés quelconques, reste simple et donne d'excellentes performances tant qu'il n'y a pas de débordements : une lecture d'article s'effectue en une entrée-sortie (lecture du paquet) alors qu'une écriture en nécessite en général deux (lecture puis réécriture du paquet). Cependant, les débordements dégradent rapidement les performances. De plus, le taux d'occupation de la mémoire secondaire réellement utilisée peut rester assez éloigné de 1. Enfin, la taille d'un fichier doit être fixée a priori. Si le nombre d'articles d'un fichier devient plus important que prévu, le fichier doit être réorganisé.

4.2. ORGANISATIONS HACHÉES DYNAMIQUES

4.2.1. Principes du hachage dynamique

La première organisation **hachée dynamique** a été proposée pour des tables en mémoire [Knott71]. Puis plusieurs techniques fondées sur le même principe mais différentes ont été proposées pour étendre les possibilités du hachage à des fichiers dynamiques [Fagin79, Larson78, Litwin78, Larson80, Litwin80]. Le principe de base de ces différentes techniques est la digitalisation progressive de la fonction de hachage : la chaîne de bits résultat de l'application de la fonction de hachage à la clé est exploitée progressivement bit par bit au fur et à mesure des extensions du fichier.

Plus précisément, les méthodes dynamiques utilisent une fonction de hachage de la clé h(K) générant une chaîne de N bits, où N est grand (par exemple 32). La fonction est choisie de sorte qu'un bit quelconque de h(K) ait la même probabilité d'être à 1 ou à 0. Lors de la première implantation du fichier haché, seuls les M premiers bits de h(K) (avec M petit devant N) sont utilisés pour calculer le numéro de paquet dans lequel placer un article. Ensuite, lorsque le fichier est considéré comme saturé (par exemple, lorsqu'un premier paquet est plein), une partie du fichier (par exemple le paquet plein) est doublée : une nouvelle région est allouée pour cette partie et les articles de l'ancienne partie sont distribués entre l'ancienne partie et la nouvelle en utilisant le bit M+1 de la fonction de hachage. Ce processus d'éclatement est appliqué chaque fois que le fichier est saturé, de manière récursive. Ainsi, les bits (M+1), (M+2), (M+3)… de la fonction de hachage sont successivement utilisés et le fichier peut grandir jusqu'à 2^N paquets. Une telle taille est suffisante sous l'hypothèse que N soit assez grand.

Les méthodes de hachage dynamique diffèrent par les réponses qu'elles apportent aux questions suivantes :

(Q1) Quel est le critère retenu pour décider qu'un fichier haché est saturé ?

(Q2) Quelle partie du fichier faut-il doubler quand un fichier est saturé ?

(Q3) Comment retrouver les parties d'un fichier qui ont été doublées et combien de fois ont-elles été doublées ?

(Q4) Faut-il conserver une méthode de débordement, et si oui laquelle ?

Nous présentons ci-dessous deux méthodes qui nous paraissent des plus intéressantes: le hachage extensible [Fagin79] et le hachage linéaire [Litwin80]. Vous trouverez des méthodes sans doute plus élaborées dans [Larson80], [Lomet83], [Samet89] ainsi que des évaluations du hachage extensible dans [Scholl81] et du hachage linéaire dans [Larson82].

4.2.2. Le hachage extensible

Le **hachage extensible** [Fagin79] apporte les réponses suivantes aux questions précédentes :

(Q1) Le fichier est étendu dès qu'un paquet est plein ; dans ce cas un nouveau paquet est ajouté au fichier.

(Q2) Seul le paquet saturé est doublé lors d'une extension du fichier. Il éclate selon le bit suivant du résultat de la fonction de hachage appliquée à la clé h(K). Les articles ayant ce bit à 0 restent dans le paquet saturé, alors que ceux ayant ce bit à 1 partent dans le nouveau paquet.

(Q3) La fonction de hachage adresse un répertoire des adresses de paquets ; la taille du répertoire est 2**(M+P) où P est le niveau du paquet qui a éclaté le plus grand nombre de fois. Chaque entrée du répertoire donne l'adresse d'un paquet. Les 2**(P-Q) adresses correspondant à un paquet qui a éclaté Q fois sont identiques et pointent sur ce paquet. Ainsi, par l'indirection du répertoire, le système retrouve les paquets.

(Q4) La gestion de débordement n'est pas nécessaire.

Le hachage extensible associe donc à chaque fichier un répertoire des adresses de paquets. Au départ, M bits de la fonction de hachage sont utilisés pour adresser le répertoire. À la première saturation d'un paquet, le répertoire est doublé, et un nouveau paquet est alloué au fichier. Le paquet saturé est distribué entre l'ancien et le nouveau paquets, selon le bit suivant (M+1) de la fonction de hachage. Ensuite, tout paquet plein est éclaté en deux paquets, lui-même et un nouveau paquet alloué au fichier. L'entrée du répertoire correspondant au nouveau paquet est mise à jour avec l'adresse de ce nouveau paquet si elle pointait encore sur le paquet plein. Sinon, le répertoire est à nouveau doublé. En résumé, le hachage extensible peut être défini comme suit :

Notion III.21 : Hachage extensible (*Extensible hashing*)

Méthode de hachage dynamique consistant à éclater un paquet plein et à mémoriser l'adresse des paquets dans un répertoire adressé directement par les (M+P) premiers bits de la fonction de hachage, où P est le nombre d'éclatements maximal subi par les paquets.

Cette organisation est illustrée figure III.9. Nous montrons ici un répertoire adressé par 3 bits de la fonction de hachage. Le fichier avait été créé avec deux paquets et était adressé par le premier bit de la fonction de hachage (celui de droite). Puis le paquet 1 a éclaté et a été distribué entre le paquet 01 et 11. Le paquet 11 a éclaté à son tour et a été distribué entre le paquet 011 et le paquet 111. Le répertoire a donc été doublé deux fois (P = 2 alors que M = 1).

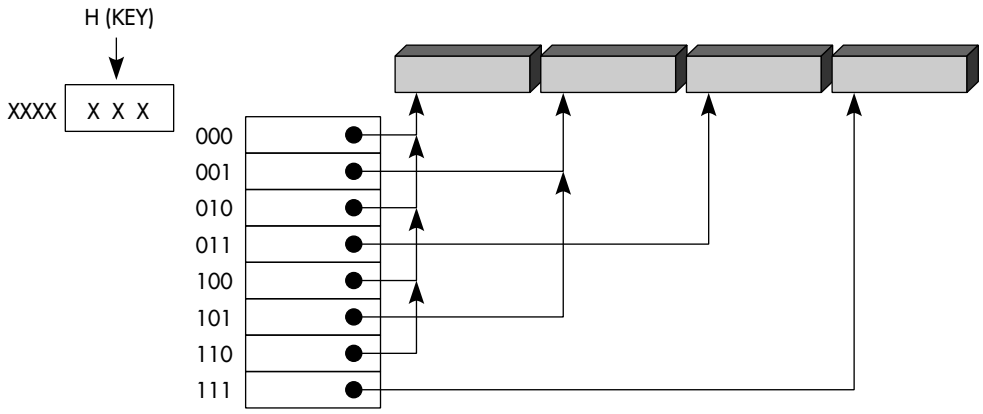


Figure III.9 : Répertoire et paquets d'un fichier haché extensible

Un fichier haché extensible est donc structuré en deux niveaux : le répertoire et les paquets. Soit P le niveau d'éclatement maximal du fichier. Le répertoire contient un en-tête qui indique la valeur de M+P, le nombre de bits de la fonction de hachage utilisés pour le paquet ayant le plus éclaté. Après l'en-tête figurent des pointeurs vers les paquets. Les M+P premiers bits de la fonction de hachage sont donc utilisés pour adresser le répertoire. Le premier pointeur correspond à la valeur 0 des (M+P) premiers bits de la fonction de hachage, alors que le dernier correspond à la valeur 2**(M+P) – 1, c'est-à-dire aux (M+P) premiers bits à 1. Soit Q le nombre d'éclatements subis par un paquet. À chaque paquet sont associés dans le répertoire 2**(P-Q) pointeurs qui indiquent son adresse. Le répertoire pointe ainsi plusieurs fois sur le même paquet, ce qui accroît sa taille.

L'insertion d'un article dans un fichier haché extensible nécessite tout d'abord l'accès au répertoire. Pour cela, les (M+P) bits de la clé sont utilisés. L'adresse du paquet dans lequel l'article doit être placé est ainsi lue dans l'entrée adressée du répertoire. Si le paquet est plein, alors celui-ci doit être doublé et son niveau d'éclatement Q augmenté de 1 ; un paquet frère au même niveau d'éclatement doit être créé ; les articles sont répartis dans les deux paquets selon la valeur du bit (M+Q+1) de la fonction de hachage. Le mécanisme d'éclatement de paquet est illustré figure III.10. Si le niveau du répertoire P est supérieur à Q, alors le répertoire doit simplement être mis à jour, 2**(P-Q+1) pointeurs étant forcés sur l'adresse du nouveau paquet. Si P est égal à Q, alors le répertoire doit être doublé.

En cas de suppression dans un paquet adressé par M+Q bits de la fonction de hachage, il est possible de tenter de regrouper ce paquet avec l'autre paquet adressé par M+Q bits s'il existe. Ainsi, la suppression d'un article dans un paquet peut théoriquement conduire à réorganiser le répertoire. En effet, si le paquet concerné est le seul paquet avec son frère au niveau d'éclatement le plus bas et si la suppression d'un article laisse assez de place libre pour fusionner les deux frères, la fusion peut être entreprise.

Le niveau d'éclatement du répertoire doit alors être réduit de 1 et celui-ci doit être divisé par deux en fusionnant les blocs jumeaux.

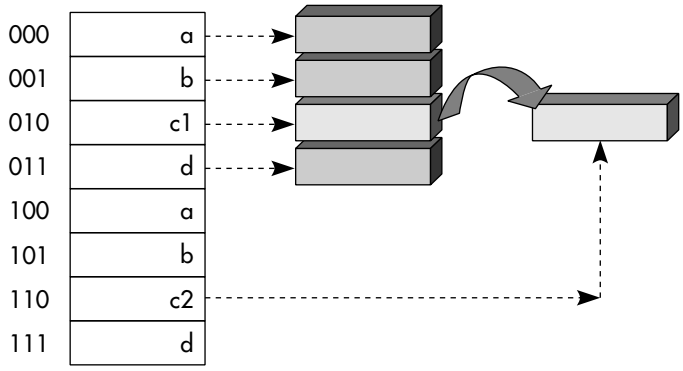


Figure III.10 : Éclatement de paquet dans un fichier haché extensible

4.2.3. Le hachage linéaire

Le **hachage linéaire** [Litwin80] apporte les réponses suivantes aux questions déterminantes d'une méthode de hachage dynamique :

(Q1) Le fichier est étendu dès qu'un paquet est plein, comme dans le hachage extensible ; un nouveau paquet est aussi ajouté au fichier à chaque extension ;

(Q2) Le paquet doublé n'est pas celui qui est saturé, mais un paquet pointé par un pointeur courant initialisé au premier paquet du fichier et incrémenté de 1 à chaque éclatement d'un paquet (donc à chaque saturation) ; lorsque ce pointeur atteint la fin de fichier, il est repositionné au début du fichier ;

(Q3) Un niveau d'éclatement P du fichier (initialisé à 0 et incrémenté lorsque le pointeur courant revient en début de fichier) est conservé dans le descripteur du fichier; pour un paquet situé avant le pointeur courant, (M+P+1) bits de la fonction de hachage doivent être utilisés alors que seulement (M+P) sont à utiliser pour adresser un paquet situé après le pointeur courant et avant le paquet 2**(M+P) ;

(Q4) Une gestion de débordement est nécessaire puisqu'un paquet plein n'est en général pas éclaté ; il le sera seulement quand le pointeur courant passera par son adresse. Une méthode de débordement quelconque peut être utilisée.

En résumé, il est possible de définir le hachage linéaire comme suit :

Notion III.22 : Hachage linéaire (*Linear hashing*)

Méthode de hachage dynamique nécessitant la gestion de débordement et consistant à : (1) éclater le paquet pointé par un pointeur courant quand un paquet est plein, (2) mémoriser le niveau d'éclatement du fichier afin de déterminer le nombre de bits de la fonction de hachage à appliquer avant et après le pointeur courant.

La figure III.11 illustre un fichier haché linéairement. Le pointeur courant est situé en début du 3e paquet (paquet 10). Les paquets 000 et 001, 100 et 101 (c'est-à-dire 0, 1, 4 et 5) sont adressés par les trois premiers bits de la fonction de hachage, alors que les paquets 10 et 11 (c'est-à-dire 2 et 3) sont seulement adressés par les deux premiers bits. Lors du prochain débordement, le paquet 10 (2) éclatera, quel que soit le paquet qui déborde.

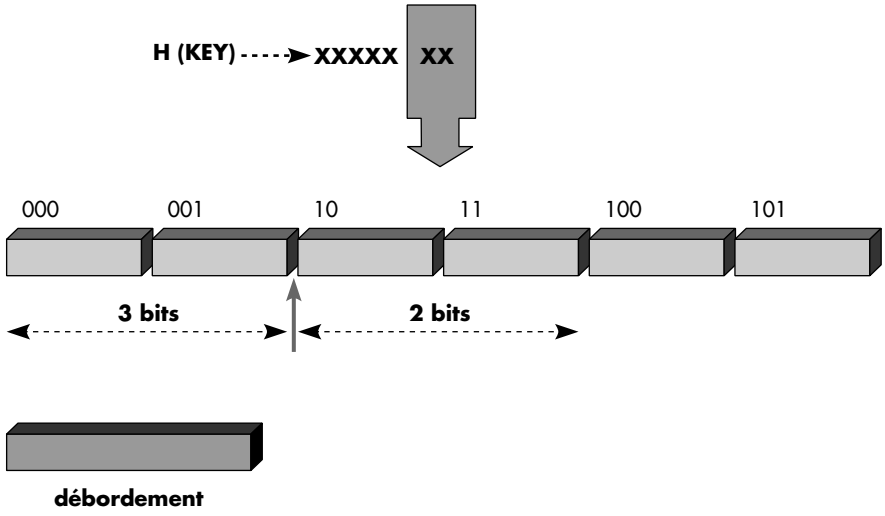


Figure III.11 : Fichier haché linéairement

Notons que le hachage linéaire peut aussi s'implémenter avec un répertoire. Dans ce cas, le pointeur courant est un pointeur sur le répertoire : il référence l'adresse du paquet suivant à éclater. L'avantage du hachage linéaire est alors la simplicité de l'algorithme d'adressage du répertoire ; on utilise tout d'abord M+P bits de la fonction de hachage ; si l'on est positionné avant le pointeur courant, on utilise un bit de plus, sinon on lit l'adresse du paquet dans le répertoire. À chaque éclatement, le répertoire s'accroît d'une seule entrée. L'inconvénient est bien sûr la nécessité de gérer des débordements.

L'insertion d'un article dans un fichier haché linéairement se fait très simplement : si P est le niveau d'éclatement du fichier, (M+P) bits de la clé sont tout d'abord pris en compte pour déterminer le numéro de paquet. Si le numéro obtenu est supérieur au pointeur courant, il est correct ; sinon, un bit supplémentaire de la fonction de hachage est utilisé pour déterminer le numéro de paquet. L'insertion s'effectue ensuite de manière classique, à ceci près que lorsqu'un paquet est saturé, le paquet pointé par le pointeur courant est éclaté ; ce pointeur courant est augmenté de 1 ; s'il atteint le paquet 2**(M+P) du fichier, il est ramené au début et le niveau d'éclatement du fichier est augmenté de un.

De manière analogue au répertoire du hachage extensible, la suppression d'article dans un paquet peut amener la fusion de deux paquets et donc le recul de 1 du pointeur courant. Attention : en général les paquets fusionnés n'incluent pas le paquet dans lequel a lieu la suppression ; la fusion peut amener des distributions d'articles en débordement.

Une variante de cette méthode consistant à changer la condition d'éclatement a été proposée dans [Larson80]. La condition retenue est l'atteinte d'un taux de remplissage maximal du fichier, le taux de remplissage étant le rapport de la place occupée par les articles sur la taille totale du fichier.

En résumé, les méthodes de hachage dynamique sont bien adaptées aux fichiers dynamiques, c'est-à-dire de taille variable, cependant pas trop gros pour éviter les saturations de paquets trop nombreuses et les accroissements de la taille du catalogue. Leurs limites sont encore mal connues. Le hachage extensible paraît plus robuste face aux mauvaises distributions de clés que le hachage linéaire. Par contre, la gestion d'un répertoire est plus lourde que celle d'un pointeur courant.

Le grand problème des méthodes par hachage reste l'absence de possibilités efficaces d'accès ordonné pour un tri total ou pour des questions sur des plages de valeur. Telle est la limite essentielle, qui nécessite l'emploi d'autres méthodes.

5. ORGANISATIONS ET MÉTHODES D'ACCÈS INDEXÉES

Dans cette section, nous étudions les principes de base des organisations avec index et les principales méthodes pratiques.

5.1. PRINCIPES DES ORGANISATIONS INDEXÉES

5.1.1. Notion d'index

Le principe de base des organisations et méthodes d'accès indexées est d'associer à la clé d'un article son adresse relative dans le fichier à l'aide d'une « table des matières » du fichier. Ainsi, à partir de la clé de l'article, un accès rapide est possible par recherche de l'adresse relative dans la table des matières, puis par un accès en relatif à l'article dans le fichier. Les principales méthodes d'accès indexées se distinguent par le mode de placement des articles dans le fichier et par l'organisation de la table des matières, appelée **index**.

Notion III.23 : Index (*Index*)

Table (ou plusieurs tables) permettant d'associer à une clé d'article l'adresse relative de cet article.

La figure III.12 illustre cette notion d'index. Le fichier contient les articles a5, a2, a57, a3 et a10. L'index est rangé en fin de fichier comme le dernier article du fichier. Il contient une entrée par article indiquant la clé de l'article et son adresse relative dans le fichier. L'index d'un fichier peut en général être rangé dans le fichier ou plus rarement dans un fichier spécial.

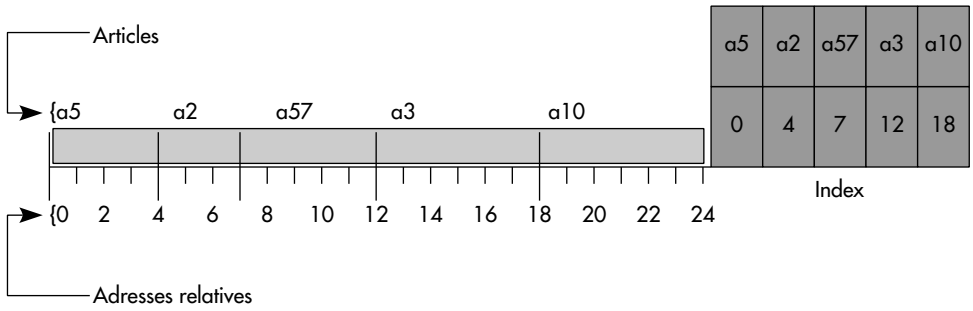


Figure III.12 : Exemple de fichier indexé

Les étapes successives exécutées pour l'accès à un article dans un fichier indexé sont les suivantes :

1. Accès à l'index qui est monté en mémoire dans un tampon.
2. Recherche de la clé de l'article désiré en mémoire afin d'obtenir l'adresse relative de l'article ou d'un paquet contenant l'article.
3. Conversion de l'adresse relative trouvée en adresse absolue par les couches internes du système de gestion de fichiers.
4. Accès à l'article (ou au paquet d'articles) sur disques magnétiques et transfert dans un tampon du système.
5. Transfert de l'article dans la zone du programme usager.

En général, l'accès à un article dans un fichier indexé nécessite une à trois entrées-sorties pour monter l'index en mémoire, puis une entrée sortie pour monter l'article en mémoire. Différentes variantes sont possibles, selon l'organisation des articles dans le fichier et de l'index.

5.1.2. Variantes possibles

Les variantes se distinguent tout d'abord par l'organisation de l'index. Celui-ci peut être trié ou non. Le fait que l'index soit trié autorise la recherche dichotomique. Ainsi,

un index contenant n clés, divisé en blocs (d'une page) de b clés, nécessitera en moyenne n/2b accès pour retrouver une clé s'il n'est pas trié ; il suffira de $\log_2 n/b$ accès s'il est trié. Par exemple, avec $n = 10^6$, b = 100 clés, on obtient 10 accès si l'index est trié contre 5 000 accès sinon.

Un index d'un fichier indexé peut contenir toutes les clés (c'est-à-dire celles de tous les articles) ou seulement certaines. Un index qui contient toutes les clés est appelé index dense. Afin de différencier plus précisément les méthodes d'accès obtenues, il est possible d'introduire la notion de **densité d'un index**.

Notion III.24 : Densité d'un index (*Index key selectivity*)

Quotient du nombre de clés dans l'index sur le nombre d'articles du fichier.

La densité d'un index varie entre 0 et 1. Un index dense a donc une densité égale à 1. Dans le cas d'index non dense, toutes les clés ne figurent pas dans l'index. Aussi, les articles du fichier ainsi que l'index sont triés. Le fichier est divisé en paquets de taille fixe et chaque paquet correspond à une entrée en index contenant le doublet : <plus grande clé du paquet-adresse relative du paquet>. La figure III.13 illustre un index non dense et le fichier correspondant. Le paquet 1 contient les articles de clé 1, 3 et 7. La plus grande clé (7) figure dans l'index non dense, etc. L'index est composé ici d'un seul bloc contenant trois clés, la plus grande de chaque paquet.

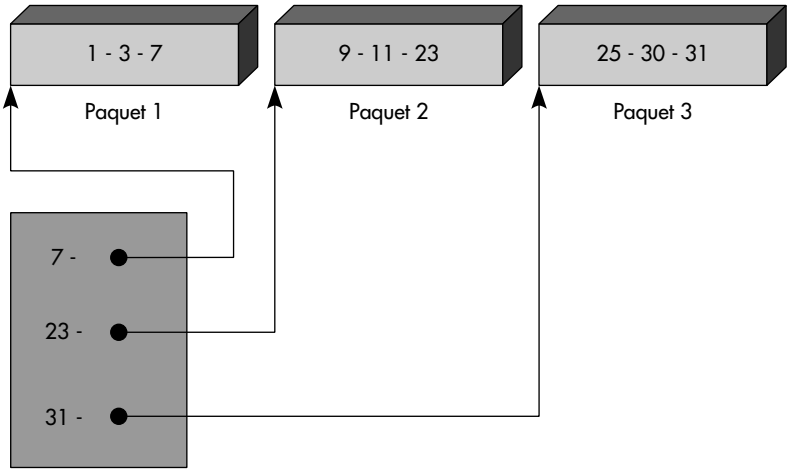


Figure III.13 : Exemple d'index non dense

Comme le fichier peut être trié ou non trié, et l'index dense ou non dense, trié ou non trié, diverses variantes sont théoriquement possibles. Deux méthodes sont particulièrement intéressantes : le fichier séquentiel non trié avec index trié dense, historiquement à la base de l'organisation IS3, et le fichier trié avec index non dense trié, sur

lequel sont fondées des organisations telles ISAM, VSAM et UFAS. Il est impossible d'associer un index non dense à un fichier non trié.

5.1.3. Index hiérarchisé

Un index peut être vu comme un fichier de clés. Si l'index est grand (par exemple plus d'une page), la recherche d'une clé dans l'index peut devenir très longue. Il est alors souhaitable de créer un index de l'index vu comme un fichier de clés. Cela revient à gérer un index à plusieurs niveaux. Un tel index est appelé **index hiérarchisé**.

Notion III.25 : Index hiérarchisé (*Multilevel index*)

Index à n niveaux, le niveau k étant un index trié divisé en paquets, possédant lui-même un index de niveau k+1, la clé de chaque entrée de ce dernier étant la plus grande du paquet.

Un index hiérarchisé à un niveau est un index trié, généralement non dense, composé de paquets de clés. Un index hiérarchisé à n niveaux est un index hiéarchisé à n – 1 niveaux, possédant lui-même un index à un niveau. La figure III.14 illustre un index hiérarchisé à 3 niveaux. Le niveau 1 comporte trois paquets de clés. Le niveau 2 en comporte deux qui contiennent les plus grandes clés des paquets de niveau inférieur. Le niveau 3 est la racine et contient les plus grandes clés des deux paquets de niveau inférieur. La notion d'index hiérarchisé est indépendante du nombre de niveaux, qui peut grandir autant que nécessaire.

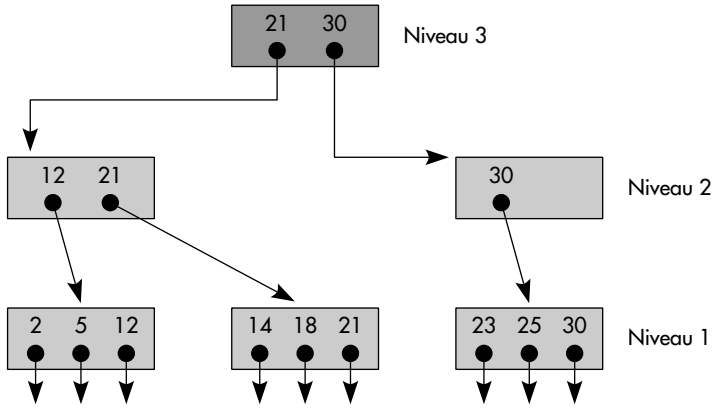


Figure III.14 : Exemple d'index hiérarchisé

5.1.4. Arbres B

Afin de mieux caractériser la notion d'index hiérarchisé et de la rendre indépendante des particularités d'implantation, on a été amené à introduire une structure d'arbre,

avec un nombre variable de niveaux. Cette structure, appelée **arbre B** [Bayer72, Comer79], peut être introduite formellement comme suit :

Notion III.26: Arbre B (*B-tree*)

Un arbre B d'ordre m est un arbre au sens de la théorie des graphes tel que :
1°) Toutes les feuilles sont au même niveau ;
2°) Tout nœud non feuille a un nombre NF de fils tel que m+1 ≤ NF ≤ 2m+1, sauf la racine, qui a

La figure III.15 représente un arbre balancé d'ordre 2. La racine a deux fils. Les deux autres nœuds non feuilles ont trois fils.

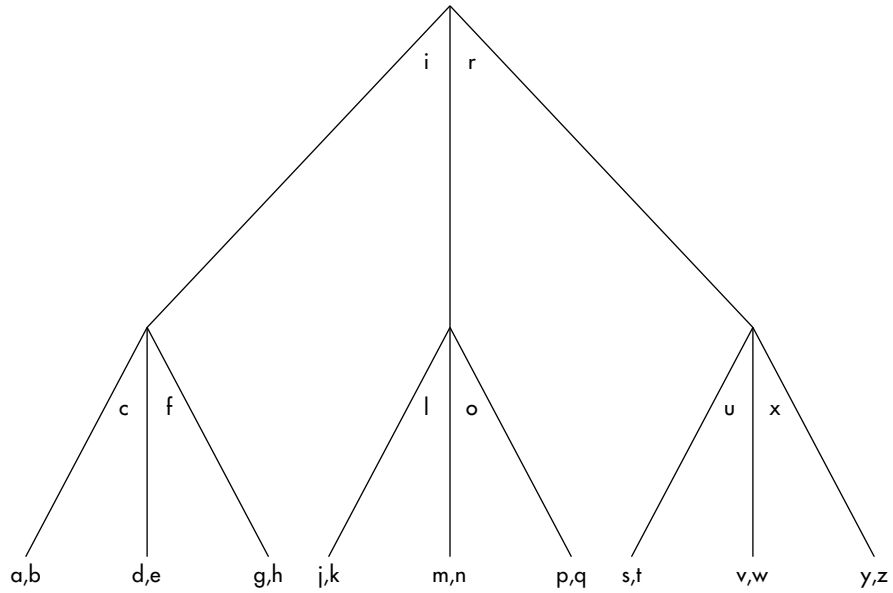
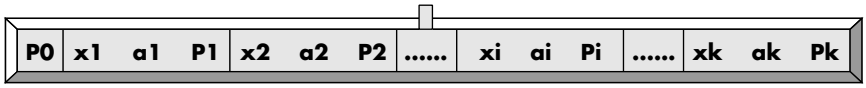


Figure III.15 : Arbre B d'ordre 2

Un arbre B peut être utilisé pour constituer un index hiérarchisé d'un fichier. Dans ce cas, les nœuds représentent des pages de l'index. Ils contiennent des clés triées par ordre croissant et des pointeurs de deux types : les pointeurs internes désignent des fils et permettent de définir les branches de l'arbre, alors que les pointeurs externes désignent des pages de données (en général, des adresses relatives d'articles). La figure III.16 précise la structure d'un nœud. Une clé xi d'un nœud interne sert de séparateur entre les deux branches internes adjacentes (Pi-1 et Pi). Un nœud contient entre m et 2m clés, à l'exception de la racine qui contient entre 1 et 2m clés.



Pi : Pointeur interne permettant de représenter l'arbre; les feuilles ne contiennent pas de pointeurs Pi ;
ai : Pointeur externe sur une page de données ;
xi : valeur de clé.

Figure III.16 : Structure d'un nœud d'un arbre B

De plus, l'ensemble des clés figurant dans l'arbre B doit être trié selon l'ordre post-fixé induit par l'arbre, cela afin de permettre les recherches en un nombre d'accès égal au nombre de niveaux. Plus précisément, en désignant par K(Pi) l'ensemble des clés figurant dans le sous-arbre dont la racine est pointée, on doit vérifier que :

1. (x1, x2…xK) est une suite croissante de clés ;
2. Toute clé y de K(P0) est inférieure à x1 ;
3. Toute clé y de K(P1) est comprise entre xi et xi+1 ;
4. Toute clé y de K(PK) est supérieure à xk.

La figure III.17 représente un exemple d'index sous forme d'arbre B d'ordre 2. Cet arbre contient les valeurs de clé de 1 à 26. Les flèches entre les nœuds représentent les pointeurs internes, les traits courts issus d'une clé les pointeurs externes vers les articles. Dans la suite, nous omettrons les pointeurs externes, qui seront donc implicites.

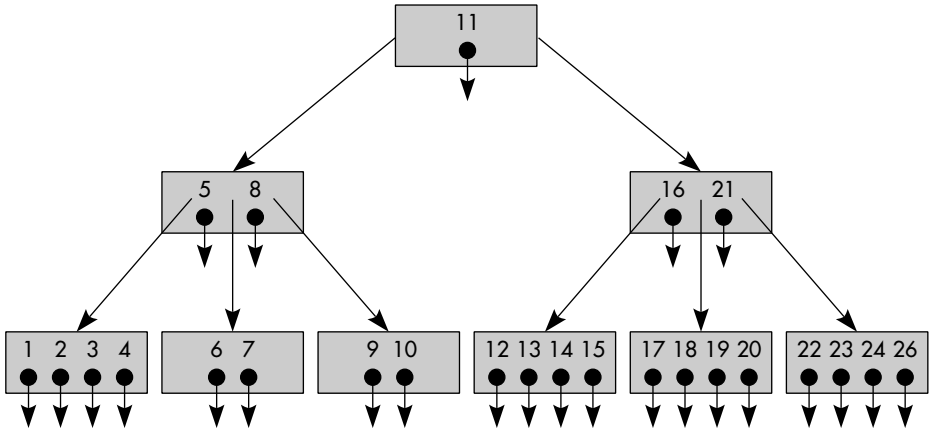


Figure III.17 : Exemple d'index sous forme d'arbre B

La recherche d'une clé dans un arbre B s'effectue en partant de la racine. En règle générale, les valeurs contenues dans un nœud partitionnent les valeurs possibles de clés en un nombre d'intervalles égal au nombre de branches. Ainsi, si la valeur cher-

chée est inférieure à la première clé du nœud, on choisit la première branche ; si elle est comprise entre la première clé et la deuxième clé, on choisit la deuxième branche, etc. Si une clé n'est pas trouvée après recherche dans un nœud terminal, c'est qu'elle n'existe pas.

Le nombre de niveaux d'un arbre B est déterminée par son degré et le nombre de clés contenues. Ainsi, dans le pire des cas, si l'arbre est rempli au minimum, il existe :

– une clé à la racine,

– deux branches en partent avec m clés,

– (m+1) branches partent de ces dernières avec m clés,

– etc.

Pour un arbre de niveaux h, le nombre de clés est donc :

$$N = 1 + 2\,m\,(1+ (m+1) + (m+1)^{2} + \ldots + (m+1)^{h-2})$$

soit, par réduction du développement limité :

$$N = 1 + 2\,((m+1)^{h-1}-1).$$

D'où l'on déduit que pour stocker N clés, il faut :

$$h = 1 + \log_{m+1}((N+1)/2) \text{ niveaux.}$$

Par exemple, pour stocker 1 999 999 clés avec un arbre B de degré 99, $h = 1 + \log_{100}10^{6} = 4$. Au maximum, quatre niveaux sont donc nécessaires. Cela implique qu'un article d'un fichier de deux millions d'articles avec un index hiérarchisé organisé comme un arbre B peut être cherché en quatre entrées-sorties.

L'insertion d'une clé dans un arbre B est une opération complexe. Elle peut être définie simplement de manière récursive comme suit :

a) Rechercher le nœud terminal qui devrait contenir la clé à insérer et l'y insérer en bonne place ;

b) Si le nombre de clés après insertion de la nouvelle clé est supérieur à 2 m, alors migrer la clé médiane au niveau supérieur, en répétant la procédure d'insertion dans le nœud supérieur.

À titre d'exemple, la figure III.18 représente les étapes nécessaires à l'insertion de la clé (25) dans l'arbre B représenté figure III.17. Tout d'abord, la place de la clé 25 est recherchée. Celle-ci doit être insérée dans le dernier nœud à droite (étape a). Cela provoque un éclatement du nœud qui a maintenant plus de 2 m clés, soit 4. La clé médiane 24 doit être remontée au niveau supérieur. Elle est alors insérée après 21 (étape b). Le nœud ayant trois clés, aucun autre éclatement n'est nécessaire.

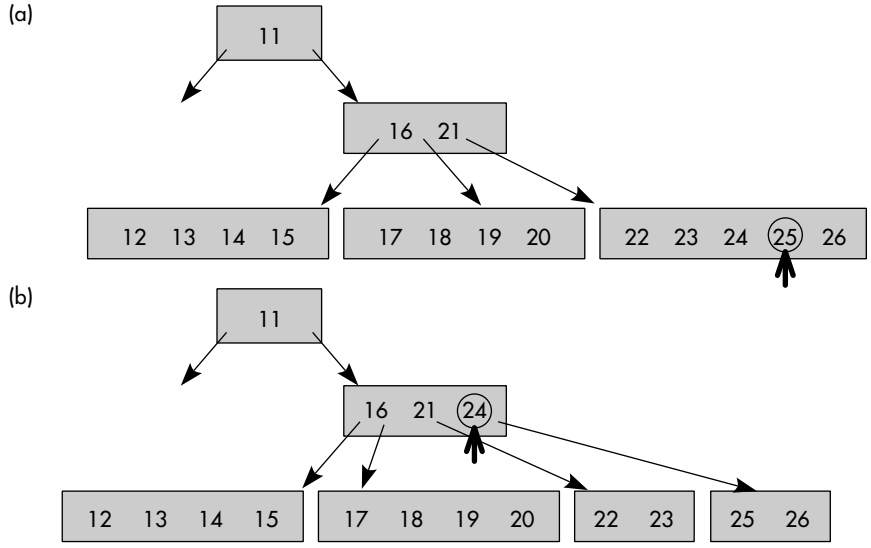


Figure III.18 : Insertion de la clé 25

La suppression d'une clé soulève également des problèmes. Tout d'abord, si l'on supprime une clé non terminale, il faut remonter la clé suivante pour garder le partitionnement. De plus, si un nœud a moins de m clés, il faut le regrouper avec le précédent de même niveau afin de respecter la définition et de conserver entre m et 2m clés dans un nœud non racine.

Une variante de l'arbre B tel que nous l'avons décrit pour réaliser des index est l'arbre B* [Knuth73, Comer79], dans lequel l'algorithme d'insertion essaie de redistribuer les clés dans un nœud voisin avant d'éclater. Ainsi, l'éclatement ne se produit que quand deux nœuds consécutifs sont pleins. Les deux nœuds éclatent alors en trois. Les pages des index de type arbre B* sont donc en général mieux remplies que celles des index de type arbre B.

5.1.5. Arbre B+

L'utilisation des arbres B pour réaliser des fichiers indexés tels que décrits ci-dessus conduit à des traitements séquentiels coûteux. En effet, l'accès selon l'ordre croissant des clés à l'index nécessite de nombreux passages des pages externes aux pages internes. Pour éviter cet inconvénient, il a été proposé de répéter les clés figurant dans les nœuds internes au niveau des nœuds externes. De plus, les pages correspondant aux feuilles sont chaînées entre elles. On obtient alors un **arbre B+.** L'arbre B+ correspondant à l'arbre B de la figure III.17 est représenté figure III.19. Les clés 11, 8 et 21 sont répétées aux niveaux inférieurs. Les pointeurs externes se trouvent seulement au niveau des feuilles.

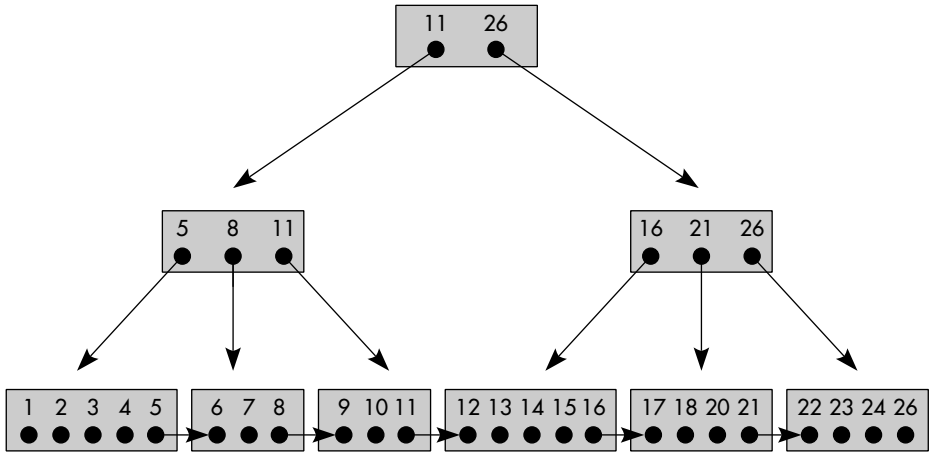


Figure III.19 : Exemple d'index sous forme d'arbre B+

Les arbres B+ peuvent être utilisés pour réaliser des fichiers à index hiérarchisés de deux manières au moins :

– L'arbre B+ peut être utilisé pour implémenter seulement les index. Autrement dit, les articles sont stockés dans un fichier séquentiel classique et l'arbre B+ contient toutes les clés ainsi que les adresses d'articles. Une telle organisation est proche de celle proposée par IBM sur les AS 400. Pour des raisons historiques, cette méthode s'appelle IS3.

– L'arbre B+ peut être utilisé pour implémenter fichiers et index. Dans ce cas, les pointeurs externes sont remplacés par le contenu des articles. Les articles sont donc triés. Seules les clés sont déplacées aux niveaux supérieurs qui constituent un index non dense. Cette méthode correspond à l'organisation séquentielle indexée régulière d'IBM sur MVS connue sous le nom de VSAM, et également à celle de BULL sur DPS7, connue sous le nom de UFAS.

5.2. ORGANISATION INDEXÉE IS3

Cette organisation est voisine de celle développée tout d'abord sur les systèmes de la série 3 d'IBM. Les articles sont rangés en séquentiel dans un fichier dont l'index est dense et organisé sous forme d'un arbre B+.

Notion III.27 : Fichier indexé (*Indexed file*)

Fichier séquentiel non trié, d'index trié dense organisé sous la forme d'un arbre B+.

L'interprétation de la définition que constitue la notion III.27 soulève plusieurs problèmes. Tout d'abord, comment est défini l'ordre de l'arbre B+ qui constitue l'index ? La solution consiste à diviser l'index en pages (une page = 1 à p secteurs). Lors de la première écriture, les pages ne sont pas complètement remplies. Lors d'une insertion, si une page est pleine elle est éclatée en deux pages à demi pleines. La clé médiane est remontée au niveau supérieur.

Un deuxième problème consiste à garder un index dense. En fait, celui-ci est dense au dernier niveau. Autrement dit, toutes les clés d'articles sont gardées au plus bas niveau. Ainsi, quand une page éclate, la clé médiane devient la plus grande clé de la page gauche résultant de l'éclatement. Cette clé est donc dupliquée au niveau supérieur de l'index. La figure III.20 illustre une insertion dans un fichier indexé IS3. L'insertion provoque l'éclatement de l'unique page index et la création d'une page index de niveau supérieur.

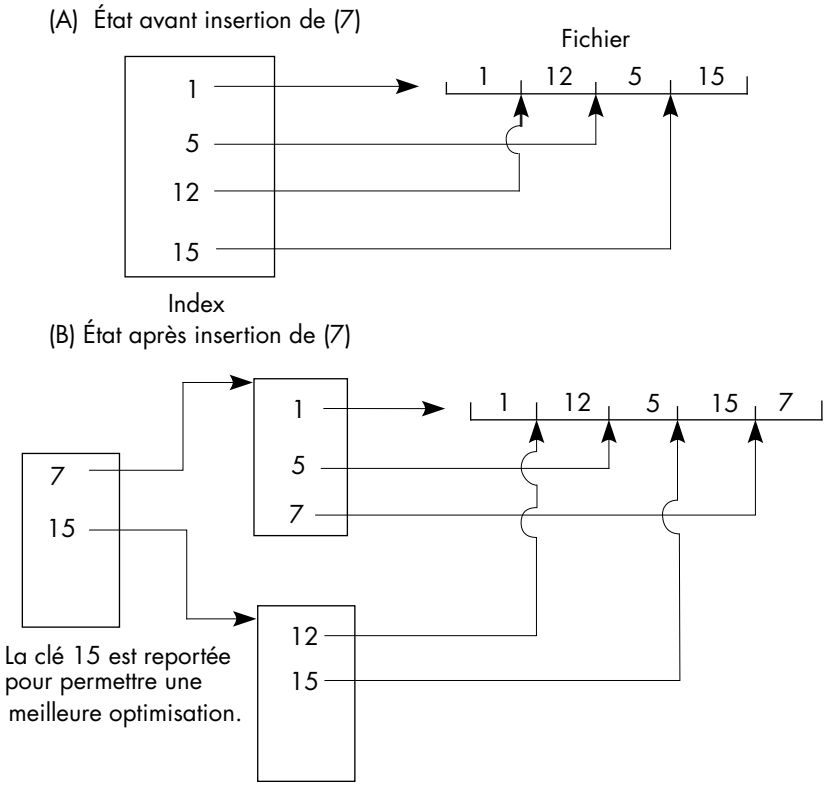


Figure III.20 : Insertion dans un fichier IS3

Un dernier problème est celui du stockage de l'index. Celui-ci peut être stocké en fin de fichier. Il est ainsi possible de lire la page supérieure de l'index en mémoire centrale lors du début d'un travail sur un fichier, puis de la réécrire en fin de travail. Il est aussi possible, avec une telle méthode d'enregistrement des index, de garder les ver-

sions historiques des index à condition que les nouveaux articles écrits le soient après le dernier index enregistré, c'est-à-dire en fin de fichier.

La méthode d'accès et l'organisation associée IS3 présentent plusieurs avantages : l'insertion des articles est simple puisqu'elle s'effectue en séquentiel dans le fichier ; il est possible de garder des versions historiques des index. Les performances de la méthode sont satisfaisantes. Si m est le nombre de clés par page d'index, du fait de l'organisation de l'index en arbre B+, le nombre d'entrées-sorties nécessaires pour lire un article dans un fichier de N articles reste inférieur ou égal à $2 + \log_{(m/2)}((N+1)/2)$. Une écriture nécessite en général deux accès, sauf dans les cas d'éclatement de page index où il faut une lecture et deux écritures de plus par niveau éclaté. En pratique, et à titre d'exemple, un fichier de moins de 10^6 articles ne nécessitera pas plus de trois entrées-sorties en lecture.

Cette méthode présente cependant trois inconvénients sérieux :

– Du fait de la séparation des articles et de l'index, les mouvements de bras des disques sont en général importants.
– La lecture en séquentiel par clé croissante doit se faire par consultation de l'index et est en conséquence très coûteuse.
– L'index est dense et donc de grande taille si aucune technique de compactage de clés n'est mise en œuvre.

5.3. ORGANISATION SÉQUENTIELLE INDEXÉE ISAM

5.3.1. Présentation générale

Il s'agit de l'organisation IBM utilisée dans les systèmes DOS, OS/VS, MVS et connue sous le nom ISAM (*Indexed Sequential Acces Method*) [IBM78]. Le fichier est organisé physiquement selon un découpage en pistes et cylindres. Cette méthode très ancienne reste populaire malgré son manque d'indépendance physique.

Notion III.28 : Fichier séquentiel indexé (*Indexed sequential file*)

Fichier trié d'index trié non dense composé d'une zone primaire et d'une zone de débordement ; une piste saturée déborde dans une extension logique constituée par une liste d'articles en débordement.

Un fichier ISAM comporte trois zones logiques :

– une zone primaire où l'on écrit les articles à la première écriture ;
– une zone de débordement où l'on transfère les articles lors des additions au fichier ;
– une zone d'index où l'on écrit les index.

Ci-dessous nous étudions successivement chacune des zones.

5.3.2. Étude de la zone primaire

La zone primaire se compose de cylindres successifs dont certaines pistes sont réservées pour l'index et les zones de débordement. En zone primaire, les articles sont enregistrés par ordre croissant des clés. Ils peuvent être bloqués ou non. Lors de la première écriture du fichier, les articles doivent être délivrés au système de fichiers par ordre croissant des clés. La figure III.21 illustre un fichier ISAM après une première écriture. Ce fichier est composé de deux cylindres. Chaque cylindre comporte deux pistes de données et une piste réservée pour les index.

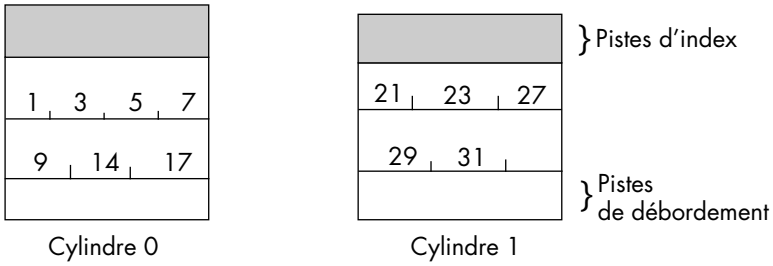


Figure III.21 : Fichier ISAM après une première écriture des articles 1, 3, 5, 7, 9, 14, 17, 21, 23, 27, 29, 31 (dans l'ordre)

5.3.3. Étude de la zone de débordement

Il existe deux types de zones de débordement : la zone de débordement de cylindre composée de quelques pistes sur chaque cylindre et la zone de débordement indépendante, composée des derniers cylindres du fichier (voir figure III.22).

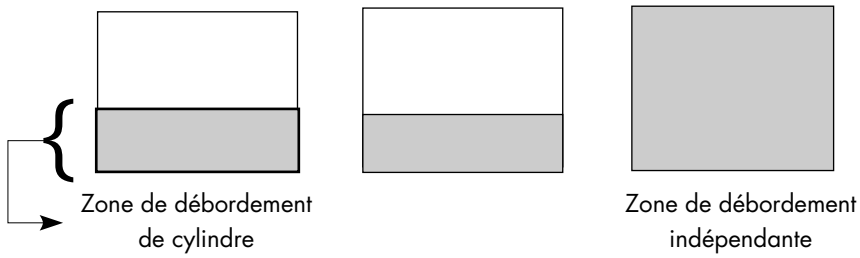


Figure III.22 : Zones de débordement d'un fichier ISAM.

En zone de débordement, les articles ne sont pas bloqués. Ils sont chaînés entre eux afin de reconstituer la piste qui a débordé de manière logique. Quand un article est inséré, on recherche sa séquence dans la piste logique. S'il est placé en zone primaire, les articles suivants sont déplacés et le dernier est écrit en zone de débordement. Les

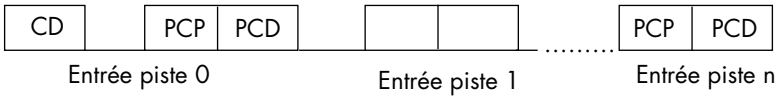
chaînages sont mis à jour. S'il vient en zone de débordement, il est écrit dans cette zone et est inséré en bonne place dans la chaîne des articles en débordement.

La zone de débordement de cylindre est tout d'abord utilisée. Lorsqu'elle est saturée, la zone de débordement indépendante est utilisée. Dans ce cas, comme le chaînage est effectué par ordre croissant des clés, il est possible qu'il parte de la zone de débordement de cylindre, pointe en zone de débordement indépendante, puis revienne en zone de débordement de cylindre, etc. Alors, la méthode devient particulièrement peu performante pour rechercher un article dans une piste ayant débordé, du fait des déplacements de bras.

5.3.4. Étude de la zone index

Il existe obligatoirement deux niveaux d'index et optionnellement trois : les index de pistes, les index de cylindres et le (ou les) index maître(s).

Le premier niveau d'index obligatoire est l'index de piste. Il en existe un par cylindre, en général contenu sur la première piste du cylindre. Chaque entrée correspond à une piste du cylindre et fournit la plus grande clé de la piste logique en zone de débordement ainsi que l'adresse du premier article en zone de débordement s'il existe. La figure III.23 illustre le format de l'index de piste. Chaque piste est décrite par une double entrée comportant, en plus des adresses, la plus grande clé en zone primaire et la plus grande clé en zone de débordement.



CD = Contrôle de débordement précisant l'adresse du dernier article de la zone de débordement.
PCP = Plus grande clé en zone primaire et adresse de la piste.
PCD = Plus grande clé en zone de débordement et adresse du premier article en zone de débordement.

Figure III.23 : Format de l'index de piste

Le deuxième niveau de l'index obligatoire est l'index de cylindre. Il existe un index de cylindre par fichier. Cet index contient une entrée par cylindre comportant la plus grande clé du cylindre ainsi que l'adresse du cylindre. L'index de cylindre est généralement rangé dans une zone particulière, par exemple en début de zone de débordement indépendante. Cet index permet, à partir d'une clé d'article, de sélectionner le cylindre où doit se trouver l'article.

Le troisième niveau d'index est optionnel : c'est l'index maître. Il est utilisé quand l'index de cylindre est trop grand afin d'accélérer les recherches. Il existe une entrée

en index maître par piste de l'index de cylindre. Cette entrée contient l'adresse de la piste et la valeur de la plus grande clé dans la piste.

5.3.5. Vue d'ensemble

La figure III.24 donne une vue d'ensemble d'un fichier ISAM après insertion d'articles, avec seulement deux niveaux d'index. Bien que seulement les chaînages du premier cylindre soient représentés, on notera le grand nombre de chaînages. La première piste logique est constituée des articles a0, a1, a2 et a4 ; a0, a1, a2 sont en zone primaire ; a3 et a4 sont en zone de débordement de cylindre. Les articles a5, a6, a7, a8 et a9 sont rangés dans la deuxième piste logique ; a8 est en débordement de cylindre et a9 en zone de débordement indépendante.

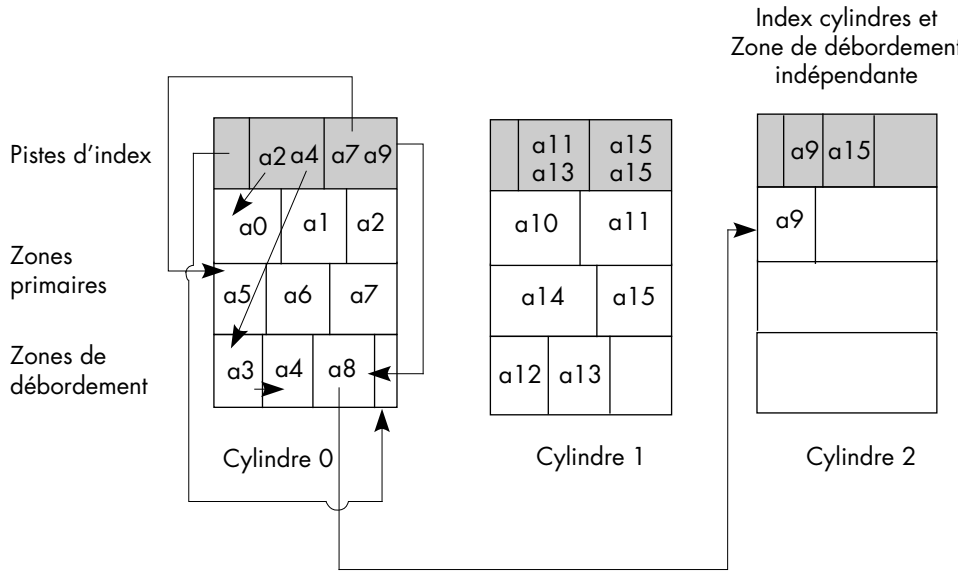


Figure III.24 : Vue d'ensemble d'un fichier ISAM
(Les pointeurs issus du cylindre 1 ne sont pas représentés.)

Les avantages de la méthode sont que le fichier est trié, ce qui facilite l'accès en séquentiel trié, ainsi que les temps d'accès tant que le fichier n'a pas débordé (3 E/S pour lire un article). Les inconvénients sont essentiellement liés aux débordements. La gestion des débordements est complexe et dégrade les performances, de sorte qu'il est nécessaire de réorganiser périodiquement les fichiers ayant débordé. Le fait que la méthode d'accès ne soit pas indépendante des caractéristiques physiques du support (pistes, cylindres…) améliore les performances, mais rend le changement de support difficile. En fait, les performances dépendent des débordements. Si une piste com-

porte des articles en débordement, une lecture nécessitera approximativement 3+[d/2] entrées-sorties alors qu'une écriture demandera 2+[d/2]+4 entrées-sorties, cela afin de trouver la position de l'article, de l'écrire et de mettre à jour les chaînages. Cela peut devenir très coûteux.

5.4. ORGANISATION SÉQUENTIELLE INDEXÉE RÉGULIÈRE VSAM

5.4.1. Présentation générale

Il s'agit de l'organisation IBM utilisée dans les systèmes IBM et connue sous le nom de VSAM (*Virtual Sequential Acces Method*) [IBM87]. À la différence de ISAM, VSAM assure l'indépendance des fichiers au support physique et une réorganisation progressive du fichier, sans gestion de débordement. VSAM est une organisation basée sur les principes des arbres B+.

Notion III.29 : Fichier séquentiel indexé régulier (*Virtual sequential file*)

Fichier trié d'index trié non dense dont l'ensemble fichier plus index est organisé sous forme d'un arbre B+.

Afin d'assurer l'indépendance des fichiers aux supports physiques, des divisions logiques sont introduites. Le fichier est divisé en aires, une aire étant un ensemble de pistes du même cylindre ou de cylindres contigus. Chaque aire est elle-même divisée en intervalles, un intervalle étant généralement composé d'une partie de piste ou de plusieurs pistes consécutives lue(s) en une seule entrée-sortie. L'intervalle correspond à la feuille de l'arbre B+. Lorsqu'un intervalle est saturé, il éclate en deux intervalles ; lorsqu'une aire est saturée, elle éclate aussi en deux aires, si bien que le fichier se réorganise de lui-même par incréments.

Cette organisation résulte d'une étude critique de ISAM. Cette fois, le fichier est plus indépendant de la mémoire secondaire : la piste est remplacée par l'intervalle qui peut être une fraction de piste ou plusieurs pistes, le cylindre est remplacé par la notion d'aire. Les débordements ne perturbent plus l'organisation puisque le mécanisme de débordement est celui de l'arbre B+, c'est-à-dire qu'un intervalle ou une aire saturés éclatent en deux.

5.4.2. Étude de la zone des données

Cette zone qui contient les articles est donc divisée en intervalles et aires, comme sur la figure III.25. Lors de la première écriture, comme avec des fichiers séquentiels indexés ISAM, les articles sont enregistrés triés, c'est-à-dire que le programme doit les délivrer à la méthode d'accès dans l'ordre croissant des clés. Cette fois, les mises à jour sont pré-

vues: de la place libre est laissée dans chaque intervalle et des intervalles libres sont laissés dans chaque aire afin de permettre les insertions de nouveaux articles.

À titre d'exemple, le fichier de la figure III.25 a été créé avec les paramètres suivants :

– pourcentage d'octets libres par intervalle = 25 ;
– pourcentage d'intervalles libres par aire = 25.

Lors de la première écriture, le programme créant le fichier a délivré au système les articles suivants (dans l'ordre) : 1, 5, 7, 9, 15, 20, 22, 27, 30, 31, 33, 37, 40, 43, 47, 51, 53.

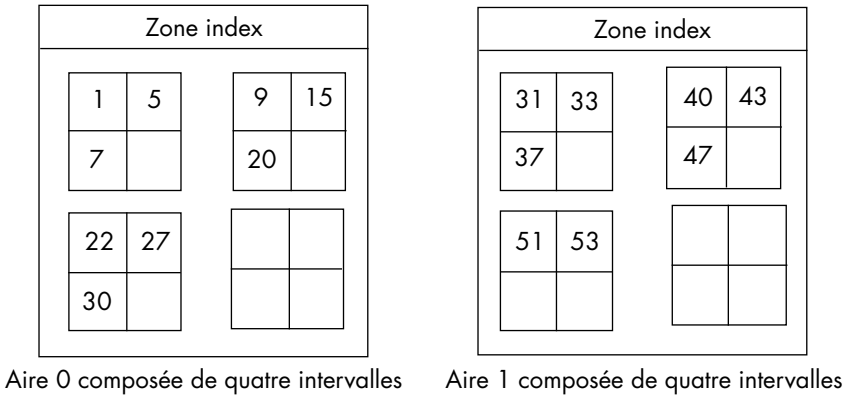


Figure III.25 : Fichier séquentiel indexé régulier après première écriture

L'algorithme d'insertion d'un article consiste à déterminer, grâce aux index, l'intervalle qui doit contenir l'article. Ensuite, deux cas sont possibles :

a) Si l'intervalle n'est pas plein, alors l'article est rangé dans l'intervalle, en bonne place dans l'ordre lexicographique des clés. Par exemple, l'insertion de l'article de clé 10 conduira à la substitution d'intervalle représentée figure III.26.

Figure III.26 : Insertion de l'article de clé 10 dans le fichier précédent

b) Si l'intervalle est plein, alors il éclate en deux intervalles à demi pleins. Deux sous-cas sont encore possibles :

b1) S'il existe un intervalle libre dans l'aire, celui-ci est utilisé afin de stocker l'un des deux intervalles résultant de l'éclatement. Par exemple, l'insertion de

l'article de clé 13 dans le fichier obtenu après insertion de l'article de clé 10 (Fig. III.26) conduit au fichier représenté figure III.27.

b2) S'il n'existe pas d'intervalle libre dans l'aire, on alloue alors une aire vide au fichier et on éclate l'aire saturée en deux à demi pleines ; pour cela, la moitié des intervalles contenant les articles de plus grandes clés sont recopiés dans l'aire nouvelle. Par exemple, l'insertion des articles de clés 11 et 12 dans le fichier de la figure III.27 conduit au fichier représenté figure III.28. Une nouvelle aire a été créée afin de dédoubler la première aire du fichier qui était saturée.

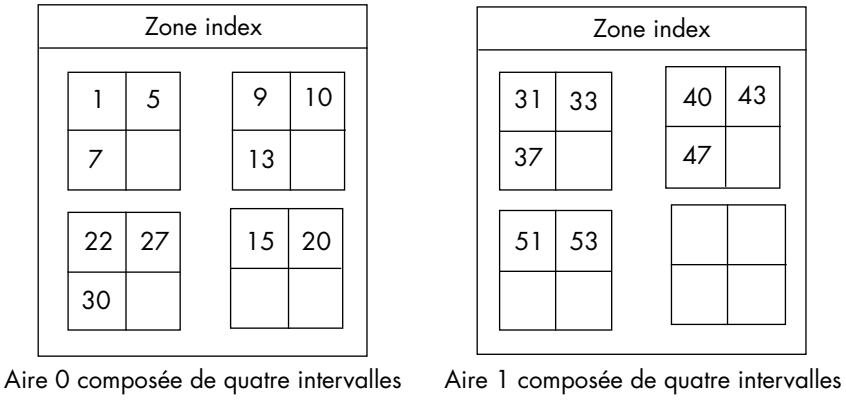


Figure III.27 : Fichier après insertion des articles de clés 10 et 13

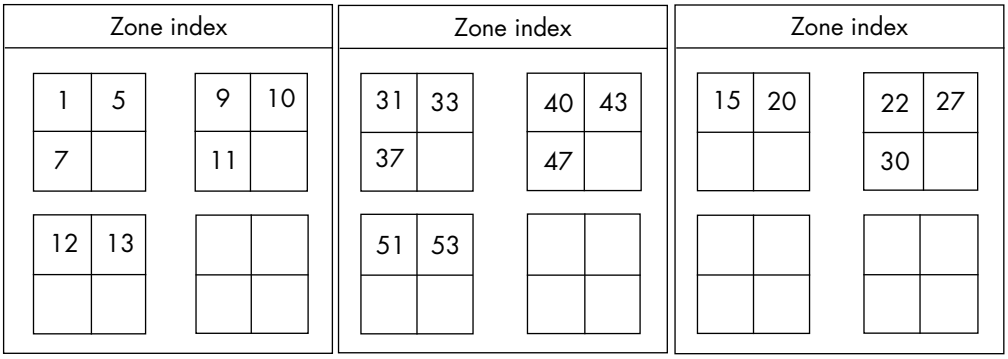


Figure III.28 : Fichier après insertion des articles 11 et 12

5.4.3. Étude de la zone index

Il peut exister deux ou trois niveaux d'index. Le premier est appelé **index d'intervalles**. Il en existe un par aire. Chaque entrée contient la plus grande clé de l'inter-

valle correspondant ainsi que son adresse. Le deuxième niveau est appelé **index d'aires**. Il en existe un par fichier. Chaque entrée contient la plus grande clé de l'aire correspondante ainsi que l'adresse de l'aire. Il est également possible, dans le cas où l'index d'aire est trop grand (par exemple plus d'une piste), de faire gérer au système un index de troisième niveau appelé **index maître** et permettant d'accéder directement à la partie pertinente de l'index d'aire. À titre d'illustration, la figure III.29 représente les index d'intervalles et d'aires du fichier représenté figure III.28. Chaque entrée représente une clé suivie d'une adresse d'intervalle ou d'aire.

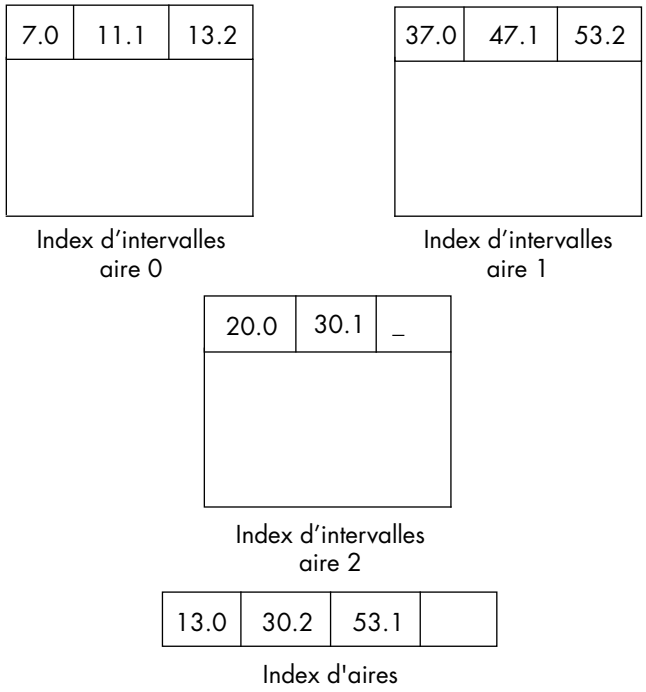


Figure III.29 : Index du fichier représenté figure III.28

5.4.4. Vue d'ensemble

La figure III.30 donne une vue d'ensemble d'un autre fichier VSAM composé de deux aires, avec seulement deux niveaux d'index. Chaque aire possède donc un index d'intervalles. L'index d'aires constitue ici la racine de l'arbre B+. Il indique que 17 est la plus grande clé de l'aire 0, alors que 32 est la plus grande de l'aire 1.

Les avantages de la méthode sont que le fichier est physiquement trié, ce qui facilite les accès en séquentiel trié, ainsi que les temps d'accès à un article : la lecture s'effectue en général en trois entrées-sorties. Les inconvénients sont peu nombreux.

Cependant l'écriture peut parfois être très longue dans le cas d'éclatement d'intervalles ou, pire, dans le cas d'éclatement d'aires. En fait, les écritures ne s'effectuent pas en un temps constant : certaines sont très rapides (4 E/S lorsqu'il n'y a pas d'éclatement), d'autres sont très longues (lorsqu'il y a éclatement d'aire).

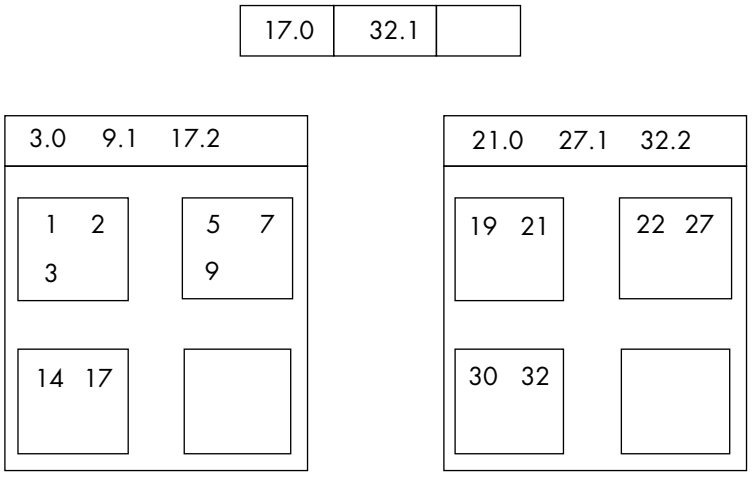


Figure III.30 : Vue d'ensemble d'un fichier séquentiel indexé régulier

6. MÉTHODES D'ACCÈS MULTI-ATTRIBUTS

Les méthodes d'accès étudiées jusque-là permettent de déterminer l'adresse d'un article dans un fichier à partir de la valeur d'une donnée unique de l'article, appelée clé. Nous allons maintenant présenter des méthodes qui permettent d'accéder rapidement à un groupe d'articles à partir des valeurs de plusieurs données, aucune d'entre elles n'étant en général discriminante.

6.1. INDEX SECONDAIRES

Le problème posé est de déterminer l'adresse d'un ou plusieurs articles à partir de la valeur de plusieurs données de cet article, aucune d'entre elles ne permettant en principe de déterminer à elle seule le ou les articles désirés. La solution la plus simple est d'utiliser plusieurs index, chacun d'eux n'étant pas discriminant. On appelle **index secondaire** un index non discriminant.

Notion III.30 : Index secondaire (*Secondary Index*)

Index sur une donnée non discriminante, donnant pour chaque valeur de la donnée la liste des adresses d'articles ayant cette valeur.

Par exemple, un index secondaire d'un fichier décrivant des vins pourra porter sur le champ cru. Les entrées correspondront aux différents crus (Chablis, Medoc, Volnay, etc.) et donneront pour chaque cru la liste des adresses d'articles correspondant à ce cru. Ainsi, en accédant l'entrée Volnay, on trouvera la liste des Volnay.

Un index secondaire est souvent organisé comme un arbre B, bien que d'autres organisations soient possibles. En particulier, un index secondaire peut être un fichier à part, servant d'index au fichier principal. On parle alors de **fichier inverse**, le fichier contenant l'index apparaissant comme une structure de données en quelque sorte renversée par rapport au fichier principal. Un fichier avec un index secondaire (ou un fichier inverse) est en général indexé ou haché sur une clé discriminante (par exemple, le numéro de vin). On parle alors de **clé primaire**. Par opposition, le champ sur lequel l'index secondaire est construit (par exemple, le cru) est appelé **clé secondaire**. Un fichier peut bien sûr avoir plusieurs clés secondaires, par exemple cru et région pour le fichier des vins.

La question qui se pose alors concerne la recherche d'un article dans le fichier. Si plusieurs clés secondaires sont spécifiées (par exemple, cru = "Volnay" et région= "Bourgogne"), on peut avoir intérêt à choisir un index (ici le cru), puis à lire les articles correspondant en vérifiant les autres critères (ici, que la région est bien Bourgogne). Dans certains cas où aucune donnée n'est a priori plus discriminante qu'une autre, on aura au contraire intérêt à lire les différentes entrées d'index sélectionnées, puis à faire l'intersection des listes d'adresses d'articles répondant aux différents critères. Finalement, un accès aux articles répondant à tous les critères (ceux dont l'adresse figure dans l'intersection) sera suffisant.

Un autre problème posé par la gestion d'index secondaires est celui de la mise à jour. Lors d'une mise à jour d'un article du fichier principal, il faut mettre à jour les index secondaires. Bien sûr, si la valeur d'indexation est changée, il faut enlever l'article de l'entrée correspondant à l'ancienne valeur, puis l'insérer dans l'entrée correspondant à la nouvelle valeur. Cette dernière doit être créée si elle n'existe pas. Plus coûteux, avec les méthodes d'accès utilisant l'éclatement de paquets, il faut aller modifier dans les index secondaires les adresses des articles déplacés lors des déplacements d'articles. On peut avoir alors intérêt à garder des références invariantes aux déplacements d'articles dans les index secondaires. Une solution consiste à garder les clés primaires à la place des adresses, mais cela coûte en général un accès via un arbre B.

En résumé, bien que présentant quelques difficultés de gestion, les index secondaires sont très utiles pour accélérer les recherches d'articles à partir de valeurs de données. Ils sont largement utilisés dans les SGBD. S'ils permettent bien d'améliorer les temps

de réponse lors des interrogations, ils pénalisent les mises à jour. Il faut donc indexer les fichiers seulement sur les données souvent documentées lors de l'interrogation.

6.2. HACHAGE MULTI-ATTRIBUT

Avec le hachage, il est possible de développer des méthodes d'accès portant sur de multiples champs (ou attributs) en utilisant plusieurs fonctions de hachage, chacune étant appliquée à un champ. En appliquant N fonctions de hachage à N attributs d'un article, on obtient un vecteur constituant une adresse dans un espace à N dimensions. Chaque coordonnée correspond au résultat d'une fonction de hachage. À partir de cette adresse, plusieurs méthodes sont possibles pour calculer le numéro de paquet où ranger l'article. Une telle approche est la base des méthodes dites de **placement multi-attribut**.

Notion III.31 : Placement multi-attribut (*Multiattribute clustering*)

Méthode d'accès multidimensionnelle dans laquelle l'adresse d'un paquet d'articles est déterminée en appliquant des fonctions de hachage à différents champs d'un article.

6.2.1. Hachage multi-attribut statique

Une méthode simple est le **placement multi-attribut statique**. La méthode consiste à concaténer les résultats des fonctions de hachage dans un ordre prédéfini. La chaîne binaire obtenue est alors directement utilisée comme adresse de paquet. On obtient donc un hachage statique à partir d'une fonction portant sur plusieurs attributs. Si A1, A2 … An sont les attributs de placement, chacun est transformé par application d'une fonction de hachage en bi bits de l'adresse du paquet. Si hi est la fonction de hachage appliquée à l'attribut bi, le numéro de paquet où placer un article est obtenu par a = h1(A1) || h2(A2) || … hn(An), où la double barre verticale représente l'opération de concaténation. Le nombre total de bits du numéro de paquet est B = b1+b2+… bn. Pour retrouver un article à partir de la valeur d'un attribut, disons Ai, le système devra seulement balayer les paquets d'adresse x || hi(Ai) ||y, où x représente toute séquence de b1 +… bi-1 bits et y toute séquence de bi+1+ … bn bits. Ainsi, le nombre de paquets à balayer sera réduit de 2**B à 2**(B-bi). Le nombre optimal de bits à allouer à l'attribut Ai dépend donc de la fréquence d'utilisation de cet attribut pour l'interrogation. Plus cette fréquence sera élevée, plus ce nombre sera important ; l'attribut sera ainsi privilégié aux dépens des autres.

6.2.2. Hachages multi-attributs dynamiques

L'utilisation de techniques de hachage dynamique est aussi possible avec des méthodes de placement multi-attribut. L'approche la plus connue est appelée **fichier**

grille, ou *grid file* en anglais [Nievergelt84]. La méthode peut être vue comme une extension du hachage extensible. Une fonction de hachage est associée à chaque attribut de placement. L'adresse de hachage multidimensionnelle est constituée d'un nombre suffisant de bits choisi en prélevant des bits de chacune des fonctions de hachage de manière circulaire. Cette adresse sert de pointeur dans le répertoire des adresses de paquets. Tout se passe donc comme avec le hachage extensible, mais avec une fonction de hachage multi-attribut mixant les bits des différentes fonctions composantes.

Afin de clarifier la méthode, une représentation par une grille multidimensionnelle du fichier est intéressante. Pour simplifier, supposons qu'il existe seulement deux attributs de placement A1 et A2. Au départ, le fichier est composé d'un seul paquet qui délimite la grille (voir figure III.31.a). Quand le fichier grossit au fur et à mesure des insertions, le paquet sature. Il est alors éclaté en deux paquets B0 et B1 selon la dimension A1 (voir figure III.31.b). Pour ce faire, le premier bit de la fonction de hachage appliquée à A1 (h1(A1)) est utilisé. Quand l'un des paquets, disons B0, est plein, il est à son tour éclaté en deux paquets B00 et B01, cette fois selon l'autre dimension. Le premier bit de la fonction de hachage appliquée à A2 (h2(A2)) est utilisé. Si l'un des paquets B00 ou B01 devient plein, il sera à son tour éclaté, mais cette fois selon la dimension A1. Le processus d'éclatement continue ainsi alternativement selon l'une ou l'autre des dimensions.

Pour retrouver un article dans un paquet à partir de valeurs d'attributs, il faut appliquer les fonctions de hachage et retrouver l'adresse du ou des paquets correspondants dans le répertoire des paquets. Pour cela, la méthode propose un répertoire organisé comme un tableau multidimensionnel. Chaque entrée dans le répertoire correspond à une région du fichier grille. Le répertoire est stocké continûment sur des pages disques, mais est logiquement organisé comme un tableau multidimensionnel. Par exemple, avec un placement bidimensionnel, l'adresse d'un paquet pour la région (i,j) sera déterminé par une transformation d'un tableau bidimensionnel à un répertoire linéaire : l'entrée 2**N*i + j sera accédée, N étant le nombre maximal d'éclatement dans la dimension A1. Ainsi, lors d'un éclatement à un niveau de profondeur supplémentaire d'une dimension, il faut doubler le répertoire selon cette dimension. Cela peut s'effectuer par recopie ou chaînage. En résumé, on aboutit à une gestion de répertoire complexe avec un répertoire qui grossit exponentiellement en fonction de la taille des données.

Pour améliorer la gestion du répertoire et obtenir une croissance linéaire avec la taille des données, plusieurs approches on été proposées. Une des plus efficaces est utilisée par les *arbres de prédicats* [Gardarin83]. Avec cette méthode, l'ensemble des fonctions de hachage est ordonné selon les fréquences décroissantes d'accès (les attributs les plus souvent documentés lors des interrogations d'abord). Pour simplifier, supposons que l'ordre soit de A0 à An. Un arbre de placement permet d'illustrer les éclatement de paquets (voir figure III.32). Au premier niveau, les paquets éclatent progressivement

selon les bits de la fonction de hachage h(A1). Quand tous les bits sont exploités, on utilise la fonction h2(A2), etc. Ainsi, chaque paquet est une feuille de l'arbre de placement caractérisée par une chaîne de bits plus ou moins longue correspondant aux éclatements successifs. Cette chaîne de bits est appelée signature du paquet.

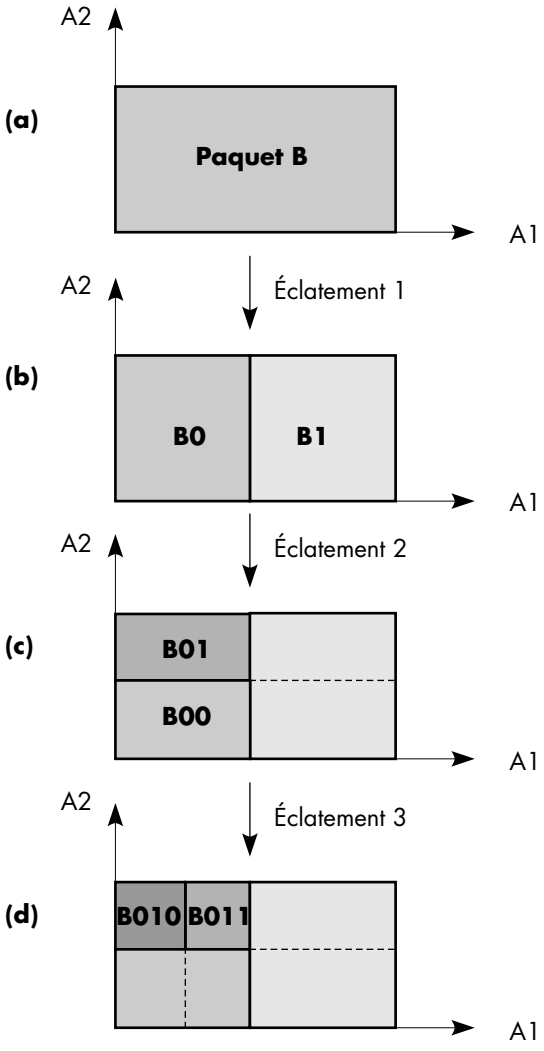
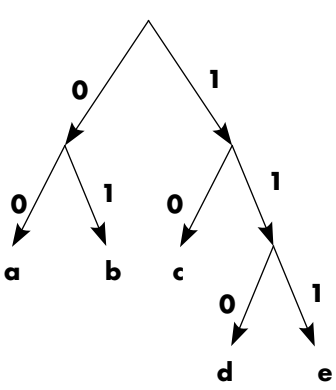


Figure III.31 : Éclatement de paquets dans un fichier grille

Le répertoire contient des doublets <signature de paquet, adresse de paquet>. Le répertoire est organisé lui-même comme un fichier placé par un arbre de prédicats correspondant aux bits de la signature (chaque bit est considéré comme un attribut haché par l'identité). Sa taille est donc linéaire en fonction du nombre de paquets du fichier

de données. Pour rechercher un article à partir d'attributs connus, un profil de signature est élaboré. Les bits correspondant aux attributs connus sont calculés et les autres mis à la valeur inconnue. Ce profil est utilisé pour accéder au répertoire par hachage. Il permet de déterminer les paquets à balayer. Des évaluations et des expérimentations de la méthode ont démontré son efficacité en nombre d'entrées-sorties [Gardarin83]. D'autres méthodes similaires ont été proposées afin de réduire la taille du répertoire et de profiter d'un accès sélectif [Freeston87].



Répertoire

Signatures	Adresses paquets
00	a
10	b
01	c
011	d
111	e

Figure III.32 : Arbre de placement et répertoire associé

6.3. INDEX BITMAP

L'utilisation d'index sous forme d'arbres B sur un champ ayant peu de valeur conduit à des difficultés. En effet, pour de gros fichiers les listes d'adresses relatives d'articles deviennent longues et lourdes à manipuler. Il est bien connu qu'un index sur le sexe d'un fichier de personnes est inutile : il alourdit fortement les mises à jour et n'aide pas en interrogation, car il conduit à accéder directement à la moitié des articles, ce qui est pire qu'un balayage. Les **index bitmap** ont été introduits dans le SGBD Model 204 [O'Neil87] pour faciliter l'indexation sur des attributs à nombre de valeurs limités. Ils ont été plus tard généralisés [O'Neil97, Chan98].

Notion : Index bitmap (*Bitmap index*)

Index sur une donnée A constitué par une matrice de bits indiquant par un bit à 1 en position (i, j) la présence de la j^e valeur possible de l'attribut indexé A pour l'article i du fichier, et son absence sinon.

Un index bitmap suppose l'existence d'une fonction permettant d'associer les N valeurs possibles de l'attribut indexé A de 0 à N-1. Cette fonction peut être l'ordre

alphanumérique des valeurs. C'est alors une bijection. La figure III.33 donne un exemple d'index bitmap pour un fichier de sportifs sur l'attribut `sport`.

Données

Personne	Sport
Perrec	Athlétisme
Cantona	Foot
Virenque	Vélo
Leconte	Tennis
Barthez	Foot
Jalabert	Vélo
Pioline	Tennis

Index bitmap

Athlétisme	Foot	Tennis	Vélo
1	0	0	0
0	1	0	0
0	0	0	1
0	0	1	0
0	1	0	0
0	0	0	1
0	0	1	0

Figure III.33 : Exemple d'index bitmap simple

Les index bitmap sont particulièrement adaptés à la recherche. Par exemple, la recherche des sportifs pratiquant le vélo s'effectue par lecture de l'index, extraction de la colonne Vélo et accès à tous les articles correspondant à un bit à 1. On trouvera ainsi Virenque et Jalabert. L'accès à l'article s'effectue à partir du rang du bit, 3 pour Virenque, 6 pour Jalabert. Pour faciliter cet accès, le fichier doit être organisé en pages avec un nombre fixe p d'articles par page. Si j est le rang du bit trouvé, la page j/p doit être lue et l'article correspondant au j^e cherché dans cette page. Fixer le nombre d'article par page peut être difficile en cas d'articles de longueurs variables. Des pages de débordement peuvent alors être prévues.

Un index bitmap peut aussi permettre d'indexer des attributs possédant des valeurs continues en utilisant une fonction non injective, c'est-à-dire en faisant correspondre à une colonne de la bitmap plusieurs valeurs d'attributs. Une technique classique consiste à associer à chaque colonne une plage de valeurs de l'attribut indexé. Par exemple, la figure III.34 illustre un index bitmap de l'attribut Coût utilisant des plages de valeurs de 0 à 100. Lors d'une recherche sur valeur, on choisira la bonne plage, puis on accédera aux articles correspondant aux bits à 1, et on testera plus exactement le critère sur les données de l'article. Par exemple, la recherche d'un produit de coût 150 conduira à accéder à la 2^e colonne, puis aux articles 1, 3, et 4. Seul l'article 3 sera finalement retenu.

Enfin, un index bitmap peut être utilisé pour indexer des attributs multivalués, tel l'attribut `Produits` figure III.34. Pour chaque article, les bits de chacune des colonnes correspondant aux valeurs de l'attributs sont positionnés à 1.

Les index bitmap sont particulièrement utiles pour les accès sur des attributs multiples ou sur des valeurs multiples d'un même attribut. Il suffit pour cela d'effectuer des unions ou intersections de vecteurs de bits. Par exemple, la sélection des ménagères ayant acheté les produits P1 ou P2 avec un coût total supérieur à 100 s'effectuera par union des colonnes 1 et 2 de la bitmap index produits, puis intersection avec la

colonne 3 unit à la colonne 2 de la bitmap index coût. On obtient alors un vecteur de bits qui désigne les ménagères 1 et 4. Les index bitmap sont donc très utile pour les accès multi-attributs. Ils sont aussi utilisés pour calculer des agrégats, voire extraire des règles des bases de données, comme nous le verrons plus tard. Les limites de cette technique développée récemment sont encore mal cernées.

Fichier de données

Ménagère	Produits	Coût
• 1	{P1, P3, P5}	120
• 2	{P2, P3}	70
• 3	{P4}	150
• 4	{P2, P5}	110
• 5	{P3,P4,P6}	220

Index coût

0-100	100-200	200-300
0	1	0
1	0	0
0	1	0
0	1	0
0	0	1

Index produits

P1	P2	P3	P4	P5	P6
1	0	1	0	1	0
0	1	1	0	0	0
0	0	0	1	0	0
0	1	0	0	1	0
0	0	1	1	0	1

Figure III.34 : Exemple d'index bitmap plus complexes

7. CONCLUSION

Nous venons de passer en revue les fonctions essentielles et les techniques de base des gestionnaires de fichiers. D'autres études peuvent être trouvées, par exemple dans [Korth91] et [Widerhold83]. Il faut savoir qu'un gestionnaire de fichiers est de plus en plus souvent la base d'un système de gestion de bases de données. Pour ce faire, des niveaux de logiciels supérieurs sont généralement implantés pour assurer l'optimisation des recherches, la description centralisée des données des articles de fichiers, des interfaces de gestion de données variées avec les programmes d'application, etc.

La nécessité de pouvoir supporter un Système de Gestion de Bases de Données (SGBD) tend aujourd'hui à rendre le gestionnaire de fichiers de plus en plus élaboré.

Il est par exemple possible de trouver des systèmes gérant plusieurs index secondaires pour un même fichier placé selon un hachage extensible éventuellement multi-attribut. En effet, pour permettre la consultation des données selon des critères variés, les SGBD nécessitent généralement plusieurs chemins d'accès à un même article. Nous reviendrons sur les problèmes de méthodes d'accès et d'organisation de fichiers pour les systèmes de gestion de bases de données dans le chapitre spécifique aux modèles d'accès, puis plus tard pour les données multimédias.

8. BIBLIOGRAPHIE

[Bayer72] Bayer R., McCreight C., « Organization and Maintenance of Large Ordered Index », *Acta Informatica*, vol. 1, n° 3, 1972, p. 173-189.

Un des premiers articles introduisant l'organisation des index par arbres B et démontrant les avantages de cette organisation pour de gros fichiers.

[Chan98] Chan C-Y., Ioannidis Y.E., « Bitmap index Design and evaluation », *ACM SIGMOD Intl. Conf.,* SIGMOD Record V° 27, n° 2, Seattle, USA, 1998.

Cet article étudie la conception d'index bitmap pour traiter des requêtes complexes sur de grandes bases de données. En particulier, les techniques de mapping des valeurs d'attributs sur les indices de colonnes sont prises en compte et différents critères d'optimalité des choix sont étudiés.

[Comer79] Comer D., « The Ubiquitous B-Tree », *Computing Surveys*, vol. 11, n° 2, juin 1979.

Une revue très complète des organisations basées sur les arbres B. Différentes variantes, dont les arbres B+, sont analysées.

[Daley65] Daley R.C., Neuman P.G., « A General Pupose File System for Secondary Storage », *Fall Joint Computer Conference*, 1965, p. 213-229.

Une présentation de la première version du système de fichiers de MULTICS. Ce système introduit pour la première fois une organisation hiérarchique des catalogues de fichiers. L'intérêt de noms hiérarchiques et basés pour désigner un fichier est démontré. Le partage des fichiers par liens est introduit.

[Fagin79] Fagin R., Nivergelt J., Pippengar N., Strong H.R., « Extendible Hahing – A Fast Access Method for Dynamic Files », *ACM TODS*, vol. 4, n° 3, septembre 1979, p. 315-344.

L'article de base sur le hachage extensible. Il propose d'utiliser un répertoire d'adresses de paquets adressé par exploitation progressive des bits du résultat de la fonction de hachage. Les algorithmes de recherche et de mise à jour sont détaillés. Une évaluation démontre les avantages de la méthode aussi bien en temps d'accès qu'en taux de remplissage.

[Freeston87] Freeston M., « The BANG file – A New Kind of Grid File », *ACM SIGMOD*, mai 1987, San Fransisco, ACM Ed., p. 260-269.

Cet article présente une variante du fichier grille, avec laquelle le répertoire reste linéaire en fonction de la taille des données. La technique est voisine de celle des signatures développées dans les arbres de prédicats, mais les attributs ne sont pas ordonnés et le répertoire des signatures dispose d'une organisation particulière. Le BANG file a été implémenté à l'ECRC – le centre de recherche commun BULL, ICL, SIEMENS à Munich – et a servi de base au système de bases de données déductives EKS.

[Gardarin83] Gardarin G., Valduriez P., Viémont Y., « Predicate Tree – An Integrated Approach to Multi-Dimensional Searching », *Rapport de recherche INRIA*, n° 203, avril 1983.

Cet article présente la méthode d'accès multi-attributs des arbres de prédicats. Cette méthode part d'une spécification d'une suite de collections de prédicats disjoints, chaque collection portant sur un attribut. La suite de collection permet d'affecter une signature à chaque tuple constituée par la concaténation des numéros de prédicats satisfaits. La signature est exploitée progressivement bit à bit pour placer les données sur disques. L' idée clé est de permettre un accès multicritères selon une hiérarchie de prédicats. Cette méthode a été mise en œuvre dans le SGBD SABRINA.

[IBM78] IBM Corporation, « Introduction to IBM Direct Access Storage Devices and Organization Methods », Student text, *Manual Form GC20-1649-10*.

Une description des supports de stockage et des méthodes d'accès supportées par IBM en 1978. Ce manuel contient en particulier une description très didactique de la méthode ISAM et une présentation de la méthode VSAM.

[Knott71] Knott G.D., « Expandable Open Addressing Hash Table Storage and Retrieval », *ACM SIGFIDET Workshop on Data Description, Access and Control*, 1971, ACM Ed., p. 186-206.

Le premier article proposant un hachage dynamique, par extension progressive de la fonction de hachage en puissances de 2 successives. L'article s'intéresse seulement au cas de tables en mémoires.

[Knuth73] Knuth D.E., *The Art of Computer Programming*, Addisson-Wesley, 1973.

Le livre bien connu de Knuth. Une partie importante est consacrée aux structures de données, plus particulièrement à l'analyse de fonctions de hachage.

[Korth91] Korth H., Silberschatz A., *Database System Concepts*, Mc Graw-Hill Ed., 2e édition, 1991.

Un des livres les plus complets sur les bases de données. Deux chapitres sont plus particulièrement consacrés aux organisations de fichiers et aux techniques d'indexation.

[Larson78] Larson P., « Dynamic Hashing », *Journal BIT, n° 18*, 1978, p. 184-201.

Un des premiers schémas de hachage dynamique proposé, avec éclatement de paquets quand un taux de remplissage est atteint.

[Larson80] Larson P., « Linear Hashing with Partial Expansions », *6th Very Large Data Bases*, Montreal, octobre 1980, p. 224-232.

Une variante du hachage linéaire ou les avancées du pointeur d'éclatement sont contrôlées par le taux de remplissage du fichier.

[Larson82] Larson P., « Performance Analysis of Linear Hashing with Partial Expansions », *ACM TODS*, vol. 7, n° 4, décembre 1982, p. 566-587.

Cet article résume la méthode présentée dans [Larson80] et démontre ses bonnes performances par une étude analytique.

[Lister84] Lister A.M., *Principes fondamentaux des systèmes d'exploitation*, Éditions Eyrolles, 4e édition, 1984.

Cet excellent livre présente les couches successives constituant un système d'exploitation. Un chapitre est plus particulièrement consacré aux techniques d'allocation mémoire.

[Litwin78] Litwin W., « Virtual Hashing: A Dynamically Changing Hashing », *4th Very Lare Data Bases*, Berlin, septembre 1978, p. 517-523.

Sans doute le premier schéma de hachage dynamique proposé pour des fichiers. Celui-ci est basé sur une table de bits pour marquer les paquets éclatés et sur un algorithme récursif de parcours de cette table pour retrouver la bonne fonction de hachage.

[Litwin80] Litwin W., « Linear Hashing – A New Tool for File and Table Addressing », *6th Very Large Data Bases*, Montreal, octobre 1980, p. 224-232.

L'article introduisant le hachage linéaire. L'idée est de remplacer la table de bits complexe du hachage virtuel par un simple pointeur circulant sur les paquets du fichier. N bits de la fonction de hachage sont utilisés avant le pointeur, N+1 après. À chaque débordement, le paquet pointé par le pointeur est éclaté.

[Lomet83] Lomet D., « A High Performance, Universal Key Associative Access Method », *ACM SIGMOD Intl. Conf.*, San José, 1983.

Une méthode d'accès intermédiaire entre hachage et indexation, avec de très complètes évaluations de performances.

[Lomet89] Lomet D., Salzberg B., « Access Methods for Multiversion Data », *ACM SIGMOD Intl. Conf.,* Portland, 1989.

Cet article discute les techniques d'archivage multiversion sur disques optiques. De tels disques sont inscriptibles une seule fois. Ils ne peuvent être corrigés, mais peuvent bien sûr être lus plusieurs fois (Write Once Read Many times – WORM). Ils nécessitent des organisations spécifiques étudiées dans cet article.

[Nievergelt84] Nievergelt J., Hinterberger H., Sevcik K., « The Grid File: An Adaptable Symetric Multi-Key File Structure », *ACM TODS*, vol. 9, n° 1, mars 1983.

Cet article introduit le fichier grille (Grid File). L'idée est simplement d'étendre le hachage extensible en composant une fonction de hachage multi-attribut, à partir de différentes sous-fonctions portant sur un seul attribut. Une prise en compte successive des bits de chaque fonction garantie la symétrie. Différentes organisations de répertoires d'adresses sont possibles. Un tableau dynamique à n dimensions est proposé.

[O'Neil87] O'Neil P. « Model 204 – Architecture and Performance », *Springer Verlag, LCNS* n° 359, Proc. of 2nd Intl. Workshop on High Performance Transactions Systems, p. 40-59, 1987.

Cet article décrit l'architecture du SGBD Model 204 de Computer Corporation of America. Il introduit pour la première fois les index bitmap.

[O'Neil97] O'Neil P., Quass D., « Improved Query Performance with Variant index », *ACM SIGMOD* Intl. Conf., SIGMOD Record V° 26, n° 2, Tucson, Arizona, USA, 1997.

Cet article présente les index bitmap comme une alternative aux index classiques à listes d'adresses. Il analyse les performances comparées de ces types d'index pour différents algorithmes de recherche.

[Samet89] Samet H., *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1989.

Ce livre présente des méthodes d'accès et organisations fondées sur les quadtrees pour stocker et retrouver des données spatiales, telles que des points, des lignes, des régions, des surfaces et des volumes. Les quadtrees sont des structures de données hiérarchiques basées sur un découpage récursif de l'espace. Par exemple, une image plane en noir et blanc peut être découpée en deux rectangles, chaque rectangle étant à son tour découpé en deux jusqu'à obtention de rectangles d'une seule couleur. Les découpages successifs peuvent être représentés par un arbre dont les feuilles sont à 0 si le rectangle correspondant est blanc, à 1 s'il est noir. Une telle structure est un quadtree. Elle permet de constituer un index ou un arbre de placement utilisable pour une recherche efficace de motifs. Le livre de Samet étudie toutes les variantes des quadtrees.

[Scholl81] Scholl M., « New File Organization Based on Dynamic Hashing », *ACM TODS*, vol. 6, n° 1, mars 1981, p. 194-211.

Une étude des performances des méthodes par hachage dynamique.

[Widerhold83] Widerhold G., *Database Design*, Mc Graw-Hill Ed., New York, 1983.

Ce livre, qui fut l'un des premiers sur les bases de données, consacre une partie importante aux disques magnétiques, aux fichiers et aux méthodes d'accès.

Chapitre IV

BASES DE DONNÉES RÉSEAUX ET HIÉRARCHIQUES

1. INTRODUCTION

Les systèmes étudiés dans ce chapitre sont aujourd'hui appelés **systèmes légataires (*legacy systems*)**, car il nous sont légués par le passé. Ils permettent de modéliser des articles stockés dans des fichiers ainsi que les liens entre ces articles. Ces modèles dérivent avant tout d'une approche système au problème des bases de données qui tend à voir une base de données comme un ensemble de fichiers reliés par des pointeurs. Ils privilégient l'optimisation des entrées-sorties. C'est pourquoi nous les appelons aussi modèles d'accès. À ces modèles sont associés des langages de manipulation de données basés sur le parcours de fichiers et de liens entre fichiers, article par article, appelés langages navigationnels. Ces langages sont très caractéristiques des modèles d'accès.

Nous présentons successivement les deux modèles les plus populaires, à savoir le modèle réseau et le modèle hiérarchique, avec le langage de manipulation spécifique associé à chacun d'eux. Le modèle réseau proposé initialement par le groupe DBTG du comité CODASYL fut et reste utilisé avec diverses variantes possibles par de nombreux systèmes tels que IDS.II (Bull), IDMS (Computer-Associate), EDMS (Xerox),

DMS/1100 (Univac), DBMS (Digital), PHOLAS (Philips) et TOTAL (Cincom). Le modèle hiérarchique étendu est employé par les systèmes anciens que sont IMS (IBM) et Systems 2000 (MRI-Intel), systèmes encore très répandus dans l'industrie. Nous concluons ce chapitre par une présentation des avantages et inconvénients des modèles d'accès.

2. LE MODÈLE RÉSEAU

Dans cette section, nous introduisons les principaux concepts du modèle réseau pour définir les bases de données et le langage associé du Codasyl.

2.1. INTRODUCTION ET NOTATIONS

Le modèle réseau a été proposé par le groupe DBTG du comité CODASYL [Codasyl71]. Des rapports plus récents [Codasyl78, Codasyl81] ont apporté des améliorations notables, en particulier une séparation plus nette entre les concepts de niveau interne et ceux de niveau conceptuel. Malheureusement, ces extensions ne sont que rarement intégrées dans les produits, pour la plupart construits au début des années 70. Le modèle réseau type CODASYL 1971 reste encore aujourd'hui utilisé par plusieurs systèmes, le plus connu en France étant IDS.II de BULL. Notre présentation est basée sur cette implémentation. Vous trouverez une présentation plus générale du modèle dans [Taylor76].

La syntaxe de présentation des clauses utilisées est dérivée de CODASYL, modifiée et étendue comme suit :

– les parenthèses carrées ([]) indiquent des éléments optionnels ;
– les pointillés suivent un élément qui peut être répété plusieurs fois ;
– les crochets groupent comme un seul élément une séquence d'éléments ;
– la barre verticale entre crochets du type {a | b | c | …} signifie que l'une des options a ou b ou c ou… doit être choisie ;
– un exposant plus ($^{+}$) indique que l'élément qui précède peut être répété n fois ($n \geq 1$), chaque occurrence étant séparée de la précédente par une virgule ; l'élément doit donc au moins apparaître une fois : il est obligatoire ;
– un exposant multiplié (*) indique que l'élément qui précède peut être répété n fois ($n \geq 0$), chaque occurrence étant séparée de la précédente par une virgule ; l'élément peut apparaître 0 fois : il est facultatif.

De plus, les mots clés obligatoires sont soulignés. Les paramètres des clauses sont précisés entre crochets triangulaires (<>).

2.2. LA DÉFINITION DES OBJETS

Les objets modélisés dans la base de données sont décrits à l'aide de trois concepts : l'**atome,** le **groupe** et l'**article**. Ces concepts permettent de décrire les données constitutives des objets stockés dans des fichiers.

Notion IV.1 : Atome (*Data Item*)

Plus petite unité de données possédant un nom.

La notion d'atome correspond classiquement au champ d'un article de fichier. Un atome possède un type qui définit ses valeurs possibles et les opérations que l'on peut accomplir sur ces valeurs. Un atome est instancié par une valeur atomique dans la base de données. Par exemple, CRU, DEGRE, AGE et NOM peuvent être des atomes d'une base de données de vins et de buveurs. Le type du CRU sera « chaîne de 8 caractères », alors que celui du degré sera « réel ».

Plusieurs atomes peuvent être groupés consécutivement pour constituer un **groupe** de données.

Notion IV.2 : Groupe (*Data Aggregate*)

Collection d'atomes rangés consécutivement dans la base et portant un nom.

Il existe deux types de groupes : les groupes simples et les groupes répétitifs. Un groupe simple est une suite d'atomes, alors qu'un groupe répétitif est une collection de données qui apparaît plusieurs fois consécutivement. Un groupe répétitif peut être composé d'atomes ou même de groupes répétitifs. Un groupe répétitif composé d'un seul atome est appelé **vecteur**. Par exemple, MILLESIME, composé d'ANNEE et QUALITE, peut être défini comme un groupe simple apparaissant une seule fois dans un vin. Au contraire, ENFANT composé de PRENOM, SEXE et AGE pourra apparaître plusieurs fois dans le descriptif d'une personne : il s'agit d'un groupe répétitif. Une personne pouvant avoir plusieurs prénoms, la donnée PRENOM pourra être un vecteur apparaissant au plus trois fois. Notons que le modèle réseau impose de limiter le nombre de répétitions possibles d'un groupe. Atomes et groupes permettent de constituer les **articles**.

Notion IV.3: Article (*Record*)

Collection d'atomes et de groupes rangés côte à côte dans la base de données, constituant l'unité d'échange entre la base de données et les applications.

Un article peut à la limite ne contenir aucune donnée. Les occurrences d'articles sont rangées dans des fichiers (AREA). Par exemple, un fichier de vins contiendra des articles composés d'atomes NUMERO, CRU et du groupe répétitif MILLESIME, répété au plus cinq fois.

Les articles sont décrits au niveau du type dans le schéma au moyen d'une clause :

```
RECORD NAME IS <nom-d'article>.
```

Par exemple, le type d'article VINS sera introduit par la clause RECORD NAME IS VINS. Suivront ensuite les descriptions détaillées des données de l'article.

Chaque atome ou groupe est défini par un nom précédé d'un niveau éventuel. Les données d'un article sont donc définies au moyen d'un arrangement hiérarchique de groupes dans l'article. Le groupe de niveau 1 est l'article proprement dit. Les données de niveau 2 sont constituées par tous les atomes non agrégés dans un groupe. Les groupes de niveau 3 sont composés de tous les groupes n'étant pas à l'intérieur d'un autre groupe. Viennent ensuite les groupes internes à un groupe (niveau 4), etc. Pour chaque atome, une spécification de type précise le domaine de la donnée (décimal, binaire, caractères). Ainsi, la clause de définition d'un atome est :

```
[<niveau>] <nom-d'atome> TYPE IS <spécification-de-type>
```

alors que celle de définition d'un groupe est simplement :

```
[<niveau>] <nom-de groupe>.
```

Les types de données possibles sont les suivants (version minimale sous IDS.II):

– décimal signé condensé ou non (n1 et n2 précisent respectivement le nombre de chiffre total et celui après la virgule) :

```
SIGNED [ {PACKED | UNPACKED} ] DECIMAL <n1>, [<n2>]
```

– entier binaire long (signe plus 31 bits) ou court (signe plus 15 bits) :

```
SIGNED BINARY {15 | 31}
```

– chaîne de caractères de longueur fixe (n1 caractères) :

```
CHARACTER <n1>.
```

Un atome ou un groupe peuvent être répétés plusieurs fois (cas des vecteurs et des groupes répétitifs). Cela est précisé par la clause OCCURS du langage de description de données (n1 est un entier quelconque) :

```
OCCURS <n1> TIMES.
```

Afin d'illustrer ces différentes clauses, nous décrivons (figure IV.1) un type d'article VINS regroupant les cinq derniers millésimes (caractérisés par une année et un degré) d'un vin défini par un numéro (NV) et un nom de cru (CRU). Nous décrivons également un type d'article PRODUCTEURS composé d'un numéro (NP), un nom (NOM), un prénom (PRENOM) et un groupe simple ADRESSE, composé de RUE, CODE et VILLE.

```
RECORD NAME IS VINS;
02 NV TYPE IS SIGNED PACKED DECIMAL 5;
02 CRU TYPE IS CHARACTER 10;
02 MILLESIME OCCURS 5 TIMES;
03 ANNEE TYPE IS SIGNED UNPACKED DECIMAL 4;
03 DEGRE TYPE IS BINARY 15.

RECORD NAME IS PRODUCTEURS;
02 NP TYPE IS SIGNED PACKED DECIMAL 5;
02 NOM TYPE IS CHARACTER 10;
02 PRENOM TYPE IS CHARACTER 10;
02 ADRESSE
      03 RUE TYPE IS CHARACTER 30;
      03 CODE TYPE IS SIGNED PACKED DECIMAL 5;
      03 VILLE TYPE IS CHARACTER 10;
```

Figure IV.1 : Description des données de deux types d'articles

2.3. LA DÉFINITION DES ASSOCIATIONS

Les possibilités offertes pour modéliser les associations entre objets constituent un des éléments importants d'un modèle de données. Historiquement, le modèle réseau est issu d'une conceptualisation de fichiers reliés par des pointeurs. De ce fait, il offre des possibilités limitées pour représenter les liens entre fichiers. Avec les recommandations du CODASYL, il est seulement possible de définir des associations entre un article appelé propriétaire et n articles membres. Une instance d'association est le plus souvent une liste circulaire d'articles partant d'un article propriétaire et parcourant n articles membres pour revenir au propriétaire. Ces associations, qui sont donc purement hiérarchiques mais qui, utilisées à plusieurs niveaux, peuvent permettre de former aussi bien des arbres, des cycles que des réseaux, sont appelées ici **liens** (en anglais *set*).

Notion IV.4 : Lien (*Set*)

Type d'association orientée entre articles de type T1 vers articles de type T2 dans laquelle une occurrence relie un article propriétaire de type T1 à n articles membres de type T2.

Un type de lien permet donc d'associer un type d'article propriétaire à un type d'article membre ; une occurrence de lien permet d'associer une occurrence d'article propriétaire à n occurrences d'articles membres. Généralement, les articles membres seront d'un type unique, mais la notion de lien peut théoriquement être étendue afin de supporter des membres de différents types.

Un lien se représente au niveau des types à l'aide d'un **diagramme de Bachman**. Il s'agit d'un graphe composé de deux sommets et d'un arc. Les sommets, représentés par des rectangles, correspondent aux types d'articles ; l'arc représente le lien de 1 vers n [Bachman69]. L'arc est orienté du type propriétaire vers le type membre. Il est valué par le nom du type de lien qu'il représente et chaque sommet par le nom du type d'article associé.

Par exemple, les types d'articles VINS et PRODUCTEURS décrits ci-dessus seront naturellement reliés par le type de lien RECOLTE, allant de PRODUCTEURS vers VINS. Une occurrence reliera un producteur à tous les vins qu'il produit. La figure IV.2 schématise le diagramme de Bachman correspondant à ce lien.

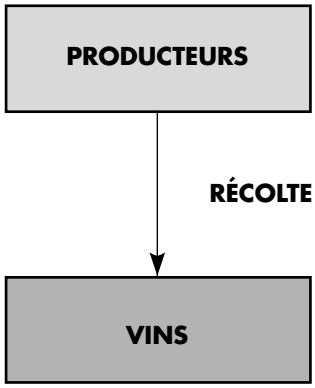


Figure IV.2 : Exemple de diagramme de Bachman

Deux limitations sont importantes : (i) un type d'article ne peut être à la fois propriétaire et membre d'un même lien ; (ii) une occurrence d'article ne peut appartenir à plusieurs occurrences du même lien. Par exemple, deux producteurs ne pourront partager la récolte d'un même vin, comme le montre la figure IV.3. Par contre, un type d'article peut être maître de plusieurs liens. Il peut aussi être membre de plusieurs liens. Deux types d'articles peuvent aussi être liés par des types de liens différents.

Afin d'illustrer les concepts introduits, la figure IV.4 présente le graphe des types d'une base de données vinicole composée des articles :

– VINS décrits ci-dessus,
– BUVEURS composés des atomes numéro de buveur, nom et prénom,
– ABUS décrivant pour chaque vin la quantité bue par un buveur et la date à laquelle celle-ci a été bue,
– PRODUCTEURS définissant pour chaque vin le nom et la région du producteur,
– COMMANDES spécifiant les commandes de vins passées par les buveurs aux producteurs.

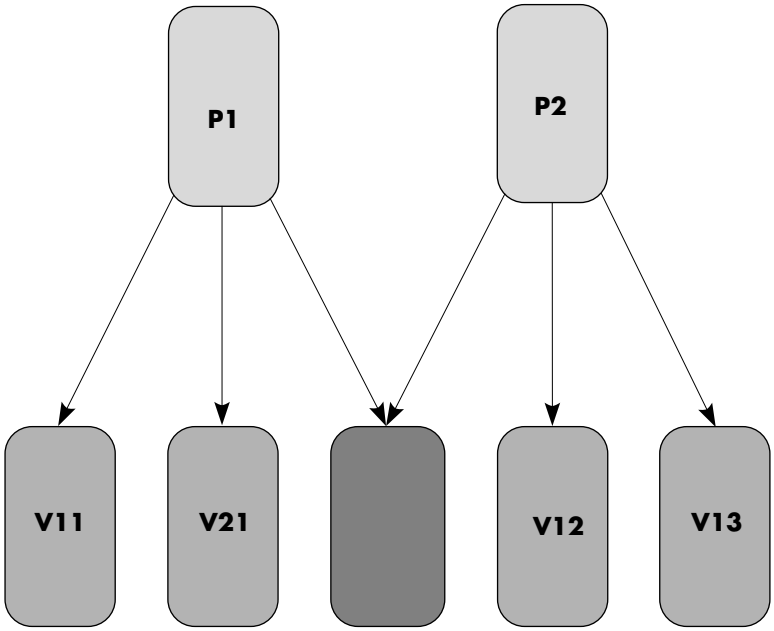


Figure IV.3 : Exemple de partage d'occurrences de membres interdit

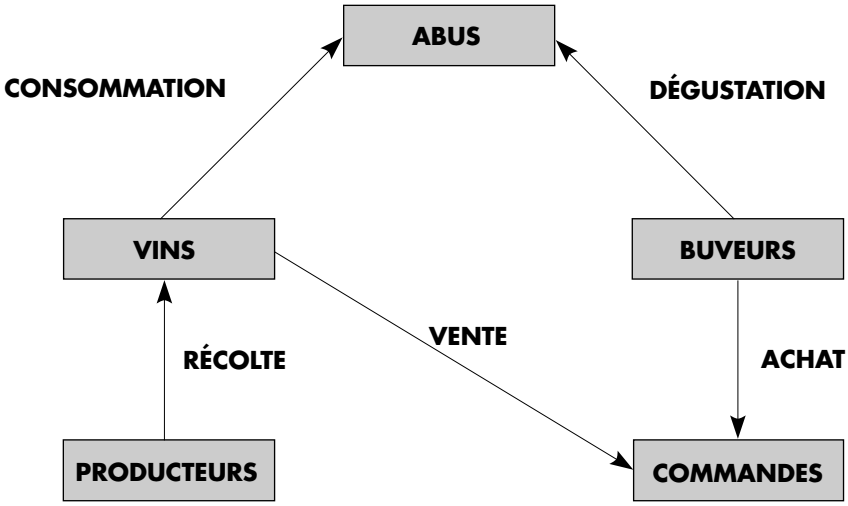


Figure IV.4 : Exemple de graphe des types

Les liens considérés sont :

– RECOLTER qui associe un producteur aux vins récoltés,

– RECEVOIR qui associe un vin aux commandes correspondantes,

– PASSER qui associe un buveur à ses commandes,
– BOIRE qui associe un buveur à une quantité de vin bue,
– PROVOQUER qui associe un vin à toutes les quantités bues par les différents buveurs.

Afin de mieux faire comprendre la notion de lien, il est possible de représenter une base de données réseau par un graphe au niveau des occurrences. Dans ce cas, les articles d'une occurrence de lien sont reliés par un cycle. Par exemple, la figure IV.5 représente des occurrences des articles VINS, ABUS et BUVEURS par des carrés arrondis, et des occurrences des liens BOIRE et PROVOQUER par des chaînes d'occurrences d'articles.

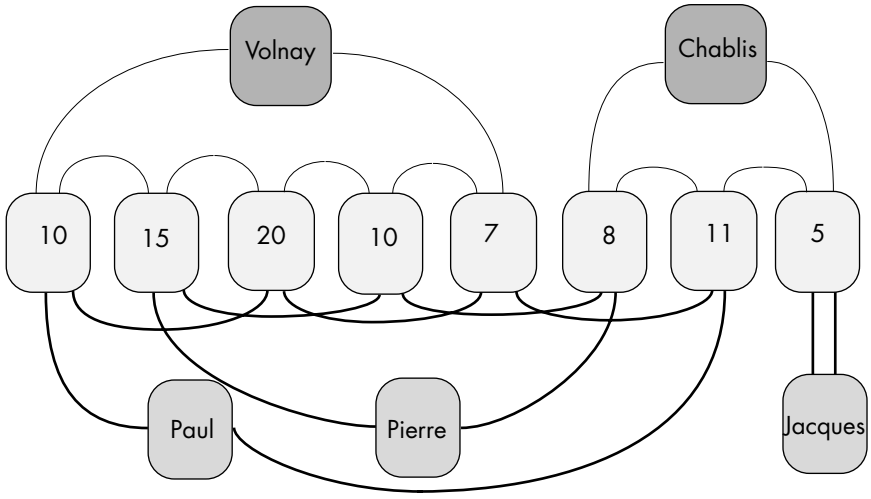


Figure IV.5 : Exemple de graphe des occurrences

En CODASYL, la clause de définition d'un lien utilisée au niveau de la description d'un schéma, qui se situe bien évidemment au niveau des types, est structurée comme suit :

```
SET NAME IS <nom-de-lien>
OWNER IS <nom-d'article>
[MEMBER IS <nom-d'article>] *
```

Nous verrons ci-dessous les détails des sous-clauses OWNER et MEMBER. Plusieurs types de membres peuvent être spécifiés par répétition de la clause MEMBER.

CODASYL autorise la définition de liens singuliers avec une seule occurrence. Pour cela, il suffit d'utiliser la définition de propriétaire :

```
OWNER IS SYSTEM
```

Tous les articles membres sont alors chaînés entre eux dans une seule occurrence de lien (une seule liste dont la tête de liste est gérée par le système). Cela permet de rechercher un article parmi les membres comme dans une occurrence de lien normale, et aussi de chaîner des articles singuliers.

2.4. L'ORDONNANCEMENT DES ARTICLES DANS LES LIENS

Les articles dans les occurrences de lien sont ordonnés. L'ordre est maintenu lors des insertions et les articles sont présentés dans l'ordre d'insertion aux programmes d'application. Lors de l'insertion d'un article dont le type est membre d'un lien, il faut tout d'abord choisir l'occurrence de lien dans laquelle l'article doit être placé. Les méthodes possibles pour ce choix sont présentées ci-dessous. Ensuite, l'article doit être inséré dans la suite ordonnée des articles composants les membres de l'occurrence de lien. Les choix possibles de la position de l'article sont les suivants :

– en début (FIRST) ou en fin (LAST) de la suite des membres,
– juste avant (PRIOR) ou juste après (NEXT) le dernier article de la suite des membres auquel a accédé le programme effectuant l'insertion,
– par ordre de tri (SORTED) croissant ou décroissant d'une donnée des articles membres ; dans ce cas, la donnée doit être définie comme une clé (KEY) dans la clause membre ; les cas de doubles doivent être prévus ; de même, si le lien comporte plusieurs types d'articles, l'ordre des types doit être précisé.

La figure IV.6 représente ces diverses possibilités pour l'insertion dans une occurrence du lien BOIRE.

Ces possibilités doivent être définies au niveau du schéma à l'aide de la clause ORDER. Celle-ci est placée dans la définition du type de lien (clause SET) juste après la définition du type du propriétaire (sous-clause OWNER). Sa syntaxe simplifiée est la suivante :

```
ORDER IS [ PERMANENT ] INSERTION IS
{FIRST | LAST | PRIOR | NEXT | SORTED <spécification de tri>}.
```

Une spécification de tri est définie comme suit :

```
<spécification de tri> ::=
[ RECORD-TYPE SEQUENCE IS <nom-d'article>+ ]
BY DEFINED KEYS
[ DUPLICATES ARE {FIRST | LAST | NOT ALLOWED} ]
```

L'option PERMANENT (obligatoire pour les liens triés) précise qu'un programme d'application ne peut modifier l'ordre d'un lien. Ainsi, tout changement effectué restera local au programme et ne perturbera pas l'ordre des articles déjà enregistrés dans la base.

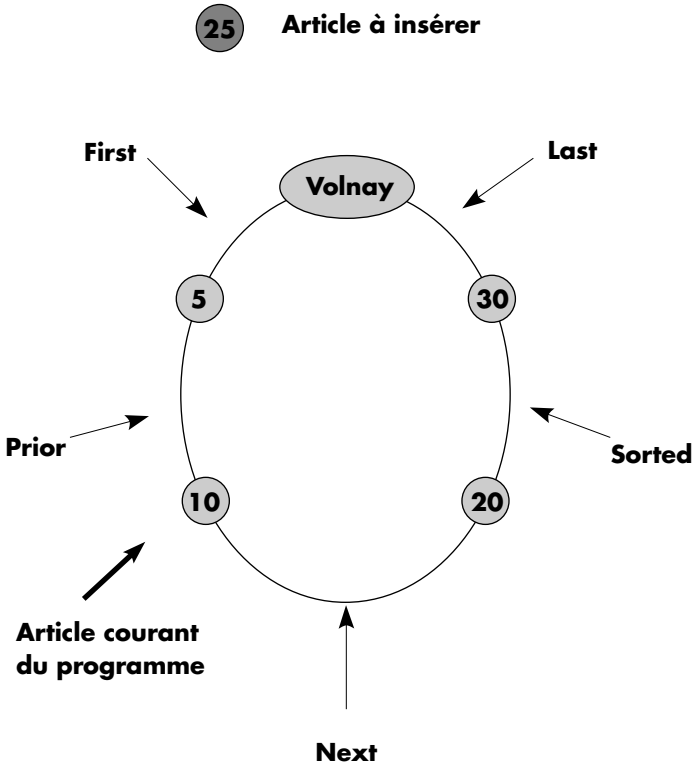


Figure IV.6 : Possibilités d'insertion dans une occurrence de lien

Si un ordre de clé est précisé à l'intérieur de chaque type d'article (option SORTED), il faut tout d'abord préciser l'ordre entre les types d'articles dans le cas où le lien est hétérogène (plusieurs types d'articles membres). C'est l'objet de la clause RECORD TYPE SEQUENCE IS. Ensuite, la clause DUPLICATES permet de spécifier si les doubles sont possibles pour les clés définies au niveau de la clause MEMBER (clause KEY ci-dessous) et, si oui, comment ils sont traités. L'option FIRST place les doubles avant ceux déjà existants ; l'option LAST les place après.

Pour chacun des types d'articles membres, si l'option SORTED BY DEFINED KEYS est retenue, il faut préciser la clé de tri ; celle-ci est spécifiée par la clause optionnelle KEY. La clé peut être utilisée pour un tri ascendant ou descendant. Elle est définie par la clause suivante :

```
KEY IS {ASCENDING | DESCENDING} <nom-de-donnée>
```

2.5. LA SÉLECTION D'OCCURRENCE D'UN TYPE DE LIEN

Il existe autant d'occurrences d'un lien que d'articles propriétaires. Lors d'une insertion ou d'une recherche, il y a donc nécessité de sélectionner l'occurrence choisie. Cela est effectué par parcours d'un chemin dans le graphe des occurrences de liens depuis un article point d'entrée, jusqu'à l'article propriétaire de l'occurrence de lien cherchée.

La sélection peut être manuelle ou automatique. Si elle est manuelle, elle est effectuée par le programme qui peut par exemple parcourir les liens de proche en proche pour trouver le bon propriétaire. Si elle est automatique, le mode de sélection doit être précisé par la clause SET SELECTION.

Nous décrivons maintenant la clause de sélection d'un chemin pour atteindre le propriétaire d'une occurrence de lien, utilisée notamment afin d'insérer automatiquement un article membre dans une occurrence de lien. Cette clause est une des plus complexe du langage de description CODASYL. Elle aboutit au lien dont une occurrence doit être sélectionnée (nom-de-lien1) en partant d'un point d'entrée (propriétaire du nom-de-lien2). La descente s'effectue de lien en lien en précisant comment doit être sélectionné le propriétaire à chaque niveau. La syntaxe simplifiée de la clause de sélection est la suivante :

```
SET SELECTION [FOR <nom-de-lien1>] IS
THRU <nom-de-lien2> OWNER IDENTIFIED BY
        {  APPLICATION
        |  DATA-BASE-KEY [EQUAL TO <nom-de-paramètre1>]
        |  CALC KEY [EQUAL TO <nom-de-paramètre2>]}

[ THEN THRU<nom-de-lien3>WHERE OWNER IDENTIFIED BY
{<nom-de-donnée3> [EQUAL TO <nom-de-paramètre3>]}+ ] …
```

La première partie de la clause (THRU) permet de spécifier le mode de sélection du point d'entrée. Deux approches sont possibles :

– par application, ce qui signifie que l'article propriétaire a dû être, préalablement à la recherche ou à l'insertion, repéré par le programme d'application, comme nous le verrons lors de l'étude du langage de manipulation ;
– par clé (DATA-BASE-KEY ou CALC KEY) fournie par le programme en paramètre ; le premier type de clé est une adresse base de données d'article (adresse relative dans le fichier, gérée par le système) et le second une clé de hachage (CALC KEY) servant au placement de l'article dans le fichier par application d'une fonction de hachage, comme nous le verrons ci-dessous.

La deuxième partie (THEN THRU) est optionnelle dans le cas où l'on désire parcourir un chemin de longueur supérieure à un dans le graphe des liens. Elle permet, de manière récursive lorsqu'elle est répétée, de sélectionner une occurrence de lien par

recherche du propriétaire dans les membres de l'occurrence précédemment sélectionnée. Cette recherche s'effectue par balayage de l'occurrence de lien de niveau supérieur jusqu'au premier article ayant pour valeur de la donnée nom-de-donnée3 la valeur contenue dans nom-de-paramètre3 ; cette donnée (nom-de-donnée3) doit être discriminante dans l'occurrence de lien (pas de doubles autorisés). Ainsi, il est possible de faire choisir l'occurrence de lien dans lequel un article est automatiquement inséré par le SGBD, à partir d'un point d'entrée préalablement sélectionné et de paramètres fournis par le programme.

2.6. LES OPTIONS D'INSERTION DANS UN LIEN

Lors de l'insertion d'un article dans la base, celui-ci peut, pour chacun des liens dont son type d'article est déclaré membre :

– être inséré automatiquement dans la bonne occurrence du lien sélectionné par le système en accord avec la SET SELECTION (option INSERTION IS AUTOMATIC) ;
– ne pas être inséré automatiquement par le système; dans ce cas (option INSERTION IS MANUAL), le programme devra, s'il le désire, faire l'insertion par une commande spéciale du langage de manipulation que nous étudierons plus loin..

De plus, une contrainte d'intégrité spécifique peut être précisée : l'association entre deux types d'articles reliés par un lien est déclarée « obligatoire » (option MANDATORY) ou « facultative » (option OPTIONAL). Dans le premier cas, tout article du type membre d'un lien sera forcément membre d'une occurrence de ce lien, alors qu'il ne le sera pas forcément dans le second. L'option MANDATORY correspond à un lien fort (qui doit forcément exister) alors que l'option OPTIONAL correspond à un lien faible (qui peut exister).

Ces options d'insertion sont précisées pour chaque type de membre, après la clause MEMBER, par la clause :

```
INSERTION IS {AUTOMATIC | MANUAL}
RETENTION IS {MANDATORY | OPTIONAL}
```

2.7. LE PLACEMENT DES ARTICLES

Une base de données CODASYL est placée dans un ensemble de fichiers appelé AREA (ou REALM dans la nouvelle version). Ces fichiers sont soit des fichiers relatifs (adressage par numéro de page et par numéro d'octet dans la page), soit des fichiers aléatoires (adresse relative calculée par une fonction de hachage). Chaque article est repéré dans la base par une **clé base de données** (*database key*) qui lui est

affectée à sa création et permet de l'identifier jusqu'à sa disparition. Un SGBD CODASYL gère donc l'identité d'objets par des clés base de données.

Notion IV.5 : Clé Base de Données (*Database key*)

Adresse invariante associée à un article lors de sa création et permettant de l'identifier sans ambiguïté.

Bien que le comité DBTG CODASYL n'ait pas spécifié le format des clés base de données, la plupart des systèmes réseaux attribuent une place fixe aux articles, si bien qu'une clé base de données peut être un numéro de fichier, suivi d'un numéro de page et d'un déplacement dans la page permettant de retrouver l'en-tête de l'article. Certains systèmes gèrent en fin de page un index des en-têtes d'articles contenus dans la page, si bien que le déplacement dans la page est simplement le numéro de l'en-tête en fin de page.

Le **placement** d'un article consiste à calculer son adresse dans la base, ainsi que sa clé base de données qui en découle en général directement. Le mode de placement est défini dans le schéma pour chaque type d'article selon plusieurs procédés possibles.

Notion IV.6 : Placement CODASYL (*CODASYL location mode*)

Méthode de calcul de l'adresse d'un article et d'attribution de la clé base de données lors de la première insertion.

De manière surprenante, la clé base de données peut être fournie directement par l'utilisateur comme un paramètre du programme. Ce mode, appelé **placement direct** (mot clé DIRECT), est en général réservé aux programmeurs système. Le mode le plus facilement utilisable est le placement aléatoire classique : une clé, pas forcément discriminante, est précisée et le système calcule l'adresse de l'article à l'aide d'une procédure de hachage. Ce mode de placement, appelé **placement calculé**, est spécifié par les mots clés CALC USING.

Un mode plus complexe, mais permettant en général de bonnes optimisations, est le mode par lien, spécifié par le mot clé VIA. Il possède deux variantes selon que l'article est placé dans le même fichier que son propriétaire via le lien considéré, ou dans un fichier différent. Dans le premier cas, l'article est placé **à proximité** du propriétaire, alors que dans le second il est placé dans un autre fichier que celui du propriétaire **par homothétie**.

Le placement à proximité consiste à placer l'article aussi près que possible du propriétaire du lien retenu pour le placement, dans la même page ou dans la première page voisine ayant de la place disponible.

Le placement par homothétie consiste à calculer l'adresse de la page dans laquelle on place l'article (dénotée Adresse Article AA) à partir de l'adresse de la page du propriétaire (dénotée Adresse Propriétaire AP) par la formule AA = AP * TA/TP, où TA

désigne la taille du fichier contenant l'article et TP la taille du fichier contenant le propriétaire. L'avantage de cette méthode est qu'en général tous les articles ayant le même propriétaire via le lien considéré sont placés dans la même page (ou des pages voisines). Ainsi la méthode par homothétie réduit le temps de parcours des membres d'une occurrence de lien sans accroître le temps de parcours du type d'article membre, alors que la méthode par proximité réduit le temps de parcours des occurrences de lien mais en général accroît celui du type d'article membre.

La figure IV.7 illustre les différents types de placements couramment utilisés : placement calculé (CALC), à proximité et par homothétie. Le choix entre ces différents modes est laissé à l'administrateur système, qui doit chercher à optimiser les performances des accès les plus fréquents aux données.

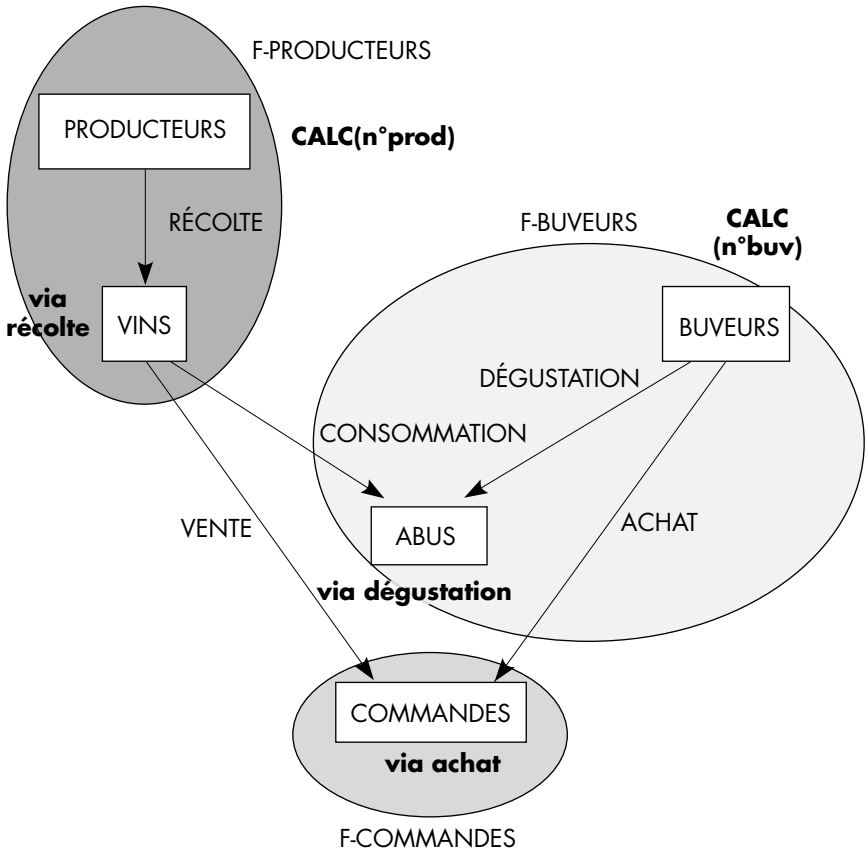


Figure IV.7 : Différents types de placements possibles

Les diverses possibilités de placement illustrées ci-dessus sont définies au niveau du schéma par la clause LOCATION MODE. Il existe en fait quatre modes possibles, le

mode SYSTEM spécifiant qu'un algorithme définit par le système est choisi. On retrouve les modes décrits ci-dessus, DIRECT par adresse calculée par le programme, CALC par clé et VIA par lien. Le fichier choisi peut être celui du propriétaire ou un autre dans le cas VIA, ce qui différencie le placement à proximité et le placement par homothétie. Dans tous les cas, le fichier choisi est précisé par la clause WITHIN. La syntaxe de la clause de placement est la suivante :

```
LOCATION MODE IS
{       SYSTEM
|       DIRECT <NOM-DE-PARAMETRE>
|       CALC USING <NOM-DE-DONNÉE>
        [ DUPLICATES ARE [NOT] ALLOWED ]
|       VIA <NOM-DE-LIEN> SET}

WITHIN {<NOM-DE-FICHIER> | AREA OF OWNER}
```

2.8. EXEMPLE DE SCHÉMA

Afin d'illustrer le langage de définition de schéma CODASYL, la figure IV.8 propose un schéma possible pour la base de données dont le graphe des types a été présenté figure IV.4. La base a été placée dans trois fichiers (F-BUVEURS, F-PRODUCTEURS et F-COMMANDES). Les articles ABUS et VINS sont placés à proximité des propriétaires, respectivement BUVEURS et PRODUCTEURS, eux-mêmes placés par hachage. Les articles COMMANDES sont placés dans le fichier F-COMMANDES par homothétie avec le fichier F-BUVEURS. Des choix sont faits pour les ordres d'insertion, les contraintes de rétention et les modes de sélection des liens. Ces choix devront être respectés lors de l'écriture des programmes de manipulation.

```
SCHEMA NAME IS VINICOLE ;
   AREA NAME IS F-BUVEURS ;
   AREA NAME IS F-PRODUCTEURS ;
   AREA NAME IS F-COMMANDES ;
RECORD NAME IS BUVEURS ;
   LOCATION MODE IS CALC USING NB DUPLICATES
   NOT ALLOWED WITHIN F-BUVEURS ;
   02 NB TYPE IS SIGNED PACKED DECIMAL 5 ;
   02 NOM TYPE IS CHARACTER 10 ;
   02 PRENOM TYPE IS CHARACTER 10 ;
RECORD NAME IS ABUS ;
   LOCATION MODE IS VIA DEGUSTATION
   WITHIN AREA OF OWNER ;
   02 QUANTITE TYPE IS SIGNED BINARY 15 ;
RECORD NAME IS PRODUCTEURS ;
   LOCATION MODE IS CALC USING NOM DUPLICATES
   ALLOWED WITHIN F-PRODUCTEURS ;
```

```
   02 NOM TYPE IS CHARACTER 10 ;
   02 REGION TYPE IS CHARACTER 8 ;
RECORD NAME IS VINS ;
   LOCATION MODE IS VIA RECOLTE WITHIN AREA OF OWNER ;
   02 NV TYPE IS SIGNED PACKED DECIMAL 5 ;
   02 CRU TYPE IS CHARACTER 10 ;
   02 MILLESIME OCCURS 5 TIMES ;
   03 ANNEE TYPE IS SIGNED UNPACKED DECIMAL 4 ;
   03 DEGRE TYPE IS SIGNED BINARY 15 ;
RECORD NAME IS COMMANDES ;
   LOCATION MODE IS VIA ACHAT WITHIN F-COMMANDES ;
   02 DATE TYPE IS CHARACTER 8 ;
   02 QUANTITE TYPE IS SIGNED BINARY 15 ;
SET NAME IS DEGUSTATION ;
   OWNER IS BUVEURS ;
   ORDER IS PERMANENT INSERTION IS LAST ;
   MEMBER IS ABUS ;
   INSERTION IS AUTOMATIC RETENTION IS MANDATORY ;
   SET SELECTION FOR DEGUSTATION IS
   THRU DEGUSTATION OWNER IDENTIFIED BY CALC KEY ;
SET NAME IS CONSOMMATION ;
   OWNER IS VINS ;
   ORDER IS PERMANENT INSERTION IS NEXT ;
   MEMBER IS ABUS ;
   INSERTION IS AUTOMATIC RETENTION IS MANDATORY ;
   SET SELECTION FOR CONSOMMATION IS
   THRU CONSOMMATION OWNER IDENTIFIED
   BY APPLICATIONS ;
SET NAME IS RECOLTE ;
   OWNER IS PRODUCTEURS ;
   ORDER IS PERMANENT INSERTION IS SORTED
   BY DEFINED KEYS DUPLICATES ARE FIRST ;
   MEMBER IS VINS ;
   INSERTION IS MANUAL RETENTION IS OPTIONAL ;
   KEY IS ASCENDING CRU ;
   SET SELECTION IS
   THRU RECOLTE OWNER IDENTIFIED BY CALC KEY ;
SET NAME IS VENTE ;
   OWNER IS VINS ;
   ORDER IS PERMANENT INSERTION IS SORTED
   BY DEFINED KEYS DUPLICATES ARE NOT ALLOWED ;
   MEMBER IS COMMANDES ;
   INSERTION IS AUTOMATIC RETENTION IS MANDATORY ;
   KEY IS DESCENDING DATE DUPLICATES NOT ALLOWED ;
   SET SELECTION IS
   THRU RECOLTE OWNER IDENTIFIED BY CALC KEY
   THEN THRU VENTE WHERE OWNER IDENTIFIED BY NV ;
SET NAME IS ACHAT ;
   OWNER IS BUVEURS ;
```

```
   ORDER IS PERMANENT INSERTION IS LAST ;
   MEMBER IS COMMANDES ;
   INSERTION IS AUTOMATIC RETENTION IS MANDATORY ;
   SET SELECTION IS
   THRU ACHAT OWNER IDENTIFIED BY APPLICATION ;
END-SCHEMA.
```

Figure IV.8 : Exemple de schéma CODASYL

3. LE LANGAGE DE MANIPULATION COBOL-CODASYL

Le langage de manipulation de données du CODASYL est fortement lié à COBOL, bien que généralisé et utilisable depuis d'autres langages de 3e génération tel Fortran.

3.1. SOUS-SCHÉMA COBOL

Un schéma externe en CODASYL est appelé **sous-schéma**. La notion de sous-schéma est en général plus restrictive que celle de schéma externe. Il s'agit d'un sous-ensemble exact du schéma sans restructuration possible. Au niveau d'un sous-schéma, l'administrateur ne peut qu'omettre des types d'articles, des fichiers (area), des liens et des données déclarés dans le schéma. Certains systèmes permettent cependant de définir des données virtuelles, non stockées dans la base, mais calculées par le SGBD à partir de données de la base.

Notion IV.7 : Sous-schéma (*Sub-schema*)

Sous-ensemble du schéma vu par un programme d'application, spécifiant la vision externe de la base par le programme.

Outre la possibilité de ne définir qu'une partie des données, des articles, des liens et des fichiers, d'autres variations importantes peuvent exister entre le schéma et un sous-schéma. En particulier :

– l'ordre des atomes dans un article peut être changé,
– les caractéristiques (types) d'un atome peuvent être changées,
– les clauses SET SELECTION peuvent être redéfinies,
– les noms de types d'articles, d'attributs et de liens peuvent être changés.

Un sous-schéma COBOL se compose en principe de trois divisions : une division de titre (title division) qui nomme le sous-schéma et le relie au schéma, une division de transformation (mapping division) qui permet de définir les synonymes et une division des structures (structure division) où sont définis les fichiers (appelés REALM en COBOL car le mot AREA est utilisé à d'autres fins), les articles et les liens vus par le programme. Afin d'illustrer le concept de sous-schéma, la figure IV.9 propose un sous-schéma de la base vinicole que pourrait définir un administrateur pour des programmes s'intéressant seulement aux articles buveurs, commandes et abus (à l'exception du numéro de buveur).

```
TITLE DIVISION
SS CLIENT WITHIN SCHEMA VINICOLE
MAPPING DIVISION
ALIAS SECTION
AD SET BOIT IS DEGUSTATION
AD SET ACHETE IS ACHAT
STRUCTURE DIVISION
REALM SECTION
RD F-BUVEURS
RD F-COMMANDES
RECORD SECTION
01 BUVEURS
   02 NV PICTURE IS 999
   02 NOM PICTURE IS X(10)
   02 PRENOM PICTURE IS X(10)
01 ABUS
   02 QUANTITE PICTURE IS 999
01 COMMANDES
   02 QUANTITE PICTURE IS 999
   02 DATE PICTURE IS X(8)
SET SECTION
SD BOIT
SD ACHETE
```

Figure IV.9 : Exemple de sous-schéma COBOL

3.2. LA NAVIGATION CODASYL

Une fois qu'un schéma et des sous-schémas ont été définis, des programmes d'applications peuvent être écrits. Ceux-ci invoquent le système de gestion de bases de données à l'aide des verbes du langage de manipulation qui sont inclus dans un programme COBOL, ou dans un autre langage (C, FORTRAN, PL1 sont en particulier supportés). Les verbes de manipulation peuvent être classés en quatre types :

– la recherche d'articles (FIND),

– les échanges d'articles (GET, STORE),
– la mise à jour (ERASE, CONNECT, DISCONNECT, MODIFY),
– le contrôle des fichiers (READY, FINISH).

Les échanges d'articles entre le SGBD et un programme d'application s'effectuent par une zone de travail tampon appelée USER WORKING AREA, dans laquelle les articles fournis par le SGBD sont chargés (GET), et dans laquelle ceux fournis par le programme sont placés avant d'être rangés dans la base (STORE). Chaque atome ou article décrit dans le sous-schéma a une place fixée dans la zone de travail, affectée lors du chargement du programme. Chaque objet peut être référencé par le programme en utilisant son nom défini dans le sous-schéma.

La recherche d'articles ne provoque pas d'échange de données entre le programme et le SGBD. Seuls des pointeurs associés au programme et à des collections d'articles, appelés **curseurs**, sont déplacés par les ordres de recherche (FIND).

Notion IV.8 : Curseur (*Currency*)

Pointeur courant contenant la clé base de données du dernier article manipulé d'une collection d'articles, et permettant au programme de se déplacer dans la base.

Il existe plusieurs curseurs associés à un programme en exécution, chacun permettant de mémoriser l'adresse du dernier article manipulé dans une collection d'articles (en général, un type) :
– un curseur indique l'article courant du programme, c'est-à-dire en principe le dernier article lu, écrit ou simplement cherché par le programme ;
– un curseur indique l'article courant de chaque type d'article référencé ;
– un curseur indique l'article courant de chaque type de lien référencé ;
– un curseur indique enfin l'article courant de chaque type de fichier référencé.

Le nombre de curseurs associés à un programme se comptabilise en fonction des types du sous-schéma. Il est obtenu par la formule : (1 + Nombre de types d'articles + Nombre de type de liens + Nombres de fichiers). Les curseurs sont déplacés grâce aux divers types de FIND que nous allons étudier. Seul l'article pointé par le curseur du programme peut être lu par le programme en zone de travail. Ainsi, un programme se déplace dans la base en déplaçant son curseur : on dit qu'il navigue [Bachman73].

3.3. LA RECHERCHE D'ARTICLES

La recherche d'articles consiste donc à déplacer les curseurs dans la base. Elle s'effectue à l'aide de l'instruction FIND. L'exécution d'une telle instruction déplace toujours le curseur du programme, mais il est possible d'empêcher sélectivement le déplace-

ment des autres curseurs. S'il n'a pas été spécifié qu'un curseur ne doit être déplacé, celui-ci est repositionné dès qu'un article de la collection à laquelle il est associé (type d'articles, liens, fichiers) est manipulé (lu, écrit ou traversé).

Le format général de l'instruction de recherche est :

```
FIND <EXPRESSION-DE-SÉLECTION> RETAINING CURRENCY FOR
{       MULTIPLE
|       REALM
|       RECORD
|       SETS
|       <NOM DE LIEN>+}
```

L'expression de sélection spécifie le critère de recherche. La rétention du déplacement de tous les curseurs, à l'exception de celui du programme, s'effectue par l'option RETAINING CURRENCY FOR MULTIPLE. Les autres choix de rétention de curseurs peuvent être combinés. Ci-dessous, nous examinons les différents types de FIND résultant des diverses expressions de sélections possibles.

3.3.1. La recherche sur clé

Comme l'indique le paragraphe IV.2, un article possède toujours une clé base de données (DATA BASE KEY) et peut posséder une clé de hachage (avec ou sans double) s'il a été placé en mode calculé. On obtient ainsi trois possibilités de recherche sur clés dont voici les syntaxes :

– Recherche connaissant la clé base de données (adresse invariante)

```
FIND <nom-d'article> DBKEY IS <nom-de-paramètre>
```

– Recherche connaissant la clé calculée unique (clé de hachage)

```
FIND ANY <nom-d'article>
```

Dans ce cas, la valeur de la clé de recherche doit préalablement être chargée par le programme en zone de travail.

– Recherche connaissant la clé calculée avec doubles

```
FIND DUPLICATE <nom-d'article>
```

Plusieurs FIND DUPLICATE permettront de retrouver les doubles successivement.

3.3.2. La recherche dans un fichier

Cette recherche est avant tout séquentielle ou en accès direct. Il est ainsi possible de sélectionner le premier article (option FIRST), le dernier (option LAST), l'article suivant (option NEXT) ou précédent (option PRIOR) l'article courant, ainsi que le i^e article d'un fichier. Le numéro de l'article à sélectionner peut être spécifié par un paramètre du programme. Le format de l'instruction de recherche dans un fichier est :

```
FIND     {FIRST | LAST | NEXT | PRIOR | <i> | <paramètre>}
<nom-d'article> WITHIN <nom-de-fichier>
```

3.3.3. La recherche dans une occurrence de lien

De même que dans un fichier, il est possible de sélectionner le premier article, le dernier, l'article suivant ou précédent l'article courant et le i^{e} article de l'occurrence courante d'un lien. Le format de l'instruction est identique à celui de la recherche en fichier :

```
FIND     {FIRST | LAST | NEXT | PRIOR | <i> | <paramètre>}
<nom-d'article> WITHIN <nom-de-lien>
```

Il est également possible de rechercher un article à partir d'une valeur de donnée dans l'occurrence courante d'un lien. La donnée dont la valeur est utilisée pour ce type de recherche associative dans une occurrence de lien est citée en argument du mot clé USING. Cette valeur doit bien sûr être préalablement chargée en zone de travail. Selon que l'on recherche la première (ou l'unique) occurrence d'article ou la suivante, on emploie :

```
FIND <nom-d'article> WITHIN <nom-de-lien>
USING <nom-de-donnée>+

FIND DULICATE WITHIN <nom-de-lien>
USING <nom-de-donnée>+
```

Un lien peut être parcouru depuis un article membre vers le propriétaire, donc en sens inverse de l'arc qui le représente dans le diagramme de Bachman. Ainsi, il est possible de sélectionner le propriétaire de l'occurrence courante d'un lien par la commande :

```
FIND OWNER WITHIN <nom-de-lien>
```

Une telle instruction permet en général de passer depuis un membre d'un lien à un propriétaire, donc de remonter les arcs du graphe des types d'une base de données CODASYL.

Il est aussi possible de tester des conditions concernant l'appartenance de l'article courant d'un programme à un lien. La condition est vraie si l'article courant du programme est propriétaire (option OWNER), membre (option MEMBER) ou plus généralement participant (option TENANT) à l'intérieur d'une occurrence du lien cité. La forme de la commande est :

```
IF [NOT] <nom-de-lien> {OWNER | MEMBER | TENANT}
EXECUTE <instructions>
```

Il est enfin possible de tester si une occurrence de lien sélectionnée par un article propriétaire ne possède aucun membre en utilisant la commande :

```
IF <nom-de-lien> IS [NOT] EMPTY EXECUTE <instructions>
```

3.3.4. Le positionnement du curseur de programme

À chaque recherche, le curseur du programme est positionné. Par suite, les FIND provoquent des changements successifs de la position du curseur du programme : on dit que le programme navigue dans la base CODASYL [Bachman73]. Après des navigations successives, il peut être utile de revenir se positionner sur l'article courant d'un type d'article d'un fichier ou d'un lien. Cela peut être fait à l'aide de la commande :

```
FIND CURRENT [<nom-d'article>]
[ WITHIN {<nom-de-fichier> | <nom-de-lien>} ]
```

Cette commande a donc pour effet de forcer le curseur du programme à celui du nom-d'article spécifié, ou à celui du fichier ou du lien spécifié, éventuellement selon un type d'article précisé.

3.4. LES ÉCHANGES D'ARTICLES

L'article pointé par le curseur d'un programme peut être amené en zone de travail du programme pour traitement. Seules certaines données de cet article peuvent être lues. Cette opération s'effectue à l'aide de la commande :

```
GET {[<type-d'article>] | {<nom-de-donnée>} *}
```

Les arguments peuvent être soit le type d'article qui doit alors être le même que celui de l'article pointé par le curseur du programme, soit une liste des données du type de l'article courant que l'on souhaite amener en zone de travail. Si aucun nom de donnée n'est précisé, toutes les données sont lues.

Le rangement d'un article dans la base s'effectue, après chargement de l'article en zone de travail, par exécution de la commande STORE. En principe, tous les curseurs sont modifiés par la commande STORE : tous pointent après exécution sur le nouvel article stocké. Il est possible de retenir le déplacement de certains curseurs en utilisant l'option RETAINING, de manière identique à FIND. Le format de la commande STORE est le suivant :

```
STORE <nom d'article> RETAINING CURRENCY FOR
                          {   MULTIPLE
                          |   REALM
                          |   RECORD
                          |   SETS
                          |   <nom de lien>+}
```

3.5. LES MISES À JOUR D'ARTICLES

Elles incluent à la fois les suppressions et les modifications de données. Alors que la suppression est simple, la modification soulève des problèmes au niveau des liens qui peuvent ou non être modifiés.

3.5.1. Suppression d'articles

Avant de supprimer un article, il faut bien sûr le rechercher. Il devient alors le courant du programme. La suppression de l'article courant d'un programme s'effectue par la commande :

```
ERASE [ALL] [<nom-d'article>].
```

Si cet article est propriétaire d'occurrences de lien, tous ses descendants sont également supprimés seulement dans le cas où le mot clé ALL est précisé. Une suppression est ainsi cascadée à tous les articles dépendants, et cela de proche en proche. Le nom d'article permet optionnellement au système de vérifier le type de l'article courant à détruire.

3.5.2. Modification d'articles

La modification de l'article courant d'un programme s'effectue en principe après avoir retrouvé l'article à modifier comme le courant du programme. Les données à modifier doivent être placées en zone article dans le programme (zone de travail). Seules certaines données sont modifiées, en principe celles spécifiées par la commande ou toutes celles dont la valeur en zone article est différente de celle dans la base. Les liens spécifiés sont changés en accord avec les clauses de sélection de liens (SET SELECTION) précisées dans le schéma. Le format de la commande est le suivant :

```
MODIFY {[<nom-d'article>] | [<nom-de-donnée>]}
[ INCLUDING {ALL | ONLY <nom-de-lien>+} MEMBERSHIP ]
```

Les données à modifier peuvent être précisées ou non ; dans le dernier cas, le système modifie toutes celles qui sont changées dans l'article en zone de travail. On doit aussi préciser les modifications à apporter aux liens dont l'article est membre ; plusieurs options sont possibles. Il est possible de demander la réévaluation, en accord avec la clause du schéma SET SELECTION de toutes (INCLUDING ALL) ou de certaines (INCLUDING liste de noms de liens) appartenances à des occurrences de liens. Il est aussi possible de ne modifier aucune donnée, mais seulement les appartenances aux liens (option MODIFY… ONLY…).

3.5.3. Insertion et suppression dans une occurrence de lien

L'insertion d'un article dans une occurrence de lien peut être effectuée par le SGBD ou par le programme, selon le choix lors de la définition du schéma. Deux commandes permettent d'insérer et d'enlever l'article courant d'un programme d'une occurrence de lien : CONNECT et DISCONNECT. Leur utilisation nécessite que le mode d'insertion dans le lien ait été défini comme manuel (MANUAL) ou alors que l'appartenance au lien soit optionnelle (OPTIONAL). La syntaxe de ces commandes est la suivante :

```
CONNNECT [<nom-d'article>] TO <nom-de-lien>.
DISCONNECT [<nom-d'article>] FROM <nom-de-lien>.
```

Le nom d'article permet de vérifier le type de l'article courant.

3.6. LE CONTRÔLE DES FICHIERS

Avant de travailler sur un fichier, un programme doit l'ouvrir en spécifiant le mode de partage souhaité. Le fichier peut être ouvert en mode exclusif (seul l'utilisateur peut alors accéder) ou en mode protégé (les accès sont alors contrôlés au niveau des articles), à fin de recherche (RETRIEVAL) ou de mise à jour (UPDATE). Cette ouverture s'effectue à l'aide de la commande :

```
READY {<nom-de-fichier> [USAGE-MODE IS
{EXCLUSIVE | PROTECTED} {RETRIEVAL | UPDATE}]} +
```

En fin de travail sur un fichier, il est nécessaire de le fermer à l'aide de la commande :

```
FINISH {<nom-de-fichier>} +
```

3.7. QUELQUES EXEMPLES DE TRANSACTION

Voici différents types d'accès réalisés par des transactions simples. Ces transactions utilisent le sous-schéma CLIENT de la figure VI.9. La signification des transactions apparaît en légende de la figure correspondante.

```
READY F-BUVEURS USAGE-MODE PROTECTED
RETRIEVAL
MOVE '100' TO NB IN BUVEURS.
FIND ANY BUVEURS.
PRINT NOM, PRENOM IN BUVEURS.
FINISH F_BUVEURS.
```

Figure IV.10 : Accès sur clé calculée unique

```
READY F-BUVEURS, F-COMMANDES.
FIND FIRST BUVEURS WITHIN F-BUVEURS.
PERFORM UNTIL "FIN-DE-FICHIER"
   GET BUVEURS.
   PRINT NOM, PRENOM IN BUVEURS.
   FIND FIRST COMMANDE WITHIN ACHETE.
   PERFORM UNTIL "FIN-D'OCCURENCE-D'ENSEMBLE"
      GET COMMANDES.
      PRINT QUANTITE, DATE IN COMMANDE.
      FIND NEXT COMMANDE WITHIN ACHETE.
   END-PERFORM.
   FIND NEXT BUVEURS WITHIN F-BUVEURS.
END-PERFORM.
FINISH.
```

Figure IV.11 : Accès séquentiel à un fichier et à des occurrences de lien

```
READY F-BUVEURS, F-COMMANDES.
MOVE '100' TO NB IN BUVEURS.
FIND ANY BUVEURS.
MOVE '10-02-81' TO DATE IN COMMANDES.
FIND COMMANDE WITHIN ACHETE USING DATE.
PERFORM UNTIL "PLUS-DE-COMMANDE"
   GET COMMANDE.
   PRINT QUANTITE, DATE IN COMMANDE.
   FIND DUPLICATE WITHIN ACHETE USING DATE.
END-PERFORM.
FINISH.
```

Figure IV.12 : Accès associatif dans une occurrence de lien

```
READY F-BUVEURS, F-COMMANDES.
FIND FIRST ABUS WITHIN F-BUVEURS.
PERFORM UNTIL « FIN-DE-FICHIER »
   GET ABUS.
   IF QUANTITE IN ABUS > 100
      FIND OWNER WITHIN BOIT.
      IF ACHETE IS NOT EMPTY
         GET BUVEURS.
         PRINT NOM, PRENOM IN BUVEURS.
      END-IF.
   END-IF.
   FIND NEXT ABUS WITHIN F-BUVEURS.
FINISH.
```

Figure IV.13 : Accès au propriétaire et utilisation d'une condition

```
READY F-BUVEURS, F-COMMANDES.
MOVE '100' TO NB IN BUVEURS.
FIND ANY BUVEURS.
ERASE ALL BUVEURS.
FINISH.
```

Figure IV.14 : Suppression d'un article et de ses descendants

```
READY F-BUVEURS.
MOVE '7' TO I.
MOVE '200' TO NB IN BUVEURS.
FIND ANY BUVEURS.
FIND I ABUS IN BOIT.
GET ABUS.
ADD 10 TO QUANTITE IN ABUS.
MODIFY ABUS.
FINISH.
```

Figure IV.15 : Modification d'un article

4. LE MODÈLE HIÉRARCHIQUE

Les bases de données modélisent des informations du monde réel. Puisque le monde réel nous apparaît souvent au travers de hiérarchies, il est normal qu'un des modèles les plus répandus soit le modèle hiérarchique. Quelques systèmes sont encore basés sur ce modèle [MRI74, IBM78]. Vous trouverez une présentation détaillée du modèle hiérarchique dans [Tsichritzis76]. Nous résumons ici les aspects essentiels.

4.1. LES CONCEPTS DU MODÈLE

Le modèle hiérarchique peut être vu comme un cas particulier du modèle réseau, l'ensemble des liens entre types d'articles devant former des graphes hiérarchiques. Cependant, les articles ne peuvent avoir de données répétitives. De plus, le modèle introduit en général des concepts spécifiques afin de modéliser les objets. Un **champ** est exactement l'équivalent d'un atome du modèle réseau. Il s'agit d'une donnée élémentaire à valeur simple.

Notion IV.9 : Champ (*Field*)

Plus petite unité de données possédant un nom.

Un **segment** correspond à un article sans groupe répétitif. Une occurrence de segment est en général de taille fixe. Les champs d'un segment sont tous au même niveau, si bien qu'une occurrence de segment est parfois qualifiée d'article plat. Un segment peut avoir un champ discriminant appelé **clé**. La valeur de la clé permet alors de déterminer une occurrence unique dans le segment.

Notion IV.10 : Segment (*Segment*)

Collection de champs rangés consécutivement dans la base, portant un nom et dont une occurrence constitue l'unité d'échange entre la base de données et les applications.

Les segments sont reliés par des liens de 1 vers N qui à un segment père font correspondre N segments fils (N est un entier positif quelconque), aussi bien au niveau des types qu'au niveau des occurrences. Ainsi, un type de segment possède en général plusieurs types de segments descendants. De même, une occurrence de segment est reliée à plusieurs occurrences de chacun des segments descendants. Pour représenter une descendance de segments reliés par des associations père-fils, on construit des **arbres de segments**.

Notion IV.11 : Arbre de segments (*Segment tree*)

Collection de segments reliés par des associations père-fils, organisée sous la forme d'une hiérarchie.

Les types de segments sont donc organisés en arbre. Un type racine possède N1 types fils, qui à leur tour possèdent chacun N2 types fils, et ainsi de suite jusqu'aux segments feuilles. Il est aussi possible de considérer une occurrence d'un arbre de segments : une occurrence d'un segment racine possède plusieurs occurrences de segments fils. Parmi celles-ci, certaines sont d'un premier type, d'autres d'un second, etc. À leur tour, les occurrences des fils peuvent avoir des fils, et ainsi de suite jusqu'aux occurrences des feuilles.

Finalement, une **base de données hiérarchique** peut être considérée comme un ensemble d'arbres, encore appelé forêt, dont les nœuds sont des segments. La définition s'applique aussi bien au niveau des types qu'au niveau des occurrences. Les arbres sont en principe indépendants. Chaque arbre possède un segment racine unique, des segments internes et des segments feuilles. Le niveau d'un segment caractérise sa distance à la racine.

Notion IV.12 : Base de données hiérarchique (*Hierarchical data base*)

Base de données constituée par une forêt de segments.

Une représentation par un graphe d'une base de données hiérarchique découle de la définition. Les nœuds du graphe sont les segments alors que les arcs représentent les associations père-fils ; ces arcs sont orientés du père vers le fils. Il est possible de considérer un graphe des types ou un graphe d'occurrences (voir figure IV.16).

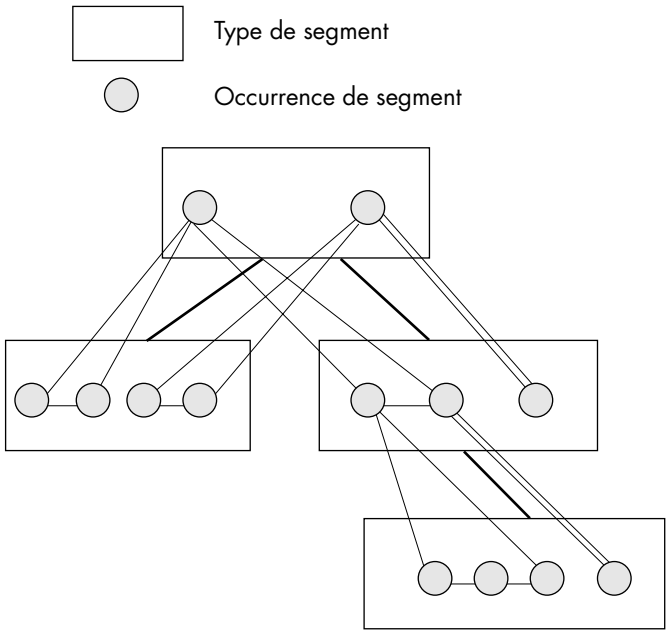


Figure IV.16 : Graphe hiérarchique de type et d'occurrence

Les deux graphes sont bien sûr des forêts, c'est-à-dire des ensembles de hiérarchies sans lien entre elles. Un graphe des types n'est pas différent d'un diagramme de Bachman d'une base de données réseau ; cependant, de chaque segment est issu au plus un lien allant vers son fils. Les liens ne sont pas nommés.

À titre d'exemple, nous avons défini une base de données hiérarchique à partir de la base vinicole. Il n'est évidemment pas possible de représenter un réseau de liens tel que défini figure IV.4. Afin de ne pas perdre d'informations, on est conduit à dupliquer certaines données, par exemple le cru du vin dans les segments ABUS et le nom du buveur dans les segments COMMANDES (Champ CLIENT). Comme les segments ne sont pas hiérarchiques, le type d'article VINS doit être éclaté en deux segments CRUS et MILLESIMES. La figure IV.17 schématise une représentation hiérarchique de la base vinicole.

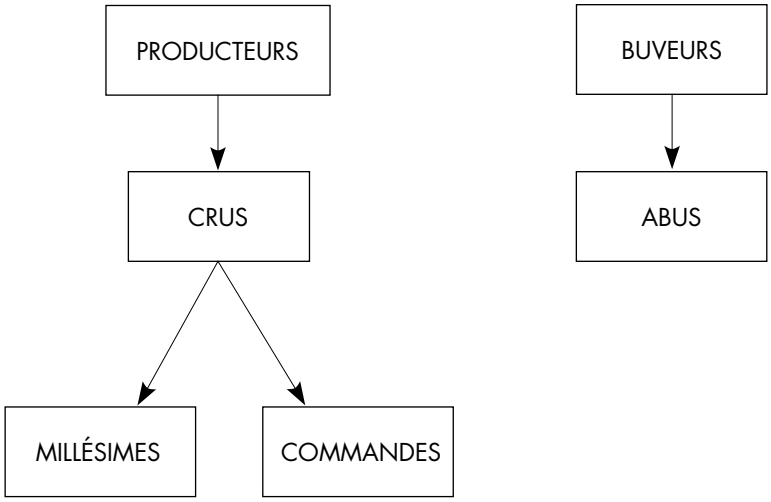


Figure IV.17 : Exemple de base hiérarchique

Certains modèles hiérarchiques autorisent les liens entre arbres pour permettre de modéliser des associations de 1 vers N sans avoir à dupliquer des segments. Tout segment d'un arbre peut alors pointer vers un segment d'un autre arbre. Cela peut être limité à un seul pointeur par segment : on parle de lien frère, pour distinguer ce lien du lien vers le fils. On aboutit alors à des modèles hiérarchiques étendus dont les possibilités se rapprochent de celles du modèle réseau [IBM78].

4.2. INTRODUCTION AU LANGAGE DL1

DL1 est un langage de manipulation de bases de données hiérarchiques proposé par IBM dans son produit IMS [IBM78]. Nous allons ci-dessous en donner quelques prin-

cipes afin d'illustrer la manipulation de bases hiérarchiques. Le langage DL1 est un langage très complexe qui permet de naviguer dans une base hiérarchique à partir de programmes écrits en PL1, COBOL ou FORTRAN. Vous trouverez diverses présentations de ce langage dans [Tsichritzis76, Cardenas85].

Les recherches s'effectuent en naviguant dans les arbres d'occurrences. Un ordre de parcours des arbres de la racine vers les fils, et de gauche à droite, est imposé. Plus précisément, cet ordre est défini comme suit :

1. Visiter le segment considéré s'il n'a pas déjà été visité.
2. Sinon, visiter le fils le plus à gauche non précédemment visité s'il en existe un.
3. Sinon, si tous les descendants du segment considéré ont été visités, remonter à son père et aller à 1.

Cet ordre de parcours des arbres est illustré figure IV.18 sur une forêt d'occurrences de l'un des arbres des types de segments d'une base hiérarchique. Une telle forêt est supposée avoir une racine virtuelle de tous les arbres pour permettre le passage d'un arbre à un autre, représentée par une ligne sur le schéma. L'ordre de parcours est utilisé pour les recherches d'occurrences de segments satisfaisant un critère, mais aussi pour les insertions de nouvelles occurrences.

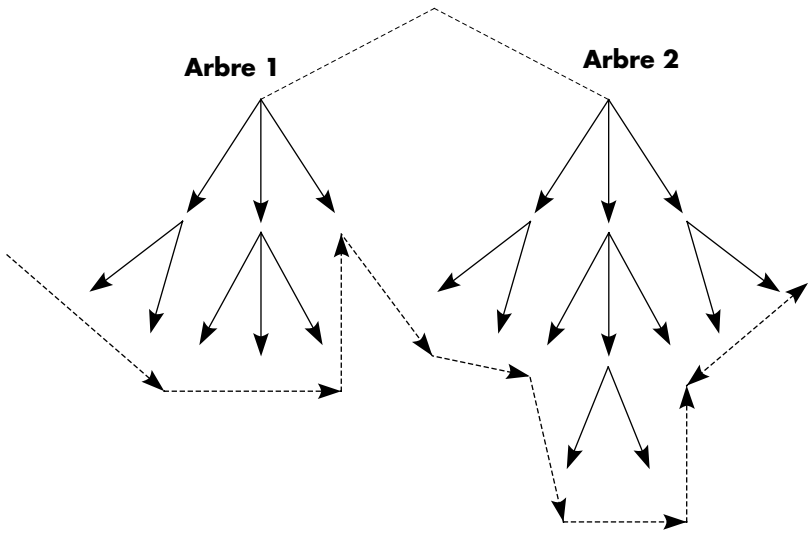


Figure IV.18 : Ordre de parcours des arbres en DL1

DL1 est un langage de manipulation navigationnel, en ce sens que des curseurs permettent de mémoriser un positionnement sur la base. Ces curseurs, appelés PCB, sont stockés dans des structures de données passées en arguments des appels DL1 effectués comme des appels de procédure depuis les programmes utilisateurs. Les PCB

permettent en particulier de mémoriser des positionnements sur la base de données par des **clés base de données** (adresses d'occurrences de segments). Une clé base de données est souvent la concaténation des clés des segments traversés depuis la racine jusqu'au segment de niveau le plus bas sélectionné. Les PCB permettent aussi de récupérer un code réponse STATUS du système après chaque appel à une commande de recherche ou de mise à jour. La figure IV.19 illustre la structure simplifiée d'un PCB. Outre le nom de la base et le code réponse, on notera la présence de la clé base de données indiquant le positionnement courant du PCB.

```
01 PCB /* BLOC DE CONTROLE DE PROGRAMME */
02 NOM DE BASE /* BASE DE DONNÉES RÉFÉRENCÉES */
02 NIVEAU /* NIVEAU MAXIMUM CONSIDÉRÉ */
02 STATUS /* CODE RÉPONSE AUX REQUETES */
02 LONGUEUR CLE /* LONGUEUR DU CHAMP SUIVANT */
02 CLE BASE DE DONNES/* POSITIONNEMENT COURANT SUR LA BASE */
```

Figure IV.19 : Curseurs de programme en DL1

Une qualification de chemin peut être associée à un appel DL1. Elle se compose d'une suite de qualifications de segments (Search Segment Argument); une qualification de segments est une structure dont une vue simplifiée est représentée figure IV.20. Il s'agit en fait d'une expression logique parenthésée ou conjonctive (ET de OU) de critères de comparaisons portant sur un segment. Outre le nom du segment concerné, chaque prédicat élémentaire contient un code commande permettant de spécifier des options telles que recherche ou insertion demandée, un nom de champ, un comparateur et une valeur. Les prédicats élémentaires sont connectés par le connecteur logique. Ils se suivent dans un SSA. Des parenthèses peuvent être utilisées pour marquer des priorités.

```
01 SSA /*(QUALIFICATION DE SEGMENTS)*/
02 NOM-DE-SEGMENT
02 CODE-COMMANDE /*(DIVERSES OPTIONS POSSIBLES)*/
02 NOM-DE-CHAMP
02 OPERATEUR-DE-COMPARAISON /*(<, ≤, >, ≥, =, ≠)*/
02 VALEUR
02 CONNECTEUR /*(AND, OR) */
   ...
```

Figure IV.20 : Qualification de segment en DL1

Un ensemble de qualifications de segments suivant la hiérarchie des segments constitue une **qualification de chemin**. Une telle qualification spécifie un chemin depuis la racine jusqu'au segment cherché dans un arbre du graphe des types en permettant de sélectionner certaines occurrences de segment à chaque niveau. Certains niveaux peu-

vent être absents, ce qui implique un balayage séquentiel du segment de ce niveau. Le code commande permet en particulier de spécifier si l'occurrence de segment sélectionnée doit ou non être lue en zone de travail lors de l'utilisation du SSA dans une commande de recherche au SGBD. Lors d'une insertion d'une hiérarchie de segments, ce code permet de préciser si le segment du niveau doit être inséré. À partir de ces éléments (PCB et qualification de chemin), différents types d'appels au SGBD peuvent être effectués. Les plus importants sont explicités ci-dessous.

GET UNIQUE (GU) permet de rechercher directement la première occurrence de segment satisfaisant la qualification, par exemple pour une recherche sur clé. Le PCB passé en argument est utilisé pour mémoriser la clé base de données du segment trouvé ; le contenu des segments retrouvés à chaque niveau est chargé en mémoire pour les segments dont le SSA demande la lecture dans le code commande.

GET NEXT (GN) permet de rechercher l'occurrence de segment suivant celle mémorisée dans le PCB (en général la dernière trouvée) satisfaisant la qualification si elle existe. Le parcours du graphe s'effectue selon l'ordre indiqué ci-dessus en balayant séquentiellement chaque occurrence de segment, sans tenir compte des liens de filiation.

GET NEXT WITHIN PARENT (GNP) a la même fonction que GET NEXT, mais il autorise la recherche seulement à l'intérieur des descendants du segment courant. Il permet donc de parcourir un lien depuis un parent.

INSERT (ISRT) permet d'insérer un nouveau segment en-dessous du parent sélectionné par la qualification.

DELETE (DLET) permet de supprimer l'occurrence du segment courant. Celle-ci a dû être précédemment sélectionnée par une instruction spéciale du type GET HOLD UNIQUE (GHU) ou GET HOLD NEXT (GHN) ou GET HOLD NEXT WITHIN PARENT (GHNP) afin d'assurer la non sélection simultanée par plusieurs utilisateurs. La suppression détruit l'occurrence de segment précédemment sélectionnée ainsi que tous ses descendants s'il en existe. Il s'agit donc d'une suppression en cascade.

REPLACE (REPL) permet de modifier l'occurrence du segment courant. Celle-ci a dû être précédemment lue par une instruction spéciale du type GET HOLD, comme pour la suppression. La clé d'un segment n'est pas modifiable par la commande REPLACE.

4.3. QUELQUES EXEMPLES DE TRANSACTIONS

Voici quelques types d'accès par des transactions simples. La syntaxe utilisée est très loin de la syntaxe pénible de DL1 [IBM78]. Il s'agit en fait d'une abstraction de cette syntaxe compatible avec les commandes décrites ci-dessus.

```
/*DECLARATIONS*/
DCL   1  SSA
      2  SEGMENT     = 'BUVEURS'
      2  CHAMP       = 'NB'
      2  OPERATEUR   = '='
      2  VALEUR      = '100'
DCL   1  PCB
      ..........
      2  STATUS

/*PROGRAMME*/
      GU (PCB, BUVEUR, SSA)
      PRINT BUVEUR.NOM, BUVEUR.PRENOM
```

Figure IV.21 : Accès sur clé

```
/*DECLARATIONS*/
DCL   1  SSA1
      2  SEGMENT     = 'PRODUCTEURS'
      2  CHAMP       = 'NOM'
      2  OPERATEUR   = '='
      2  VALEUR      = 'MARTIN'
DCL   1  SSA2
      2  SEGMENT     = 'CRUS'
      2  CHAMP       = 'CRU'
      2  OPERATEUR   = '='
      2  VALEUR      = 'BEAUJOLAIS'
DCL   1  PCB
      ..........
      2  STATUS

/*PROGRAMME*/
      GU (PCB, CRUS, SSA1, SSA2)
      WHILE STATUS ≠ 'fin-de-descendance'
      GNP
      PRINT "segment lu"
      END-WHILE
```

Figure IV.22 : Balayage des descendants d'une occurrence de segment

```
/*DECLARATIONS*/
DCL   1  SSA1
      2  SEGMENT     = 'PRODUCTEURS'
      2  CHAMP       = 'NOM'
      2  OPERATEUR   = '='
      2  VALEUR      = 'MARTIN'
DCL   1  SSA2
      2  SEGMENT     = 'CRUS'
      2  CHAMP       = 'CRU'
      2  OPERATEUR   = '='
      2  VALEUR      = 'BEAUJOLAIS'
DCL   1  SSA3
      2  SEGMENT     = 'COMMANDES'
      2  CHAMP       = 'DATE'
      2  OPERATEUR   = '='
      2  VALEUR      = '10-02-81'
      2  CONNECTEUR  = 'AND'
      2  CHAMP       = 'CLIENT'
      2  OPERATEUR   = '='
      2  VALEUR      = 'DUPONT'
DCL   1  PCB
      ..........
      2  STATUS

/*PROGRAMME*/
      GHU (PCB, COMMANDES, SSA1, SSA2, SSA3)
      COMMANDES.QUANTITE = COMMANDES.QUANTITE + 100
      RPL
```

Figure IV.23 : Mise à jour d'une occurrence de segment

5. CONCLUSION

Dans ce chapitre, nous avons présenté les principaux modèles d'accès, c'est-à-dire ceux directement issus de la modélisation d'organisation d'articles stockés dans des fichiers et reliés entre eux par des pointeurs. Ces modèles sont encore très utilisés : une part significative des grandes bases de données sont sans doute encore hiérarchiques ou organisées en réseaux. Ces modèles sont aussi très performants car très proches du niveau physique. Le modèle réseau permet des organisations de liens plus générales que le modèle hiérarchique, bien que celui-ci ait souvent été étendu par des liens inter-hiérarchies.

Compte tenu de l'accroissement des possibilités des machines, les modèles d'accès sont aujourd'hui inadaptés en tant que modèles conceptuels ou externes de données. Ils assu-

rent en effet une faible indépendance des programmes aux données. Ils peuvent rester très compétitifs au niveau interne. Les critiques proviennent sans doute aussi de la complexité des langages de manipulation historiquement associés à ces modèles, basés sur la navigation. Cependant, il est tout à fait possible de définir des langages de plus haut niveau fondés sur la logique des prédicats pour un modèle hiérarchique ou réseau.

Finalement, la question s'est posée de savoir si le modèle entité-association est un meilleur candidat pour modéliser les données au niveau conceptuel que le modèle réseau. La réponse est positive. En effet, un modèle se voulant un cadre conceptuel pour définir une large classe de base de données structurées et un médiateur entre des représentations de données stockées et des vues usagers, devrait probablement avoir au moins quatre caractéristiques [Codd79] :

1. permettre une représentation sous forme de tableaux de données ;
2. permettre une interrogation ensembliste afin d'autoriser des requêtes sans préciser les ordres élémentaires de navigation dans la base ;
3. permettre une interprétation des objets en termes de formules de la logique des prédicats afin de faciliter les inférences de connaissances ;
4. supporter une représentation graphique simple afin de pouvoir visualiser les objets et leurs associations pour concevoir la base.

Ajoutons également que la simplicité est une caractéristique essentielle d'un modèle. C'est là un avantage important du modèle relationnel que nous allons étudier dans la partie suivante de cet ouvrage.

6. BIBLIOGRAPHIE

[Bachman64] Bachman C., Williams S., « A General Purpose Programming System for Random Access Memories », *Fall Joint Computer Conference*, vol. 26, AFIPS Press, p. 411-422.

La première présentation du système IDS, réalisé à General Electric au début des années 60.

[Bachman69] Bachman C., « Data Structure Diagrams », *Journal of ACM, SIGBDP* vol. 1, n° 2, mars 1969, p. 4-10.

Cet article introduit les diagrammes de Bachman. Ceux-ci permettent de modéliser les types d'articles par des boîtes et les liens (sets) par des flèches depuis le propriétaire vers le membre. Ils modélisent une base de données comme un graphe dont les nœuds sont les types d'articles et les arcs les associations de 1 vers n articles. Ce type de modélisation est toujours très utilisé, notamment dans l'outil de conception BACHMAN, du nom de l'inventeur.

[Bachman73] Bachman C., « The Programmer as Navigator », *Communication of the ACM*, vol. 16, n° 1, novembre 1971.

Cet article correspond à la présentation de Charlie Bachman lorsqu'il a reçu le Turing Award en 1973. Bachman compare le programmeur à un navigateur qui suit les chemins d'accès aux bases de données à l'aide de curseurs, afin d'atteindre les articles désirés.

[Cardenas85] Cardenas A., *Data Base Management Systems*, 2nd Edition, Allyn and Bacon, Boston, Ma, 1985.

Ce livre présente plus particulièrement les techniques de base des modèles réseau et hiérarchique. Il décrit de nombreux systèmes dont TOTAL, IDS II, SYSTEM 2000 et IMS.

[Codasyl71] Codasyl Database Task Group, *DBTG April 71 Report*, ACM Ed., New York, 1971.

Les spécifications de référence des langages de description et de manipulation du CODASYL. La présentation effectuée dans ce chapitre est très proche de ces propositions.

[Codasyl78] Codasyl Database Task Group, *Codasyl Data Description Language Journal of Development*, Codasyl Ed., New York, 1978.

La nouvelle version du langage de définition de données du CODASYL. Cette version est probablement moins implémentée que la précédente.

[Codasyl81] Codasyl Database Task Group, *Codasyl Data Description Language Journal of Development – Draft Report*, Codasyl Ed., New-York, 1981.

Encore une nouvelle version du langage de définition et de manipulation de données du CODASYL. Cette version n'a jamais été adoptée.

[IBM78] IBM Corporation, « Information Management System/ Virtual Storage General Information », *IBM Form Number GH20-1260*, SH20-9025-27.

La description générale du système IMS II d'IBM.

[MRI74] MRI Systems Corporation, *System 2000 Reference Manual*, Document UMN-1, 1974.

Le manuel de référence de la première version de System 2000.

[Taylor76] Taylor R.W., Frank R.L., « Codasyl Data Base Management Systems », *ACM Computing Surveys*, vol. 8, n° 1, mars 1976.

Un panorama très complet résumant les caractéristiques et les évolutions du modèle réseau.

[Tsichritzis76] Tsichritzis D.C., Lochovsky F.H., « Hierarchical Database Management – A Survey », *ACM Computing Surveys*, vol. 8, n° 1, mars 1976.

Un panorama très complet résumant les caractéristiques et les évolutions du modèle hiérarchique.

Chapitre V

LOGIQUE ET BASES DE DONNÉES

1. INTRODUCTION

Historiquement, les chercheurs ont tout d'abord essayé de modéliser les langages d'interrogation de bases de données par la logique. Ainsi sont nés les calculs relationnels, véritables langages d'interrogation fondés sur la logique du premier ordre. Ce n'est qu'à partir de la fin des années 70 que l'on a cherché à comprendre globalement les bases de données par la logique. Cela a permis de montrer qu'un SGBD était un démonstrateur de théorèmes très particulier, raisonnant sur des faits et donnant les preuves d'un théorème représentant la question.

Les travaux sur l'introduction dans les SGBD de raisonnements généraux incluant non seulement des faits, mais aussi des règles déductives exprimées en logique du premier ordre, ont débuté principalement au CERT à Toulouse à la fin de la décennie 1970-1980 [Gallaire78]. Ils se sont surtout concentrés sur la compréhension des bases de données déductives par la logique. Afin de permettre la gestion de grandes bases de connaissances (quelques milliers de règles associées à quelques millions voire milliards de faits), des problèmes de recherche complexes ont dû être résolus. Ces problèmes sont à la croisée des chemins des bases de données et de l'intelligence artificielle et présentent des aspects à la fois théoriques et pratiques. D'un point de vue théorique, les approches par la logique permettent une meilleure compréhension des langages des SGBD, des mécanismes de raisonnements sous-jacents et des techniques d'optimisation. D'un point de vue pratique, le développement de SGBD basés sur la

logique supportant la déduction a conduit à des prototypes opérationnels, mais encore rarement à des produits.

Ce chapitre se propose tout d'abord d'introduire les notions de base de logique nécessaires à la bonne compréhension des bases de données. Nous définissons ce qu'est une base de données logique vue comme une interprétation d'un langage logique. Il s'agit simplement d'un ensemble de faits listés dans des tables. Historiquement, cette vision est une compréhension des bases de données relationnelles proposées par Gallaire, Minker et Nicolas à la fin des années 70 [Gallaire81]. C'est une vision élémentaire appelée théorie du modèle (ou de l'interprétation). Pour beaucoup, le terme BD logique inclut les facilités déductives que nous étudierons dans le chapitre sur les BD déductives.

Dans ce chapitre, nous introduisons aussi les langages logiques d'interrogation de bases de données que sont les calculs de domaines et de tuples. Ils sont une formalisation des langages des bases de données relationnelles que nous étudierons dans la partie suivante. Pour bien les comprendre, il est bon d'avoir quelques notions sur les BD relationnelles, mais ce n'est sans doute pas indispensable.

Ce chapitre comporte cinq sections. Après cette introduction, la section 2 introduit la logique du premier ordre et les manipulations de formules. La section 3 définit ce qu'est une BD logique non déductive. La section 4 est consacrée au calcul de domaine et à son application pratique, le langage QBE. La section 5 traite de la variante calcul de tuple, qui était à la base du langage QUEL, somme toute assez proche de SQL. La section 6 introduit les techniques de raisonnement et de preuve de théorème. Il est nécessaire de connaître un peu ces techniques pour aborder ultérieurement les bases de données déductives.

2. LA LOGIQUE DU PREMIER ORDRE

Dans cette partie, nous rappelons les principes de la logique du premier ordre et nous introduisons la méthode de réécriture sous forme de clauses. La logique permet une modélisation simple des bases de données, l'introduction de langages de requêtes formels et le support de la déduction.

2.1. SYNTAXE DE LA LOGIQUE DU PREMIER ORDRE

Rappelons que la logique du premier ordre, aussi appelée **calcul de prédicats**, est un langage formel utilisé pour représenter des relations entre objets et pour déduire de

nouvelles relations à partir de relations connues comme vraies. Elle peut être vue comme un formalisme utilisé pour traduire des phrases et déduire de nouvelles phrases à partir de phrases connues.

La logique du premier ordre repose sur un alphabet qui utilise les symboles suivants :

1. Des variables généralement notées x, y, z …
2. Des constantes généralement notées a, b, c …
3. Des prédicats généralement notées P, Q, R …, chaque prédicat pouvant recevoir un nombre fixe d'arguments (n pour un prédicat n-aire).
4. Les connecteurs logiques $\wedge$, $\vee$, $\Rightarrow$, $\neg$.
5. Des fonctions généralement dénotées f, g, h…, chaque fonction pouvant recevoir un nombre fixe d'arguments (n pour une fonction n-aire).
6. Les quantificateurs $\forall$ et $\exists$.

Des règles syntaxiques simples permettent de construire des formules. Un **terme** est défini récursivement comme une variable ou une constante ou le résultat de l'application d'une fonction à un terme. Par exemple, x, a, f(x) et f(f(x)) sont des termes. Une **formule** est une phrase bien construite du langage. Une formule est définie comme suit :

1. Si P est un prédicat à n arguments (n places) et t1, t2…tn des termes, alors P(t1,t2…tn) est une **formule atomique**.
2. Si F1 et F2 sont des formules, alors F1 $\wedge$ F2, F1 $\vee$ F2, F1 $\Rightarrow$ F2 et $\neg$ F1 sont des formules.
3. Si F est une formule et x une variable non quantifiée (on dit aussi libre) dans F, alors $\forall$x F et $\exists$x F sont des formules.

Nous résumons ci-dessous les concepts essentiels introduits jusque-là sous forme de notions.

Notion V.1 : Terme (*Term*)

Constante, variable ou fonction appliquée à un terme.

Notion V.2 : Formule atomique (*Atomic Formula*)

Résultat de l'application d'un prédicat à n-places à n termes, de la forme P(t1,t2…tn).

Notion V.3 : Formule (*Formula*)

Suite de formules atomiques ou de négation de formules atomiques séparées par des connecteurs logiques `et`, `ou`, `implique`, *avec d'éventuelles quantifications des variables.*

Pour indiquer les priorités entre connecteurs logiques, il est possible d'utiliser les parenthèses : si F est une formule valide, (F) en est aussi une. Afin d'éviter les paren-

thèses dans certains cas simples, nous supposerons une priorité des opérateurs logiques dans l'ordre descendant ¬, ∨, ∧, et ⇒. Voici quelques exemples de formules formelles :

$P(a,x) \wedge Q(x,y)$,
$\neg Q(x,y)$,
$\forall x \exists y (Q(x,y) \vee P(a,x))$,
$\forall x (R(x) \wedge Q(x,a) \Rightarrow P(b,f(x)))$.

Afin de donner des exemples plus lisibles de formules, nous choisirons un vocabulaire moins formel, les prédicats et fonctions étant des noms ou des verbes du langage courant et les constantes des chaînes de caractères désignant par exemple des services ou des employés. Voici quelques exemples de formules proches du langage naturel :

$$\text{SERVICE(informatique,pierre)} \vee \text{EMPLOYE (informatique,marie)}$$
$$\forall x ((\text{DIRIGE(pierre,x)} \vee \neg \text{EMPLOYE(informatique,x)} \Rightarrow \text{EMPLOYE(finance,x)})$$
$$\forall x \exists y ((\text{DIRIGE(x,y)} \vee \text{DIRIGE(y,x)}) \Rightarrow \text{MEMESERVICE(x,y)})$$

2.2. SÉMANTIQUE DE LA LOGIQUE DU PREMIER ORDRE

Une formule peut être interprétée comme une phrase sur un ensemble d'objets : il est possible de lui donner une signification vis-à-vis de cet ensemble d'objets. Pour ce faire, il faut assigner un objet spécifique à chaque constante. Par exemple, on assigne un objet à la constante a, le service Informatique de l'entreprise considérée à la constante `informatique`, l'employée Marie à la constante `marie`, etc. L'ensemble d'objets considérés est appelé le **domaine de discours** ; il est noté D. Chaque fonction à n arguments est interprétée comme une fonction de D^n dans D. Un prédicat représente une relation particulière entre les objets de D, qui peut être vraie ou fausse. Définir cette relation revient à définir les tuples d'objets qui satisfont le prédicat. L'ensemble des tuples satisfaisants le prédicat constitue son extension.

Notion V.4 : Domaine de discours (*Domain of Discourse*)

Ensemble d'objets sur lequel une formule logique prend valeur par interprétation des constantes comme des objets particuliers, des variables comme des objets quelconques, des prédicats comme des relations entre objets et des fonctions comme des fonctions particulières entre objets.

Par exemple, la formule :

$$\forall x \exists y ((\text{DIRIGE(x,y)} \vee \text{DIRIGE(y,x)}) \Rightarrow \text{MEMESERVICE(x,y)})$$

peut être interprétée sur l'ensemble des personnes {Pierre, Paul, Marie}, qui constitue alors le domaine de discours. DIRIGE peut être interprété comme la relation binaire « est chef de » ; MEMESERVICE peut être interprété comme la relation binaire « tra-

vaille dans le même service ». La formule est vraie si Pierre, Paul et Marie travaillent dans le même service.

Un univers de discours infini peut être retenu pour interpréter la formule $\forall x\ (x < SUCC(x))$. En effet, cette formule peut être interprétée sur l'ensemble des entiers positifs {1,2,3…} qui est infini. < est alors la relation « est inférieur à » et SUCC la fonction qui à tout entier associe son successeur. Il est clair que cette formule est vraie sur les entiers.

Ainsi, étant donné une interprétation d'une formule sur un domaine de discours, il est possible d'associer une valeur de vérité à cette formule. Pour éviter les ambiguïtés (les formules pouvant avoir plusieurs valeurs de vérité), nous considérons seulement les formules dans lesquelles toutes les variables sont quantifiées, appelées **formules fermées**. Toute formule fermée F a une unique valeur de vérité pour une interprétation donnée sur un domaine de discours D. Cette valeur notée V(F) se calcule ainsi :

$V(F1 \vee F2) = V(F1) \vee V(F2)$

$V(F1 \wedge F2) = V(F1) \wedge V(F2)$

$V(\neg\ F1) = \neg\ V(F1)$

$V(\forall x\ F) = V(F_{x=a}) \wedge V(F_{x=b}) \wedge \ldots$ où a, b, … sont les constantes de D.

$V(\exists x\ F) = V(F_{x=a}) \vee V(F_{x=b}) \vee \ldots$ où a, b, … sont les constantes de D.

V(P(a,b…)) = Vrai si [a,b…] satisfait la relation P et Faux sinon.

Un **modèle** d'une formule logique est une interprétation sur un domaine de discours qui rend la formule vraie. Par exemple, les entiers sont un modèle pour la formule $\forall x\ (x < SUCC(x))$ avec l'interprétation indiquée ci-dessus. En effet, en appliquant les règles ci-dessus, on calcule :

$V(\forall x\ (x < SUCC(x))) = V(1 < 2) \wedge V(2<3) \wedge V(3<4) \wedge \ldots$ = Vrai.

2.3. FORME CLAUSALE DES FORMULES FERMÉES

Toute formule fermée, c'est-à-dire à variables quantifiées, peut se simplifier et s'écrire sous une forme canonique sans quantificateurs, appelée forme clausale. La forme clausale nécessite d'écrire la formule comme un ensemble de **clauses**.

Notion V.5 : Clause (*Clause*)

Formule de la forme $P1 \wedge P2 \wedge \ldots \wedge Pn \Rightarrow Q1 \vee Q2 \vee \ldots \vee Qm$, ou les Pi et les Qj sont des littéraux positifs (c'est-à-dire des prédicats atomiques sans négation devant).

Une clause ayant un seul littéral situé après l'implication (on dit en tête de clause), donc un seul Qi, est dite **clause de Horn**.

Notion V.6 : Clause de Horn (*Horn Clause*)

Clause de la forme P1∧ P2 ∧ … ∧ Pn ⇒ Q1.

Par exemple :

$$\text{ENTIER}(x) \wedge (x > 0) \Rightarrow (x < \text{SUCC}(x)) \vee (x > \text{SUCC}(x))$$

$$\text{DIRIGE}(x,y) \wedge \text{MEMESERVICE}(x,y) \Rightarrow \text{AIME}(x,y) \vee \text{DETESTE}(x,y)$$

sont des clauses.

$$\text{ENTIER}(x) \wedge (x > 0) \Rightarrow (x < \text{SUCC}(x))$$

$$\text{DIRIGE}(x,y) \wedge \text{MEMESERVICE}(x,y) \Rightarrow \text{AIME}(x,y)$$

sont des clauses de Horn.

La transformation d'une formule fermée en clauses s'effectue par des transformations successives que nous décrivons brièvement ci-dessous. Nous illustrons les transformations avec la formule :

$$\forall x\ \exists y\ ((\text{DIRIGE}(x,y) \vee \text{DIRIGE}(y,x)) \Rightarrow \text{MEMESERVICE}(x,y)).$$

1. **Élimination des implications**. Ceci s'effectue simplement en remplaçant toute expression de la forme $A \Rightarrow B$ par $\neg A \vee B$. En effet, ces expressions sont équivalentes du point de vue logique. La formule choisie en exemple devient :
   $$\forall x\ \exists y\ (\neg\ (\text{DIRIGE}(x,y) \vee \text{DIRIGE}(y,x)) \vee \text{MEMESERVICE}(x,y)).$$
2. **Réduction de la portée des négations**. Le but est de faire que les négations s'appliquent directement à des prédicats atomiques. Pour cela, on utilise de manière répétée les transformations :
   - $\neg\ (A \vee B)$ devient $\neg A \wedge \neg B$;
   - $\neg\ (A \wedge B)$ devient $\neg A \vee \neg B$;
   - $\neg\ (\forall x A)$ devient $\exists x \neg A$;
   - $\neg\ (\exists x A)$ devient $\forall x \neg A$;
   - $\neg\ (\neg\ A)$ devient A.

   L'exemple devient :
   $$\forall x\ \exists y\ ((\neg\ \text{DIRIGE}(x,y) \wedge \neg\ \text{DIRIGE}(y,x)) \vee \text{MEMESERVICE}(x,y)).$$
3. **Mise en forme prenex**. Il s'agit simplement de mettre les quantificateurs avec la variable quantifiée en-tête de formule. Cette étape peut nécessiter de renommer les variables afin d'éviter les confusions de quantification. Notre exemple reste ici inchangé.
4. **Élimination des quantificateurs**. Afin de simplifier encore, les variables quantifiées existentiellement sont remplacées par une constante paramètre dite **constante de Skolem**. En effet, s'il existe x satisfaisant une formule, il est possible de choisir une constante s désignant cette valeur de x. Attention, si une variable quantifiée par « quel que soit » apparaît avant la variable quantifiée par « il existe », la constante choisie dépend de la première variable. Par exemple, dans le cas $\forall x\ \exists y$ (pour tout x,

il existe y, mais le y dépend de x), on remplacera donc y par s(x), c'est-à-dire une fonction de Skolem qui matérialise la constante y dépendant de x. Ainsi, il est possible d'éliminer les variables quantifiées par « il existe ». Après cette transformation, toute variable restante est quantifiée par « quel que soit ». On peut donc oublier d'écrire les quantificateurs (c'est implicitement $\forall$). Notre exemple devient alors :

$((\neg \text{DIRIGE}(x,s(x)) \wedge \neg \text{DIRIGE}(s(x),x)) \vee \text{MEMESERVICE}(x,s(x)))$.

5. **Mise en forme normale conjonctive**. La formule restante est une combinaison par des « ou » et des « et » de littéraux positifs ou négatifs (prédicats atomiques précédés ou non d'une négation). Elle peut être écrite comme une conjonction ($\wedge$) de disjonction ($\vee$) en distribuant les « et » par rapport aux « ou » à l'aide de la règle :

   $A \vee (B \wedge C)$ devient $(A \vee B) \wedge (A \vee C)$.

   L'exemple devient ainsi :

   $(\neg \text{DIRIGE}(x,s(x)) \vee \text{MEMESERVICE}(x,s(x))) \wedge (\neg \text{DIRIGE}(s(x),x)) \vee \text{MEMESERVICE}(x,s(x)))$.

6. **Écriture des clauses**. Finalement, il est simple d'écrire chaque clause sur une ligne en remplaçant les « et » par des changements de lignes. De plus, profitant du fait que $\neg A \vee B$ s' écrit $A \Rightarrow B$, on peut écrire tous les prédicats négatifs d'abord en éliminant la négation et en les connectant par $\wedge$, puis tous les prédicats positifs connectés par $\vee$. On obtient bien ainsi une suite de clauses, définies comme ci-dessus. Avec l'exemple, on obtient deux clauses (d'ailleurs de Horn):

   $\text{DIRIGE}(x,s(x)) \Rightarrow \text{MEMESERVICE}(x,s(x))$

   $\text{DIRIGE}(s(x),x)) \Rightarrow \text{MEMESERVICE}(x,s(x)))$.

3. LES BASES DE DONNÉES LOGIQUES

La notion de base de données logique a été introduite en particulier à Toulouse à la fin des années 70 [Gallaire84]. Nous définissons ici ce qu'est une base de données logique non déductive. Il s'agit d'une première approche simple des bases de données par la logique. Nous verrons plus loin que cette approche peut être étendue avec la déduction.

3.1. LA REPRÉSENTATION DES FAITS

Une base de données peut être définie comme **l'interprétation d'un langage logique** du premier ordre L. En pratique, définir l'interprétation consiste à définir les prédicats

vrais. Le langage permet d'exprimer les requêtes, comme nous le verrons ci-dessous. Une contrainte d'intégrité peut être vue comme une requête toujours vraie, donc exprimée avec le langage.

Notion V.7 : Base de données logique (*Logic database*)

Ensemble de faits constituant l'interprétation d'un langage du premier ordre avec lequel il est possible d'exprimer des questions et des contraintes d'intégrité sur les faits.

En première approche, le langage logique L ne comporte pas de fonctions. Plus particulièrement, L se compose :

1. de prédicats à n arguments, chaque argument correspondant à un type de données élémentaire ;
2. de constantes, une pour chaque valeur possible des type de données élémentaires.

Comme dans toute interprétation d'un langage logique, un prédicat représente une relation particulière entre les objets du domaine de discours, qui peut être vraie ou fausse. Ici les objets du domaine de discours sont donc les valeurs possibles de la base. Définir cette relation revient à définir les articles ou n-uplets qui satisfont le prédicat. L'ensemble des n-uplets satisfaisant le prédicat constitue son **extension**. Cette extension peut être assimilée à un fichier dans une base en réseau ou à une table relationnelle, comme nous le verrons plus loin. Pour rester cohérent avec les bases de données relationnelles qui peuvent être perçues comme une implémentation simple des bases de données logiques, nous appellerons l'extension d'un prédicat une **table**. Chaque colonne correspond à un argument et est aussi appelée **attribut** dans le contexte relationnel. Comme déjà dit, une base de données logique pourra être complétée par des règles de déduction : nous parlerons alors de **base de données déductive** (voir partie 4 de cet ouvrage).

La figure V.1 illustre une base de données logique. En termes de tables, cette base correspond à celles représentées figure V.2. Elle est composée de trois prédicats définis comme suit :

– PRODUIT avec les arguments Numéro de produit (NPRO), nom de produit (NPRO), quantité en stock (QTES), et couleur (COULEUR) ;
– VENTE avec les arguments numéro de vente (NVEN), nom du client (NOMC), numéro du produit vendu (NPRV), quantité vendue (QTEV) et date (DATE) ;
– ACHAT avec les arguments numéro d'achat (NACH), date de l'achat (DATE), numéro du produit acheté (NPRA), quantité achetée (QTEA) et nom du fournisseur (NOMF).

Comme on le voit, seul les faits positifs, c'est-à-dire ceux satisfaisant l'un des trois prédicats, sont enregistrés dans la base. Ceci constitue l'**hypothèse du monde fermé**. Les faits non enregistrés dans une extension de prédicat sont supposés faux. Il est vrai que si vous interrogez la base de données que gère votre banque et que vous ne voyez pas de chèque de 100 000 euros crédités ce jour, c'est que le fait que cette somme ait

été créditée sur votre compte est faux ! Des chercheurs ont essayé de lever cette hypothèse : on tombe alors dans l'hypothèse du monde ouvert, ou un fait non enregistré peut être inconnu [Reiter78].

```
PRODUIT (100, BILLE, 100     , VERTE)
PRODUIT (200, POUPEE, 50, ROUGE)
PRODUIT (300, VOITURE, 70, JAUNE)
PRODUIT (400, CARTE, 350, BLEU)
VENTE (1, DUPONT, 100, 30, 08-03-1999)
VENTE (2, MARTIN, 200, 10, 07-01-1999)
VENTE (3, CHARLES, 100, 50, 01-01-2000)
VENTE (4, CHARLES, 300, 50, 01-01-2000)
ACHAT (1, 01-03-1999, 100, 70, FOURNIER)
ACHAT (2, 01-03-1999, 200, 100, FOURNIER)
ACHAT (3, 01-09-1999, 100, 50, DUBOIS)
```

Figure V.1 : Faits constituant une base de données logique

PRODUIT	NPRO	NOMP	QTES	COULEUR
	100	Bille	100	Verte
	200	Poupée	50	Rouge
	300	Voiture	70	Jaune
	400	Carte	350	Bleu

VENTE	NVEN	NOMC	NPRV	QTEV	DATE
	1	Dupont	100	30	08-03-1999
	2	Martin	200	10	07-01-1999
	3	Charles	100	50	01-01-2000
	4	Charles	300	50	01-01-2000

ACHAT	NACH	DATE	NPRA	QTEA	NOMF
	1	01-03-1999	100	70	Fournier
	2	01-03-1999	200	100	Fournier
	3	01-09-1999	100	50	Dubois
	4	01-09-1999	300	50	Dubois

Figure V.2 : Tables représentant la base de données logique

3.2. QUESTIONS ET CONTRAINTES D'INTÉGRITÉ

Les questions et les contraintes d'intégrité sur la base peuvent alors être exprimées comme des formules dans le langage logique. Cependant, celui-ci doit être étendu

pour inclure les opérateurs de comparaison arithmétique {=, <, ≤, >, ≥, ≠} comme des prédicats particuliers, dont la valeur de vérité est calculée par les opérations usuelles des calculateurs.

La réponse à une question F(x1, …, xn) – où x1, …xn sont des variables libres dans la formule F – est alors l'ensemble des n-uplets <e1, …,ep> tels que F(e1, …ep) est vraie. Certaines formules doivent être toujours vraies sur l'interprétation que constitue la base : ce sont alors des contraintes d'intégrité. Cette vue logique des bases de données a donné naissance à deux calculs permettant d'exprimer des questions et des contraintes d'intégrité sur les bases : le calcul de domaine et le calcul de tuple. Dans le premier, les objets de l'interprétation logique sont les valeurs atomiques de données ; dans le second ce sont les faits composites correspondant aux n-uplets, encore appelés tuples. Nous étudions ces deux calculs ci-dessous.

4. LE CALCUL DE DOMAINES

Les calculs relationnels de domaine et de tuples [Codd72] permettent d'exprimer des requêtes à l'aide de formules du premier ordre sur une base de données logique, ou sa représentation tabulaire qui est une base de données relationnelles. Ils ont été utilisés pour formaliser les langages d'interrogation pour les bases de données relationnelles. Ci-dessous, nous étudions tout d'abord le calcul de domaine et sa représentation bi-dimensionnelle QBE [Zloof77], puis le calcul de tuples. Le langage QUEL introduit au chapitre II peut être perçu comme un paraphrasage en anglais du calcul de tuples.

4.1. PRINCIPES DE BASE

Le calcul relationnel de domaines [Codd72] se déduit donc de la logique du premier ordre. Les données sont les constantes. Les prédicats utilisés sont :

1. les **prédicats extensionnels** contenant les enregistrements de la base ; chaque enregistrement est un tuple typé selon un prédicat extensionnel ; les prédicats étant n-aire, une variable ou une constante est utilisée comme argument de prédicat ; les variables sont alors implicitement typées selon le type de l'argument pour lequel elles apparaissent ;
2. les **prédicats de comparaison** entre une variable et une constante, ou entre deux variables, construits à l'aide des opérateurs {=, ≤, <, ≥, >, ≠}.

Une question en calcul relationnel de domaine s'écrit donc sous la forme suivante :

{x, y… | F(x,y, …)}

F est une formule logique composée à partir de prédicats extensionnels et de comparaison ; les variables résultats x, y, …, sont des variables libres dans F.

La notion suivante résume la définition du calcul de domaine.

Notion V.8 : Calcul relationnel de domaines (*Domain relational calculus*)

Langage d'interrogation de données formel permettant d'exprimer des questions à partir de formules bien formées dont chaque variable est interprétée comme variant sur le domaine d'un argument d'un prédicat.

La BNF du calcul relationnel de domaine est définie figure V.3. Pour simplifier, les formules sont écrites en forme prenex (quantificateurs devant). En pratique, une simplification d'écriture permet de regrouper des prédicats de restriction du type (x = a) et des prédicats de définition de variable du type P (x, …) en écrivant P (a, …). Toute variable apparaissant dans le résultat doit être définie dans un prédicat extensionnel et doit rester libre dans la formule spécifiant la condition. Une variable non utilisée ni en résultat ni dans un critère de comparaison peut être remplacée par la variable muette positionnelle notée « – ».

```
<question> : := "{" (<résultat>) "|" <formule> "}"
<résultat> : := <variable> | <variable>, <résultat>
<formule> : := <quantification> <formule libre> | <formule libre>
<quantification> ::= ∀ <variable> | ∃ <variable>
                   | <quantification> <quantification>
<formule libre> : := <formule atomique>
                   | <formule libre> ∧ <formule atomique>
                   | <formule libre> ∨ <formule atomique>
                   | <formule libre> ⇒ <formule libre>
                   | ¬ <formule libre>
                   | (<formule libre>)
<formule atomique> : := <prédicat extensionnel> (<arguments>)
                   | <terme> <comparateur> <terme>
<arguments> : := <terme> | <terme>, <terme>
<terme> : := <variable> | <constante>
<comparateur> : := = | < | ≤ | > | ≥ | ≠
```

Figure V.3 : BNF du calcul relationnel de domaines

4.2. QUELQUES EXEMPLES DE CALCUL DE DOMAINE

Nous illustrons le calcul de domaine sur la base logique introduite ci-dessus décrivant des achats et des ventes de produits stockés dans un magasin. Le schéma de la base est le suivant :

```
PRODUIT (NPRO, NOMP, QTES, COULEUR)
VENTE (NVEN, NOMC, NPRV, QTEV, DATE)
ACHAT (NACH, DATE, NPRA, QTEA, NOMF)
```

Le prédicat extensionnel PRODUIT se compose des attributs numéro de produit (NPRO), nom du produit (NOMP), quantité en stock (QTES) et couleur (COULEUR). Le prédicat VENTE décrit les ventes de produits effectuées et se compose des attributs numéro de vente (NVEN), nom du client (NOMC), numéro du produit vendu (NPRV), quantité vendue (QTEV) et date de la vente (DATE). Le prédicat ACHAT définit les achats effectués aux fournisseurs. Il contient les attributs numéro d'achat (NACH), date d'achat (DATE), numéro du produit acheté (NPRA), quantité achetée (QTEA) et nom du fournisseur (NOMF). Pour simplifier la lecture, nous utilisons des variables rappelant le nom du domaine. Notez que les quantificateurs existentiels dans les formules peuvent être omis, car ils sont implicites si la variable est libre et ne figure pas en résultat.

(Q1) Donner la liste des noms et des couleurs de tous les produits :
$\{(p,c) \mid \text{PRODUIT}(-,p,-,c)\}$

(Q2) Donner les noms et les quantités en stock des produits de couleur rouge :
$\{(p,q) \mid \text{PRODUIT}(-, p, q, \text{“Rouge”})\}$

(Q3) Donner, pour chaque produit en stock, le nom du fournisseur associé :
$\{(p,f) \mid \exists n\ (\text{PRODUIT}(n, p, -, -) \wedge \text{ACHAT}(-, -, n, -, f))\}$

(Q4) Donner, pour chaque produit en stock en quantité supérieure à 100 et de couleur rouge, les triplets nom de fournisseurs ayant vendu ce type de produit et nom de client ayant acheté ce type de produit et nom du produit :
$\{(f, c, p) \mid \exists n\ \exists q\ (\text{PRODUIT}(n, p, q, \text{“Rouge”}) \wedge \text{ACHAT}(-, -, n, -, f)$
$\wedge\ \text{VENTE}(-, c, n, -, -) \wedge (q > 100))\}$

(Q5) Donner les noms des clients ayant acheté au moins un produit de couleur verte :
$\{(c) \mid \exists n\ (\text{VENTE}(-, c, n, -, -) \wedge \text{PRODUIT}(n, -, -, \text{“Verte”}))\}$

(Q6) Donner les noms des clients ayant acheté tous les produits stockés :
$\{(c) \mid \forall p\ (\text{PRODUIT}(p, -, -, -) \Rightarrow \text{VENTE}(-, c, p, -, -))\}$

(Q7) Donner les noms des produits fournis par tous les fournisseurs et achetés par au moins un client :
$\{(p) \mid \forall f\ (\text{ACHAT}(-,-, -, -, f) \Rightarrow \text{ACHAT}(-, -, p, -, f))$
$\wedge\ \exists c\ \text{VENTE}(-, c, p, -, -)\}$

4.3. LE LANGAGE QBE

Le langage Query-By-Exemple (QBE) [Zloof77] est conçu pour être utilisé à partir d'un terminal. Il a été développé par M. ZLOFF à IBM Yorkton Heights. Il est com-

mercialisé par IBM au-dessus de DB2 depuis 1983 et plus récemment par de nombreux autres constructeurs avec des présentation variées, par exemple par Microsoft dans ACCESS. Ce langage peut être considéré comme une mise en œuvre visuelle (basée sur des tableaux affichés) du calcul relationnel de domaine.

L'idée de base du langage est de faire formuler une question à l'utilisateur à partir d'un exemple d'une réponse possible à la question. Pour cela, l'utilisateur provoque tout d'abord l'affichage du schéma des tables nécessaires à la question qu'il désire poser (sous forme de squelettes de tables) par frappe du nom des tables correspondantes. Ainsi, des tables vides (avec seulement les en-têtes de colonnes et le nom de la relation associée) apparaissent sur l'écran. Elles correspondent bien sûr aux définitions des prédicats extensionnels de la base logique. Par exemple, il est possible d'afficher le schéma des tables PRODUIT, VENTE et ACHAT comme représenté figure V.4.

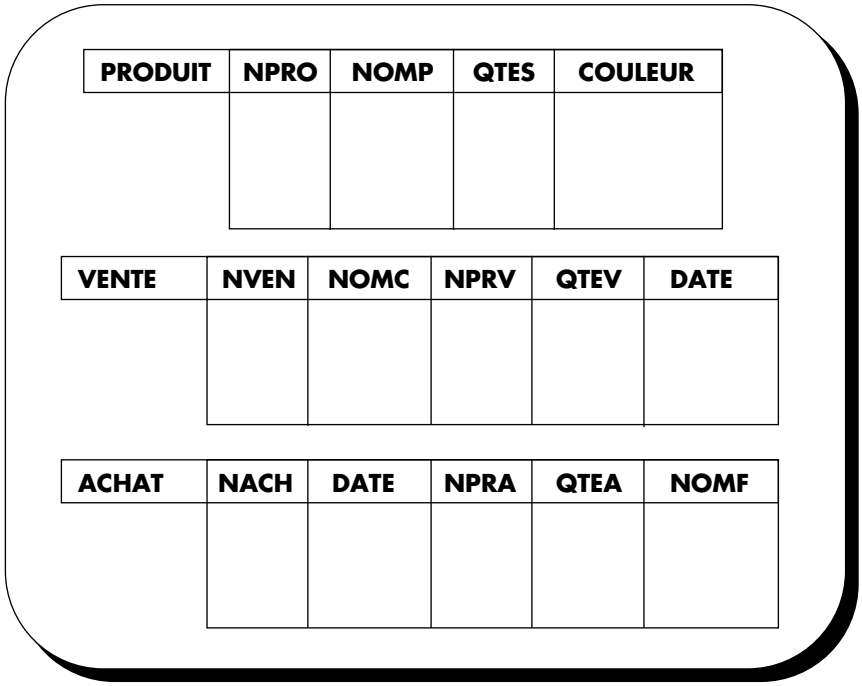


Figure V.4 : Schémas de tables affichés par QBE

Comme indiqué ci-dessus, QBE peut être vu comme une implantation à deux dimensions du calcul relationnel de domaines. Pour cela, les règles suivantes sont utilisées :

1. Les résultats (attributs à projeter en relationnel) sont définis en frappant « P. » (PRINT) dans la colonne associée. Il est possible d'imprimer tous les attributs d'une table en frappant « P. » dans la colonne contenant le nom de la table.

2. Les constantes sont tapées directement dans la colonne de l'attribut associé, précédées de l'opérateur de comparaison les reliant à l'attribut si ce n'est pas = (c'est-à-dire $<$, $\leq$, $>$, $\geq$, $\neq$). Dans le cas où un critère complexe est nécessaire avec des conjonctions (and) et disjonctions (or), il est possible d'ouvrir une boîte condition et de taper le critère dans cette boîte (par exemple A < 10 AND A > 5).
3. Les variables domaines sont désignées par des valeurs exemples soulignées tapées dans la colonne les spécifiant ; la valeur exemple soulignée est en fait le nom de la variable domaine ; par suite, QBE associe deux valeurs soulignées identiques comme définissant une même variable.
4. Le quantificateur « quel-que-soit » est appliqué à une variable en tapant « ALL. » devant son nom (l'exemple souligné) alors que toute variable non imprimée non précédée de P. est implicitement quantifiée par « il-existe ».
5. Le connecteur ou (or) est exprimé en utilisant deux lignes (deux exemples) comme si l'on avait en fait deux questions, l'une avec la partie gauche et l'autre avec la partie droite de la qualification (après mise en forme normale disjonctive).

À l'aide de ces règles simples, il est possible d'exprimer toute question du calcul relationnel de domaines. Nous avons formulé (voir figure V.5) les questions introduites ci-dessus (Q1 à Q7) sur la base des produits. Remarquez l'aspect naturel de ce langage par l'exemple qui peut être simplement paraphrasé.

(a) Noms et couleurs des produits

PRODUIT	NPRO	NOMP	QTES	COULEUR
		P.		P.

(b) Noms et quantités en stock des produits rouges

PRODUIT	NPRO	NOMP	QTES	COULEUR
		P.	P.	Rouge

(c) Noms des produits et des fournisseurs associés

PRODUIT	NPRO	NOMP	QTES	COULEUR
	100	P.		

ACHAT	NACH	DATE	NPRA	QTEA	NOMF
			100		P.

(d) Noms des fournisseurs vendant tous les produits

PRODUIT	NPRO	NOMP	QTES	COULEUR
	ALL.100			

ACHAT	NACH	DATE	NPRA	QTEA	NOMF
			100		P.

(e) Descriptifs des produits rouge ou vert en stock en plus de 1 000 unités

PRODUIT	NPRO	NOMP	QTES	COULEUR
P.			> 1000	—

boite condition
COULEUR = "Rouge" OR COULEUR = "Verte"

Figure V.5 : Exemples de questions QBE

QBE offre quelques possibilités supplémentaires par rapport au calcul de domaines. En particulier, il est possible d'enlever l'élimination automatique des doubles lors des projections finales en spécifiant le mot clé ALL devant une variable résultat à imprimer. Par exemple, l'édition de tous les noms des clients à qui l'on a vendu des produits (sans élimination des doubles) sera effectuée en réponse à la question représentée figure V.6.

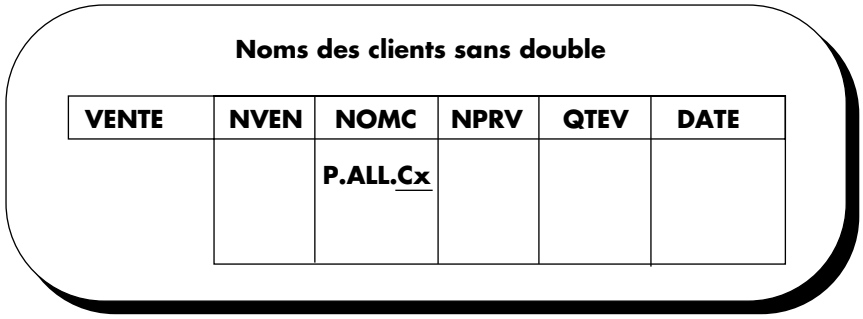


Figure V.6 : Non-élimination des doubles en QBE

Il est également possible d'obtenir des résultats triés par ordre croissant (mot clé AO.) ou décroissant (mot clé DO.). Par exemple, l'édition des noms de produits en stock en quantité supérieure à 100 par ordre alphabétique descendant sera obtenue par exécution de la question représentée figure V.7.

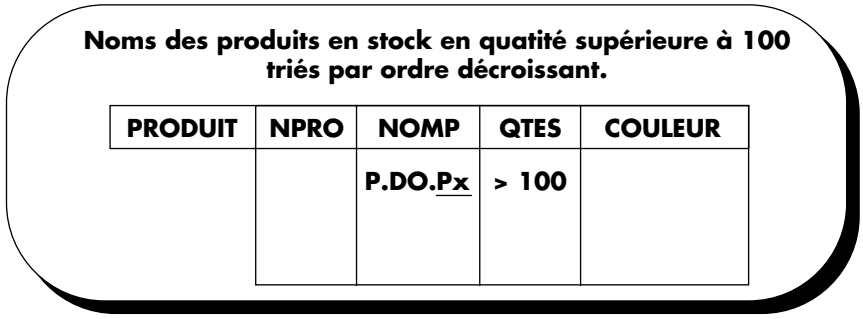


Figure V.7 : Exemple de question avec résultats triés

De plus, QBE offre des facilités pour accomplir les opérations de type fermeture transitive de graphe. Le calcul de la fermeture transitive est impossible avec les langages basés sur le calcul de prédicats. Soit par exemple la relation représentée figure V.8. Tout composant peut aussi être un composé. Si l'on veut rechercher les composants de voiture au deuxième niveau, il est possible avec QBE d'utiliser la question représentée figure V.9.

NOMENCLATURE	COMPOSE	COMPOSANT
	Voiture	Porte
	Voiture	Chassis
	Voiture	Moteur
	Moteur	Piston
	Moteur	Bielle
	Moteur	Chemise
	Piston	Axe
	Piston	Segment

Figure V.8 : Table composé-composant

NOMENCLATURE	COMPOSE	COMPOSANT
	Voiture	**Cx**
	Cx	**P. Cy**

Figure V.9 : Recherche des composants de second niveau

Pour rechercher les composants de niveau n à partir d'un composant (ou les composés de niveau n à partir d'un composé), il faut une question à n lignes. Ceci peut être fastidieux. Afin de simplifier, QBE autorise le mot clé L entre parenthèses précédé d'un nombre de niveaux (par exemple (2L)). Ainsi, la question précédente pourra aussi être formulée comme sur la figure V.10.

NOMENCLATURE	COMPOSE	COMPOSANT
	Voiture	**P. Cy (2 L)**

Figure V.10 : Autre manière de rechercher les composants de second niveau

La fermeture transitive du composé VOITURE consiste à rechercher les composants de tout niveau. Ceci est effectué en fixant un niveau variable, comme représenté

figure V.11. Une telle question n'est pas exprimable à l'aide du calcul relationnel de domaines, mais nécessite la récursion que nous étudierons dans le cadre des BD déductives. Il est aussi possible d'obtenir les composants de dernier niveau à l'aide du mot clé LAST.

NOMENCLATURE	COMPOSE	COMPOSANT
	Voiture	P. Cy (2 L)

Figure V.11 : Fermeture transitive du composé VOITURE

Finalement, QBE dispose également des fonctions d'agrégats qui dépassent la logique du premier ordre. Les fonctions CNT (décompte), SUM (somme), AVG (moyenne), MAX (maximum) et MIN (minimum) permettent de faire des calculs sur plusieurs valeurs de variables, celles-ci étant sélectionnées par des critères exprimés par une question. Ces fonctions peuvent être appliquées à toute variable résultat à condition qu'elle soit précédée de ALL. Si l'on désire éliminer les doubles avant application de la fonction, il faut appliquer avant l'opérateur unique (mot clé UNQ). Par exemple, si l'on veut connaître la quantité en stock des produits de nom « Vins » on écrira la question représentée figure V.12.

Somme des quantités de vins en stock

PRODUIT	NPRO	NOMP	QTES	COULEUR
		"Vins"	P.SUM.ALL.Q	

Figure V.12 : Utilisation de la fonction SUM

QBE permet aussi la mise à jour des prédicats extensionnels, c'est-à-dire des tables. Il est possible :

– d'insérer des n-uplets dans une relation (opérateur I. en première colonne) ; la figure V.13 illustre l'insertion du produit de clé 200 ;

– de modifier des attributs de n-uplet (opérateur U. en première colonne) ; la figure V.14 illustre la mise à jour des quantités en stock pour les produits de couleur rouge ;
– de supprimer les n-uplets dans une relation obéissant à un critère de sélection donné (opérateur « D. » en première colonne) ; la figure V.15 illustre la suppression des produits rouges de la relation produit.

Insertion du tuple 200 dans la relation Produit

PRODUIT	NPRO	NOMP	QTES	COULEUR
I.	200	"Balais"	100	"Bleu"

Figure V.13 : Insertion du tuple de clé 200

Incrément de 100 des quantités en stock de produits rouges

PRODUIT	NPRO	NOMP	QTES	COULEUR
U.	200 200		10 10 + 100	"Rouge"

Figure V.14 : Mise à jour des quantités en stock des produits rouges

Suppression des produits rouges

PRODUIT	NPRO	NOMP	QTES	COULEUR
S.				"Rouge"

Figure V.15 : Suppression des produits rouges

QBE permet enfin une définition très dynamique de la base de données. Il est possible de créer et détruire des prédicats extensionnels (tables). De plus, il est permis d'étendre un prédicat en ajoutant des attributs. QBE est donc un langage très souple et complet, issu de la logique du premier ordre appliquée aux tables relationnelles.

5. LE CALCUL DE TUPLES

Le calcul relationnel de tuples [Codd72] se déduit de la logique du premier ordre. Comme le calcul de domaine, le calcul de tuples permet d'exprimer une question comme une formule. Cette fois, les constantes sont interprétées comme les n-uplets de la base. Les variables parcourent donc les lignes des tables. Afin de distinguer les deux calculs, nous noterons les variables tuples par des majuscules X, Y, Z… Il est d'ailleurs possibles de mixer calcul de domaines et calcul de tuples en utilisant à la fois des variables tuples et des variables domaines [Reiter84].

5.1. PRINCIPES DU CALCUL DE TUPLE

Les seuls termes considérés sont les constantes associées aux n-uplets composant les faits et les fonctions génératrices des attributs notées par le nom de l'attribut appliqué à une variable par la notation pointée (par exemple X.COULEUR). Les prédicats utilisés sont ceux correspondant aux relations extensionnelles ainsi que les prédicats de comparaison $\{=, <, \leq, \geq, >, \neq\}$. Les prédicats extensionnels permettent la définition de la portée d'une variable sur une table R par une formule atomique du type R (X), X étant donc une variable tuple et R une table.

L'élément essentiel différenciant le calcul de tuples par rapport aux calculs de domaines est bien sûr l'association aux variables de tuples des extensions de prédicats et non plus de valeurs de domaines, comme avec le calcul de domaine. Cela change légèrement la syntaxe du calcul. Nous définissons plus précisément le calcul de tuples ci-dessous. La syntaxe du calcul est définie en BNF figure V.16.

Notion V.9 : Calcul relationnel de tuples (*Tuple relational calculus*)

Langage d'interrogation de données formel permettant d'exprimer des questions à partir de formules bien formées dont les variables sont interprétées comme variant sur les tuples d'une table.

```
<question> : := "{" (<résultat>) "|" <formule> "}"
<résultat> : := <variable>.<attribut> | <résultat>, <résultat>
<formule> : := <quantification> <formule libre> | <formule libre>
<quantification> ::= ∀ <variable> | ∃ <variable>
                   | <quantification> <quantification>
<formule libre> : := <formule atomique>
                   | <formule libre> ∧ <formule atomique>
                   | <formule libre> ∨ <formule atomique>
                   | <formule libre> ⇒ <formule libre>
                   | ¬ <formule libre>
                   | (<formule libre>)
<formule atomique> : := <predicat extensionnel> (<variable>)
                   | <terme> <comparateur> <terme>
<terme> : := <variable>.<attribut> | <constante>
<comparateur> : := = | < | ≤ | > | ≥ | ≠
```

Figure V.16 : BNF du calcul relationnel de tuples

5.2. QUELQUES EXEMPLES DE CALCUL DE TUPLE

Nous exprimons maintenant en calcul relationnel de tuples les questions exprimées ci-dessus en calcul de domaines sur la base des produits.

(Q1) Donner la liste des noms et des couleurs de tous les produits :
{(P.NOMP, P.COULEUR) | PRODUIT(P)}

(Q2) Donner les noms et les quantités en stock des produits de couleur rouge :
{(P.NOMP, P.QTES) | PRODUIT(P) ∧ P.COULEUR = "ROUGE"}

(Q3) Donner pour chaque produit en stock, le nom du fournisseur associé :
{(P.NOMP, A.NOMF) | PRODUIT(P)∧ ACHAT(A) ∧ P.NPRO=A.NPRA}

(Q4) Donner, pour chaque produit en stock en quantité supérieure à 100 et de couleur rouge, les couples nom de fournisseurs ayant vendu ce type de produit et nom de client ayant acheté ce type de produit, avec le nom du produit :
{(P.NOMP, A.NOMF, V.NOMC) | PRODUIT(P) ∧ ACHAT(A) ∧ VENTE(V)
∧ P.QTES > 100 ∧ P.COULEUR = "Rouge" ∧ P.NPRO = V.NPRV
∧ P.NPRO = A.NPRA}

(Q5) Donner les noms des clients ayant acheté au moins un produit de couleur verte :
{(V.NOMC) | VENTE (V) ∧ ∃P (PRODUIT (P) ∧ V.NPRV = P.NPRO
∧ P.COULEUR = "Verte")}

(Q6) Donner les noms des clients ayant acheté tous les produits stockés :
{(V_1.NOMC) | VENTE (V_1) ∧ ∀P (PRODUIT (P) ⇒ (∃V_2 (VENTE(V_2)
∧ V_2.NPRV = P.NPRO ∧ V_2.NOMC = V_1.NOMC)))}

(Q7) Donner les noms des produits fournis par tous les fournisseurs et achetés par au moins à un client :

$\{(P.NOMP) \mid PRODUIT(P) \wedge (\forall A_1\ (ACHAT(A_1) \Rightarrow (\exists A_2\ (ACHAT(A_2)$
$\wedge\ A_2.NOMF = A_1.NOMF \wedge A_2.NPRA = P.NPRO)))$
$\wedge\ (\exists V\ (VENTE(V) \wedge V.NPRV = P.NPRO))\}$

Les questions (Q6) et (Q7) démontrent la difficulté d'utilisation des quantificateurs « quel-que-soit » et « il existe ». En général, toute variable apparaissant dans la réponse doit être libre dans la condition. Les « il existe » ne sont utiles qu'à l'intérieur des quantifications universelles.

6. LES TECHNIQUES D'INFÉRENCE

La logique permet la déduction. La déduction, principe de l'intelligence, permet de prouver des théorèmes à partir d'axiomes. Les systèmes de bases de données déductifs sont fondés sur l'inférence. Dans cette section, nous rappelons les principes fondamentaux des techniques de déduction. Ces résultats sont supposés connus lors de l'étude des bases de données déductives, dans la quatrième partie de cet ouvrage.

6.1. PRINCIPE D'UN ALGORITHME DE DÉDUCTION

Un algorithme de déduction est une procédure pour prouver une formule T à partir d'un ensemble de formules {A1, A2…An} connues comme vraies. T est le théorème à démontrer. A1, A2…An sont les axiomes. L'existence d'une preuve de T à partir de A1, A2…An est formellement notée {A1,A2…An} |== T.

Notion V.10 : Déduction (*Deduction*)

Procédure permettant de prouver un théorème à partir d'un ensemble d'axiomes connus comme vrais sur tout domaine de discours considéré.

Par exemple, à partir des axiomes :

DIRIGE(pierre, marie)
DIRIGE(marie, julie)
$\forall x\ \forall y\ \forall z$ (DIRIGE(x,y) $\wedge$ DIRIGE(y,z) $\Rightarrow$ DIRIGE(x,z))

on aimerait déduire le théorème:

DIRIGE(pierre, julie).

Pour prouver un théorème, un algorithme de déduction dérive à partir des axiomes une séquence de formules dont le théorème est la dernière en utilisant des **règles**

d'inférence. Une règle d'inférence est une règle permettant de générer une formule à partir de deux ou plus. Une règle d'inférence correcte permet de générer une formule valide (vraie sur les univers de discours considérés) à partir de formules valides.

Deux règles d'inférences bien connues sont :

– Le **modus ponens**. Il permet de générer P à partir des deux formules F et $F \Rightarrow P$. Intuitivement, cela signifie que si F et F implique P sont prouvées, alors P est prouvée. On écrit formellement :

$$\frac{F \qquad F \Rightarrow P}{P}$$

– La **spécialisation**. Elle permet de générer F(a) à partir de la formule $\forall x F(x)$. Intuitivement, cela signifie que si F(x) est prouvée pour toute valeur de x, alors F(a) est prouvée. On écrit formellement :

$$\frac{\forall x\ F(x)}{F(a)}$$

Il existe une règle d'inférence générale pour les formules du premier ordre en forme clausale. Cette règle génère par application récursive toutes les formules qui peuvent être déduites à partir de deux axiomes sous forme de clauses. Il s'agit de la **règle d'inférence de Robinson** [Robinson65]. Elle s'appuie sur l'**algorithme d'unification** qui permet de comparer deux formules atomiques.

6.2. ALGORITHME D'UNIFICATION

La capacité à décider si deux formules atomiques peuvent être identifiées par substitution de paramètres est centrale à la plupart des méthodes de déduction. Une **unification** de deux formules atomiques consiste à les rendre identiques par remplacement de variables dans une formule par des termes de l'autre.

Notion V.11 : Unification (*Unification*)

Remplacement de variables dans une formule atomique par des termes (constantes, autres variables, fonctions appliquées à des constantes ou des variables) de manière à la rendre identique à une autre formule atomique.

Deux formules atomiques L1(t1,t2…tn) et L2(s1,s2…sn) ne sont pas toujours unifiables. Un algorithme d'unification décide si les deux formules peuvent être identifiées par une substitution de variable valide (renommage ou assignation de valeur). Un tel algorithme (voir figure V.17) retourne :

– SUCCES et une substitution de variable minimale si les deux formules peuvent être identifiées en appliquant la substitution ;

– ECHEC s'il est impossible de les rendre identiques par une substitution de terme unique à chaque variable.

L'algorithme est récursif : il exploite les termes à partir du début en unifiant tête et queue successivement. Il existe beaucoup d'algorithmes d'unification, celui-là n'est qu'indicatif.

Quelques exemples simples permettent d'illustrer l'algorithme. On note {x/t1; y/t2; …} la substitution qui remplace x par t1, y par t2, etc. Les formules DIRIGE(pierre,marie) et DIRIGE(x,y) sont simplement unifiables par la substitution {x/pierre; y/marie}. Les formules DIRIGE(chef(x),pierre) et DIRIGE(chef(chef(pierre)),y) sont unifiables par la substitution {x/chef(pierre); y/pierre}. Les formules DIRIGE(chef(x),x) et DIRIGE(chef(chef(x)),x) ne sont pas unifiables: il faudrait que x devienne à la fois chef(x) et x, ce qui est syntaxiquement impossible.

```
Bool Function Unifier(L1, L2, S) {; // Unification de L1 et L2
   S := ϕ ; // S est la substitution résultat en cas de succès
   Si (L1 est un atome) alors {// unification d'atome
      Si L1 = L2 alors Retour(Succès)
       Si L1 est une variable alors
         Si L1 ∈ L2 alors Retour(Echec)
         Sinon {S : = S ∪ [L1/L2]
               Retour(Succès) } ;
   Si (L2 est un atome) alors {… idem avec L2} ;
   M1 = Elément suivant (L1) ; // prendre éléments suivants
   M2 = Elément suivant (L2) ;
   Suite1 = unifier(M1, M2, S1) ; // tenter unification
   Si (Suite1 = Echec) alors Retour(Echec) ;
   N1 = [Reste(L1)/S1)] ; // substituer si succès
   N2 = [Reste(L1)/S1)] ;
   Suite2 = unifier(N1, N2, S2) ; // unifier le reste
   Si (Suite2 = Echec) alors Retour(Echec) ;
   S = S1 ∪ S2 ; // composer les substitutions
   Retour(Succès) ;}
```

Figure V.17 : Un algorithme d'unification

6.3. MÉTHODE DE RÉSOLUTION

La méthode de résolution est très simplement basée sur la règle d'inférence de Robinson. Cette règle s'énonce comme suit. Soient C1 et C2 deux clauses de la forme :

$C1 = F1 \vee L1$
$C2 = L2 \Rightarrow F2.$

De plus, moyennant une substitution s, L1 et L2 sont unifiables (on note L1[s] = L2[s]). La règle de Robinson permet d'inférer la clause F1[s] ∨ F2[s] ; cette nouvelle clause obtenue par disjonction de F1 et F2 et application de la substitution s est appelée **résolvant** de C1 et C2. Formellement, la règle de Robinson s'écrit :

$$\frac{F1 \vee F(x) \quad L2 \Rightarrow F2 \quad L1[s] = L2[s]}{F1[s] \vee F2\ [s]}$$

En remarquant que L2 ⇒ F2 peut aussi s'écrire ¬ L2 ∨ F2, puis en remplaçant le « ou » par + et la négation par –, on obtient :

$$\frac{F1 + L1 \quad F2 - L2 \quad L1[s] = L2[s]}{F1[s] + F2\ [s]}$$

On voit alors que la règle de Robinson permet finalement de réaliser l'addition de clauses avec suppression des parties s'annulant moyennant une substitution s. Il s'agit de la règle de base du calcul symbolique. Elle a une propriété remarquable de complétude : toute formule pouvant être démontrée à partir d'un ensemble d'axiomes se déduit par applications successives de la règle de Robinson aux axiomes et aux résolvants obtenus.

À partir de cette règle d'inférence est construite la **méthode de résolution**. Cette méthode permet de prouver un théorème à partir d'axiomes non contradictoires. C'est une méthode par l'absurde qui consiste à partir des axiomes et de la négation du théorème et à prouver qu'il y a contradiction. Donc, sous l'hypothèse de non-contradiction des axiomes, c'est que le théorème est valide (car son contraire contredit les axiomes).

En résumé, la méthode procède comme suit :

1. Mettre les axiomes et la négation du théorème (¬T) sous forme de clauses.
2. Ajouter récursivement les résolvants que l'on peut obtenir en appliquant la règle d'inférence de Robinson à l'ensemble des clauses jusqu'à obtention de la clause vide.

La clause vide (tout s'est annulé) ne peut jamais être satisfaite (son modèle est vide) ; par suite, c'est que les axiomes contredisent la négation du théorème. Donc celui-ci est démontré. Il a été démontré que si une preuve du théorème existe, la méthode de résolution se termine et la trouve. Si aucune preuve n'existe, la méthode peut se perdre dans des univers infinis et boucler (de plus en plus de clauses sont générées). La logique du premier ordre est semi-décidable. Pour un approfondissement de cette méthode, consultez [Manna85].

Nous nous contenterons d'illustrer la méthode par un arbre de preuve (figure V.18). Soit à prouver le théorème DIRIGE(pierre, julie) à partir des axiomes non contradictoires :

(A1) DIRIGE(pierre, marie),
(A2) DIRIGE(marie, julie),

(A3) $\forall x \forall y \forall z$ (DIRIGE(x,y) $\wedge$ DIRIGE(y,z) $\Rightarrow$ DIRIGE(x,z)).

La négation du théorème est ¬ DIRIGE(pierre,julie). Les deux premiers sont des clauses. Le troisième se met simplement sous la forme de la clause :

(A'3) DIRIGE(x,y) $\wedge$ DIRIGE(y,z) $\Rightarrow$ DIRIGE(x,z)

qui s'écrit encore :

(A''3) ¬DIRIGE(x,y) $\vee$ ¬DIRIGE(y,z) $\vee$ DIRIGE(x,z).

L'arbre de preuve (encore appelé arbre de réfutation) représenté figure V.18 montre des unifications et additions successives de clauses qui conduisent à la clause vide. Il permet donc par applications successives de la règle d'inférence de Robinson (chaque nœud R non initial dérive de deux nœuds précédents N1 et N2) de tirer le résolvant vide et ainsi de démontrer que Pierre dirige Julie.

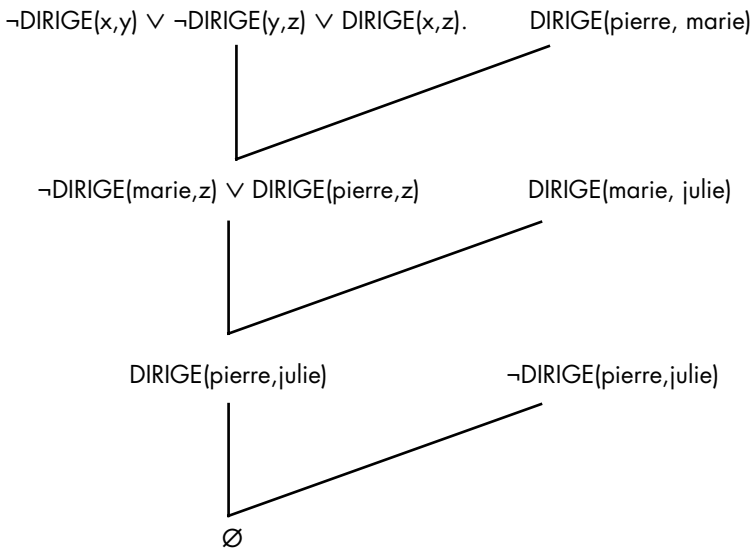


Figure V.18 : Exemple d'arbre de preuve

7. CONCLUSION

Nous avons dans ce chapitre rappelé les concepts de base de la logique du premier ordre. Une base de données peut être vue comme l'interprétation d'un langage logique. Cette vision est limitée et nous irons beaucoup plus loin avec les bases de données déductives, traitées dans la 4e partie de cet ouvrage.

Quoi qu'il en soit, la logique constitue une bonne base pour comprendre les bases de données, en particulier les BD relationnelles. Une BD relationnelle est un ensemble de tables donnant les extensions valides des prédicats. Le calcul de tuples et le calcul de domaines sont des formalisations logiques des langages d'interrogation des bases de données relationnelles. Nous allons étudier le modèle relationnel indépendamment de la logique dans la partie qui suit, mais il ne faut pas perdre de vue que la logique est son fondement.

8. BIBLIOGRAPHIE

[Ceri91] Ceri S., Gottlob G., Tanca L., *Logic Programming and Databases*, Surveys in Computer Sciences, Springer Verlag, 1990.

Un livre fondamental sur le modèle logique. Le livre introduit Prolog comme un langage d'interrogation de données, les bases de données relationnelles vues d'un point de vue logique, et enfin les couplages de Prolog et des bases de données. Dans une deuxième partie, Datalog et ses fondements sont présentés. La troisième partie est consacrée aux techniques d'optimisation de Datalog et à un survol des prototypes implémentant ces techniques.

[Clark78] Clark C. « Negation as failure » in *Logic and databases*, Édité par Gallaire et Minker, Plenum Press, New York, 1978.

Un article de base sur la négation en programmation logique. Il est proposé d'affirmer qu'un fait est faux s'il ne peut être démontré vrai (négation par échec). Cela conduit à interpréter les règles comme des équivalences : « si » peut être lu comme « si et seulement si » à condition de collecter toutes les règles menant à un même but en une seule.

[Clocksin81] Clocksin W.F., Mellish C.S., *Programming in Prolog*, Springer Verlag, Berlin-Heidelberg-New York, 1981.

Un livre de base sur le langage Prolog. Prolog utilise des bases de faits en mémoire qui sont similaires aux bases logiques décrites dans ce chapitre. En plus, Prolog gère la déduction.

[Codd72] Codd E.F., « Relational Completness of Data Base Sublanguages », In *Data Base Systems, Courant Computer Science Symposia Series*, n° 6, Prentice-Hall, 1972.

Cet article introduit la notion de complétude pour un langage d'interrogation de bases de données relationnelles. Il donne la définition du calcul relationnel de tuple et de l'algèbre relationnelle. Il démontre que l'algèbre est aussi puissante que le calcul en donnant un algorithme de traduction de calcul en

algèbre. Codd argumente aussi sur les avantages du calcul comme base d'un langage utilisateur.

[Gallaire78] Gallaire H., Minker J., *Logic and Databases,* Plenum Press, 1978.

Le premier livre de base sur la logique et les bases de données. Ce livre est une collection d'articles présentés à Toulouse lors d'un « workshop » tenu en 1977 sur le sujet.

[Gallaire81] Gallaire H., Minker J., Nicolas J.M., *Advances in database theory*, vol. 1, Plenum Press, 1981.

Le second livre de base sur la logique et les bases de données. Ce livre est une collection d'articles présentés à Toulouse lors d'un « workshop » tenu en 1979 sur le sujet.

[Gallaire84] Gallaire H., Minker J., Nicolas J.M., « *Logic and databases: a deductive approach* », ACM Computing Surveys, vol. 16, n° 2, juin 1984.

Un article de synthèse sur les bases de données et la logique. Différents types de clauses (fait positif ou négatif, contrainte d'intégrité, loi déductive) sont isolés. L'interprétation des bases de données comme un modèle ou comme une théorie de la logique est discutée. Les différentes variantes de l'hypothèse du monde fermé sont résumées.

[Lloyd87] Lloyd J., *Foundations of logic programming*, 2e édition, Springer Verlag Ed., 1987.

Le livre de base sur les fondements de la programmation logique. Les différentes sémantiques d'un programme logique sont introduites. Les techniques de preuve de type résolution avec négation par échec sont développées.

[Manna85] Manna Z., Waldinger R., *The Logical Basis for Computer Programming*, vol. 1, 618 pages, Addison-Wesley, 1985.

Le livre de base sur la logique. La syntaxe, la sémantique et les méthodes de preuves pour la logique du premier ordre sont développées. La logique du deuxième ordre, où des variables peuvent représenter des prédicats, est aussi étudiée.

[Merrett84] Merrett T.H., *Relational Information Systems*, Prentice Hall, 1984, chapitre 5.

Un bon livre très synthétique sur les bases de données, avec de nombreux exemples pratiques. La relation composant-composé (et plus généralement l'application Facture de Matériel – « Bill of Material ») est introduite.

[Nilsson80] Nilsson N., *Principles of Artificial Intelligence*, Tioga Publication, 1980.

Un des livres de base de l'intelligence artificielle. Les systèmes de règles de production pertinents aux bases de données déductives sont particulièrement développés.

[Reiter78] Reiter R., « On closed world data bases », in *Logic and databases*, Édité par Gallaire et Minker, Plenum Press, New York, 1978.

L'article de base sur l'hypothèse du monde fermé.

[Reiter84] Reiter R., « Towards a Logical Reconstruction of Relational Database Theory », in *On Conceptual Modelling*, p. 191-234, Springer-Verlag Ed., 1984.

Une redéfinition des bases de données relationnelles en logique. Les différents axiomes nécessaires à l'interprétation d'une base de données comme une théorie en logique sont présentés: fermeture des domaines, unicité des noms, axiomes d'égalité, etc. Les calculs relationnels sont unifiés et généralisés.

[Robinson65] Robinson J.A., « A Machine oriented Logic based on the Resolution Principle », *Journal of the ACM*, 12, 1965.

L'article introduisant la méthode de résolution.

[Ullman88] Ullman J.D., *Principles of Database and Knowledge-base Systems*, vol. I et II, Computer Science Press, 1988.

Deux volumes très complets sur les bases de données, avec une approche plutôt fondamentale. Jeffrey Ullman détaille tous les aspects des bases de données, depuis les méthodes d'accès jusqu'aux modèles objets en passant par le modèle logique. Les livres sont très centrés sur une approche par la logique des bases de données. Les calculs de tuples et domaines, les principaux algorithmes d'accès, d'optimisation de requêtes, de concurrence, de normalisation, etc. sont détaillés.

[Zloof77] Zloof M., « Query-by-Example: A Data Base Language », *IBM Systems Journal*, vol. 16, n° 4, 1977, p. 324-343.

Cet article présente QBE, le langage par grille dérivé du calcul de domaines proposé par Zloof, alors chercheur à IBM. Ce langage bi-dimensionnel est aujourd'hui opérationnel en sur-couche de DB2 et aussi comme interface externe du système Paradox de Borland. Zloof discute aussi des extensions bureautiques possibles, par exemple pour gérer le courrier (OBE).

Partie 2

LE RELATIONNEL

VI – Le modèle relationnel (*The relational model*)

VII – Le langage SQL2 (*The SQL2 language*)

VIII – Intégrité et BD actives (*Integrity and Triggers*)

IX – La gestion des vues (*View management)*

X – L'optimisation de questions (*Query optimization*)

Chapitre VI

LE MODÈLE RELATIONNEL

1. INTRODUCTION : LES OBJECTIFS DU MODÈLE

Le modèle relationnel a été introduit par E. F. Codd [Codd70], qui travaillait au fameux centre de recherche d'IBM San-José et s'opposait au développement du modèle DIAM, un modèle utilisant des entités liées par de multiples pointeurs. La première volonté du modèle relationnel fut d'être un modèle ensembliste simple, supportant des ensembles d'enregistrements aussi bien au niveau de la description que de la manipulation. Les premières idées d'un modèle ensembliste avaient été proposées un peu avant, notamment dans [Childs68]. Le modèle relationnel est aujourd'hui la base de nombreux systèmes, et les architectures permettant d'accéder depuis une station de travail à des serveurs de données s'appuient en général sur lui. Le relationnel a donc atteint ses objectifs au-delà de toute espérance.

Les premiers objectifs du modèle ont été formulés par E. F. Codd [Codd70] comme suit :

1. Permettre un haut degré d'indépendance des programmes d'applications et des activités interactives à la représentation interne des données, en particulier aux choix des ordres d'implantation des données dans les fichiers, des index et plus généralement des chemins d'accès.
2. Fournir une base solide pour traiter les problèmes de cohérence et de redondance des données.

Ces deux objectifs, qui n'étaient pas atteints par les modèles réseau et hiérarchique, ont été pleinement satisfaits par le modèle relationnel, d'une part grâce à la simplicité des vues relationnelles qui permettent de percevoir les données sous forme de tables à deux dimensions, et d'autre part grâce aux règles d'intégrité supportées par le modèle et ses fondements logiques.

En addition aux objectifs fixés par E. F. Codd, le modèle relationnel a atteint un troisième objectif :

3. Permettre le développement de langages de manipulation de données non procéduraux basés sur des théories solides.

Ce troisième objectif est atteint d'une part à l'aide de l'algèbre relationnelle qui permet de manipuler des données de manière très simple et formelle – comme les opérateurs arithmétiques permettent de manipuler des entiers –, et d'autre part à l'aide des langages assertionnels basés sur la logique qui permettent de spécifier les données que l'on souhaite obtenir sans dire comment les obtenir.

Finalement, le modèle relationnel a atteint deux autres objectifs non prévus initialement :

4. Être un modèle extensible permettant de modéliser et de manipuler simplement des données tabulaires, mais pouvant être étendu pour modéliser et manipuler des données complexes.
5. Devenir un standard pour la description et la manipulation des bases de données.

L'objectif 4 est très important, car il a permis d'intégrer de nouveaux concepts au modèle relationnel, notamment la plupart des concepts de l'orienté objets que nous étudierons dans la troisième partie de cet ouvrage. Les premiers travaux de recherche dans ce sens ont été effectués par Codd lui-même [Codd79], puis poursuivis à Bell Laboratories [Zaniolo83], à Berkeley [Stonebraker87] et, en France, à l'INRIA [Gardarin89].

L'objectif 5 a été réalisé en particulier grâce à IBM : le modèle relationnel et son langage SQL ont été normalisés au niveau international en 1986 [ISO89]. Nous ferons un point sur le langage SQL et sa standardisation dans le chapitre suivant.

Dans ce chapitre, nous allons tout d'abord présenter les concepts structuraux de base du modèle relationnel qui permettent de modéliser les données sous forme de tables à deux dimensions. Nous exposerons ensuite les règles de cohérence des relations qui sont considérés comme partie intégrante du modèle. Puis nous introduirons l'algèbre relationnelle, qui est l'outil formel indispensable pour manipuler des relations. Les nombreuses extensions de cette algèbre seront également présentées. En conclusion, nous démontrerons les vastes possibilités d'enrichissement offertes par le modèle relationnel, possibilités qui seront étudiées plus en détail au niveau des modèles objets et logiques, sujets d'autres parties de ce livre.

2. LES STRUCTURES DE DONNEES DE BASE

2.1. DOMAINE, ATTRIBUT ET RELATION

Le modèle relationnel est fondé sur la théorie mathématique bien connue des relations. Cette théorie se construit à partir de la théorie des ensembles. Trois notions sont importantes pour introduire les bases de données relationnelles. La première permet de définir les ensembles de départ. Ces ensembles sont les **domaines** de valeurs.

Notion VI.1 : Domaine (*Domain*)

Ensemble de valeurs caractérisé par un nom.

Les domaines sont donc les ensembles dans lesquels les données prennent valeur. Comme un ensemble, un domaine peut être défini en extension, en donnant la liste des valeurs composantes, ou en intention, en définissant une propriété caractéristique des valeurs du domaine. Au départ, les domaines `ENTIER, REEL, BOOLEEN, CARACTERES` (une chaîne de caractères de longueur fixe ou variable) sont définis en intention. À partir de ces domaines, il est possible de définir en intention des domaines plus spécifiques tels que `MONNAIE` (réel avec 2 chiffres derrière la virgule), `DATE` (entier de 6 chiffres jour, mois et an), `TEMPS` (heure en secondes), etc. Un domaine peut toujours être défini en extension, par exemple le domaine des couleurs de vins : `COULEUR-VINS = {Rosé, Blanc, Rouge}`.

Rappelons que le produit cartésien d'un ensemble de domaines `D1, D2...`, Dn est l'ensemble des vecteurs `<v1, v2..., vn>` où, pour i variant de 1 à n, `vi` est une valeur de Di. Par exemple, le produit cartésien des domaines `COULEUR-VINS = {ROSE, BLANC, ROUGE}` et `CRUS = {VOLNAY, SANCERRE, CHABLIS}` est composé des neuf vecteurs représentés figure VI.1.

`COULEUR-VINS = {ROSE, BLANC, ROUGE}`

`CRUS = {VOLNAY, SANCERRE, CHABLIS}`

ROSE	VOLNAY
ROSE	SANCERRE
ROSE	CHABLIS
BLANC	VOLNAY
BLANC	SANCERRE
BLANC	CHABLIS
ROUGE	VOLNAY
ROUGE	SANCERRE
ROUGE	CHABLIS

Figure VI.1 : Exemple de produit cartésien

Nous pouvons maintenant introduire la notion de **relation**, à la base du modèle.

Notion VI.2 : Relation (*Relation*)

Sous-ensemble du produit cartésien d'une liste de domaines caractérisé par un nom.

Sous-ensemble d'un produit cartésien, une relation est composée de vecteurs. Une représentation commode d'une relation sous forme de table à deux dimensions a été retenue par Codd. Elle est généralement utilisée. Chaque ligne correspond à un vecteur alors que chaque colonne correspond à un domaine du produit cartésien considéré ; un même domaine peut bien sûr apparaître plusieurs fois. Par exemple, à partir des domaines `COULEURS_VINS` et `CRUS`, il est possible de composer la relation `COULEURS_CRUS` représentée sous forme tabulaire figure VI.2.

COULEURS_CRUS	Couleur	Cru
	ROSE	SANCERRE
	ROSE	CHABLIS
	BLANC	SANCERRE
	ROUGE	VOLNAY
	ROUGE	SANCERRE
	ROUGE	CHABLIS

Figure VI.2 : Un premier exemple de relation

Pour pouvoir distinguer les colonnes d'une relation sans utiliser un index, et ainsi rendre leur ordre indifférent tout en permettant plusieurs colonnes de même domaine, il est nécessaire d'associer un nom à chaque colonne. Une colonne se distingue d'un domaine en ce qu'elle prend valeur dans un domaine et peut à un instant donné comporter seulement certaines valeurs du domaine. Par exemple, si le domaine est l'ensemble des entiers, seules quelques valeurs seront prises à un instant donné, par exemple {10, 20, 30}. L'ensemble des valeurs d'une colonne de relation est en général fini. Afin de bien distinguer domaine et colonne, on introduit la notion d'**attribut** comme suit :

Notion VI.3 : Attribut (*attribute*)

Colonne d'une relation caractérisée par un nom.

Le nom associé à un attribut est souvent porteur de sens. Il est donc en général différent de celui du domaine qui supporte l'attribut. Ainsi, la première colonne de la relation COULEUR-`CRUS` pourra être appelée simplement `COULEUR` et la deuxième `CRU`. `COULEUR` et `CRU` seront donc deux attributs de la relation `COULEUR-CRUS`.

Les lignes d'une relation correspondent à des n-uplets de valeurs. Une ligne est aussi appelée **tuple** (du mot anglais *tuple*) : un tuple correspond en fait à un enregistrement dans une relation (encore appelée table).

Notion VI.4 : Tuple (*tuple*)

Ligne d'une relation correspondant à un enregistrement.

La notion de relation est bien connue en mathématiques : une relation n-aire est un ensemble de n-uplets. Un cas particulier souvent employé est la relation binaire, qui est un ensemble de paires ordonnées. Une relation n-aire est souvent représentée par un graphe entre les ensembles composants. Ainsi, la relation `LOCALISATION`, donnant la région de chaque cru et certains prix moyens des vins associés, est représentée par un graphe figure VI.3.

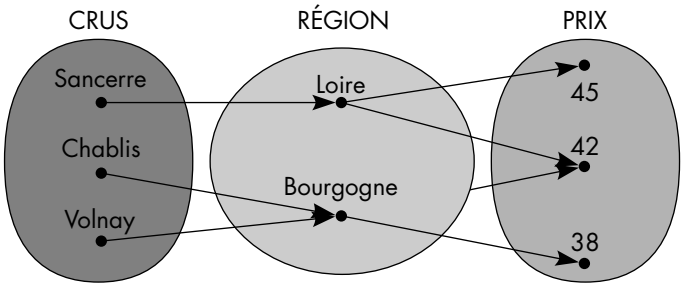


Figure VI.3 : Graphe de la relation `LOCALISATION`

Une relation peut aussi être représentée sur un diagramme à n dimensions dont les coordonnées correspondent aux domaines participants. Ainsi, la relation `STATISTIQUE`, d'attributs `AGE` et `SALAIRE`, donnant la moyenne des salaires par âge, est représentée figure VI.4.

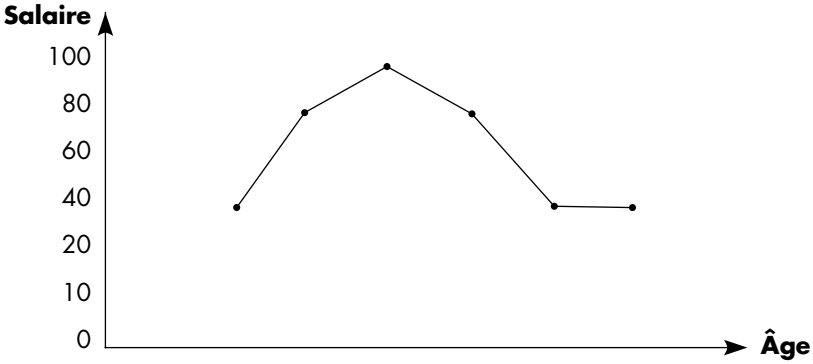


Figure VI.4 : Diagramme représentant une relation binaire

Une autre perception mathématique d'une relation consiste à voir chaque tuple t comme une fonction t : `Ai -> Di pour i = 1, 2... n` qui applique donc les attributs sur les domaines. Ainsi, une relation peut être vue comme un ensemble fini de fonctions. Bien sûr, cet ensemble varie en fonction du temps. Tout ceci montre la diversité des outils mathématiques qui peuvent être appliqués aux relations. En conséquence, le modèle relationnel peut apparaître comme un modèle mathématique simple des données. De plus, grâce à la représentation tabulaire, il est simple à appréhender pour les non mathématiciens.

2.2. EXTENSIONS ET INTENTIONS

Comme tous les modèles de données, le modèle relationnel permet de décrire des données dont les valeurs varient en fonction du temps. Les relations varient en fonction du temps en ce sens que des tuples sont ajoutés, supprimés et modifiés dans une relation au cours de sa vie. Cependant, la structure d'une relation caractérisée par les trois concepts de domaine, relation et attribut est un invariant pour la relation (elle ne change pas en fonction du temps). Cette structure est capturée dans le **schéma de la relation**.

Notion VI.5 : Schéma de relation (*Relation Schema*)

Nom de la relation suivi de la liste des attributs et de la définition de leurs domaines.

Un schéma de relation est noté sous la forme :

```
R (A1 : D1, A2 : D2..., Ai : Di... An : Dn)
```

R est le nom de la relation, Ai les attributs et Di les domaines associés. À titre d'exemple, la figure VI.5 propose deux schémas de relations : celui de la relation `LOCALISATION` introduite ci-dessus et celui de la relation `VINS` qui représente des vins caractérisés par un numéro, un cru, un millésime et un degré. Les domaines utilisés sont ceux des entiers, des réels et des chaînes de caractères de longueur variable (`CHARVAR`). La plupart du temps, lorsqu'on définit le schéma d'une relation, les domaines sont implicites et découlent du nom de l'attribut ; aussi omet-on de les préciser.

```
LOCALISATION (CRU : CHARVAR, REGION : CHARVAR, PRIX : FLOAT)
VINS (NV :INTEGER, CRU : CHARVAR, MILL :INTEGER, DEGRE : FLOAT)
```

Figure VI.5 : Schémas des relations `LOCALISATION` et `VINS`

Soulignons que le schéma d'une relation représente son **intention**, c'est-à-dire les propriétés (au moins certaines) communes et invariantes des tuples qu'elle va contenir

au cours du temps. Au contraire, une table représente une **extension** d'une relation (voir figure VI.2 par exemple), c'est-à-dire une vue des tuples qu'elle contient à un instant donné. Une extension d'une relation R est aussi appelée **instance** de R. L'intention est le résultat de la description des données, alors qu'une extension (ou instance) fait suite à des manipulations et représente un état de la base.

3. LES RÈGLES D'INTÉGRITÉ STRUCTURELLE

Les règles d'intégrité sont les assertions qui doivent être vérifiées par les données contenues dans une base. Il est possible de distinguer les règles structurelles qui sont inhérentes au modèle de données, c'est-à-dire nécessaires à sa mise en œuvre, et les règles de comportement propres au schéma particulier d'une application. Le modèle relationnel impose a priori une règle minimale qui est l'unicité des clés, comme nous allons le voir ci-dessous. Il est commode et courant [Date81] d'ajouter trois types de règles d'intégrité supplémentaires afin d'obtenir les règles d'intégrité structurelle supportées par le modèle relationnel : les contraintes de références, les contraintes d'entité et les contraintes de domaine.

3.1. UNICITÉ DE CLÉ

Par définition, une relation est un ensemble de tuples. Un ensemble n'ayant pas d'élément en double, il ne peut exister deux fois le même tuple dans une relation. Afin d'identifier les tuples d'une relation sans donner toutes les valeurs et d'assurer simplement l'unicité des tuples, la notion de **clé** est utilisée.

Notion VI.6 : Clé (*Key*)

Ensemble minimal d'attributs dont la connaissance des valeurs permet d'identifier un tuple unique de la relation considérée.

De manière plus formelle, une clé d'une relation `R` est un ensemble d'attributs K tel que, quels que soient les tuples `t1` et `t2` d'une instance de `R`, `t1(K)` ≠ `t2(K)`, c'est-à-dire que `t1` et `t2` ont des valeurs de K différentes. Un ensemble d'attributs contenant une clé est appelée super-clé [Ullman88].

Toute relation possède au moins une clé car la connaissance de tous les attributs permet d'identifier un tuple unique. S'il existe plusieurs clés, on en choisit en général une

arbitrairement qui est appelée **clé primaire.** Par exemple, NV peut constituer une clé primaire pour la relation VINS. Le couple <CRU,MILLESIME> est une clé alternative.

Soulignons que la notion de clé caractérise l'intention d'une relation : dans toute extension possible d'une relation, il ne peut exister deux tuples ayant même valeur pour les attributs clés. La détermination d'une clé pour une relation nécessite donc une réflexion sur la sémantique de la relation, c'est-à-dire sur toutes les extensions possibles et non pas sur une extension particulière.

3.2. CONTRAINTES DE RÉFÉRENCES

Le modèle relationnel est souvent utilisé pour représenter des entités du monde réel qui sont les objets ayant une existence propre, et des associations entre ces objets [Chen76]. Une entité correspond alors à un tuple dans une relation qui comporte à la fois la clé de l'entité et ses caractéristiques sous forme d'attributs. Une entité est identifiée par la valeur de sa clé. Un type d'association est modélisé par une relation comportant les clés des entités participantes ainsi que les caractéristiques propres à l'association.

À titre d'exemple, nous considérons les entités BUVEURS et VINS du monde réel : la clé de l'entité BUVEURS est le numéro de buveur NB et celles de l'entité VINS le numéro de vin NV. Ces entités peuvent être associées par une consommation d'un vin par un buveur : une consommation sera par exemple modélisée par une association ABUS entre le buveur et le vin. Cette base typique et simple, qui servira souvent d'exemple, est appelée DEGUSTATION. Le diagramme entité-association de cette base est représenté figure VI.6.

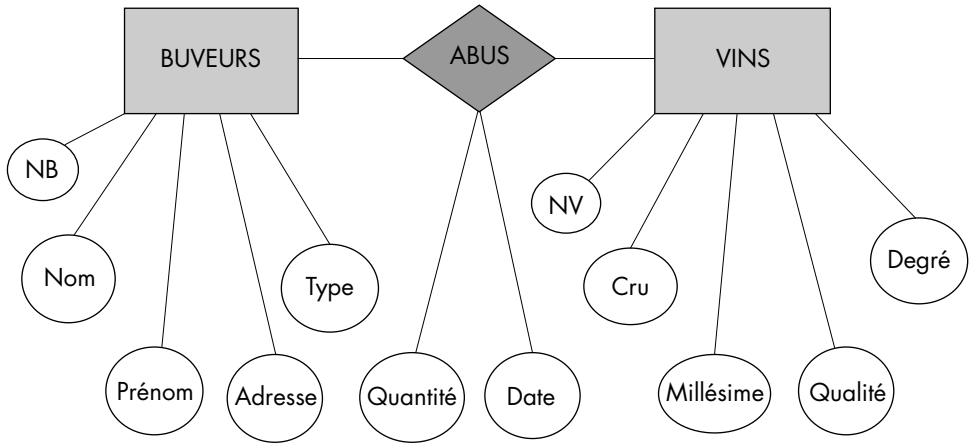


Figure VI.6 : Diagramme entité-association de la base DEGUSTATION

En relationnel, chaque entité est représentée par une table. Une association est aussi représentée par une table, dont les attributs seront les clés des entités participantes, c'est-à-dire NB et NV, ainsi que les attributs caractérisant l'association, par exemple la date et la quantité bue. On aboutit ainsi au schéma relationnel représenté figure VI.7.

```
BUVEURS (NB, NOM, PRENOM, ADRESSE, TYPE)
VINS (NV, CRU, MILL QUALITE, DEGRE)
ABUS (NB, NV, DATE, QUANTITE)
```

Figure VI.7 : Schéma relationnel de la base DEGUSTATION

L'utilisation du modèle relationnel pour représenter des entités et des associations ne permet pas jusque-là de représenter le fait que l'association entre une consommation et un vin est obligatoire, c'est-à-dire que tout abus doit correspondre à un vin existant dans la base. De même, le fait que le lien entre une consommation et un buveur soit obligatoire est perdu. Le maintient de liens obligatoires a justifié l'introduction de la notion de **contrainte référentielle** [Date81].

Notion VI.7 : Contrainte référentielle (*Referential constraint*)

Contrainte d'intégrité portant sur une relation R1, consistant à imposer que la valeur d'un groupe d'attributs apparaisse comme valeur de clé dans une autre relation R2.

Une telle contrainte d'intégrité s'applique en général aux associations obligatoires : une telle association ne peut exister que si les entités participant à l'association existent déjà. Notez que dans la définition, R1 et R2 ne sont pas forcément distinctes : l'association peut en effet porter sur deux entités de même type, par exemple entre une personne et ses parents.

La représentation de contraintes de référence peut s'effectuer par la définition de **clés étrangères** dans une relation : une clé étrangère est un groupe d'attributs qui doit apparaître comme clé dans une (autre) relation. Par exemple, la relation ABUS (NB, NV, DATE, QUANTITE) a pour clé <NB, NV, DATE> et a deux clés étrangères, NB dans BUVEURS et NV dans VINS.

Les contraintes référentielles définissent des liens obligatoires entre relations. Ce sont des contraintes très fortes qui conditionnent le succès des opérations de mises à jour. Lors de l'insertion d'un tuple dans une relation référençante, il faut vérifier que les valeurs de clés étrangères existent dans les relations référencées. Par exemple, lors de l'insertion d'un abus, il faut que le vins et le buveurs associés existent, sinon l'insertion est refusée pour cause de violation d'intégrité. Lors de la suppression d'un tuple dans une relation référencée, il faut vérifier qu'aucun tuple n'existe dans chaque relation référençante ; s'il en existe, le système peut soit refuser la suppression, soit la

répercuter en cascade (c'est-à-dire, supprimer les tuples référençants). Par exemple, la suppression d'un vin entraînera la vérification qu'aucun abus ne référence ce vin. Les contraintes d'intégrité référentielles sont donc des liens forts qui rendent les relations dépendantes ; en quelque sorte, elles introduisent des liens hiérarchiques depuis les relation référencées vers les relations référençantes.

3.3. VALEURS NULLES ET CLÉS

Lors de l'insertion de tuples dans une relation, il arrive fréquemment qu'un attribut soit inconnu ou non applicable ; par exemple, la quantité de vin bu par un buveur à une certaine date peut être inconnue ; si un vin ne possède pas de millésime, l'attribut millésime est inapplicable à ce vin. On est alors amené à introduire dans la relation une valeur conventionnelle, appelée **valeur nulle**.

Notion VI.8 : Valeur nulle (*Null value*)

Valeur conventionnelle introduite dans une relation pour représenter une information inconnue ou inapplicable.

La signification précise d'une valeur nulle est souvent ambiguë ; le groupe ANSI/X3/SPARC a recensé quatorze significations possibles, parmi lesquelles valeur inconnue et valeur inapplicable sont les plus caractéristiques.

Tout attribut dans une relation ne peut prendre une valeur nulle ; en effet, l'existence d'une clé unique impose la connaissance de la clé afin de pouvoir vérifier que cette valeur de clé n'existe pas déjà. Ainsi, une contrainte structurelle du modèle relationnel est la **contrainte d'entité** définie ci-dessous [Date81].

Notion VI.9 : Contrainte d'entité (*Entity constraint*)

Contrainte d'intégrité imposant que toute relation possède une clé primaire et que tout attribut participant à cette clé primaire soit non nul.

À moins qu'il n'en soit spécifié autrement par une contrainte sémantique, le modèle relationnel n'impose pas que les clés étrangères qui n'appartiennent pas à une clé primaire soient non nulles. Cela peut permettre une certaine souplesse, par exemple d'enregistrer des employés qui ne sont attachés à aucun service.

3.4. CONTRAINTES DE DOMAINES

En théorie, une relation est construite à partir d'un ensemble de domaines. En pratique, les domaines gérés par les systèmes sont souvent limités aux types de base

entier, réel, chaîne de caractères, parfois monnaie et date. Afin de spécialiser un type de données pour composer un domaine plus fin (par exemple, le domaine des salaires mensuels qui sont des réels compris entre 6 000 et 1 000 000 de francs), la notion de **contrainte de domaine** est souvent ajoutée aux règles d'intégrité structurelle du relationnel. Cette notion peut être introduite comme suit :

Notion VI.10 : Contrainte de domaine (*Domain constraint*)

Contrainte d'intégrité imposant qu'une colonne d'une relation doit comporter des valeurs vérifiant une assertion logique.

L'assertion logique est l'appartenance à une plage de valeurs ou à une liste de valeurs. Par exemple, `SALAIRE ≥ 5 000 et ≤ 1 000 000, COULEUR ∈ {BLEU, BLANC, ROUGE}`, etc. Les contraintes permettent de contrôler la validité des valeurs introduites lors des insertions ou mises à jour. La non-nullité d'une colonne peut aussi être considérée comme une contrainte de domaine, par exemple `DEGRE IS NULL`.

4. L'ALGÈBRE RELATIONNELLE : OPÉRATIONS DE BASE

L'algèbre relationnelle a été inventée par E. Codd comme une collection d'opérations formelles qui agissent sur des relations et produisent les relations en résultats [Codd70]. On peut considérer que l'algèbre relationnelle est aux relations ce qu'est l'arithmétique aux entiers. Cette algèbre, qui constitue un ensemble d'opérations élémentaires associées au modèle relationnel, est sans doute une des forces essentielles du modèle. Codd a initialement introduit huit opérations, dont certaines peuvent être composées à partir d'autres. Dans cette section, nous allons introduire six opérations qui permettent de déduire les autres et qui sont appelées ici opérations de base. Nous introduirons ensuite quelques opérations additionnelles qui sont parfois utilisées. Des auteurs ont proposé d'autres opérations qui peuvent toujours se déduire des opérations de base [Delobel83, Maier83].

Les opérations de base peuvent être classées en deux types : les opérations ensemblistes traditionnelles (une relation étant un ensemble de tuples, elle peut être traitée comme telle) et les opérations spécifiques. Les opérations ensemblistes sont des opérations binaires, c'est-à-dire qu'à partir de deux relations elles en construisent une troisième. Ce sont l'union, la différence et le produit cartésien. Les opérations spécifiques sont les opérations unaires de projection et restriction qui, à partir d'une relation, en construisent une autre, et l'opération binaire de jointure. Nous allons définir toutes ces opérations plus précisément.

4.1. LES OPÉRATIONS ENSEMBLISTES

4.1.1. Union

L'**union** est l'opération classique de la théorie des ensembles adaptée aux relations de même schéma.

Notion VI.11 : Union (*Union*)

Opération portant sur deux relations de même schéma `RELATION1` *et* `RELATION2` *consistant à construire une relation de même schéma* `RELATION3` *ayant pour tuples ceux appartenant à* `RELATION1` *ou* `RELATION2` *ou aux deux relations.*

Plusieurs notations ont été introduites pour cette opération, selon les auteurs :

– `RELATION1 ∪ RELATION2`

– `UNION (RELATION1, RELATION2)`

– `APPEND (RELATION1, RELATION2)`

La notation graphique représentée figure VI.8 est aussi utilisée. À titre d'exemple, l'union des relations `VINS1` et `VINS2` est représentée figure VI.9.

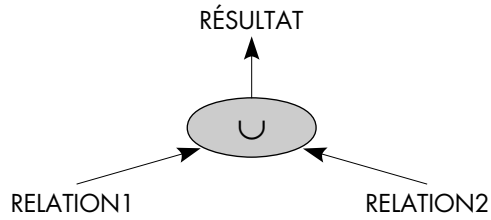


Figure VI.8 : Représentation graphique de l'union

Vins1	Cru	Mill	Région	Couleur
	CHENAS	1983	BEAUJOLAIS	ROUGE
	TOKAY	1980	ALSACE	BLANC
∪	TAVEL	1986	RHONE	ROSE

Vins2	Cru	Mill	Région	Couleur
↓	TOKAY	1980	ALSACE	BLANC
	CHABLIS	1985	BOURGOGNE	ROUGE

Vins	Cru	Mill	Région	Couleur
	CHENAS	1983	BEAUJOLAIS	ROUGE
	TOKAY	1980	ALSACE	BLANC
	TAVEL	1986	RHONE	ROSE
	CHABLIS	1985	BOURGOGNE	ROUGE

Figure VI.9 : Exemple d'union

4.1.2. Différence

La **différence** est également l'opération classique de la théorie des ensembles adaptée aux relations de même schéma.

Notion VI.12 : Différence (*Difference*)

Opération portant sur deux relations de même schéma `RELATION1` *et* `RELATION2`*, consistant à construire une relation de même schéma* `RELATION3` *ayant pour tuples ceux appartenant à* `RELATION1` *et n'appartenant pas à* `RELATION2`*.*

La différence est un opérateur non commutatif : l'ordre des relations opérandes est donc important. Plusieurs notations ont été introduites pour cette opération, selon les auteurs :

– `RELATION1 - RELATION2`
– `DIFFERENCE (RELATION1, RELATION2)`
– `REMOVE (RELATION1, RELATION2`
– `MINUS (RELATION1, RELATION2)`

La notation graphique représentée figure VI.10 est aussi utilisée. À titre d'exemple, la différence des relations `VINS1 - VINS2` est représentée figure VI.11.

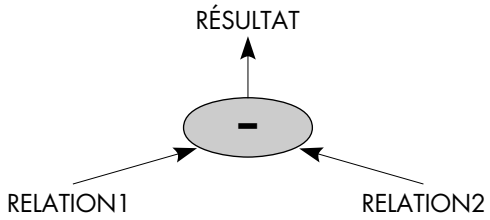


Figure VI.10 : Représentation graphique de la différence

Vins1	**Cru**	**Mill**	**Région**	**Couleur**
	CHENAS	1983	BEAUJOLAIS	ROUGE
	TOKAY	1980	ALSACE	BLANC
–	TAVEL	1986	RHONE	ROSE

Vins2	**Cru**	**Mill**	**Région**	**Couleur**
↓	TOKAY	1980	ALSACE	BLANC
	CHABLIS	1985	BOURGOGNE	ROUGE

Vins	**Cru**	**Mill**	**Région**	**Couleur**
	CHENAS	1983	BEAUJOLAIS	ROUGE
	TAVEL	1986	RHONE	ROSE

Figure VI.11 : Exemple de différence

4.1.3. Produit cartésien

Le **produit cartésien** est l'opération ensembliste que nous avons rappelée ci-dessus pour définir le concept de relations. Elle est adaptée aux relations. Cette fois, les deux relations n'ont pas nécessité d'avoir même schéma.

Notion VI.13 : Produit cartésien (*Cartesian product*)

Opération portant sur deux relation `RELATION1` *et* `RELATION2`, *consistant à construire une relation* `RELATION3` *ayant pour schéma la concaténation de ceux des relations opérandes et pour tuples toutes les combinaisons des tuples des relations opérandes.*

Des notations possibles pour cette opération sont :

– `RELATION1 X RELATION2`
– `PRODUCT (RELATION1, RELATION2)`
– `TIMES (RELATION1, RELATION2)`

La notation graphique représentée figure VI.12 est aussi utilisée. À titre d'exemple, la relation VINS représentée figure VI.13 est le produit cartésien des deux relations `CRUS` et `ANNEES` de la même figure.

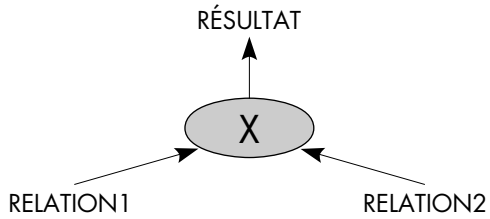


Figure VI.12 : Représentation graphique du produit cartésien

Vins1

Cru	Région	Couleur
CHENAS	BEAUJOLAIS	ROUGE
TOKAY	ALSACE	BLANC
TAVEL	RHONE	ROSE

X

Années	Mill
	1980
	1985

↓

Vins

Cru	Région	Couleur	Mill
CHENAS	BEAUJOLAIS	ROUGE	1980
TOKAY	ALSACE	BLANC	1980
TAVEL	RHONE	ROSE	1980
CHENAS	BEAUJOLAIS	ROUGE	1985
TOKAY	ALSACE	BLANC	1985
TAVEL	RHONE	ROSE	1985

Figure VI.13 : Exemple de produit cartésien

4.2. LES OPÉRATIONS SPÉCIFIQUES

4.2.1. Projection

La **projection** est une opération spécifique aux relations qui permet de supprimer des attributs d'une relation. Son nom provient du fait qu'elle permet de passer d'une relation n-aire à une relation p-aire avec `p<n`, donc d'un espace à n dimensions à un espace à moins de dimensions.

Notion VI.14 : Projection (*Projection*)

Opération sur une relation `RELATION1` *consistant à composer une relation* `RELATION2` *en enlevant à la relation initiale tous les attributs non mentionnés en opérandes (aussi bien au niveau du schéma que des tuples) et en éliminant les tuples en double qui sont conservés une seule fois.*

Les notations suivantes sont utilisées pour cette opération, en désignant par `Attributi, Attributj... Attributm` les attributs de projection :

– Π`Attributi, Attributj... Attributm (RELATION1)`

– `RELATION1 [Attributi, Attributj... Attributm]`

– `PROJECT (RELATION1, Attributi, Attributj... Attributm)`

La notation graphique représentée figure VI.14 est aussi utilisée. Le trapèze horizontal signifie que l'on réduit le nombre de colonnes de la relation : partant du nombre représenté par la base, on passe au nombre représenté par l'anti-base. La figure VI.15 donne un exemple de projection d'une relation `VINS` comportant aussi l'attribut `QUALITE` sur les attributs `MILLESIME` et `QUALITE`.

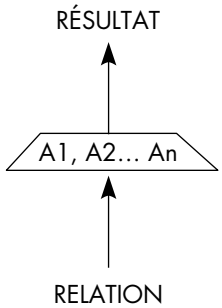


Figure VI.14 : Représentation graphique de la projection

VINS	Cru	Mill	Région	Qualité
	VOLNAY	1983	BOURGOGNE	A
	VOLNAY	1979	BOURGOGNE	B
	CHENAS	1983	BEAUJOLAIS	A
Π**cru, Région**	JULIENAS	1986	BEAUJOLAIS	C

↓

Π(VINS)	Cru	Région
	VOLNAY	BOURGOGNE
	CHENAS	BEAUJOLAIS
	JULIENAS	BEAUJOLAIS

Figure VI.15 : Exemple de projection

4.2.2. Restriction

La **restriction** est aussi une opération spécifique unaire, qui produit une nouvelle relation en enlevant des tuples à la relation opérande selon un critère.

Notion VI.15 : Restriction (*Restriction*)

Opération sur une relation `RELATION1` *produisant une relation* `RELATION2` *de même schéma, mais comportant les seuls tuples qui vérifient la condition précisée en argument.*

Les conditions possibles sont du type :

```
<Attribut> <Opérateur> <Valeur>
```

où l'opérateur est un opérateur de comparaison choisi parmi {=, <, ≤, ≥, >, ≠}. L'attribut doit appartenir à la relation sur laquelle s'applique le critère. Par exemple, pour la relation `VINS`, `CRU = "Chablis"` est une condition de restriction possible. `DEGRE > 12` est une autre condition possible. Il est aussi possible d'utiliser des compositions logiques de critères simples, c'est-à-dire des « et » et « ou » de conditions élémentaires. On pourra par exemple utilisé le critère CRU = "Chablis" et DEGRE > 12, ou encore le critère `CRU = "Chablis"` ou `DEGRE = 12`. Toute composition de critères valides par conjonction et disjonction (des parenthèses peuvent être utilisées pour préciser les priorités) est valide. Notons que les compositions logiques peuvent aussi être obtenues par union et intersection de relations restreintes (voir ci-dessous).

Les notations suivantes sont utilisées pour la restriction :

– $\sigma_{condition}$ (RELATION1)
– RELATION1 [Condition]
– RESTRICT (RELATION1, Condition)

ainsi que la notation graphique représentée figure VI.16. Le trapèze vertical signifie que l'on réduit le nombre de tuples de la relation : partant du nombre représenté par le côté gauche, on passe au nombre représenté par le côté droit. La figure VI.17 représente la restriction d'une relation VINS enrichie d'un attribut `QUALITE` par la condition `QUALITE = "BONNE"`.

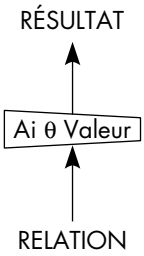


Figure VI.16 : Représentation graphique de la restriction

VINS

Cru	Mill	Région	Qualité
VOLNAY	1983	BOURGOGNE	A
VOLNAY	1979	BOURGOGNE	B
CHENAS	1983	BEAUJOLAIS	A
JULIENAS	1986	BEAUJOLAIS	C

$\sigma_{\text{cru} > 1983}$

VINS

Cru	Mill	Région	Qualité
JULIENAS	1986	BEAUJOLAIS	C

Figure VI.17 : Exemple de restriction

4.2.3. Jointure

La **jointure** est une des opérations essentielles de l'algèbre relationnelle, sans doute la plus difficile à réaliser dans les systèmes. La jointure permet de composer deux relations à l'aide d'un critère de jointure. Elle peut être vue comme une extension du produit cartésien avec une condition permettant de comparer des attributs. Nous la définirons comme suit :

Notion VI.16 : Jointure (*Join*)

Opération consistant à rapprocher selon une condition les tuples de deux relations `RELATION1` *et* `RELATION2` *afin de former une troisième relation* `RELATION3` *qui contient l'ensemble de tous les tuples obtenus en concaténant un tuple de* `RELATION1` *et un tuple de* `RELATION2` *vérifiant la condition de rapprochement.*

La jointure de deux relations produit donc une troisième relation qui contient toutes les combinaisons de tuples des deux relations initiales satisfaisant la condition spécifiée. La condition doit bien sûr permettre le rapprochement des deux relations, et donc être du type :

```
<Attribut1> <opérateur> <Attribut2>
```

où Attribut1 appartient à RELATION1 et Attribut2 à RELATION2. Selon le type d'opérateur, on distingue :

– l'**équi-jointure** dans le cas où l'opérateur est =, qui est une véritable composition de relations au sens mathématique du terme ;
– l'**inéqui-jointure** dans les autres cas, c'est-à-dire avec un des opérateurs <, ≤, >, ≥, ≠.

Dans le cas d'équi-jointure, les deux attributs égaux apparaissent chacun dans le résultat : il y a donc duplication d'une même valeur dans chaque tuple. Afin d'éliminer cette redondance, on définit la **jointure naturelle** comme suit :

Notion VI.17 : Jointure naturelle (*Natural join*)

Opération consistant à rapprocher les tuples de deux relations RELATION1 *et* RELATION2 *afin de former une troisième relation* RELATION3 *dont les attributs sont l'union des attributs de* RELATION1 *et* RELATION2*, et dont les tuples sont obtenus en composant un tuple de* RELATION1 *et un tuple de* RELATION2 *ayant mêmes valeurs pour les attributs de même nom.*

L'opération de jointure est représentée par l'une des notations suivantes, la condition étant simplement omise dans le cas de jointure naturelle (c'est alors l'égalité des attributs de même nom) :

– RELATION1 $\bowtie_{\text{Condition}}$ RELATION2
– JOIN (RELATION1, RELATION2, Condition)

La figure VI.18 donne la représentation graphique de l'opération de jointure ; la figure VI.19 illustre cette opération en effectuant la jointure naturelle des deux relations VINS et LOCALISATION. L'inéqui-jointure de ces deux relations selon la condition QUALITE > QUALITE MOYENNE est représentée figure VI.20. On suppose que les qualités sont codées par ordre décroissant A, B, C, D, E.

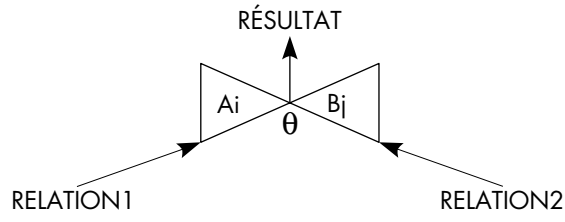
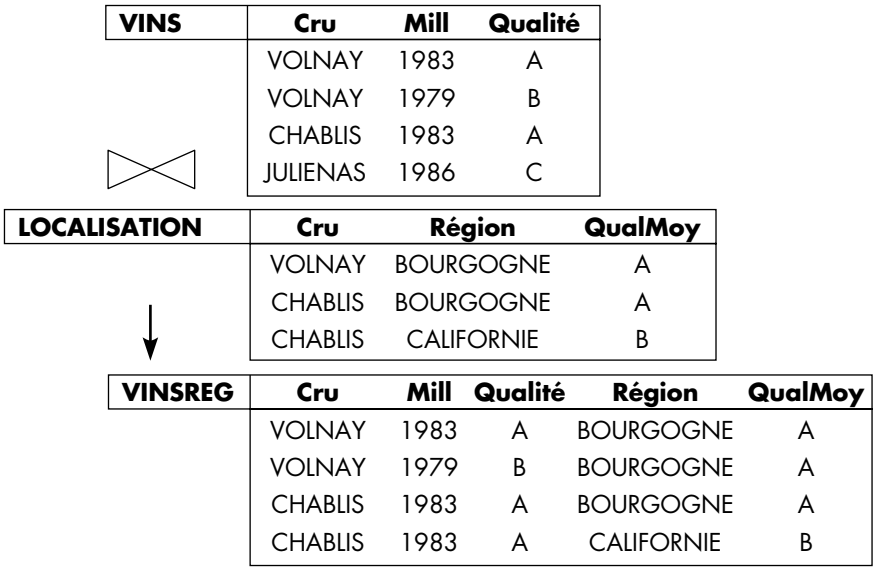


Figure VI.18 : Représentation graphique de la jointure

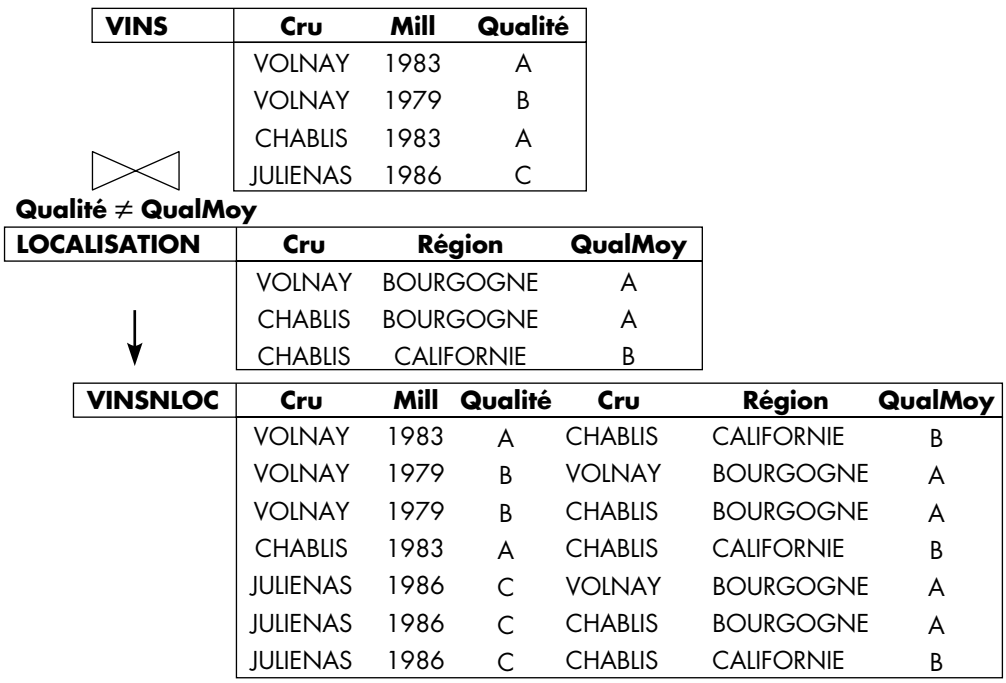


VINS	Cru	Mill	Qualité
	VOLNAY	1983	A
	VOLNAY	1979	B
	CHABLIS	1983	A
	JULIENAS	1986	C

LOCALISATION	Cru	Région	QualMoy
	VOLNAY	BOURGOGNE	A
	CHABLIS	BOURGOGNE	A
	CHABLIS	CALIFORNIE	B

VINSREG	Cru	Mill	Qualité	Région	QualMoy
	VOLNAY	1983	A	BOURGOGNE	A
	VOLNAY	1979	B	BOURGOGNE	A
	CHABLIS	1983	A	BOURGOGNE	A
	CHABLIS	1983	A	CALIFORNIE	B

Figure VI.19 : Jointure naturelle des relations `VINS` *et* `LOCALISATION`



VINS	Cru	Mill	Qualité
	VOLNAY	1983	A
	VOLNAY	1979	B
	CHABLIS	1983	A
	JULIENAS	1986	C

Qualité ≠ QualMoy

LOCALISATION	Cru	Région	QualMoy
	VOLNAY	BOURGOGNE	A
	CHABLIS	BOURGOGNE	A
	CHABLIS	CALIFORNIE	B

VINSNLOC	Cru	Mill	Qualité	Cru	Région	QualMoy
	VOLNAY	1983	A	CHABLIS	CALIFORNIE	B
	VOLNAY	1979	B	VOLNAY	BOURGOGNE	A
	VOLNAY	1979	B	CHABLIS	BOURGOGNE	A
	CHABLIS	1983	A	CHABLIS	CALIFORNIE	B
	JULIENAS	1986	C	VOLNAY	BOURGOGNE	A
	JULIENAS	1986	C	CHABLIS	BOURGOGNE	A
	JULIENAS	1986	C	CHABLIS	CALIFORNIE	B

Figure VI.20 : Inéqui-jointure des relations `VINS` *et* `LOCALISATION`

La jointure n'est pas toujours considérée comme une opération de base de l'algèbre relationnelle. En effet, si l'on étend la définition de la restriction de manière à considérer des conditions multi-attributs du type `<Attribut1> <Opérateur> <Attribut2>`, alors la jointure peut être obtenue par un produit cartésien suivi d'une restriction du résultat comme suit :

```
JOIN (RELATION1, RELATION2, Condition) =
RESTRICT ((RELATION1 X RELATION2), Condition)
```

Compte tenu de son jointure, nous avons préféré ici définir la jointure comme une opération de base.

5. L'ALGÈBRE RELATIONNELLE : OPÉRATIONS DÉRIVÉES

Les opérations présentées ci-dessous sont parfois utilisées pour manipuler des relations. Elles peuvent en général être obtenues par combinaison des opérations précédentes. Dans certains cas (complément, jointure externe), leur expression à partir des opérations de base nécessite la manipulation de relations constantes à tuples prédéfinis, telles que la relation obtenue par le produit cartésien des domaines, ou encore celle composée d'un seul tuple à valeurs toutes nulles.

5.1. INTERSECTION

L'**intersection** est l'opération classique de la théorie des ensembles adaptée aux relations de même schéma.

Notion VI.18 : Intersection (*Intersection*)

Opération portant sur deux relations de même schéma `RELATION1` *et* `RELATION2` *consistant à construire une relation de même schéma* `RELATION3` *ayant pour tuples ceux appartenant à la fois à* `RELATION1` *et* `RELATION2`.

Plusieurs notations ont été introduites pour cette opération selon les auteurs :

– `RELATION1 ∩ RELATION2`

– `INTERSECT (RELATION1, RELATION2)`

– `AND (RELATION1, RELATION2)`

La notation graphique représentée figure VI.21 est aussi utilisée.

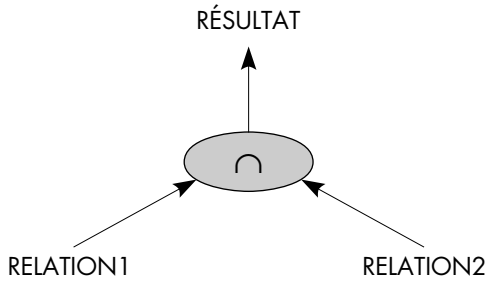


Figure VI.21 : Représentation graphique de l'intersection

L'intersection est une opération redondante avec les opérations de base, puisqu'il est possible de l'obtenir à partir de la différence à l'aide d'une des formules suivantes :

```
RELATION1 ∩ RELATION2 = RELATION1 – (RELATION1 – RELA-
                                                TION2)
RELATION1 ∩ RELATION2 = RELATION2 – (RELATION2 – RELA-
                                                TION1)
```

5.2. DIVISION

La **division** permet de rechercher dans une relation les sous-tuples qui sont complétés par tous ceux d'une autre relation. Elle permet ainsi d'élaborer la réponse à des questions de la forme « quel que soit x, trouver y » de manière simple.

Notion VI.19 : Division (*Division*)

Opération consistant à construire le quotient de la relation `D (A1, A2... Ap, Ap+1... An)` *par la relation* `d(Ap+1... An)` *comme la relation* `Q(A1, A2... Ap)` *dont les tuples sont ceux qui concaténés à tout tuple de d donnent un tuple de* `D`.

De manière plus formelle, désignons par ai une valeur quelconque de l'attribut Ai. Un tuple est alors une suite de valeurs <a1, a2, a3...>. Utilisant ces notations, le quotient de D par D est défini par :

```
Q = {<a1, a2... ap> tel-que quel-que-soit <ap+1... an> de d,
<a1, a2... ap, ap+1..., an> appartient à D}
```

Les notations possibles pour la division sont :

– `D ÷ d`

– `DIVISION (D, d)`

ainsi que la représentation graphique de la figure VI.22. Un exemple de division est représenté figure VI.23.

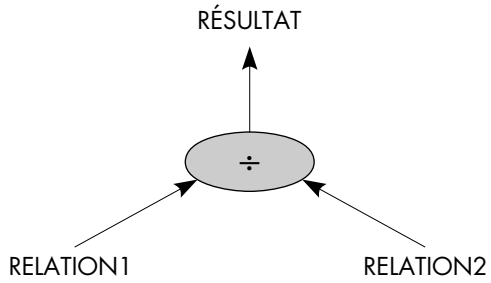


Figure VI.22 : Représentation graphique de la division

VINS	Cru	Mill	Qualité
	VOLNAY	1983	A
	VOLNAY	1979	B
	CHABLIS	1983	A
	CHABLIS	1979	A
	JULIENAS	1986	A

÷

QUALITE	Mill	Qualité
	1983	A
	1979	A

↓

CRU	Cru
	CHABLIS

Figure VI.23 : Exemple de division

La division peut être obtenue à partir de la différence, du produit cartésien et de la projection comme suit :

– `D - d = R1 - R2 avec`

– `R1 =` Π`A1, A2... Ap(D)`

– `R2 =` $\Pi_{\texttt{A1, A2... Ap}}$ `((R1 X d) - D)`

5.3. COMPLÉMENT

Le **complément** permet de trouver les tuples qui n'appartiennent pas à une relation. Il suppose a priori que les domaines sont finis (sinon on obtient des relations infinies).

Notion VI.20 : Complément (*Complement*)

Ensemble des tuples du produit cartésien des domaines des attributs d'une relation n'appartenant pas à cette relation.

Le complément d'une relation `RELATION1` est noté au choix :

– `RELATION1`

– `NOT(RELATION1)`

– `COMP(RELATION1)`

La figure VI.24 illustre cette opération dans un cas simple. C'est une opération peu utilisée du fait qu'elle permet de générer des tuples qui ne sont pas dans la base, en général très nombreux. Si l'on note par X Di le produit cartésien des domaines, le complément d'une relation `RELATION1` est obtenu à partir de la différence comme suit :

– `NOT (RELATION1) = X Di – RELATION1`

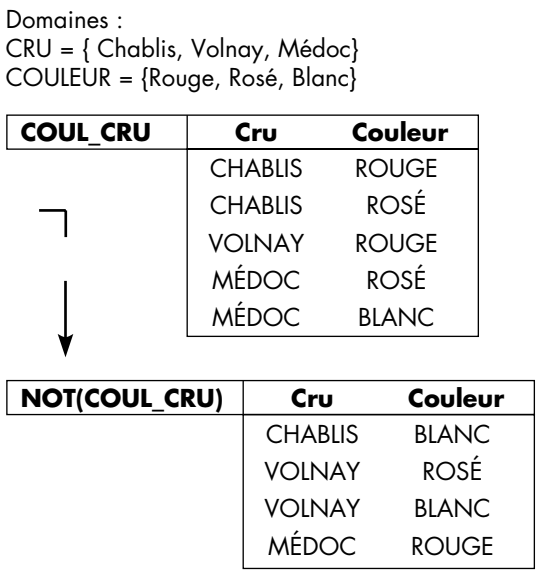


Figure VI.24 : Exemple de complément

5.4. ÉCLATEMENT

L'**éclatement** [Fagin80] est une opération qui n'appartient pas vraiment à l'algèbre rationnelle puisqu'elle donne deux relations en résultats, à partir d'une. Elle est cepen-

dant utile pour partitionner une relation horizontalement en deux sous-relations. À ce titre, elle est considérée comme une extension de l'algèbre.

Notion VI.21 : Éclatement (*Split*)

Opération consistant à créer deux relations à partir d'une relation `RELATION1` *et d'une condition, la première contenant les tuples de* `RELATION1` *et la deuxième ceux ne la vérifiant pas.*

Cet opérateur appliqué à la relation `RELATION1` génère donc deux relations R1 et R2 qui seraient obtenues par restriction comme suit :

– `R1 = RESTRICT (RELATION1, Condition)`

– `R2 = RESTRICT (RELATION1, ¬Condition)`

5.5. JOINTURE EXTERNE

Les jointures définies ci-dessus perdent des tuples d'au moins une relation quand les relations jointes n'ont pas de projections identiques sur l'attribut de jointure. Pour préserver toutes les informations dans tous les cas, il est nécessaire de définir des jointures qui conservent les tuples sans correspondant avec des valeurs nulles associées quand nécessaire. C'est dans ce but que Codd [Codd79] a introduit les **jointures externes**.

Notion VI.22 : Jointure externe (*External Join*)

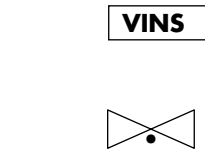
Opération générant une relation `RELATION3` *à partir de deux relations* `RELATION1` *et* `RELATION2`*, par jointure de ces deux relations et ajout des tuples de* `RELATION1` *et* `RELATION2` *ne participant pas la jointure, avec des valeurs nulles pour les attributs de l'autre relation.*

Cette opération est très utile, en particulier pour composer des vues sans perte d'informations. Elle se note en général comme suit :

– `RELATION1` $\overset{\cdot}{\bowtie}$ `RELATION2`

– `EXT-JOIN (RELATION1, RELATION2)`

La jointure externe permet par exemple de joindre des tables `CLIENTS` et `COMMANDES` sur un numéro de client commun, en gardant les clients sans commande et les commandes sans client associé. Elle est donc très utile en pratique. On peut aussi distinguer la jointure externe droite qui garde seulement les tuples sans correspondant de la relation de droite. On notera celle-ci $\bowtie$ ou `REXT-JOIN`. De même, on peut distinguer la jointure externe gauche ($\overset{\cdot}{\bowtie}$ ou `LEXT-JOIN`). Un exemple de jointure externe complète apparaît figure VI.25.



VINS	Cru	Mill	Qualité
	VOLNAY	1983	A
	VOLNAY	1979	B
	JULIENAS	1986	C

LOCALISATION	Cru	Région	QualMoy
	VOLNAY	BOURGOGNE	A
	CHABLIS	BOURGOGNE	A
	CHABLIS	CALIFORNIE	B

VINSREG	Cru	Mill	Qualité	Région	QualMoy
	VOLNAY	1983	A	BOURGOGNE	A
	VOLNAY	1979	B	BOURGOGNE	A
	CHABLIS	—	—	BOURGOGNE	A
	CHABLIS	—	—	CALIFORNIE	B
	JULIENAS	1986	C	—	—

Figure VI.25 : Exemple de jointure externe

5.6. SEMI-JOINTURE

Dans certains cas, lors de l'exécution d'une jointure, il n'est pas nécessaire de conserver tous les attributs des deux relations en résultat : seuls les attributs d'une des deux relations sont conservés. Une opération spécifique de **semi-jointure**, très utile pour optimiser l'évaluation des questions, a été définie [Berstein81].

Notion VI.23 : Semi-jointure (*Semi-join*)

Opération portant sur deux relations `RELATION1` *et* `RELATION2`*, donnant en résultat les tuples de* `RELATION1` *qui participent à la jointure des deux relations.*

La semi-jointure de la relation `RELATION1` par la relation `RELATION2` est notée :

– `RELATION1` ⋉ `RELATION2`

– `SEMI-JOIN (RELATION1, RELATION2)`

Elle est équivalente à la jointure des relations `RELATION1` et `RELATION2` suivie par une projection du résultat sur les attributs de la relation `RELATION1`. Notez que l'opération n'est pas symétrique puisque seuls des tuples de la première relation sont conservés. Elle peut être vue comme une restriction de la relation `RELATION1` par les valeurs des attributs de jointure figurant dans la relation `RELATION2`. La figure VI.26 illustre cette opération de semi-jointure.

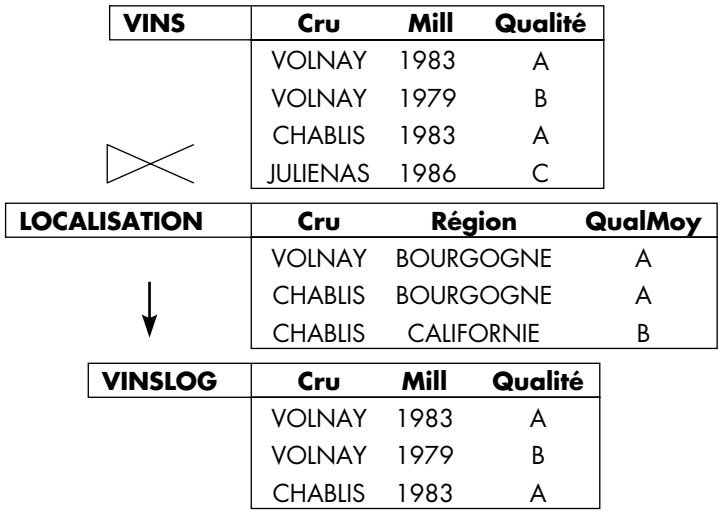


Figure VI.26 : Exemple de semi-jointure

5.7. FERMETURE TRANSITIVE

La **fermeture transitive** est une opération très particulière qui permet d'ajouter des tuples à une relation. Elle n'appartient pas à l'algèbre rationnelle, mais peut être vue comme une de ses extensions. Il n'est pas possible de constituer cette opération avec un nombre fixe d'opérations de l'algèbre rationnelle : elle peut être effectuée par une série de jointure/projection/union, mais le nombre d'opérations à effectuer dépend du contenu de la relation. Certains auteurs considèrent que c'est une faiblesse de l'algèbre relationnelle que de ne pas pouvoir exprimer une fermeture transitive par une expression constante d'opérations élémentaires.

Notion VI.24 : Fermeture transitive (*Transitive closure*)

Opération sur une relation `R` *à deux attributs (*`A1, A2`*) de même domaine consistant à ajouter à R tous les tuples qui se déduisent successivement par transitivité, c'est-à-dire que si l'on a des tuples* `<a, b>` *et* `<b, c>`*, on ajoute* `<a, c>`*.*

Cette opération se note suivant les auteurs :

– $\tau(R)$

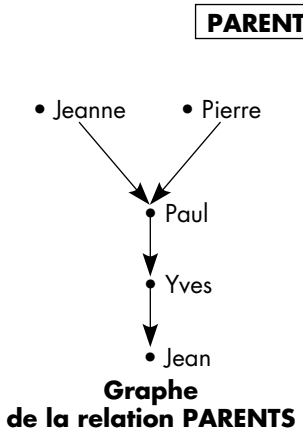
– R^+

– `CLOSE(R)`

Pour effectuer une fermeture transitive, il est nécessaire d'effectuer une boucle d'opérations jusqu'à obtention de la fermeture complète. On doit donc utiliser un langage de programmation avec une boucle *while*, comme suit :

$$\tau(R) = \textit{while}\ \tau(R)\ \textit{change do}\ \tau(R)$$
$$= \tau(R) \cup \Pi_{A1,A4}\ (R \bowtie \tau(R)).$$

Cette opération permet par exemple de calculer à partir d'une relation PARENTS (Parent, Enfant) la relation ANCETRES (Ascendant, Descendant), qui donne toute la filiation connue d'une personne. La figure VI.27 illustre cette opération. La fermeture transitive de PARENTS est calculée par jointures/projections/unions successives de la relation PARENTS avec elle-même jusqu'à saturation, c'est-à-dire jusqu'à obtention d'une relation stable à laquelle une nouvelle jointure/projection/union n'apporte plus de tuples. On voit que la relation ANCETRES représente le graphe correspondant à la fermeture transitive du graphe de la relation PARENTS.



Graphe de la relation PARENTS

PARENTS	Parent	Enfant
	Pierre	Paul
	Jeanne	Paul
	Paul	Yves
	Yves	Jean

ANCÊTRES	Ascendant	Descendant
	Pierre	Paul
	Jeanne	Paul
	Paul	Yves
	Yves	Jean
	Pierre	Yves
	Jeanne	Yves
	Pierre	Jean
	Jeanne	Jean
	Paul	Jean

Figure VI.27 : Exemple de fermeture transitive

6. LES EXPRESSIONS DE L'ALGÈBRE RELATIONNELLE

À partir des opérations de l'algèbre relationnelle, il est possible de composer un langage d'interrogation de bases de données. Une question peut alors être représentée par un arbre d'opérateurs relationnels. Le paraphrasage en anglais de telles expressions est à la base du langage SQL que nous étudierons dans le chapitre suivant.

6.1. LANGAGE ALGÉBRIQUE

En utilisant des expressions d'opérations de l'algèbre relationnelle, il est possible d'élaborer les réponses à la plupart des questions que l'on peut poser à une base de données relationnelle. Plus précisément, les opérations de base de l'algèbre relationnelle constituent un **langage complet**, c'est-à-dire ayant la puissance de la logique du premier ordre.

Notion VI.25 : Langage complet (*Complete language*)

Langage équivalent à la logique du premier ordre.

Nous avons étudié plus précisément la logique du premier ordre et les langages d'interrogation qui en découlent au chapitre V. Disons simplement que l'algèbre relationnelle permet d'exprimer toute question exprimable avec la logique du premier ordre sans fonction, par exemple avec le calcul de tuple ou de domaine. L'algèbre relationnelle peut d'ailleurs être étendue avec des fonctions [Zaniolo85].

Voici quelques questions sur la base DEGUSTATION dont le schéma a été représenté figure VI.7. Elles peuvent être exprimées comme des expressions d'opérations, ou comme des opérations successives appliquées sur des relations intermédiaires ou de base, générant des relations intermédiaires. Pour des raisons de clarté, nous avons choisi cette deuxième représentation. L'expression peut être obtenue simplement en supprimant les relations intermédiaires notées Ri.

(Q1) Donner les degrés des vins de crus Morgon et de millésime1978 :

```
R1 = RESTRICT (VINS, CRUS = "MORGON")
R2 = RESTRICT (VINS, MILLESIME = 1978)
R3 = INTERSECT (R1, R2)
RESULTAT = PROJECT (R3, DEGRE)
```

(Q2) Donner les noms et prénoms des buveurs de Morgon ou Chenas :

```
R1 = RESTRICT (VINS, CRU = "MORGON")
R2 = RESTRICT(VINS, CRUS = "CHENAS")
R3 = UNION (R1, R2)
```

```
R4 = JOIN (R3, ABUS)
R5 = JOIN (R4, BUVEURS)
RESULTAT = PROJECT (R5, NOM, PRENOM)
```

(Q3) Donner les noms et adresses des buveurs ayant bu plus de 10 bouteilles de Chablis 1976 avec le degré de ce vin :

```
R1 = RESTRICT (ABUS, QUANTITE > 10)
R2 = RESTRICT (VINS, CRU = "CHABLIS")
R3 = RESTRICT (VINS, MILLESIME = 1976)
R4 = INTERSECT (R2, R3)
R5 = JOIN (R1, R4)
R6 = PROJECT (R5, NB, DEGRE)
R7 = JOIN (R6, BUVEURS)
RESULTAT = PROJECT (R7, NOM, ADRESSE, DEGRE)
```

(Q4) Donner les noms des buveurs n'ayant bu que du Morgon :

```
R1 = JOIN(BUVEURS,ABUS,VINS)
R2 = RESTRICT(R1, CRU = "Morgon")
R3 = PROJECT(R2, NOM)
R4 = RESTRICT(R1, CRU ≠ "Morgon")
R5 = PROJECT(R1, NOM)
RESULTAT = MINUS(R3 – R5)
```

Si l'on accepte les relations constantes, l'algèbre relationnelle permet aussi d'exécuter les mises à jour. Par exemple, l'intersection du vin [100, TOKAY, 1978, 13] s'effectuera comme suit :

```
R1 = CONSTANTE (VINS, [100, TOKAY, 1978, 13])
VINS = UNION (VINS, R1)
```

où `CONSTANTE` permet de définir une relation de même schéma que la relation `VINS` contenant le seul tuple indiqué en argument.

6.2. ARBRE ALGÉBRIQUE

Une question exprimée sous forme d'un programme d'opérations de l'algèbre relationnelle peut être représentée par un **arbre relationnel**. Les nœuds correspondent aux représentations graphiques des opérations indiquées ci-dessus et les arcs aux flots de données entre opérations.

Notion VI.26 : Arbre relationnel (*Relational tree*)

Arbre dont les nœuds correspondent à des opérations de l'algèbre relationnelle et les arcs à des relations de base ou temporaires représentant des flots de données entre opérations.

Ainsi, la question « Noms et Prénoms des buveurs habitant Paris ayant bu du Chablis depuis le 1er janvier 1992 » peut être exprimée à l'aide de l'arbre représenté

figure VI.28. Plusieurs arbres équivalents peuvent être déduits d'un arbre donné à l'aide de règles de transformation simples, telles que la permutation des jointures et restrictions, la permutation des projections et des jointures, le regroupement des intersections sur une même relation, etc. Ces transformations sont à la base des techniques d'optimisation de questions qui dépassent le sujet de ce chapitre. La figure VI.29 propose un arbre équivalent à celui de la figure VI.28, obtenu par descente de projections et regroupement d'intersections.

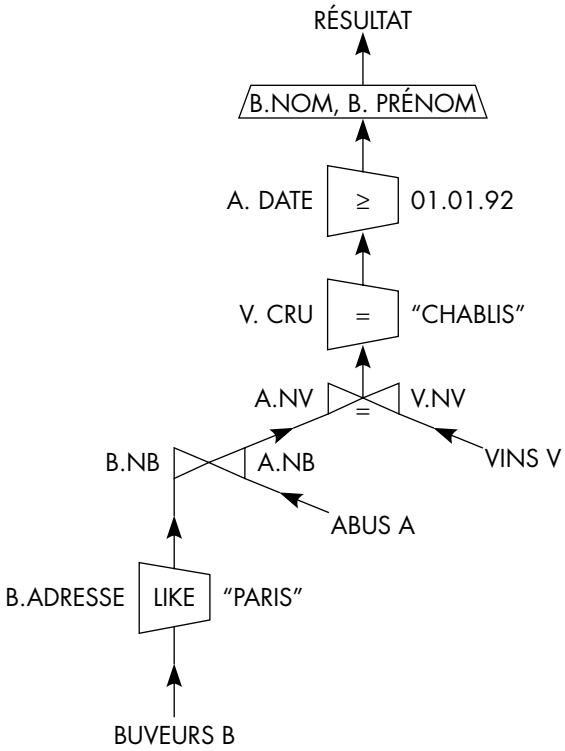


Figure VI.28 : Exemple d'arbre représentant une question

La composition d'opérations de l'algèbre relationnelle ne nécessite pas toujours d'attendre le résultat de l'opération précédente pour exécuter l'opération suivante. Restriction, projection et jointure peuvent ainsi être exécutées par des algorithmes à flots de données. Des opérateurs se succédant sur un arbre peuvent donc être exécutés en parallèle par des algorithmes « pipe-line ». Des opérateurs figurant sur des branches distinctes d'un arbre peuvent aussi être exécutés en parallèle de manière indépendante. La représentation par arbre algébrique met ainsi en évidence les possibilités de parallélisme et les enchaînements nécessaires. Ces propriétés des arbres relationnels sont importantes pour optimiser les questions dans des contextes parallèles.

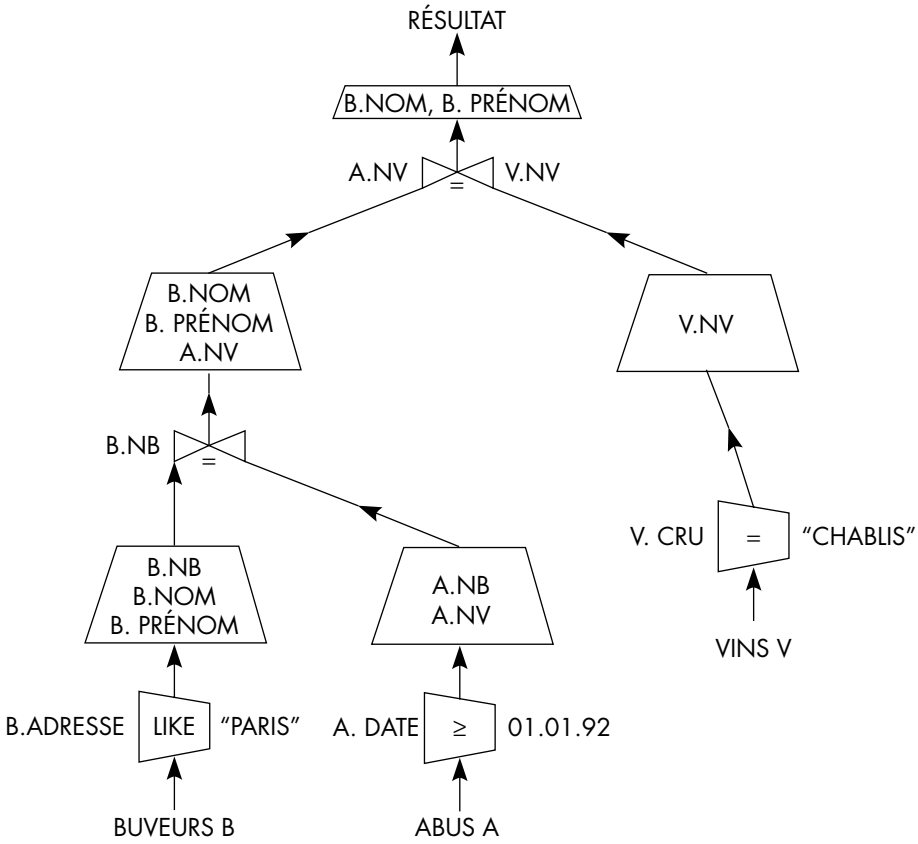


Figure VI.29 : Arbre équivalent à l'arbre précédent

7. FONCTIONS ET AGRÉGATS

L'algèbre relationnelle est insuffisante pour traiter de véritables applications des bases de données, telles la suivie de production, la gestion de budget, etc. Il est en effet nécessaire d'effectuer des calculs sur la base pour supporter de telles applications. C'est l'objet de l'introduction des fonctions de calcul au sein de l'algèbre et du support des agrégats.

7.1. FONCTIONS DE CALCUL

La possibilité d'effectuer des calculs sur les attributs est simplement introduite en généralisant projection, restriction et jointure par introduction de fonctions. Ainsi, tout attribut apparaissant en argument d'une opération est remplacé par une **expression d'attributs**.

Notion VI.27 : Expression d'attributs (*Attribut expression*)

Expression arithmétique construite à partir d'attributs d'une relation et de constantes, par application de fonctions arithmétiques successives.

Voici des exemples d'expressions d'attributs :

```
10 + 50 – 23 ;
QUANTITE * 50 ;
QUANTITE * DEGRE / 100 ;
QUANTITE – QUANTITE * DEGRE / 100.
```

De telles expressions peuvent donc être utilisées comme arguments de projections, de restrictions voire de jointures. Le programme d'algèbre relationnelle suivant illustre ces possibilités :

```
R1 = JOIN (VINS, ABUS, DEGRE*QUANTITE/100 > QUANTITE/10)
R2 = RESTRICT(R1, DEGRE*QUANTITE/100 > 10)
RESULTAT = PROJECT(R2, CRU, QUANTITE – QUANTITE*DEGRE/100)
```

La notion d'expression d'attributs peut être généralisée avec des fonctions autres que les fonctions arithmétiques, par exemple des fonctions écrites dans un langage externe. On aboutit ainsi à une généralisation de l'algèbre relationnelle aux fonctions [Zaniolo85].

7.2. SUPPORT DES AGRÉGATS

Les expressions d'attributs permettent d'effectuer des opérations de calcul en ligne, sur des attributs de relations. En pratique, il est nécessaire d'effectuer des opérations de calcul en colonnes, sur des tuples de relations, cela par exemple afin de sommer des dépenses, etc. Le concept d'**agrégat** permet de telles opérations.

Notion VI.28 : Agrégat (*Aggregat*)

Partitionnement horizontal d'une relation en fonction des valeurs d'un groupe d'attributs, suivi d'un regroupement par application d'une fonction de calcul sur ensemble.

Les fonctions de calcul sur ensemble les plus souvent proposées sont :

– SOMME (SUM) permettant de calculer la somme des éléments d'un ensemble ;
– MOYENNE (AVG) permettant de calculer la moyenne des éléments d'un ensemble ;
– MINIMUM (MIN) permettant de sélectionner l'élément minimum d'un ensemble ;
– MAXIMUM (MAX) permettant de sélectionner l'élément maximum d'un ensemble ;
– COMPTE (COUNT) permettant de compter les éléments d'un ensemble.

La figure VI.30 illustre le concept d'agrégat. La table VINS est enrichie d'un attribut QUANTITE. L'agrégat représenté calcule la somme des quantités par CRU. L'opération générique s'écrit :

```
R = AGREGAT(<RELATION> ; <ATTRIBUT1> ; <FONCTION>{<ATTRIBUT2>})
```

<Attribut1> représente un ou plusieurs attributs utilisés pour le partitionnement. Fonction est la fonction d'ensemble appliquée à l'attribut <Attribut2>. Par exemple, on écrit :

```
RESULTAT = AGREGAT(VINS ; CRU ; SUM{QUANTITE})
```

pour l'agrégat illustré figure VI.30. Notez qu'un agrégat peut s'effectuer sans partitionnement ; on peut par exemple calculer simplement la moyenne de tous les degrés comme indiqué figure VI.30 par la commande :

```
RESULTAT = AGREGAT(VINS ; ; AVG{DEGRE}).
```

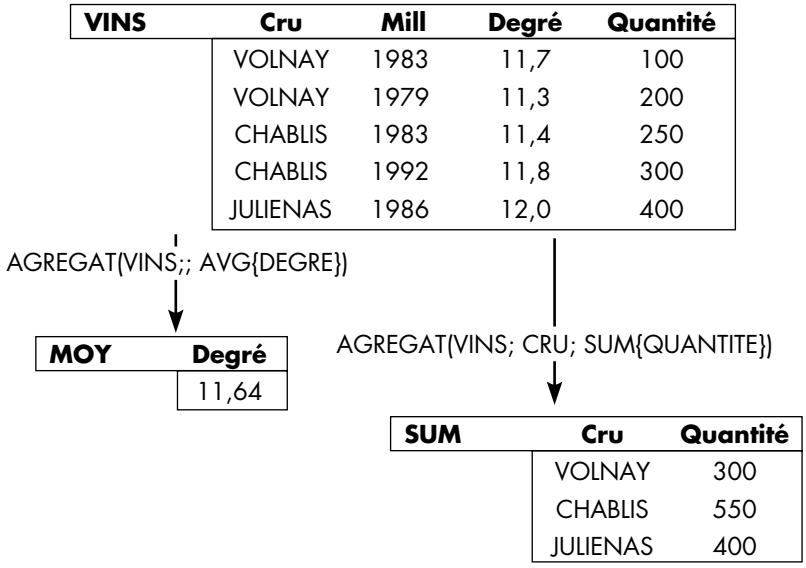


Figure VI.30 : Exemples de calcul d'agrégats

8. CONCLUSION

Dans ce chapitre, nous avons introduits les concepts essentiels aujourd'hui supportés par le modèle relationnel dans les grands systèmes industriels tels que ORACLE, INGRES, DB2, SYBASE, etc. Ces concepts constituent la base du langage SQL, le langage des systèmes relationnels. Nous allons étudier ce langage et ses extensions dans les chapitres qui suivent.

Le modèle relationnel fait aujourd'hui autorité dans l'industrie. Issu de la théorie des relations, il est à l'origine une remarquable construction de la recherche. Il a su progressive-

ment intégrer des concepts de plus en plus riches, tels que l'intégrité référentielle, les règles actives, etc. Le modèle a aujourd'hui intégré les concepts de l'objet pour fonder l'objet-relationnel, comme nous le verrons dans la troisième partie de cet ouvrage. Il a encore un bel avenir devant lui, bien qu'il soit parfois contesté par les tenants de l'objet.

9. BIBLIOGRAPHIE

[Berstein81] Bernstein P., Goodman N., « The Power of Natural Semijoins », *Siam Journal of Computing*, vol. 10, n° 4, décembre 1981, p. 751-771.

Une discussion de la puissance de l'opérateur de semi-jointure. Après une définition de la semi-jointure, Phil Bernstein et Nathan Goodman montrent que cet opérateur permet d'exprimer un grand nombre de questions avec jointures, plus spécifiquement toutes celles dont le graphe des jointures est un arbre.

[Chen76] Chen P.P., « The Entity-Relationship Model – Towards a Unified View of Data », *ACM Transactions on Database Systems*, vol. 1, n° 1, mars 1976.

L'article de base sur le modèle entité-association. Il introduit ce modèle pour décrire la vue des données d'une entreprise. En particulier, les diagrammes de Chen sont présentés. Il est montré que le modèle permet d'unifier les différents points de vue et de passer simplement à une implémentation relationnelle. Ce dernier point explique le fait que beaucoup d'outils d'aide à la conception de bases de données relationnelles utilisent une variante du modèle de Chen.

[Childs68] Childs D.L., « Feasibility of a Set-Theoretic Data Structure – A General Structure Based on a Reconstituted Definition of a Relation », *Congrès IFIP, Genève*, 1968, p. 162-172.

Un des premiers articles proposant un modèle basé sur le concept de relation et des opérateurs ensemblistes. La proposition de Codd s'est inspirée des travaux de Childs.

[Codd70] Codd E.F., « A Relational Model for Large Shared Data Banks », *Communications de l'ACM, vol. 13, n° 6*, juin 1970, p. 377-387.

L'article de base proposant le modèle relationnel. Il introduit les concepts essentiels de relation, domaine, attribut et clé. L'algèbre relationnelle est proposée comme langage de manipulation des relations.

[Codd79] Codd E.F., « Extending the Relational Model to Capture More Meaning », *ACM TODS, vol. 4, n° 4*, décembre 1979, p. 397-433.

Une proposition d'extension du modèle relationnel pour mieux décrire la sémantique des applications. L'idée de base est de distinguer différents types de

relations (entité, association, généralisation, etc.) et d'étendre les opérateurs relationnels en conséquence. Un traitement des valeurs nulles avec une logique trivaluée est aussi proposé. L'ensemble du modèle étendu, appelé RM/T, est décrit par un métamodèle relationnel.

[Date81] Date C.J., « Referential Integrity », *7e Very Large Data Bases*, Cannes, France, 1981, IEEE Ed.

L'article proposant le support de l'intégrité référentiel au sein du modèle relationnel. Chris Date introduit les contraintes référentielles et les contraintes d'entité comme contraintes de base à intégrer au modèle relationnel pour un meilleur support de la sémantique des données. Un langage de déclaration de contraintes est aussi esquissé.

[Delobel83] Delobel C., Adiba M., *Bases de Données et Systèmes Relationnels*, livre, Dunod Informatique, 1983.

Un des premiers livres français sur les bases de données relationnelles. Alors que le livre Bases de Données – Les Systèmes et Leurs Langages de Georges Gardarin paru à la même époque propose une vue simplifiée des différents concepts, ce livre offre une vision plus formelle, souvent fondée sur les travaux de l'université de Grenoble. Des opérateurs relationnels spécifiques et une approche originale à la normalisation des relations sont notamment développés.

[Fagin80] Fagin R., « A Normal Form for Relational Databases that is Based on Domains and Keys », *ACM TODS, vol. 6, n° 3*, septembre 1981, p. 387-415.

Un des articles prosant une forme ultime de normalisation des relations. Une relation est en 5e forme normale si elle ne peut plus être décomposée par projection sur différents sous-schémas, de sorte à obtenir la relation de départ par jointures naturelles des sous-relations. Fagin introduit une méthode de normalisation fondée sur les domaines et les clés. Il montre qu'il s'agit là de la forme ultime de normalisation par projection et jointure. Il introduit aussi un opérateur d'éclatement qui permet d'envisager d'autres formes de normalisation horizontale.

[Gardarin89] Gardarin G., Cheiney J.P., Kiernan J., Pastre D., « Managing Complex Objects in an Extensible DBMS », *15th Very Large Data Bases International Conference*, Morgan Kaufman Pub., Amsterdam, Pays-Bas, août 1989.

Une présentation détaillée du support d'objets complexes dans le SGBD extensible Sabrina. Ce système est dérivé du SGBD relationnel SABRE et supporte des types abstraits comme domaines d'attributs. Il a aujourd'hui évolué vers un SGBD géographique (GéoSabrina) et est commercialisé par INFOSYS.

[ISO89] International Organization for Standardization, « Information Processing Systems – Database Language SQL with Integrity Enhancement », *International Standard ISO/IEC JTC1 9075* : 1989(E), 2e édition, avril 1989.

La norme SQL aujourd'hui en vigueur. Ce document de 120 pages présente la norme SQL1 : concepts de base, éléments communs, langage de définition de schémas, définition de modules de requêtes, langage de manipulation. Toutes la grammaire de SQL1 est décrite en BNF. Ce document résulte de la fusion du standard de 1986 et des extensions pour l'intégrité de 1989.

[Maier83] Maier D., *The Theory of Relational Databases*, livre, Computer Science Press, 1983.

Le livre synthétisant tous les développements théoriques sur les bases de données relationnelles. En 600 pages assez formelles, Maier fait le tour de la théorie des opérateurs relationnels, des dépendances fonctionnelles, multivaluées et algébriques, et de la théorie de la normalisation.

[Stonebraker87] Stonebraker M., « The Design of the POSTGRES Storage System », *13th Very Large Databases International Conference,* Morgan & Kauffman Ed., Brighton, Angleterre, 1987.

Une description assez complète du système POSTGRES, successeur d'INGRES développé à Berkeley. M. Stonebraker présente la conception du noyau de stockage du système POSTGRES. Outre un contrôle de concurrence original permettant le support de déclencheurs (ou réflexes), ce sytème intégre les types abstraits au modèle relationnel. Un type abstrait permet de définir un type d'attribut ou de tuple. Le système POSTGRES a ainsi permis de prototyper les fonctionnalités orientées objets intégrées à la version 7 d'INGRES.

[Ullman88] Ullman J.D., *Principles of Database and Knowledge-base Systems,* livres, volumes I et II, Computer Science Press, 1988.

Deux volumes très complets sur les bases de données, avec une approche plutôt fondamentale. Jeffrey Ullman détaille tous les aspects des bases de données, des méthodes d'accès aux modèles objets en passant par le modèle logique. Les livres sont finalement centrés sur une approche par la logique aux bases de données. Les principaux algorithmes d'accès, d'optimisation de requêtes, de concurrence, de normalisation, etc. sont détaillés.

[Zaniolo83] Zaniolo C., « The Database Language GEM », *ACM SIGMOD Conférence*, San José, Ca., ACM Ed., 1983.

Une extension du modèle relationnel vers le modèle entité-association et du langage QUEL pour supporter un tel modèle. Carlo Zaniolo propose d'utiliser les entité pour spécifier les domaines lors de la définition des associations. Il propose alors une extension de QUEL avec une notation pointée pour naviguer depuis les associations vers les attributs des entités. L'ensemble constitue le langage GEM, qui a été implémenté à Bell Labs au-dessus de la machine bases de données IDM.

[Zaniolo85] Zaniolo C., « The Representation and Deductive Retrieval of Complex Objects », *11th Very Large Data Bases International Conference*, Morgan Kaufman Pub., Stockholm, Suède, août 1985.

Une extension de l'algèbre relationnelle au support de fonctions. Cet article présente une extension de l'algèbre relationnelle permettant de référencer des fonctions symboliques dans les critères de projection et de sélection, afin de manipuler des objets complexes. Des opérateurs déductifs de type fermeture transitive étendue sont aussi intégrés.

Chapitre VII

LE LANGAGE SQL2

1. INTRODUCTION

Les serveurs de données relationnels présentent aujourd'hui une interface externe sous forme d'un langage de recherche et mise à jour, permettant de spécifier les ensembles de données à sélectionner ou à mettre à jour à partir de propriétés des valeurs, sans dire comment retrouver les données. Ainsi, les opérations directement utilisables par les usagers sont en général celles des langages dits assertionnels. Plusieurs langages assertionnels permettant de manipuler des bases de données relationnelles ont été proposés, en particulier QUEL [Zook77], QBE [Zloof77] et SQL [IBM82, IBM87]. Aujourd'hui, le langage SQL est normalisé [ISO89, ISO92] et constitue le standard d'accès aux bases de données relationnelles. Les autres interfaces par menus, fenêtres, grilles, etc., ou de programmation type langage de 3e ou 4e génération, sont le plus souvent offertes au-dessus du langage SQL. Celui-ci constitue donc le point d'entrée obligatoire des SGBD relationnels. QUEL était le langage proposé par l'université de Berkeley pour son système INGRES : il est aujourd'hui peu utilisé dans l'industrie. QBE est un langage par grille dérivé de la logique qui est souvent offert au-dessus de SQL.

De manière générale, SQL comme les autres langages qui ont été proposés (e.g., QUEL) utilisent tous des critères de recherche (encore appelés qualifications) construits à partir de la logique des prédicats du premier ordre. Ils comportent quatre opérations de base :

– la **recherche** (mot clé SELECT en SQL, RETRIEVE en QUEL) permet de retrouver des tuples ou parties de tuples vérifiant la qualification citée en arguments ;
– l'**insertion** (mot clé INSERT en SQL, APPEND en QUEL) permet d'ajouter des tuples dans une relation ; les tuples peuvent être fournis par l'utilisateur ou construits à partir de données existant déjà dans la base ;
– la **suppression** (mot clé DELETE en SQL, SUPPRESS en QUEL) permet de supprimer d'une relation les tuples vérifiant la qualification citée en argument ;
– la **modification** (mot clé UPDATE en SQL, REPLACE en QUEL) permet de mettre à jour les tuples vérifiant la qualification citée en argument à l'aide de nouvelles valeurs d'attributs ou de résultats d'opérations arithmétiques appliquées aux anciennes valeurs.

Le langage assertionnel SQL fut introduit commercialement tout d'abord par IBM [IBM82]. Il résultait alors d'une évolution du langage SEQUEL 2 [Chamberlin76] initialement développé au centre de recherches de San-José comme un langage expérimental appelé SQUARE [Boyce75] pour paraphraser en anglais les expressions de l'algèbre relationnelle. Aujourd'hui, l'ISO a normalisé le langage SQL pour manipuler les bases de données relationnelles et ceci à plusieurs niveaux. Le niveau SQL1 [ISO89] correspond à la norme de base acceptée en 1989, telle qu'elle est aujourd'hui appliquée par la plupart des constructeurs. SQL1 permet l'expression des requêtes composées d'opérations de l'algèbre relationnelle et d'agrégats. En plus des fonctionnalités de définition, de recherche et de mise à jour, SQL1 comporte aussi des fonctions de contrôle qui n'appartiennent pas à proprement parler au modèle relationnel, mais qui sont nécessaires pour programmer des applications transactionnelles.

Le niveau SQL2 [ISO92] est sous-divisé en trois niveaux, respectivement entrée, intermédiaire et complet. Le niveau entrée peut être perçu comme une amélioration de SQL1, alors que les niveaux intermédiaire et complet permettent de supporter totalement le modèle relationnel avec des domaines variés, tels date et temps. SQL3 est constitué d'un ensemble de propositions nouvelles traitant plus particulièrement des fonctionnalités objets et déductives.

Ce chapitre est organisé comme suit. Les quatre sections qui suivent sont consacrées à l'étude du standard SQL1. Nous étudions successivement la définition de schéma, la recherche de données, l'expression des mises à jour et l'intégration aux langages de programmation pour écrire des transactions. La section qui suit présente les fonctionnalités du nouveau standard SQL2. En conclusion, nous introduisons brièvement les propositions émises dans le cadre SQL3 (voir chapitre XIII) et nous soulignons l'importance de SQL.

Nous utilisons des notations syntaxiques en BNF (*Backus-Naur Form*), avec les extensions suivantes :
– les crochets ([]) indiquent des éléments optionnels ;
– les pointillés suivent un élément qui peut être répété plusieurs fois ;

– les accolades groupent comme un seul élément une séquence d'éléments ;
– un exposant plus ($^+$) indique que l'élément qui précède peut être répété n fois (n>0), chaque occurrence étant séparée de la précédente par une virgule ;
– un exposant multiplié (*) indique que l'élément qui précède peut être répété n fois (n≥0), chaque occurrence étant séparée de la précédente par une virgule.

2. SQL1 : LA DÉFINITION DE SCHÉMAS

SQL1 permet de définir des schémas de bases de données composés de tables et de vues. Un schéma est simplement identifié par un identifiant autorisant l'accès à la base. Au niveau d'un schéma sont associés des autorisations d'accès aux tables. Au niveau de la table, des contraintes d'intégrité peuvent être définies.

2.1. CRÉATION DE TABLES

SQL1 permet de créer des relations sous forme de tables et de définir lors de la création des contraintes d'intégrité variés sur les attributs. Ainsi, une commande de création permet de spécifier le nom de la table et de définir les éléments de table correspondant aux colonnes ou aux contraintes, selon la syntaxe suivante :

```
CREATE TABLE <nom de table> (<élément de table>+)
```

Un nom de table peut être un nom simple (par exemple VINS) ou un nom composé d'un nom de schéma (identifiant d'autorisation d'accès à la base, par exemple DEGUSTATION) suivi par le nom simple de table en notation pointée (par exemple, DEGUSTATION.VINS).

Un élément de table est soit une définition de colonne, soit une définition de contrainte, comme suit :

```
<ÉLÉMENT DE TABLE> ::=  <DÉFINITION DE COLONNE> |
                        <CONTRAINTE DE TABLE>
```

Une colonne est définie par un nom et un type de données. Une valeur par défaut peut être précisée. Une contrainte de colonne peut aussi être définie à ce niveau. On obtient donc la syntaxe suivante :

```
<DÉFINITION DE COLONNE> : := <NOM DE COLONNE> <TYPE DE DONNÉES>
                        [<CLAUSE DÉFAUT>] [<CONTRAINTE DE COLONNE>]
```

Les types de données supportés sont les chaînes de caractères de longueurs fixes – CHAR(<longueur>) –, la valeur par défaut de la longueur étant 1, les numériques

exactes – NUMERIC et DECIMAL avec précision et échelle optionnelles, INTEGER et SMALLINT –, les numériques approchés – FLOAT avec précision optionnelle, REAL et DOUBLE PRECISION.

La clause défaut permet simplement de spécifier une valeur par défaut selon la syntaxe DEFAULT <valeur>, la valeur NULL étant permise. Nous examinerons les contraintes d'intégrité de table et de colonne dans la section qui suit.

Afin d'illustrer les possibilité introduites, la figure VII.1 présente les commandes permettant de créer la base dégustation, pour l'instant sans contrainte d'intégrité. Le schéma de la base obtenu est composé des trois relations suivantes :

```
VINS (NV, CRU, MILLESIME, DEGRE, QUALITE)
BUVEURS (NB, NOM, ADRESSE, TYPE)
ABUS (NB, NV, DATE, QUANTITE).

CREATE SCHEMA AUTHORIZATION DEGUSTATION
CREATE TABLE VINS (NV INT, CRU CHAR(12), MILLESIME INT, DEGRE DEC(3,1),
   QUALITE CHAR)
CREATE TABLE BUVEURS (NB INT, NOM CHAR(20), ADRESSE CHAR(30), TYPE
   CHAR(4))
CREATE TABLE ABUS(NB INT, NV INT, DATE DEC(6), QUANTITE SMALLINT)
```

Figure VII.1 : Création de la base Dégustation

2.2. EXPRESSION DES CONTRAINTES D'INTÉGRITÉ

Les **contraintes de colonnes** permettent de spécifier différentes contraintes d'intégrité portant sur un seul attribut, y compris les contraintes référentielles. Les différentes variantes possibles sont :

– valeur nulle impossible (syntaxe NOT NULL),

– unicité de l'attribut (syntaxe UNIQUE ou PRIMARY KEY),

– contrainte référentielle – syntaxe REFERENCES <table référencée> [(<colonne référencée>)] –, le nom de la colonne référencée étant optionnel s'il est identique à celui de la colonne référençante,

– contrainte générale (syntaxe CHECK <condition>) ; la condition est une condition pouvant spécifier des plages ou des listes de valeurs possibles (voir condition de recherche ci-dessous).

Les **contraintes de relations** peuvent porter sur plusieurs attributs. Ce peut être des contraintes d'unicité, référentielles ou générales. Elles sont exprimées à l'aide des phrases suivantes :

– contrainte d'unicité UNIQUE <attribut>$^{+}$,

– contrainte référentielle, permettant de spécifier quelles colonnes référencent celles d'une autre table, [FOREIGN KEY (<colonne référençante>$^+$)] REFERENCES <table référencée> [(<colonne référencée>$^+$)],
– contrainte générale CHECK <condition>.

Par exemple, la création de la relation ABUS avec contraintes d'intégrité pourra être effectuée par la commande de la figure VII.2. Cette commande précise que les attributs NB et NV ne doivent pas être nuls et doivent exister dans les relations BUVEURS et VINS respectivement, que la date est comprise entre le premier janvier 80 et le 31 décembre 99, que la quantité doit être comprise entre 1 et 100 avec une valeur par défaut de 1.

```
CREATE TABLE ABUS (
NB INT NOT NULL,
NV INT NOT NULL REFERENCES VINS(NV),
DATE DEC(6) CHECK (DATE BETWEEN 010180 AND 311299),
QUANTITE SMALLINT DEFAULT 1,
PRIMARY KEY(NB, NV, DATE),
FOREIGN KEY NB REFERENCES BUVEURS,
CHECK (QUANTITE BETWEEN 1 AND 100))
```

Figure VII.2 : Création de la table ABUS avec contraintes d'intégrité

2.3. DÉFINITION DES VUES

SQL1 permet de définir des vues au niveau du schéma. Rappelons qu'une vue est une table virtuelle calculée à partir des tables de base par une question. La syntaxe de la commande de création de vues est la suivante :

```
CREATE VIEW <NOM DE TABLE> [(<NOM DE COLONNE>+)]
AS <SPÉCIFICATION DE QUESTION>
[WITH CHECK OPTION]
```

Une vue est modifiable s'il est possible d'insérer et de supprimer des tuples dans la base au-travers de la vue. Dans ce cas, les tuples sont insérés ou supprimés dans la première table référencée par la question. La vue doit alors contenir toutes les colonnes de cette table. L'option WITH CHECK OPTION permet de s'assurer que les tuples insérés vérifient les conditions exprimées dans la question, c'est-à-dire qu'ils appartiennent bien à la vue. Cela permet d'imposer des contraintes d'intégrité lors des mises à jour au travers de la vue (par exemple, le fait qu'un tuple de la vue doit référencé un tuple d'une autre table).

À titre d'illustration, voici une vue GROS-BUVEURS définie simplement comme la table virtuelle contenant le nom et le prénom des buveurs de type « GROS ». Cette

vue n'est pas modifiable. Cette définition montre déjà un premier exemple de question SQL très simple, de type sélection.

```
CREATE VIEW GROS-BUVEURS (NOM, PRENOM)
AS    SELECT NOM, PRENOM
FROM BUVEURS
WHERE TYPE = "GROS"
```

Figure VII.3 : Exemple de définition de vue

2.4. SUPPRESSION DES TABLES

La suppression des tables n'est pas permise en SQL1. La plupart des systèmes permettent cependant de détruire une relation par la commande :

DROP TABLE <NOM DE TABLE>.

Par exemple, la destruction de la relation VINS s'effectuera par la commande :

DROP TABLE **VINS.**

2.5. DROITS D'ACCÈS

La gestion des droits d'accès aux tables est décentralisée : il n'existe pas d'administrateur global attribuant des droits. Chaque créateur de table obtient tous les droits d'accès à cette table, en particulier les droits d'effectuer les actions de sélection (SELECT), d'insertion (INSERT), de suppression (DELETE), de mise à jour (UPDATE) et aussi de référencer la table dans une contrainte (REFERENCES). Il peut ensuite passer ses droits sélectivement à d'autres utilisateurs ou à tous le monde (PUBLIC). Un droit peut être passé avec le droit de le transmettre (WITH GRANT OPTION) ou non.

SQL1 propose ainsi une commande de passation de droits dont la syntaxe est la suivante :

GRANT **<PRIVILÈGES> ON <NOM DE TABLE>** TO **<RÉCEPTEUR>**$^+$
[WITH GRANT OPTION**]**

avec :

<PRIVILÈGES> **::= ALL PRIVILEGES** | <ACTION>$^+$

<ACTION> **::= SELECT** | **INSERT** | **UPDATE** [(<NOM DE COLONNE>$^+$)]
| **REFERENCE** [(<NOM DE COLONNE>$^+$)]

<RÉCEPTEUR> **::= PUBLIC** | <IDENTIFIANT D'AUTORISATION>

L'ensemble des privilèges (ALL PRIVILEGES) inclut les droits d'administration (changement de schéma et destruction de la relation). Le récepteur peut être un utilisateur ou un groupe d'utilisateurs, selon l'identifiant d'autorisation donné.

Par exemple, la passation des droits de consultation et mise à jour de la table VINS à l'utilisateur Poivrot s'effectuera comme indiqué ci-dessous. La présence de l'option de passation (GRANT OPTION) permet à Poivrot de passer ce droit.

```
GRANT SELECT, UPDATE ON VINS TO POIVROT
WITH GRANT OPTION
```

Bien que non prévue dans la norme de 1989, la commande REVOKE permet de retirer un droit à un utilisateur. Sa syntaxe est la suivante :

```
REVOKE <privilèges> ON <nom de table> FROM <récepteur>.
```

Par exemple, la commande qui suit permet de retirer le droit donné ci-dessus ainsi que tous les droits qui en dépendent (c'est-à-dire ceux de sélection ou mise à jour de la table VINS passés par Poivrot).

```
REVOKE SELECT, UPDATE ON VINS FROM POIVROT.
```

3. SQL1 : LA RECHERCHE DE DONNÉES

Dans cette section, nous étudions la requête de recherche qui est à la base de SQL, le fameux SELECT. Nous commençons par des exemples à partir de cas simples dérivés de l'algèbre relationnelle pour aboutir au cas général.

3.1. EXPRESSION DES PROJECTIONS

Rappelons qu'une projection effectue l'extraction de colonnes (attributs) spécifiées d'une relation, puis élimine les tuples en double. SQL n'élimine pas les doubles, à moins que cela soit explicitement demandé par le mot clé DISTINCT, l'option par défaut étant ALL. SQL généralise la projection en ce sens qu'il est possible d'appliquer des fonctions de calculs sur les colonnes extraites. Les fonctions de calculs permises sont en particulier les fonctions arithmétiques d'addition, soustraction, multiplication et division. Une projection s'exprime à l'aide du langage SQL par la clause :

```
SELECT [ALL|DISTINCT] <EXPRESSION DE VALEURS>+
FROM <NOM DE TABLE> [<NOM DE VARIABLE>]
```

Une expression de valeurs est une expression arithmétique (composée avec les opérateurs binaires +, –, * et /), éventuellement parenthésée, de spécifications de constantes

ou de colonnes. Une spécification de constante est soit une constante, une variable de programme ou le nom de l'usager (mot clé USER). Une spécification de colonne désigne le nom d'une colonne précédé d'un désignateur de relation éventuel (nom de table ou variable), comme suit :

```
<SPÉCIFICATION DE COLONNE> ::= [<DÉSIGNATEUR>.] <NOM DE COLONNE>
<DÉSIGNATEUR> ::= <NOM DE TABLE> | <NOM DE VARIABLE>
```

L'usage d'un nom de variable nécessite de définir dans la clause FROM une variable dite de corrélation, permettant de désigner la table par cette variable. De telles variables évitent de répéter le nom de la table dans le cas de questions portant sur plusieurs tables, comme nous le verrons ci-dessous. Notez aussi qu'il est possible d'utiliser une étoile (*) à la place de la liste d'expression de valeurs, cela signifiant simplement que l'on désire lister tous les attributs de la table référencée dans la clause FROM.

Nous illustrons la projection en utilisant la table VINS dont le schéma a été défini ci-dessus. La première question (Q1) indiquée figure VII.4 permet d'obtenir pour tous les vins les crus, millésimes et quantité d'alcool pur contenue dans 1 000 litres, avec doubles éventuels. La deuxième question (Q2) effectue la recherche de tous les tuples de la table VINS sans double.

```
(Q1)  SELECT CRU, MILLESIME, (DEGRE/100)*1000
      FROM VINS

(Q2)  SELECT DISTINCT *
      FROM VINS
```

Figure VII.4 : Exemples de projections

3.2. EXPRESSION DES SÉLECTIONS

Une sélection est une combinaison d'une restriction suivie d'une projection. Une sélection s'exprime comme une projection avec en plus une condition de recherche selon la syntaxe suivante :

```
SELECT [ALL|DISTINCT] {<expression de valeurs>+ | *}
FROM <nom de table> [<nom de variable>]
WHERE <condition de recherche>
```

Une condition de recherche définit un critère, qui appliqué à un tuple, est vrai, faux ou inconnu, selon le résultat de l'application d'opérateurs booléens (ET, OU, NOT) à des conditions élémentaires. L'expression booléenne de conditions élémentaires peut être parenthésée. La figure VII.5 donne les tables de vérité permettant de calculer la valeur de vérité d'une condition de recherche. En principe, les seuls tuples satisfaisant la condition de recherche sont sélectionnés par la requête.

AND	***VRAI***	***FAUX***	***INCONNU***
VRAI	VRAI	FAUX	INCONNU
FAUX	FAUX	FAUX	FAUX
INCONNU	INCONNU	FAUX	INCONNU

OR	***VRAI***	***FAUX***	***INCONNU***
VRAI	VRAI	VRAI	VRAI
FAUX	VRAI	FAUX	INCONNU
INCONNU	VRAI	INCONNU	INCONNU

NOT	***VRAI***	***FAUX***	***INCONNU***
	FAUX	VRAI	INCONNU

Figure VII.5 : Calcul de la valeur d'une condition de recherche

Une condition de recherche élémentaire est appelée **prédicat** en SQL. Un prédicat de restriction permet de comparer deux expressions de valeurs ; la première contenant des spécifications de colonnes est appelée **terme** ; la seconde contenant seulement des spécifications de constantes est appelée **constante**. Il existe une grande diversité de prédicats en SQL1. On trouve en effet :

1. un prédicat de comparaison permettant de comparer un terme à une constante à l'aide des opérateurs {=, ≠, <, >, <=, >=} ;
2. un prédicat d'intervalle `BETWEEN` permettant de tester si la valeur d'un terme est comprise entre la valeur de deux constantes ;
3. un prédicat de comparaison de texte `LIKE` permettant de tester si un terme de type chaîne de caractères contient une ou plusieurs sous-chaînes ;
4. un prédicat de test de nullité qui permet de tester si un terme a une valeur convenue `NULL`, signifiant que sa valeur est inconnue ;
5. un prédicat d'appartenance `IN` qui permet de tester si la valeur d'un terme appartient à une liste de constantes.

La figure VII.6 donne quelques exemples de sélections illustrant ces différents types de prédicats. La question (Q3) effectue la restriction de la relation VINS par la qualifi-

cation Millésime = 1977 et Degré > 13. La question (Q4) délivre les crus et degrés des vins de millésime 1977 et de degré compris entre 11 et 13. La question (Q5) retrouve les crus, années de production (millésime diminué de 1900) et qualité des Beaujolais. Elle illustre le prédicat de recherche textuel (`LIKE`) ; le caractère `%` signifie dans le cas du `LIKE` une quelconque sous-chaîne de caractères ; par exemple, `BEAUJOLAIS NOUVEAUX` et `NOUVEAUX BEAUJOLAIS` rendent le prédicat `CRU LIKE "%BEAUJOLAIS%"` vrai. La question (Q6) recherche tous les vins de degré nul. La question (Q7) délivre les crus des vins de qualité A, B ou C.

```
(Q3)  SELECT  *
      FROM    VINS
      WHERE   MILLESIME = 1977
      AND     DEGRE > 13

(Q4)  SELECT  CRU, DEGRE
      FROM    VINS
      WHERE   MILLESIME = 1977
      AND     DEGRE BETWEEN 11 AND 13

(Q5)  SELECT  DISTINCT CRU, MILLESIME - 1900, QUALITE
      FROM    VINS
      WHERE   CRU LIKE

(Q6)  SELECT  *
      FROM    VINS
      WHERE   CRU IS NULL

(Q7)  SELECT  CRU
      FROM    VINS
      WHERE   QUALITE IN (A,B,C)
```

Figure VII.6 : Exemples de sélections

3.3. EXPRESSION DES JOINTURES

Un cas particulier simple de jointure sans qualification est le produit cartésien. Celui-ci s'exprime très simplement en incluant plusieurs relations dans la clause `FROM`. Par exemple, le produit cartésien des relations `VINS` et `ABUS` se calcule à l'aide de la question (Q8) représentée figure VII.7.

```
(Q8)  SELECT  *
      FROM    VINS, ABUS
```

Figure VII.7 : Exemple de produit cartésien

La jointure avec qualification peut s'exprimer de plusieurs manières. Une première expression naturelle est la restriction du produit cartésien par un prédicat permettant de comparer deux termes. Les prédicats de comparaison (=, ≠, ≤, ≥, <, >), d'intervalle (BETWEEN), d'appartenance (IN) et de comparaison textuelle (LIKE) sont utilisables. La combinaison des opérations de jointures, restrictions et projections peut être effectuée à l'intérieur d'un même bloc SELECT.

La figure VII.8 illustre différents cas de jointures. La question (Q9) effectue la jointure de VINS et ABUS sur l'attribut numéro de vins (NV) et permet de lister en préfixe de chaque tuple de ABUS le vin correspondant. La question (Q10) recherche les buveurs ayant un nom similaire à celui d'un cru. La question (Q11) retourne le nom des buveurs ayant bu du Chablis, sans double. Notez l'usage des variables B, V et A comme alias des relations BUVEURS, VINS et ABUS, afin d'éviter de répéter le nom complet des relations dans le critère.

```
(Q9)  SELECT   *
      FROM     VINS, ABUS
      WHERE    VINS.NV = ABUS.NV

(Q10) SELECT   NB, NOM
      FROM     BUVEURS, VINS
      WHERE    NOM LIKE CRU

(Q11) SELECT   DISTINCT NOM
      FROM     BUVEURS B, VINS V, ABUS A
      WHERE    B.NB = A. NB
      AND      A.NV = V.NV
      AND      V.CRU = "Chablis"
```

Figure VII.8 : Exemples de jointures

3.4. SOUS-QUESTIONS

SQL permet l'imbrication de sous-questions au niveau de la clause WHERE, si bien que l'on pet écrire des questions du type SELECT ... FROM ... WHERE ... SELECT .... En effet, le résultat d'une question peut être considéré comme une valeur simple ou comme un ensemble de valeurs avec doubles éventuels (multi-ensemble) ; dans ce dernier cas, chaque valeur de l'ensemble correspond à un tuple du résultat. Ainsi, il est possible de considérer une sous-question comme argument particulier des prédicats de comparaison (=, ≠, <, >, ≥, ≤) et d'appartenance à une liste (IN). Toute sous-question peut elle-même invoquer des sous-questions, si bien qu'il est possible d'imbriquer des blocs SELECT à plusieurs niveaux. L'imbrication de blocs SELECT par le prédicat IN permet d'exprimer en particulier des jointures d'une manière plus procédurale.

Par exemple, la question (Q12) de la figure VII.9 effectue la jointure des tables VINS et ABUS sur numéro de vin et sélectionne les crus des vins résultants sans double. Elle est équivalente à la question (Q13). Il est aussi possible d'utiliser des variables définies dans un bloc interne au niveau d'un bloc externe. On parle alors de variable de corrélation. La question (Q14) illustre l'usage d'une variable de corrélation pour retrouver cette fois le cru du vins mais aussi la quantité bue à partir de la jointure de VINS et ABUS. Finalement, la question (Q15) recherche les noms des buveurs de Chablis. Elle est équivalente à la question (Q11), mais est écrite de manière plus procédurale avec trois blocs imbriqués.

```
(Q12) SELECT   DISTINCT CRU
      FROM     VINS
      WHERE    NV IN
               SELECT NV
               FROM ABUS

(Q13) SELECT   DISTINCT CRU
      FROM     VINS V, ABUS A
      WHERE    V.NV = A.NV

(Q14) SELECT   DISTINCT CRU, A.QUANTITE
      FROM     VINS
      WHERE    NV IN
               SELECT NV
               FROM ABUS A

(Q15) SELECT   DISTINCT NOM
      FROM     BUVEURS
      WHERE    NB IN
               SELECT   NB
               FROM     ABUS
               WHERE    NV IN
                        SELECT   NV
                        FROM     VINS
                        WHERE    CRU = "CHABLIS"
```

Figure VII.9 : Exemples de blocs imbriqués

3.5. QUESTIONS QUANTIFIÉES

Il est aussi possible de vouloir comparer une expression de valeurs à tous les résultats d'une sous-question ou seulement à l'une quelconque des valeurs générées. SQL propose pour cela l'usage de sous-questions quantifiées par « quel que soit » (ALL) ou « il existe » (ANY ou SOME). Ces quantificateurs permettent de tester si la valeur d'un terme satisfait un opérateur de comparaison avec tous (ALL) ou au moins un (ANY ou

SOME) des résultats d'une sous-question. Un prédicat quantifié par ALL est vrai s'il est vérifié pour tous les éléments de l'ensemble. Un prédicat quantifié par ANY ou SOME est vrai s'il est vérifié par au moins un élément de l'ensemble.

Ainsi, la question (Q16) recherche les noms des buveurs n'ayant commis que des abus en quantité supérieure ou égale à toutes les quantités bues, alors que la question (Q17) recherche ceux ayant commis au moins un abus en quantité supérieure ou égale à toutes les quantités bues. La première n'aura probablement pas de réponse alors que la deuxième éditera le nom de la personne ayant effectué le plus gros abus.

SQL offre une autre possibilité de quantification pour tester si le résultat d'une sous question est vide ou non. Il s'agit du prédicat d'existence EXISTS. EXISTS <sous-question> est vrai si et seulement si le résultat de la sous-question est non vide. Ainsi, la question (Q18) recherche les buveurs ayant bu du Volnay alors que la question (Q19) recherche ceux n'ayant bu que du Volnay.

```
(Q16) SELECT   NOM
      FROM     BUVEURS B, ABUS A
      WHERE    B.NB = A.NB
      AND      A.QUANTITE ≥ ALL
               SELECT     QUANTITE
               FROM       ABUS

(Q17) SELECT   B.NOM
      FROM     BUVEURS B, ABUS A
      WHERE    B.NB = A.NB
      AND      A.QUANTITE ≥ ANY
               SELECT     QUANTITE
               FROM       ABUS

(Q18) SELECT   B.NOM
      FROM     BUVEURS B
      WHERE    EXISTS
               SELECT     *
               FROM       ABUS A, VINS V
               WHERE      A.NV = V.NV
               AND        A.NB = B.NB
               AND        CRU = "VOLNAY"

(Q19) SELECT   B.NOM
      FROM     BUVEURS B
      WHERE    NOT EXISTS
               SELECT     *
               FROM       ABUS A, VINS V
               WHERE      A.NV = V.NV
               AND        A.NB = B.NB
               AND        CRU = "VOLNAY"
```

Figure VII.10 : Exemples de questions quantifiées

3.6. EXPRESSION DES UNIONS

SQL1 permet également d'exprimer l'opération d'union. Par exemple, l'obtention des crus de degré supérieur à 13 ou de millésime 1977 peut s'effectuer par la question (Q20) représentée figure VII.11.

```
(Q20) SELECT   CRU
      FROM     VINS
      WHERE    DEGRE > 13
      UNION
      SELECT   CRU
      FROM     VINS
      WHERE    MILLESIME = 1977
```

Figure VII.11 : Exemple d'union

3.7. FONCTIONS DE CALCULS ET AGRÉGATS

Au-delà de l'algèbre relationnelle, des possibilités de calcul de fonctions existent. Les fonctions implantées sont :

– COUNT qui permet de compter le nombre de valeurs d'un ensemble ;
– SUM qui permet de sommer les valeurs d'un ensemble ;
– AVG qui permet de calculer la valeur moyenne d'un ensemble ;
– MAX qui permet de calculer la valeur moyenne d'un ensemble ;
– MIN qui permet de calculer la valeur moyenne d'un ensemble.

Les fonctions peuvent être utilisées dans la clause SELECT, par exemple pour calculer le degré moyen des « Chablis » comme dans la question (Q21) figure VII.12. Les fonctions sont aussi utilisables pour effectuer des calculs d'agrégats.

Rappelons qu'un agrégat est un partitionnement horizontal d'une table en sous-tables en fonction des valeurs d'un ou de plusieurs attributs de partitionnement, suivi de l'application d'une fonction de calculs à chaque attribut des sous-tables obtenues. Cette fonction est choisie parmi celles indiquées ci-dessus. En SQL, le partitionnement s'exprime par la clause :

GROUP BY <SPÉCIFICATION DE COLONNE>$^{+}$

Cette dernière permet de préciser les attributs de partitionnement, alors que les fonctions de calculs appliquées aux ensembles générés sont directement indiquées dans les expressions de valeurs suivant le SELECT.

Une restriction peut être appliquée avant calcul de l'agrégat au niveau de la clause WHERE, mais aussi après calcul de l'agrégat sur les résultats de ce dernier. Pour cela, une clause spéciale est ajoutée à la requête SELECT :

```
HAVING <EXPRESSION DE VALEURS>+
```

La figure VII.12 propose quelques exemples de calculs d'agrégats. La question (Q21) calcule simplement la moyenne des degrés des Chablis. La question (Q22) mixte jointure et agrégat : elle affiche pour chaque cru la somme des quantités bues ainsi que la moyenne des degrés des vins du cru. Les questions (Q23) et (Q24) combinent agrégats et restrictions : (Q23) calcule la moyenne des degrés pour tous les crus dont le degré minimal est supérieur à 12 ; (Q24) recherche tous les numéros de vins bus en quantité supérieure à 10 par plus de 100 buveurs. La question (Q25) démontre une combinaison de blocs imbriqués et d'agrégats avec restriction portant sur des calculs ; elle retrouve les noms des buveurs ayant bu depuis 1983 une quantité d'alcool supérieur à 100.

```
(Q21) SELECT   AVG(DEGRE)
      FROM     VINS
      WHERE    CRU = "Chablis"

(Q22) SELECT   CRU, SUM(QUANTITE), AVG(DEGRE)
      FROM     VINS V, ABUS A
      WHERE    V.NV = A.NV
      GROUP BY CRU

(Q23) SELECT   CRU, AVG(DEGRE)
      FROM     VINS
      GROUP BY CRU
      HAVING   MIN(DEGRE)>12

(Q24) SELECT   NV
      FROM     ABUS
      WHERE    QUANTITE>10
      GROUP BY NV
      HAVING   COUNT(NB)>100

(Q25) SELECT   NOM
      FROM     BUVEURS
      WHERE    NB IN
               SELECT   NB
               FROM     ABUS A, VINS V
               WHERE    A.NV = V.NV
               AND      DATE > 01-01-83
               GROUP BY NB
               HAVING   SUM(QUANTITE*DEGRE/100)>100
```

Figure VII.12 : Exemple de calculs d'agrégats

4. SQL1 : LES MISES A JOUR

SQL1 offre trois commandes de mise à jour : INSERT (insérer), DELETE (supprimer) et UPDATE (modifier). Toute mise à jour s'effectue par recherche des tuples à modifier et application des modifications. La recherche peut s'effectuer directement par une question intégrée dans la commande de mise à jour ou préalablement par utilisation d'un curseur depuis un programme traditionnel. Nous détaillons tout d'abord les commandes de mise à jour à partir de questions. La syntaxe des mises à jour est cependant précisée pour l'utilisation de curseurs, que nous verrons dans la section suivante.

4.1. INSERTION DE TUPLES

L'insertion de tuples dans une relation permet de créer de nouvelles lignes. Elle peut s'effectuer soit par fourniture directe au terminal d'un tuple à insérer (ou d'une partie de tuple, les valeurs inconnues étant positionnées à NULL), soit par construction à partir d'une question des tuples à insérer. La première variante est l'insertion directe, la deuxième l'insertion via question. La syntaxe de la commande d'insertion de SQL1 est :

```
INSERT INTO <NOM DE TABLE> [(<NOM DE COLONNE> +)]
{VALUES (<CONSTANTE>+) | <COMMANDE DE RECHERCHE>}
```

Dans le cas où la liste de colonnes n'est pas spécifiée, tous les attributs de la relation doivent être fournis dans l'ordre de déclaration. Si seulement certaines colonnes sont spécifiées, les autres sont insérées avec la valeur nulle.

Une insertion à partir d'une commande de recherche permet de composer une relation à partir des tuples d'une relation existante, par recherche dans la base.

En guise d'illustration, la commande d'insertion d'un Juliénas 1983 de degré inconnu sous le numéro de vins 112 est représentée figure VII.13 (requête (I1)). Nous donnons aussi la commande (I2) insérant dans une table BONSBUVEURS tous les buveurs ayant bu des vins de qualité A. La table BONSBUVEURS de schéma (NB, NOM, PRENOM) a du être créée auparavant.

```
(I1)  INSERT INTO VINS (NV, CRU, MILLESIME)
      VALUE (112, JULIENAS, 1983)

(I2)  INSERT   INTO BONSBUVEURS
      SELECT   NB, NOM, PRENOM
      FROM     BUVEURS B, ABUS A, VINS V
      WHERE    B.NB = A.NB
      AND      A.NV = V.NV
      AND      V.QUALITE = "A"
```

Figure VII.13 : Exemples de commandes d'insertion

4.2. MISE À JOUR DE TUPLES

La mise à jour permet de changer des valeurs d'attributs de tuples existants. La mise à jour d'une relation peut s'effectuer soit par fourniture directe des valeurs à modifier, soit par l'élaboration de ces valeurs à partir d'une expression. Les seuls tuples mis à jour sont ceux vérifiant une condition de recherche optionnelle fournie en argument d'une clause WHERE. Il est aussi possible de faire une mise à jour d'un seul tuple pointé par un curseur préalablement ouvert.

La syntaxe de la commande de mise à jour SQL1 est :

```
UPDATE <NOM DE TABLE>
SET {<NOM DE COLONNE> = {<EXPRESSION DE VALEUR> | NULL}}+
WHERE {<CONDITION DE RECHERCHE> | CURRENT OF <NOM DE CURSEUR>}
```

Par exemple, la mise à jour du degré du Juliénas 1983 par la valeur 13 s'effectuera par la requête (U1). L'accroissement des quantités bues de Volnay 1983 de 10% s'effectuera par la commande (U2).

```
(U1)  UPDATE   VINS
      SET      DEGRE = 13
      WHERE    CRU = "JULIENAS" AND MILLESIME = 1983

(U2)  UPDATE   ABUS
      SET      QUANTITE = QUANTITE * 1.1
      WHERE    NV IN
               SELECT   NV
               FROM     VINS
               WHERE    CRUS = "JULIENAS"
               AND      MILLESIME = "1983"
```

Figure VII.14 : Exemples de commandes de mise à jour

4.3. SUPPRESSION DE TUPLES

L'opération de suppression permet d'enlever d'une relation des tuples existants. Les tuples sont spécifiés à l'aide d'une condition de recherche, à moins que l'on désire supprimer tous les tuples d'une relation. La syntaxe de la commande de suppression est :

```
DELETE FROM <NOM DE TABLE>
[WHERE {<CONDITION DE RECHERCHE> | CURRENT OF <NOM DE CURSEUR>}]
```

Par exemple, la suppression de tous les abus de vins de degré inconnu s'effectuera par la commande (D1), et la suppression de tous les vins bus par MARTIN par la commande (D2).

```
(D1)  DELETE
      FROM      ABUS
      WHERE     NV IN
                SELECT    NV
                FROM      VINS
                WHERE     DEGRE IS NULL

(D2)  DELETE
      FROM      VINS
      WHERE     NV IN
                SELECT NV
                FROM BUVEURS, ABUS
                WHERE ABUS.NB = BUVEURS.NB
                AND BUVEURS.NOM = "MARTIN"
```

Figure VII.15 : Exemples de commandes de suppression

5. SQL1 : LA PROGRAMMATION DE TRANSACTIONS

Une transaction est une séquence d'opérations, incluant des opérations bases de données, qui est atomique vis-à-vis des problèmes de concurrence et reprise après panne. Une transaction est généralement programmée dans un langage hôte. Dans cette section, nous présentons les commandes de gestion de transactions incluses dans SQL1 et l'intégration de SQL dans un langage de programmation.

5.1. COMMANDES DE GESTION DE TRANSACTION

Une transaction est initiée par un appel de procédure base de données, si une transaction n'est pas déjà active. Elle est terminée soit avec succès par un ordre de validation des mises à jour `COMMIT WORK`, soit en échec suite à un problème par un ordre `ROLLBACK WORK`. Tout se passe comme si les mises à jour étaient préparées seulement en mémoire lors des commandes `UPDATE`, `INSERT` et `DELETE`, puis intégrées de manière atomique à la base de données par l'ordre `COMMIT WORK`, ou annulées par l'ordre `ROLLBACK WORK` (voir figure VII.16).

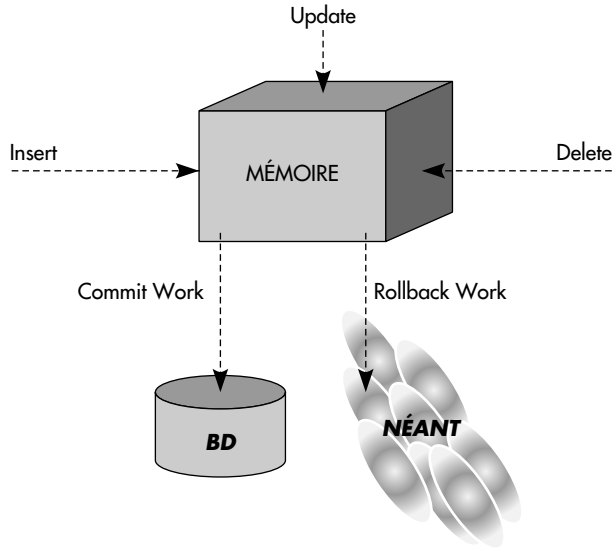


Figure VII.16 : Terminaison avec succès ou échec d'une transaction

5.2. CURSEURS ET PROCÉDURES

La programmation de transaction dans un langage hôte nécessite le travail tuple à tuple. Pour cela, SQL propose la notion de curseur. Un **curseur** permet de repérer un tuple dans la base de données à l'aide d'une commande `SELECT` et d'un balayage séquentiel des tuples résultats. Il s'agit du moyen offert par SQL1 à la fois pour balayer séquentiellement les tuples résultats d'une question, et pour repérer un tuple dans la base. Un curseur est déclaré par la commande suivante :

```
DECLARE <NOM DE CURSEUR> CURSOR
FOR <COMMANDE DE RECHERCHE>
[ORDER BY {<SPÉCIFICATION DE COLONNE> [{ASC | DESC}]}+]
```

L'utilisation d'un curseur nécessite son ouverture par la commande :

```
OPEN <NOM DE CURSEUR>.
```

Cette commande provoque l'exécution logique de la question et le positionnement du curseur sur le premier tuple résultat. Il est ensuite possible de lire successivement chaque tuple résultat et de faire avancer le curseur par la commande :

```
FETCH <NOM DE CURSEUR> INTO <NOM DE VARIABLE>+
```

La fin d'utilisation d'un curseur se signale au système par la commande :

```
CLOSE <NOM DE CURSEUR>.
```

SQL1 permet de composer des **modules**, eux-mêmes composés de **procédures** constituées d'une commande SQL, pouvant inclure des curseurs. Une procédure peut posséder des paramètres d'appel ou de retour. Elle peut être exécutée depuis un langage hôte ou plus généralement par une commande d'appel du type :

```
EXEC <NOM DE PROCÉDURE> [(<PARAMÈTRES>+)].
```

En guise d'illustration, nous avons représenté figure VII.17 un module définissant trois procédures pour ouvrir, lire et fermer les bons vins (de qualité A). Nous touchons là à l'intégration de SQL dans un langage de programmation tel PASCAL, C ou FORTRAN, que nous détaillerons dans le paragraphe suivant. Les curseurs ont un rôle essentiel pour réaliser une telle intégration ; il permettent d'effectuer des commandes de mises à jour tuple à tuple.

```
MODULE EXEMPLE
DECLARE BONVIN CURSOR
   FOR SELECT NV
   FROM VINS
   WHERE QUALITE = "A" ;
PROCEDURE OPENVIN ;
   OPEN BONVIN ;
PROCEDURE NEXTVIN(NV) NV INTEGER ;
   FETCH BONVIN INTO NV ;
PROCEDURE CLOSEVIN ;
   CLOSE BONVIN ;
```

Figure VII.17 : Exemple de module

5.3. INTÉGRATION DE SQL ET DES LANGAGES DE PROGRAMMATION

L'intégration de SQL à un langage procédural tel que C, COBOL, FORTRAN, PASCAL ou PL1 pose différents problèmes :

1. L'exploitation tuple à tuple des résultats des requêtes nécessite l'usage de curseurs. La commande FETCH est alors utilisée dans une boucle pour lire un à un les tuples résultats. Des commandes de mises à jour avec l'option de recherche WHERE CURRENT OF <nom de curseur> peuvent être employées pour les mises à jour.
2. Les variables du langage de programmation doivent être passées au SGBD par l'intermédiaire des requêtes SQL. Toute constante peut ainsi être remplacée par une variable de programmation qui doit être déclarée dans une section de déclaration de variables SQL. Un résultat peut être assigné à une variable de programme déclarée à SQL en liste résultat de l'ordre FETCH.

La figure VII.18 donne un exemple de programme PASCAL utilisant SQL. Ce programme ajoute un abus à la base pour les buveurs de Lyon ayant participé à une réception le 10-08-96. Ceux-ci sont déterminés par interrogation de l'utilisateur qui doit répondre O (oui) suite à l'affichage du nom du buveur si celui-ci a participé à la réception. Il doit alors préciser la quantité bue ce jour par le buveur. Le vin bu est déterminé par la constante `numvin`.

```
PROGRAM EXEMPLE
REPONSE STRING ;
RECORD TYPE BUVEUR
   NB INTEGER ;
   NOM STRING ;
   PRENOM STRING ;
END RECORD ;
EXEC SQL DECLARE SECTION END EXEC ;
   CONS NUMVIN = 100 ;
   VAR B BUVEUR ;
   VAR QUANTITÉ INTEGER ;
   VAR CODE SQLCODE ;
EXEC SQL END DECLARE SECTION END EXEC ;
EXEC SQL DECLARE CURSOR PARTICIPANT FOR
   SELECT NB, NOM, PRENOM
   FROM BUVEURS
   WHERE ADRESSE LIKE "%LYON%"
END EXEC ;
BEGIN
   EXEC SQL OPEN PARTICIPANT END EXEC ;
   WHILE CODE ≠ 100 DO
      BEGIN
         EXEC SQL FETCH PARTICIPANT INTO B END EXEC ;
         PRINT "NOM :", B.NOM, "PRENOM", B.PRENOM ;
         PRINT "CE BUVEUR A-T-IL PARTICIPÉ ?"
         READ REPONSE ;
         IF REPONSE = "O" THEN
            BEGIN
               PRINT "QUANTITÉ ?" ;
               READ QUANTITÉ ;
               EXEC SQL INSERT INTO ABUS
               (B.NB, NUMVIN, "10-08-96", QUANTITÉ) END EXEC ;
            END ;
      END ;
    EXEC SQL CLOSE PARTICIPANT END EXEC ;
END.
```

Figure VII.18 : Exemple de programme intégrant des commandes SQL

En conclusion, on notera la lourdeur des programmes intégrant SQL à un langage de programmation. Ceci résulte tout d'abord de la différence de point de vue entre le

modèle relationnel qui supporte des requêtes ensemblistes et les langages classiques qui sont avant tout séquentiels. Ces derniers permettent seulement d'écrire des programmes traitant un tuple (record) à la fois. D'où la nécessité de boucles complexes. La complexité résulte aussi de l'existence de deux systèmes de définition de types différents, dans notre exemple celui de PASCAL (`RECORD TYPE`) et celui de SQL (`CREATE TABLE`). D'où la nécessité de doubles déclarations. Tout ces problèmes sont souvent mieux résolus au niveau des langages de 4e génération, qui malheureusement ne sont pas standardisés.

6. SQL2 : LE STANDARD DE 1992

SQL2 [ISO92] est une extension de SQL1 devenu le standard d'accès aux bases de données relationnelles depuis 1992. Compte tenu de la complexité de SQL2 qui se veut très complet, trois niveaux sont distingués :

1. **SQL2 entrée** est avant tout une correction de la norme SQL1 de 1989 et un complément avec les commandes manquantes indispensables.
2. **SQL2 intermédiaire** apporte les compléments relationnels indispensables au support complet du modèle et de l'algèbre ainsi que des types de données plus variés.
3. **SQL2 complet** apporte des types de données encore plus variés et quelques compléments non indispensables.

Dans la suite, nous détaillons successivement ces trois niveaux supplémentaires de SQL. Ils sont en principe des extensions compatibles de SQL1.

6.1. LE NIVEAU ENTRÉE

SQL2 entrée propose une standardisation des codes réponses en ajoutant une variable retour des commandes appelées `SQLSTATE`. En effet, avec SQL1 un seul code réponse est retourné dans une variable de type `SQLCODE`. Trois valeurs sont spécifiées : 0 pour exécution correcte, +100 pour absence de données et une valeur négative – n pour toute erreur, la valeur étant spécifiée par le concepteur du SGBD. Cela rend les programmes non portables. Afin d'augmenter la portabilité, un code retour `SQLSTATE` est ajouté (`SQLCODE` est gardé afin d'assurer la compatibilité avec SQL1). Le code `SQLSTATE` est composé de deux caractères spécifiant la classe d'erreur (par exemple 22 pour exception erreurs sur les données) et de trois caractères précisant la sous-classe (par exemple, 021 pour caractère invalide). Les codes classes et sous-classes sont partiellement standardisés.

Avec SQL2, il est possible de renommer des colonnes résultats. Cette fonctionnalité est très utile, par exemple lors du calcul d'un agrégat. Avec SQL1, la colonne de la table résultat prend souvent un nom dépendant du système. La clause AS de SQL2 permet de résoudre ce problème. Par exemple, une question calculant la moyenne des degrés par crus pourra maintenant spécifier un nom de colonne résultat MOYENNE comme suit :

```
SELECT   CRU, AVG(DEGRE) AS MOYENNE
FROM     VINS
GROUP BY CRU
```

Un autre ajout proposé à SQL1 au niveau de SQL2 entrée est la possibilité d'utiliser les mots clés de SQL comme des noms de table, d'attributs, etc. Pour ce faire, il suffit d'inclure ces mots clés entre double cotes. Par exemple, on pourra créer une table de nom SELECT par la commande :

```
CREATE TABLE "SELECT" (QUESTION CHAR(100)).
```

En résumé, les extensions de SQL1 proposées au niveau de SQL2 entrée sont donc des corrections et clarifications nécessaires au langage. SQL2 entrée est donc le nouveau standard minimal que devrait à terme fournir les constructeurs de SGBD. Il permettra une meilleure portabilité des programmes. L'interface avec C est d'ailleurs aussi précisée au niveau de SQL2 entrée.

6.2. LE NIVEAU INTERMÉDIAIRE

6.2.1. Les extensions des types de données

SQL2 intermédiaire offre un meilleur support du temps. Trois nouveaux types de données DATE, TIME et TIMESTAMP sont introduits. Le type DATE est spécifié pour un attribut contenant une année, un mois et un jour (par exemple 1992/08/21). Le type TIME correspond à un groupe heures, minutes et secondes (par exemple, 09 :17 :23). Le type TIMESTAMP correspond à une concaténation d'une date et d'une heure (DATE concaténé à TIME). SQL2 offre également un type de données permettant de spécifier un intervalle de temps (INTERVAL) avec une précision en mois-année ou en seconde-minute-heure-jour.

Les opérations arithmétiques d'addition et soustraction sur les dates-heures et intervalles sont supportées avec SQL2. Il est aussi possible de multiplier ou diviser des intervalles par des valeurs numériques. Par exemple, en supposant l'attribut DATE de la table ABUS défini comme une date, il est possible de retrouver tous les abus commis dans un intervalle de temps de N mois (N est de type intervalle) par rapport à la date du 21-08-96 par la question suivante :

```
SELECT   *
FROM     ABUS
WHERE    DATE - 21/08/1996 < N
```

SQL2 admet également la création de domaines par l'utilisateur. Cela permet en particulier l'introduction de types de données par l'utilisateur au sein du modèle avec un meilleur contrôle de l'intégrité de domaine ; ces types restent en SQL2 purement des sous-ensembles des types prédéfinis ; aucune opération spécifique ne peut leur être associée. Par exemple, la création d'un domaine MONNAIE s'effectuera par la commande :

```
CREATE DOMAINE MONNAIE IS DECIMAL(5,2)
DEFAULT (-1)
CHECK (VALUE = -1 OR VALUE > 0)
NOT NULL
```

Ce domaine pourra être utilisé lors d'une création de table. Il s'agit essentiellement d'une macro-définition permettant d'inclure les contrôles d'intégrité, par exemple dans une table FACTURE :

```
CREATE TABLE FACTURE (NOM CHAR(5), MONTANT MONNAIE)
```

D'autres extensions sont proposées pour un meilleur support de types de données variés. On citera en particulier :

– le support de multiples alphabets et ensembles de caractères (CHARACTER SET) ;
– le support de différents ordres des lettres (COLLATE) ;
– la possibilité de concaténer des chaînes de caractères (||) et d' extraire des sous-chaînes (SUBSTRING) ;
– les facilités de conversion de types de données (CAST) ;
– les chaînes de caractères de longueur variables (CHAR VARYING) et la possibilité d'extraire la longueur d'une chaîne (fonction LENGTH).

6.2.2. Un meilleur support de l'intégrité

L'intégrité référentielle est supportée en SQL1, mais toute tentative de violation entraîne le rejet de la mise à jour. Au contraire, SQL2 intermédiaire permet de spécifier certaines actions correctives en cas de violation d'intégrité lors d'une suppression d'un tuple référencé, selon les options :

– cascader les suppressions (ON DELETE CASCADE),
– rendre nul l'attribut référençant (ON DELETE SET NULL).

L'option doit être précisée lors de la définition de la contrainte référentielle dans la table référençante.

Plus généralement, les contraintes d'intégrité peuvent être nommées lors de leurs définitions. La validation des contraintes d'intégrité peut être immédiate à la fin de chaque opération ou différée en fin de transaction. Ceci est indiqué par une clause :

```
SET CONSTRAINTS <NOM DE CONTRAINTE>+ {DEFERRED | IMMEDIATE}
```

Le nom de contrainte ALL indique que toutes les contraintes sont concernées.

Différents niveaux de contrôle de transactions sont aussi possibles. Une clause

```
SET TRANSACTION <MODE>
```

est introduite, permettant de préciser le niveau d'isolation désiré (0 = aucune pour lecture non protégée, 1 = écriture protégée, lecture non protégée, 2 = écriture et lecture protégées, 3 = écriture et lecture exclusives), le mode d'accès (lecture seule READ ONLY, ou lecture-écriture READ-WRITE), et le nombre de diagnostics possibles.

6.2.3. Une intégration étendue de l'algèbre relationnelle

L'algèbre relationnelle offre les opérations ensemblistes d'union, intersection et différence de relations. SQL1 ne supporte que l'union. SQL2 généralise les expressions de SELECT en introduisant intersection (INTERSECT) et différence (EXCEPT). Des expressions parenthésées de SELECT sont même possibles. De plus, l'union est généralisée à l'union externe, avec complément des schémas des relations au même schéma par ajout de valeurs nulles aux tuples, cela préalablement à l'union proprement dite.

Afin d'illustrer ces possibilités, imaginons une table NONBUVEURS (NB, NOM, PRENOM, ADRESSE), complémentaire de la table BUVEURS(NB, NOM, PRENOM, ADRESSE, TYPE). La requête suivante construit une table contenant buveurs et non buveurs avec un type nul, à l'exception des gros buveurs :

```
SELECT          *
FROM            NONBUVEURS
OUTER UNION
   (SELECT      *
   FROM      BUVEURS
EXCEPT
   SELECT       *
   FROM         BUVEURS
   WHERE          TYPE = "GROS")
```

Les jointures externes sont utiles pour retenir lors d'une jointure les tuples d'une table n'ayant pas de correspondant dans l'autre table, avec des valeurs nulles associées. On distingue ainsi des jointures externes droite, gauche et complète selon que l'on retient les tuples sans correspondant des deux tables ou seulement d'une. Rappelons aussi qu'une jointure est dite naturelle si elle porte sur des attributs de même nom. SQL2 offre la possibilité de spécifier de telles jointures au niveau de la clause FROM, selon la syntaxe suivante :

```
FROM <NOM DE TABLE> [NATURAL] [{LEFT | RIGHT}]
   JOIN <NOM DE TABLE>
   [ON (<SPÉCIFICATION DE COLONNE>+=<SPÉCIFICATION DE COLONNE>+)]
```

On peut par exemple retrouver la somme des quantités bues par chaque buveur, y compris les buveurs n'ayant pas bu par la requête suivante :

```
SELECT      NB, NOM, SUM(QUANTITE)
FROM        BUVEURS NATURAL LEFT JOIN ABUS
GROUP BY    NB
```

6.2.4. La possibilité de modifier les schémas

SQL2 permet de modifier un schéma de table à l'aide de la commande :

```
ALTER TABLE <nom de table> <altération>
```

Différents types d'altérations sont possibles :

– ajout d'une colonne (ADD COLUMN)
– modification de la définition d'une colonne (ALTER COLUMN)
– suppression d'une colonne (DROP COLUMN)
– ajout d'une contrainte (ADD CONSTRAINT)
– suppression d'une contrainte (DROP CONSTRAINT).

Par exemple, l'ajout de l'attribut REGION à la table VINS pourra s'effectuer par la commande :

```
ALTER TABLE VINS ADD COLUMN REGION CHAR VARYING.
```

6.2.5. L'exécution immédiate des commandes SQL

Avec SQL1, toute commande SQL est compilée puis exécutée depuis un programme d'application. Les processus de compilation et d'exécution sont séparés et non contrôlés. Au contraire, SQL2 supporte une option dynamique incluant les possibilités d'exécution différée ou immédiate, cette dernière étant utile par exemple en interactif. Des commandes PREPARE <commande SQL> et EXECUTE [IMMEDIATE] <commande SQL> sont introduites afin de permettre l'exécution immédiate (EXECUTE IMMEDIATE) ou la compilation puis l'exécution de multiples fois à partir du résultat de la compilation (PREPARE suivi de plusieurs EXECUTE). Tous ces aspects de mode de fonctionnement sont ainsi intégrés au langage.

6.3. LE NIVEAU COMPLET

SQL2 offre un niveau complet (FULL SQL2) qui comporte en plus les fonctionnalités suivantes :

– type de données chaînes de bits (BIT) pour supporter des objets complexes tels des images ;

– extension du support des dates et temps, notamment avec la possibilité de définir des zones d'heures, des intervalles de temps d'échelles variées, etc. ;
– expressions de requêtes `SELECT` étendues avec correspondances de colonnes possibles ; par exemple, lors d'une union deux colonnes de nom différents peuvent être transformées en une seule colonne ;
– support de tables temporaires privées détruites en fin de transaction ;
– possibilité de `SELECT` en argument d'un `FROM` afin de construire une table temporaire à l'intérieur d'une requête SQL ;
– maintenance de vues concrètes facilitant l'interrogation de vues peu modifiées, notamment de vues avec agrégats ;
– support de contraintes d'intégrité multitables à l'aide de sous-questions intégrables dans la clause `CHECK` de déclaration de contraintes.

Ces multiples extensions et quelques autres, par exemple pour généraliser la maintenance automatique des contraintes référentielles lors des mises à jour, font de SQL2 complet un langage plutôt complexe pour manipuler des bases de données relationnelles. La spécification de SQL2 comporte 522 pages de syntaxe.

7. CONCLUSION

Le standard SQL2 est adopté depuis 1992 par les organismes de standardisation internationaux (ISO). Une large extension appelée SQL3 est en cours de finition et devrait être adoptée en 1999. SQL3 intègre les fonctionnalités nouvelles non purement relationnelles. En particulier, sont traitées au niveau de SQL3 les fonctionnalités orientées objet, le support de questions récursives et le support de règles déclenchées par des événements bases de données (en anglais, *triggers*). Les fonctionnalités orientées objet sont offertes sous forme de constructeurs de types abstraits de données permettant la définition d'objets complexes par l'utilisateur. Les questions récursives permettent d'intégrer l'opérateur de fermeture transitive à SQL. Une syntaxe est proposée pour définir des *triggers*. Ces fonctionnalités seront étudiées dans les chapitres qui suivent. Plus spécifiquement, SQL3 se veut le langage des SGBD objet-relationnel ; à ce titre, nous l'étudierons donc dans la troisième partie de cet ouvrage.

En résumé, SQL est un langage standardisé de programmation de bases de données. Bien qu'à l'origine issu du modèle relationnel, SQL est aujourd'hui un langage qui couvre un spectre plus large. Les standards SQL1, SQL2 et SQL3 traitent de l'ensemble des aspects des bases de données, dont la plupart seront étudiés dans la suite. SQL sert aussi de langage d'échange de données entre SGBD. Cela explique donc son très grand succès, qui se traduit par le fait que tous les SGBD offrent une

variante de SQL. La normalisation du langage assure une bonne portabilité des programmes bases de données. Elle a été possible grâce au soutient des grands constructeurs, tels IBM, DEC, ORACLE et SYBASE.

Certains critiquent le manque de rigueur et de cohérence de SQL [Date84]. Il est vrai que l'ensemble peut paraître parfois hétéroclite. La syntaxe est souvent lourde. Quoi qu'il en soit, SQL est et reste la référence. Il est le langage de nombreux systèmes commercialisés tels Oracle, DB2, Sybase, Informix et SQL Server.

8. BIBLIOGRAPHIE

[Boyce75] Boyce R., Chamberlin D.D., King W.F., Hammer M., « Specifying Queries as Relational Expressions », *Comm. de l'ACM*, vol. 18, n° 11, novembre 1975.

Une présentation du language SQUARE qui permet d'exprimer en anglais simplifié des expressions de l'algèbre relationnelle. SQUARE est à l'origine du langage SQL.

[Chamberlin76] Chamberlin D.D., Astrahan M.M., Eswaran K.P., Griffiths P., Lorie R.A, *et al.*, « SEQUEL 2 : A Unified Approach to Data Definition, Manipulation and Control », *IBM Journal of Research and Development*, vol. 20, n° 6, novembre 1976.

Cet article décrit la deuxième version du langage SEQUEL, le langage du fameux système R, le premier SGBD relationnel prototypé à IBM San-José de 1974 à 1980. Pendant cette période, l'université de Berkeley réalisait INGRES. SEQUEL 2 étend SEQUEL 1 avec des constructions dérivées de QUEL – le langage de INGRES – et permet de paraphraser en anglais les expressions de l'algèbre. Il introduit aussi les commandes de description et contrôle de données et constitue en cela un langage unifié. C'est en tout cas un langage assez proche de SQL1.

[Date84] Date C.J., « A Critique of the SQL Database Language », *ACM SIGMOD Record*, vol. 14, n° 3, novembre 1984.

Chris Date, qui vient de quitter IBM en cette fin de 1984, critique la cohérence du langage SQL et démontre quelques insuffisances.

[Date89] Date C.J., *A Guide to the SQL Standard*, 2e édition, Addison-Wesley, Reading, Mass., 1989.

Une présentation didactique du standard SQL, avec beaucoup d'exemples.

[IBM82] IBM Corporation, « SQL/Data System Terminal Users Guide », *IBM Form Number SH24-5017-1, 1982.*

La présentation du langage du système SQL/DS d'IBM, disponible sur DOS/VSE. Il s'agit de la première implémentation commercialisée du langage SQL.

[IBM87] IBM Corporation, « Systems Application Architecture (SAA) : Common *Programming Interface, Database Reference », IBM Form Number SC26-4348-0, 1987.*

La définition du standard de convergence des systèmes IBM supportant SQL, dans le cadre de l'architecture unifiée SAA. En principe, tous les systèmes IBM réalisés dans des centres différents (DB2, SQL/DS, SQL AS/400, SQL OS2) ont convergé ou convergeront vers un même langage défini dans ce document, très proche de la norme SQL1.

[ISO89] International Organization for Standardization, « Information Processing Systems – Database Language SQL with Integrity Enhancement », *International Standard ISO/IEC JTC1 9075* : 1989(E), 2e édition, avril 1989.

Ce document de 120 pages présente la norme SQL1 : concepts de base, éléments communs, langage de définition de schémas, définition de modules de requêtes, langage de manipulation. Toute la grammaire de SQL1 est décrite en BNF. Ce document résulte de la fusion du standard de 1986 et des extensions pour l'intégrité de 1989.

[ISO92] International Organization for Standardization, « Database Language SQL », *International Standard ISO/IEC JTC1/SC21* Doc. 9075 N5739, 1992.

Ce document de 522 pages présente la norme SQL2 : définitions, notations et conventions, concepts, expressions de scalaires, expressions de questions, prédicats, langage de définition et manipulation de schémas, langage de manipulation de données, langage de contrôle, SQL dynamique, SQL intégré à un langage, codes réponses, etc. Les niveaux entrée et intermédiaire sont clairement distingués. L'ensemble constitue un document très complet qui doit être accepté par l'ISO en 1992. L'approbation de cette nouvelle norme demande un vote positif de 75% des corps représentatifs de l'ISO dans les différents pays habilités.

[Melton96] Melton J., « An SQL3 Snapshot », *Proc. Int. Conf. On Data Engineering*, IEEE Ed., p. 666-672, 1996.

Ce bref article donne un aperçu du langage SQL3 tel qu'il était en 1996. SQL3 est la nouvelle version de SQL pour les systèmes objet-relationnel. Nous étudierons la version actuelle de SQL3 plus loin dans cet ouvrage. J. Melton était à cette époque le responsable de la normalisation de SQL.

[Shaw90] Shaw Ph., « Database Language Standards : Past, Present, and Future », *Lecture Notes in Computer Science*, n° 466, Database Systems of the 90s, A. Blaser Ed., Springer Verlag, novembre 1990.

Cet article fait le point sur les efforts de standardisation de SQL. Il résume les développements passés en matière de standardisation des bases de données et introduit les propositions SQL2 et SQL3. Les niveaux de SQL2 sont particulièrement développés et illustrés par des exemples. Phil Shaw était à cette époque le responsable de la normalisation de SQL.

[X/Open92] X/Open Group, « Structured Query Language (SQL) » Common Application Environment *CAE Specification C201*, septembre 1992.

Ce document est une présentation du langage SQL2 élaborée par l'X/OPEN Group.

[Zook77] Zook W. *et al.,* « INGRES Reference Manual », Dept. of EECS, University of California, Berkeley, CA, 1977.

Ce document décrit les interfaces externes de la première version d'INGRES et plus particulièrement le langage QUEL.

[Zloof77] Zloof M., « Query-by-Example : A Data Base Language », *IBM Systems Journal*, vol. 16, n° 4, 1977, p. 324-343.

Cet article présente QBE, le langage par grille proposé par Zloof, alors chercheur à IBM. Ce langage bidimensionnel est aujourd'hui opérationnel en surcouche de DB2 et aussi comme interface externe du système Paradox de Borland. Zloof discute aussi des extensions bureautiques possibles, par exemple pour gérer le courrier (OBE).

Chapitre VIII

INTÉGRITÉ ET BD ACTIVES

1. INTRODUCTION

Un SGBD doit garantir la cohérence des données lors des mises à jour de la base. En effet, les données d'une base ne sont pas indépendantes, mais obéissent à des règles sémantiques appelées **contraintes d'intégrité**. Ces règles peuvent être déclarées explicitement et mémorisées dans le dictionnaire de données, ou plus discrètement implicites. Les transactions sont des groupes de mises à jour dépendantes qui font passer la base d'un état cohérent à un autre état cohérent. À la fin de chaque transaction, ou plus radicalement après chaque mise à jour, il est nécessaire de contrôler qu'aucune règle d'intégrité n'est violée.

Une contrainte d'intégrité peut spécifier l'égalité de deux données ; par exemple, un numéro de vins dans la table VINS doit être égal au numéro du même vin dans la table `ABUS`. De manière plus complexe, elle peut spécifier une assertion comportant de multiples données ; par exemple, la somme des avoirs des comptes doit rester égale à l'avoir de la banque. Nous étudions ci-dessous les différents types de contraintes supportées par le modèle relationnel. Quelle que soit la complexité de la contrainte, le problème est de rejeter les mises à jour qui la violent. Pour ce faire, différentes techniques sont possibles, fondées soit sur la prévention qui consiste à empêcher les mises à jour non valides de se produire, soit sur la détection impliquant de défaire les transactions incorrectes.

Une autre manière de protéger l'intégrité des données est l'utilisation de **déclencheurs** (en anglais, *triggers*). Ceux-ci permettent de déclencher une opération conséquente suite à une première opération sur la base. La forme générale d'un déclencheur est `ON <événement> IF <condition> THEN <action>`. L'événement est souvent une action de mise à jour de la base. La condition est un prédicat logique vrai ou faux. L'action peut permettre d'interdire la mise à jour (`ABORT`) ou de la compenser (`UPDATE`). Ainsi, en surveillant les mises à jour et en déclenchant des effets de bord, il est possible de maintenir l'intégrité d'une base.

Mieux, les déclencheurs permettent de modéliser au sein de la base de données le comportement réactif des applications. Les SGBD traditionnels sont passifs, en ce sens qu'ils exécutent des commandes de mises à jour et de recherche en provenance des applications. Avec des déclencheurs, ils deviennent actifs et sont capables de réagir à des événements externes. Par exemple, la surveillance d'un commerce électronique peut nécessiter le refus de vente à un client suspect, une demande d'approvisionnement en cas de rupture de stock, etc. Tous ces événements peuvent être capturés directement par le SGBD avec des déclencheurs appropriés. On passe alors à la notion de **base de données actives**, qui peut comporter des règles avec conditions déclenchées par des événements composées de plusieurs sous-événements (par exemple, une conjonction d'événements simples). Une base de données active permet donc de déplacer le comportement réactif des applications dans le SGBD. Ceci nécessite la prise en compte d'un modèle de définition de connaissances et d'un modèle d'exécution de règles au sein du SGBD. Nous examinerons ces aspects dans la deuxième partie de ce chapitre.

Ce chapitre traite donc des règles d'intégrité et des bases de données actives. Après cette introduction, la section 2 examine les différents types de contraintes d'intégrité et résume le langage de définition de contraintes de SQL2. La section 3 introduit quelques techniques d'analyse (contrôle de cohérence, de non-redondance) de contraintes et quelques techniques de simplification : simplifications possibles compte tenu du type d'opération, différenciations en considérant les delta-relations, etc. La section 4 montre comment contrôler les contraintes lors des mises à jour : diverses techniques curatives ou préventives sont étudiées, pour aboutir à la technique préventive au vol souvent appliquée pour les contraintes simples exprimables en SQL. La section 5 introduit les notions de base de données active et de déclencheur, et analyse les composants d'un SGBD actif. La section 6 étudie plus en détail les déclencheurs et donne l'essentiel de la syntaxe SQL3, les déclencheurs n'apparaissant qu'à ce niveau dans la norme. La section 7 montre comment sont exécutés les déclencheurs dans un SGBD actif. Au-delà de SQL, elle soulève quelques problèmes épineux liés aux déclencheurs.

2. TYPOLOGIE DES CONTRAINTES D'INTÉGRITÉ

Dans cette section, nous étudions les différents types de contraintes d'intégrité. Celles-ci sont classées selon leur utilisation à modéliser des relations statiques entre données, ou des relations dynamiques d'évolution des données. À un second niveau, nous énumérons les différents types de contraintes.

2.1. CONTRAINTES STRUCTURELLES

Une **contrainte structurelle** fait partie intégrante du modèle et s'applique sur les structures de base (table, colonne, ligne).

Notion VIII.1 : Contrainte structurelle (*Structural constraint*)

Contrainte d'intégrité spécifique à un modèle exprimant une propriété fondamentale d'une structure de données du modèle.

Les contraintes structurelles sont généralement statiques. Pour le modèle relationnel, elles permettent d'exprimer explicitement certaines propriétés des relations et des domaines des attributs. Ces contraintes sont donc partie intégrante du modèle et ont déjà été introduites lors de sa présentation. Il est possible de distinguer les contraintes structurelles suivantes :

1. **Unicité de clé**. Elle permet de préciser les attributs clés d'une relation, c'est-à-dire un groupe d'attributs non nul dont la valeur permet de déterminer un tuple unique dans une table. Par exemple, la table `VINS` possède une clé unique `NV` ; la table `ABUS` possède une clé multiple (`NV, NB, DATE`).
2. **Contrainte référentielle**. Elle spécifie que toute valeur d'un groupe de colonnes d'une table doit figurer comme valeur de clé dans une autre table. Une telle contrainte représente une association obligatoire entre deux tables, la table référencée correspondant à l'entité, la table référençante à l'association. Par exemple, toute ligne de la table `ABUS` référencera un numéro de vin existant dans la table `VINS`, ou toute ligne de commande référencera un produit existant, etc.
3. **Contrainte de domaine**. Ce type de contrainte permet de restreindre la plage de valeurs d'un domaine. En général, un domaine est défini par un type et un éventuel domaine de variation spécifié par une contrainte de domaine. Une contrainte de domaine peut simplement préciser la liste des valeurs permises (définition en extension) ou une plage de valeurs (contrainte en intention). Par exemple, un cru sera choisi parmi {Volnay, Beaujolais, Chablis, Graves, Sancerre} ; une quantité de vin sera comprise entre 0 et 100.

4. **Contrainte de non nullité**. Une telle contrainte spécifie que la valeur d'un attribut doit être renseignée. Par exemple, le degré d'un vin ne pourra être nul, et devra donc être documenté lors de l'insertion d'un vin dans la base, ou après toute mise à jour.

Le choix des contraintes structurelles est effectué lors de la définition d'un modèle. Codd a par exemple retenu la notion de clé composée d'attributs visibles à l'utilisateur pour identifier les tuples dans une table. Ce choix est plutôt arbitraire. Le modèle objet a choisi d'utiliser des identifiants système appelés identifiants d'objets. Codd aurait pu retenir les identifiants de tuples invariants (TID) pour identifier les tuples. On aurait alors un modèle relationnel différent, mais ceci est une autre histoire. Au contraire, dans sa première version, le modèle relationnel n'introduisait pas les contraintes référentielles : elles ont été ajoutées en 1981 pour répondre aux critiques des tenants du modèle réseau, qui trouvaient que le modèle relationnel perdait la sémantique des associations [Date81].

2.2. CONTRAINTES NON STRUCTURELLES

Les autres contraintes d'intégrité, non inhérentes au modèle de données, sont regroupées dans la classe des contraintes non structurelles. La plupart traitent plutôt de l'évolution des données suite aux mises à jour ; elles sont souvent appelées **contraintes de comportement**.

Notion VIII.2 : Contrainte de comportement (*Behavioral constraint*)

Contrainte d'intégrité exprimant une règle d'évolution que doivent vérifier les données lors des mises à jour.

Certaines contraintes structurelles peuvent aussi être qualifiées de contraintes de comportement (par exemple l'unicité de clé). Quoi qu'il en soit, par opposition aux contraintes structurelles, les non structurelles ne font pas partie intégrante du modèle relationnel, et ne sont donc pas définies dans la commande `CREATE TABLE`. Elles sont définies par la commande additionnelle `CREATE ASSERTION`. Dans le cas du modèle relationnel, la plupart peuvent être exprimées sous forme d'assertions de la logique du premier ordre, éventuellement temporelles. On distingue en particulier :

1. Les **dépendances fonctionnelles**. Celles-ci expriment l'existence d'une fonction permettant de déterminer la valeur d'un groupe d'attributs à partir de celle d'un autre groupe. Comme nous le verrons dans le chapitre sur la conception, on dit que $X \rightarrow Y$ (X détermine Y) si pour toute valeur de X il existe une valeur unique de Y associée. Par exemple, le cru détermine uniquement la région (dans une table de vins français).

2. Les **dépendances multivaluées**. Ce sont une généralisation des précédentes au cas de fonctions multivaluées. On dit que `X->> Y` (X multidétermine Y) dans une relation R si pour toute valeur de X il existe un ensemble de valeur de Y, et ceci indépendamment des valeurs des autres attributs Z de la relation R. Par exemple, dans une relation `BUVEURS (NOM, CRU, SPORT)` décrivant les vins bus (`CRU`) et les sports pratiqués (`SPORT`) par les buveurs, `NOM ->> CRU` indépendamment de `SPORT`.
3. Les **dépendances d'inclusion**. Elles permettent de spécifier que les valeurs d'un groupe de colonnes d'une table doivent rester incluses dans celles d'un groupe de colonnes d'une autre table. Elles généralisent donc les contraintes référentielles vues ci-dessus aux cas de colonnes quelconques. Par exemple, la colonne `VILLE` de la table `BUVEURS` doit rester incluse dans la colonne `VILLE` de la table `REGION`.
4. Les **contraintes temporelles**. Plus sophistiquées que les précédentes, elles font intervenir le temps. Elles permettent de comparer l'ancienne valeur d'un attribut à la nouvelle après mise à jour. On exprimera par exemple avec une telle contrainte le fait qu'un salaire ne peut que croître.
5. Les **contraintes équationnelles**. Il s'agit là de comparer deux expressions arithmétiques calculées à partir de données de la base et de forcer l'égalité ou une inégalité. La dimension temporelle peut être prise en compte en faisant intervenir des données avant et après mise à jour. Les calculs d'agrégats sont aussi possibles. Un exemple simple est une contrainte permettant d'exprimer, dans une base de gestion de stocks, le fait que la quantité en stock est égale à la somme des quantités achetées moins la somme des quantités vendues, et ce pour tout produit. Il est aussi possible d'exprimer des invariants en utilisant la dimension temps, par exemple le fait que l'avoir d'une banque reste le même après un transfert de fonds d'un compte à un autre.

Nous regroupons sous le terme **dépendances généralisées** les différents types de dépendances entre attributs (dépendances fonctionnelles, multivaluées et d'inclusion). Très utiles pour la conception des bases de données, elles permettent de mieux contrôler les redondances au sein des relations.

2.3. EXPRESSION DES CONTRAINTES EN SQL

SQL1 permet la création de contraintes lors de la création des tables, par la commande indiquée figure VIII.1. On retrouve les contraintes de non nullité, d'unicité de clé, référentielles et de domaines. Les contraintes mono-attributs peuvent être déclarées à chaque attribut, alors que les celles portant sur plusieurs attributs, appelées contraintes de relation, sont factorisées à la fin de la déclaration. La figure VIII.2 illustre de telles contraintes classiques pour la table ABUS.

```
CREATE TABLE <nom de table>
({<Attribut> <Domaine> [<CONTRAINTE D'ATTRIBUT>]}+)
   [<CONTRAINTE DE RELATION>]

<CONTRAINTE D'ATTRIBUT> ::=
      NOT NULL |
      UNIQUE | PRIMARY KEY
      REFERENCES <Relation> (<Attribut>) |
      CHECK <Condition>

<CONTRAINTE DE RELATION> ::=
      UNIQUE (<Attribut>+) | PRIMARY KEY (<Attribut>+) |
      FOREIGN KEY (<Attribut>+)
         REFERENCES <Relation> (<Attribut>+) |
      CHECK <Condition>
```

Figure VIII.1 : Création de table et contrainte d'intégrité en SQL1

```
CREATE TABLE ABUS (
      NB INT NOT NULL,
      NV INT NOT NULL REFERENCES VINS(NV),
      DATE DEC(6) CHECK BETWEEN 010180 AND 311299,
      QUANTITE SMALLINT DEFAULT 1,
      PRIMARY KEY(NB, NV, DATE),
      FOREIGN KEY NB REFERENCES BUVEURS,
      CHECK (QUANTITE BETWEEN 1 AND 100))
```

Figure VIII.2 : Exemple de création de table avec contrainte en SQL1

SQL2 étend d'une part les contraintes attachées à une table et permet de déclarer d'autres contraintes par une commande séparée `CREATE ASSERTION`. L'extension essentielle au niveau du `CREATE TABLE` porte sur les contraintes référentielles. Il devient possible de répercuter certaines mises à jour de la relation référencée. La nouvelle syntaxe est donnée figure VIII.3. La clause `ON DELETE` indique l'action que doit exécuter le système dans la table dépendante lors d'une suppression dans la table maître. `NO ACTION` ne provoque aucune action, et a donc un effet identique à l'absence de la clause comme en SQL1. `CASCADE` signifie qu'il faut enlever les tuples correspondant de la table dépendante. `SET DEFAULT` indique qu'il faut remplacer la clé étrangère des tuples correspondant de la table dépendante par la valeur par défaut qui doit être déclarée au niveau de l'attribut. `SET NULL` a un effet identique, mais cette fois avec la valeur nulle. La clause `ON UPDATE` indique comment le système doit modifier la clé étrangère dans la table dépendante lors de la mise à jour d'une clé primaire dans la table maître. Les effets sont identiques au `ON DELETE`, à ceci près que CASCADE provoque la modification de la clé étrangère de la même manière que la clé primaire.

De même pour la clause ON UPDATE lors d'une mise à jour de la clé référencée dans la table maître.

```
FOREIGN KEY (<Attribut>+)
REFERENCES <Relation> (<Attribut>+)
[ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
[ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
```

Figure VIII.3 : Contrainte référentielle en SQL2

La syntaxe de la nouvelle clause de définition de contraintes est résumée figure VIII.4. Une telle contrainte peut être vérifiée immédiatement après chaque mise à jour précisée (clause AFTER), ou en fin de transaction (clause BEFORE COMMIT). La condition peut porter sur une table entière (option FOR), ou sur chaque ligne de la table (option FOR EACH ROW OF).

```
CREATE ASSERTION <nom de contrainte>
[{BEFORE COMMIT |
AFTER {INSERT|DELETE|UPDATE[OF(Attribut+)]} ON <Relation>}...]
CHECK <Condition>
[FOR [EACH ROW OF] <Relation>]
```

Figure VIII.4 : Création de contrainte indépendante en SQL2

Voici quelques exemples. L'assertion suivante permet de vérifier d'une manière un peu détournée que chaque vin à bien un degré supérieur à 10 :

```
CREATE ASSERTION MINDEGRÉ
BEFORE COMMIT
CHECK (SELECT MIN(DEGRÉ) FROM VINS > 10)
FOR VINS ;
```

Celle qui suit vérifie que la quantité totale bue reste inférieure à 100 pour chaque buveur :

```
CREATE ASSERTION SOMMEQUANTITÉBUE
BEFORE COMMIT
CHECK (SELECT SUM(QUANTITE) FROM ABUS GROUP BY NB) < 100
FOR ABUS.
```

En supposant que la table VINS possède un attribut QUALITÉ, on pourra par exemple vérifier que chaque vin de qualité supérieure a au moins dix tuples d'ABUS le référençant. Une telle contrainte nécessitera d'insérer les ABUS d'abord et de reporter la vérification de contrainte référentielle au COMMIT, ce qui peut être fait par la clause BEFORE COMMIT.

```
CREATE ASSERTION QUALITÉSUP
AFTER INSERT ON VINS
CHECK ((SELECT COUNT(*)
        FROM ABUS A, VINS V
        WHERE A.NV = V.NV
        AND V.QUALITÉ = "SUPÉRIEURE") > 10).
```

On peut donc ainsi écrire des contraintes très complexes, difficiles à vérifier pour le système.

3. ANALYSE DES CONTRAINTES D'INTÉGRITÉ

Les contraintes d'intégrité définies par un administrateur de données sont créées en SQL. Avant d'être stockées dans la méta-base, elles doivent être analysées et contrôlées sur la base si celle-ci n'est pas vide. L'analyse doit mettre les contraintes sous une forme interne facilement exploitable, et aussi répondre aux questions suivantes :

1. Les contraintes sont-elles cohérentes entre elles ?
2. Ne sont-elles pas redondantes ?
3. Quelles contraintes doit-on vérifier suite à une insertion, une suppression, une mise à jour d'une table ?
4. Est-il possible de simplifier les contraintes pour faciliter le contrôle ?

Ces différents points sont examinés ci-dessous.

3.1. TEST DE COHÉRENCE ET DE NON-REDONDANCE

Soit I = {I1, I2...In} un ensemble de contraintes. Existe-t-il une base de données capable de satisfaire toutes ces contraintes ? Si oui, on dira que l'ensemble de contraintes est **cohérent**.

Notion VIII.3 : Cohérence de contraintes (*Constraint consistency*)

Ensemble de contraintes non contradictoires, pouvant en conséquence être satisfait par au moins une base de données.

EXEMPLE

```
CREATE ASSERTION
AFTER INSERT ON VINS CHECK DEGRÉ > 12 ;
```

et :

```
CREATE ASSERTION
AFTER INSERT ON VINS CHECK DEGRÉ < 11 ;
```

sont deux contraintes contradictoires, un vin ne pouvant être de degré à la fois supérieur à 12 et inférieur à 11. ■

Si les contraintes sont exprimables en logique du premier ordre, il est possible d'utiliser une méthode de preuve pour tester la cohérence des contraintes. De nos jours, aucun SGBD n'est capable de vérifier la cohérence d'un ensemble de contraintes (sauf peut-être les trop rares SGBD déductifs).

Étant donné un ensemble de contraintes, il est aussi possible que certaines contraintes puissent être déduites des autres, donc soient **redondantes**.

Notion VIII.4 : Contraintes redondantes (*Redundant constraints*)

Ensemble de contraintes dont l'une au moins peut être déduite des autres.

EXEMPLE

```
CREATE ASSERTION
AFTER INSERT ON VINS CHECK DEGRÉ > 12 ;
```

et :

```
CREATE ASSERTION
AFTER INSERT ON VINS CHECK DEGRÉ > 11 ;
```

sont deux contraintes redondantes, la seconde pouvant être réduite de la première. ■

Là encore, si les contraintes sont exprimables en logique du premier ordre, il est possible d'utiliser une méthode de preuve pour tester leur non-redondance. En cas de redondance, il n'est pas simple de déterminer quelle contrainte éliminer. Le problème est de trouver un ensemble minimal de contraintes à vérifier permettant de démontrer que toutes les contraintes sont satisfaites. L'ensemble retenu doit être optimal du point de vue du temps de vérification, ce qui implique l'utilisation d'une fonction de coût. De nos jours, aucun SGBD (sauf peut-être les trop rares SGBD déductifs) n'est capable de vérifier la non-redondance d'un ensemble de contraintes, et encore moins de déterminer un ensemble minimal de contraintes. Cela n'est pas très grave car les contraintes non structurelles restent peu utilisées.

3.2. SIMPLIFICATION OPÉRATIONNELLE

Certaines contraintes d'intégrité ne peuvent être violées que par certains types de mise à jour sur une relation donnée. Par exemple, l'unicité de clé dans une relation `R` ne

peut être violée par une suppression sur R. Pour éviter des contrôles inutiles, il est important d'identifier quel type de mise à jour peut violer une contrainte donnée. SQL distingue les opérations d'insertion (INSERT), de suppression (DELETE) et de mise à jour (UPDATE). Il est alors intéressant de marquer une contrainte générale I avec des étiquettes (R,U), R indiquant les relations pouvant être violées et U le type de mise à jour associé. Par exemple, l'unicité de clé K sur une relation R sera étiquetée (R,INSERT) et (R,UPDATE).

Des règles générales d'étiquetage peuvent être simplement énoncées :

1. Toute contrainte affirmant l'existence d'un tuple dans une relation R doit être étiquetée (R,DELETE).
2. Toute contrainte vraie pour tous les tuples d'une relation R doit être étiquetée (R,INSERT).
3. Toute contrainte étiquetée (R,DELETE) ou (R,INSERT) doit être étiquetée (R,MODIFY).

Soit par exemple une contrainte référentielle de R vers S. Celle-ci affirme que pour tout tuple de R il doit exister un tuple de S vérifiant R.A = S.K. Un tuple de S doit exister, d'où l'étiquette (S,DELETE). Tout tuple de R doit vérifier la contrainte, d'où l'étiquette (R,INSERT). Il faut donc ajouter les étiquettes (S,MODIFY) et (R,MODIFY). Ces petites manipulations peuvent être plus formellement définies en utilisant le calcul relationnel de tuples avec quantificateurs que nous verrons dans le contexte des bases de données déductives.

3.3. SIMPLIFICATION DIFFÉRENTIELLE

Les contraintes d'intégrité sont vérifiées après chaque transaction ayant modifiée la base. Ainsi, une transaction transforme une base de données d'état cohérent en état cohérent (voir figure VIII.5).

Lors de la vérification d'une contrainte d'intégrité, il est possible de profiter du fait que la contrainte était vérifiée en début de transaction, avant mise à jour. Dans le meilleur des cas, seules les données mises à jour doivent être vérifiées. Plus généralement, il est souvent possible de générer une forme simplifiée d'une contrainte d'intégrité qu'il suffit de vérifier sur la base après mise à jour pour garantir la satisfaction de la contrainte.

Cherchons donc à exprimer une contrainte d'intégrité par rapport aux modifications apportées à la base afin de la simplifier par prise en compte de la véracité de la contrainte avant modifications. Toute modification d'une relation R est soit une insertion, soit une suppression, soit une mise à jour pouvant être modélisée par une suppression suivie d'une insertion du même tuple modifié. Considérons une transaction t modifiant une relation R. Notons R^+ les tuples insérés et R^- les tuples supprimés

dans R. La transaction t fait passer R de l'état R à l'état Rt comme suit : Rt := (R − R⁻) ∪ R⁺. Considérons une contrainte d'intégrité I portant sur la relation R. Avant mise à jour, la contrainte est vraie sur R, ce que nous noterons R |= I (R satisfait I). Après mise à jour, il faut vérifier que Rt satisfait I, soit ((R − R⁻) ∪ R⁺)|= I. Dans certains cas de contrainte et de transaction comportant un seul type de mise à jour (Insertion, Suppression ou Mise à jour), cette dernière forme peut être simplifiée par l'introduction de **tests différentiels** [Simon87].

Notion VIII.5 : Test différentiel (*Differential test*)

Contrainte d'intégrité simplifiée à vérifier après une opération sur R *afin de garantir la satisfaction de la contrainte après application de l'opération.*

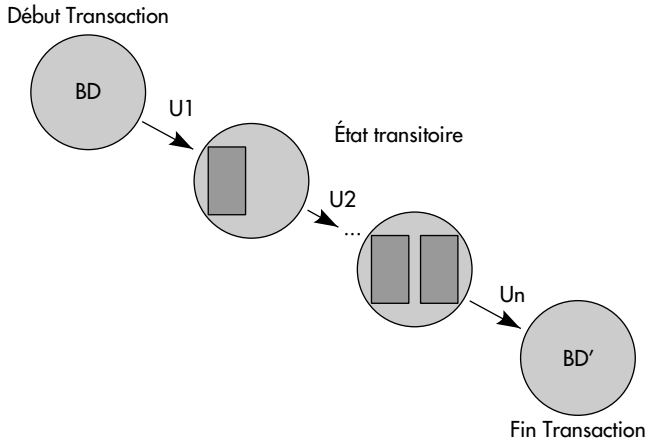


Figure VIII.5 : Modification d'une base de données par une transaction

EXEMPLE

Considérons une contrainte de domaine telle que DEGRE > 10 et une transaction d'insertion dans la table VINS. Il est clair que seuls les nouveaux vins insérés doivent être vérifiés. Dans ce cas R⁻ = ∅. On en déduit que ((R − R⁻) ∪ R⁺) |= I est équivalent à (R ∪ R⁺) |= I. La contrainte devant être vérifiée pour chaque tuple, sa vérification commute avec l'union, et il suffit donc vérifier que R⁺ |= I. Ceci est d'ailleurs vrai lors d'une insertion pour toute contrainte vérifiée par chaque tuple. ■

De manière générale, l'établissement de tests différentiels est possible pour les différentes contraintes du modèle relationnel étudiées ci-dessus dans la section 2. Le tableau de la figure VIII.6 donne quelques tests différentiels à évaluer pour les opérations d'insertion, de suppression et de différence.

Opération → **Type de contrainte ↓**	**Insertion**	**Suppression**	**Mise à jour**
Clé primaire K de R	Les clés de R^+ sont uniques et ne figurent pas dans R.	Pas de vérification	Les clés de R^+ sont uniques et ne figurent pas dans $R–R^-$.
Clé étrangère A de R Ref K de S	Les tuples de R^+ référencent un tuple de S.	R : Pas de vérification S : Les clés K de S^- ne figurent pas dans A de R.	Les tuples de R^+ référencent un tuple de S.
Domaine A de R	Domaine A sur R^+	Pas de vérification	Domaine A sur R^+
Non nullité	Non-nullité sur R^+	Pas de vérification	Non-nullité sur R^+
Dépendance fonctionnelle A→B	Pour tout tuple t de R^+, s'il existe u de R tel que t.A = u.A alors t.B = u.B	Pas de vérification	Pas de forme simplifiée
Contrainte temporelle sur attribut	Pas de vérification	Pas de vérification	Vérifier les tuples de R^+ par rapport à ceux de R-

Figure VIII.6 : Quelques tests différentiels de contraintes typiques

4. CONTRÔLE DES CONTRAINTES D'INTÉGRITÉ

Pour garder la base cohérente, le SGBD doit vérifier que les contraintes d'intégrité sont satisfaites à chaque fin de transaction. En pratique, cette opération s'accomplit à chaque mise à jour en SQL. Cette approche oblige à ordonner les mises à jour en cas de contraintes référentielles : il faut par exemple insérer dans la relation maître avant d'insérer dans la relation dépendante les tuples liés. Le problème essentiel étant les performances, on pousse d'ailleurs la vérification avant la mise à jour, ce qui peut s'effectuer par des tests appropriés, comme nous allons le voir.

4.1. MÉTHODE DE DÉTECTION

Une **méthode de détection** consiste simplement à évaluer la contrainte d'intégrité, éventuellement simplifiée, sur la base après exécution de la transaction. Pour amélio-

rer les performances, la détection recherche en général les tuples ne satisfaisant pas la contrainte.

Notion VIII.6 : Méthode de détection (*Detection method*)

Méthode consistant à retrouver les tuples ne satisfaisant pas une contrainte dans la base après une mise à jour, et à rejeter la mise à jour s'il en existe.

Ainsi, soit I une contrainte d'intégrité mettant en jeu des relations R1, R2...Rn dans une base de données. Une méthode de détection naïve lors d'une mise à jour consiste à exécuter la requête :

```
SELECT COUNT(*)
FROM R1, R2...Rn
WHERE NOT (I)
```

et à défaire la mise à jour si le résultat n'est pas 0.

En utilisant les stratégies de simplification vues ci-dessus, il est possible d'améliorer la méthode en remplaçant la requête par une requête plus élaborée mettant en jeu les relations différentielles. Plus généralement, il est possible de remplacer la requête vérifiant qu'aucun tuple ne viole la contrainte par un **post-test** pour une opération de modification donnée (`INSERT`, `DELETE` ou `UPDATE`).

Notion VIII.7 : Post-test (*Posttest*)

Test appliqué à la base après mise à jour permettant de déterminer si une contrainte d'intégrité I est violée.

Un post-test différentiel prendra en compte le fait que la contrainte était vérifiée avant la modification. Il portera sur les relations différentielles R^- et R^+. Par exemple, pour une contrainte de domaine et une insertion, un post-test possible consistera à vérifier simplement que les tuples insérés satisfont la condition :

```
(SELECT COUNT(*)
FROM R+
WHERE NOT (I)) = 0.
```

4.2. MÉTHODE DE PRÉVENTION

Une méthode efficace souvent appliquée consiste à empêcher les modifications invalides. Une telle méthode est appelée **méthode de prévention**.

Notion VIII.8 : Méthode de prévention (*Prevention Method*)

Méthode consistant à empêcher les modifications de la base qui violeraient une quelconque contrainte d'intégrité.

Pour cela, un test avant mise à jour garantissant l'intégrité de la base si elle est mise à jour est appliqué. Un tel test est appelé un **pré-test**.

Notion VIII.9 : Pré-test (*Pretest*)

Test appliqué à la base avant une mise à jour permettant de garantir la non-violation d'une contrainte d'intégrité par la mise à jour.

Un pré-test peut éventuellement modifier la mise à jour en la restreignant aux tuples conservant l'intégrité de la base. La détermination d'un pré-test peut être effectuée en transformant la contrainte d'intégrité en lui appliquant l'inverse de la mise à jour. Plus précisément, soit u une mise à jour sur une relation R et I(R) une contrainte d'intégrité portant sur R. La contrainte d'intégrité modifiée I(u(R)) est l'image de I obtenue en remplaçant R par l'effet de u sur R. Par exemple, soient la contrainte d'intégrité (Pour tout `VINS : DEGRE < 15`) et la mise à jour u :

```
UPDATE VINS
SET DEGRE = DEGRE + 1.
```

La contrainte d'intégrité modifiée est (Pour tout `VINS : DEGRE+1 < 15`), puisque l'effet de la mise à jour est de remplacer `DEGRE` par `DEGRE+1`.

À partir d'une contrainte d'intégrité modifiée `I(u(R))`, il est possible de générer un pré-test en vérifiant simplement que la requête suivante a une réponse égale à 0 :

```
SELECT COUNT(*)
FROM R
WHERE NOT (I(U(R)).
```

Dans le cas des vins de degré inférieur à 15, on obtient :

```
SELECT COUNT(*)
FROM VINS
WHERE (DEGRE + 1) ≥ 15
```

Ceci permet donc la mise à jour si aucun vin n'a un degré supérieur ou égal à 14. En effet, dans ce cas aucun vin ne pourra avoir un degré supérieur à 15 après mise à jour.

Une méthode de prévention plus connue est la modification de requêtes [Stonebraker75] appliquée dans INGRES. Elle améliore la méthode précédente en intégrant le pré-test à la requête de mise à jour, ce qui permet de restreindre cette mise à jour aux seuls tuples respectant la contrainte d'intégrité après mise à jour. Bien que définie en QUEL, la méthode est transposable en SQL. Soit la mise à jour générique :

```
UPDATE R
SET R = U(R)
WHERE Q.
```

Soit donc I(R) une contrainte d'intégrité sur R et I(u(R)) la contrainte modifiée par la mise à jour inverse, comme vu ci-dessus. La mise à jour est alors transformée en :

```
UPDATE R
SET R = U(R)
WHERE Q AND I(U(R)).
```

EXEMPLE

Dans le cas des vins de degré inférieur à 15, on exécutera la mise à jour modifiée :

```
UPDATE VINS
SET DEGRE = DEGRE+1
WHERE DEGRE < 14,
```

ce qui fait que seuls les vins de degré inférieur à 14 seront incrémentés. La contrainte ne sera pas violée, mais la sémantique de la mise à jour sera quelque peu changée. ■

L'optimisation des pré-tests peut être plus poussée et prendre en compte la sémantique des mises à jour. Par exemple, si la mise à jour décroît le degré d'un vin, il n'est pas nécessaire d'ajouter un pré-test pour vérifier que le degré ne dépassera pas 15 ! Plus généralement, si la mise à jour est intégrée à un programme, par exemple en C/SQL, il est possible de bénéficier de la sémantique du programme pour élaborer un pré-test. Une technique pour élaborer des pré-tests en PASCAL/SQL a été proposée dans [Gardarin79]. L'idée est d'utiliser les axiomes de Hoare définissant la sémantique de PASCAL pour pousser les contraintes d'intégrité écrites en logique du premier ordre depuis la fin d'une transaction vers le début, en laissant ainsi au début de chaque mise à jour les pré-tests nécessaires.

Dans le cas de contrainte avec agrégats, les pré-tests constituent une des rares méthodes efficaces de contrôle. L'idée simple développée dans [Bernstein80] est de gérer dans la méta-base des agrégats redondants. Par exemple, si la moyenne des salaires dans une relation `EMPLOYES` doit rester inférieure à 20 000 F, on gérera cette moyenne (notée `MOYENNE`) ainsi que le nombre d'employés (noté `COMPTE`) dans la méta-base. Un pré-test simple lors de l'insertion d'un nouvel employé consistera alors à vérifier que `(MOYENNE*COMPTE+NOUVEAU SALAIRE)/(COMPTE+1) < 2000`. De même, pour une contrainte spécifiant que toute valeur d'une colonne A doit rester inférieure à toute valeur d'une colonne B, on pourra garder le minimum de B dans la méta-base. Lors d'une mise à jour de A, un pré-test efficace consistera simplement à vérifier que la nouvelle valeur de A reste inférieure au minimum de B.

Bien que les mises à jour ne soient pas effectuées lors de l'évaluation des pré-tests, une méthode intermédiaire a été appliquée dans le système SABRE à l'INRIA [Simon87], basée sur des pré-tests différentiels. Elle comporte deux étapes :

1. Préparer la mise à jour en construisant les relations R^+ et R^-, comme vu ci-dessus.
2. Pour chaque contrainte menacée, appliquer un pré-test différentiel pour contrôler la validité de la mise à jour.

C'est seulement à la fin de la transaction que R^- et R^+ sont appliquées à la relation R.

Pour chaque contrainte d'intégrité et chaque type de mise à jour, un pré-test différentiel était élaboré. Pour les cas usuels, ces pré-tests correspondent à peu près à ceux du tableau de la figure VIII.6. Pour les cas plus complexes, une méthode systématique de différentiation d'expression logique était appliquée [Simon87].

4.3. CONTRÔLES AU VOL DES CONTRAINTES SIMPLES

La plupart des systèmes implémentent une version des contraintes possibles en SQL2 réduite à l'unicité de clé, aux contraintes référentielles, et aux contraintes de domaines de type liste ou plage de valeurs, ou comparaison avec une autre colonne de la même table (`CHECK <condition>`, où la condition peut être toute condition SQL sans variable). Ces contraintes sont relativement simples à vérifier, bien que les actions possibles en cas de violation d'une contrainte référentielle (`SET NULL`, `SET DEFAULT`, ou `CASCADE`) impliquent une complexité que nous verrons ci-dessous. Elles peuvent pour la plupart être vérifiées au vol, c'est-à-dire lors de la mise à jour du tuple, ou plutôt juste avant.

La méthode de contrôle généralement employée consiste à effectuer une prévention au vol en employant un pré-test à chaque modification de tuple. Cette méthode est efficace car elle réduit au maximum les entrée-sorties nécessaires. Nous examinons ci-dessous les pré-tests mis en œuvre dans chacun des cas.

4.3.1. Unicité de clé

Tout SGBD gère en général un index sur les clés primaires. Un pré-test simple consiste à vérifier que la nouvelle clé ne figure pas dans l'index. Ce pré-test est effectué lors de l'insertion d'un tuple ou de la modification d'une clé dans la table, en général d'ailleurs juste avant mise à jour de l'index.

4.3.2. Contrainte référentielle

Du fait que deux tables, la table maître et la table dépendante, sont mises en jeu par une contrainte référentielle, quatre types de modifications nécessitent des vérifications, comme vu ci-dessus :

1. **Insertion dans la table dépendante**. La colonne référencée dans la table maître étant une clé primaire, il existe un index. Un pré-test simple consiste donc à vérifier l'existence dans cet index de la valeur de la colonne référençante à insérer.
2. **Mise à jour de la colonne référençante dans la table dépendante**. Le pré-test est identique au précédent pour la nouvelle valeur de la colonne.

3. **Suppression dans la table maître**. Le pré-test consiste à vérifier qu'il n'existe pas de tuple contenant la valeur de clé à supprimer dans la colonne référençante. Si le pré-test est faux, une complexité importante surgit du fait de la multitude des actions prévues dans ce cas en SQL2. Il faut en effet soit rejeter la mise à jour, soit modifier voire supprimer les tuples de la table dépendante correspondant à cette valeur de clé. Ceci peut nécessiter d'autres contrôles d'intégrité, source de complexité examinée plus loin.
4. **Modification de clé dans la table maître**. Le pré-test est identique au précédent pour la valeur de clé avant modification.

Finalement, sauf pour les suppressions de clé dans la table maître, les vérifications sont simples. Elles peuvent devenir très complexes en cas de suppression en cascade le long de plusieurs contraintes référentielles.

4.3.3. Contrainte de domaine

Pour les contraintes de type `CHECK <condition>` avec une condition simple, il suffit de vérifier avant d'appliquer la mise à jour que la valeur qui serait donnée à l'attribut après mise à jour vérifie la condition. Ce pré-test est simple et peut être effectué au vol.

4.4. INTERACTION ENTRE CONTRAINTES

Un problème difficile est celui posé par les interactions possibles entre contraintes. Il serait souhaitable que quel que soit l'ordre d'évaluation des contraintes ou de mise à jour de colonnes, une mise à jour donne le même effet. Ceci n'est pas simple à réaliser, notamment en cas de cascade de contraintes référentielles. Les interactions avec les mises à jour au travers des vues avec option de contrôle (`WITH CHECK OPTION`) sont aussi une source d'interaction [Cochrane96]. Afin de garantir le déterminisme des mises à jour, une sémantique de type point fixe doit être appliquée. Elle consiste à appliquer une procédure de contrôle comportant les étapes suivantes :

1. Évaluer tous les pré-tests des modifications directement issues de l'ordre original.
2. Si l'un au moins n'est pas vérifié, accomplir toutes les modifications à cascader (`SET NULL`, `SET DEFAULT` ou `DELETE`) et déclencher récursivement les contrôles induits par ces modifications.
3. Évaluer à nouveau tous les pré-tests des modifications directement issues de l'ordre original. Si tous sont vérifiés, exécuter la mise à jour, sinon rejeter la mise à jour et défaire toutes les modifications faites à tous les niveaux.

Cette procédure récursive permet de se prémunir contre les interactions entre contraintes référentielles, mais aussi avec les contraintes de non-nullité, de domaine,

etc. Elle est certes un peu complexe, ce qui démontre finalement la difficulté de traiter efficacement les contraintes exprimables en SQL2.

5. SGBD ACTIFS ET DÉCLENCHEURS

Dans cette section, nous précisons ce qu'est un **SGBD actif**, puis ce qu'est un **déclencheur**. Enfin nous étudions une architecture type pour un SGBD actif.

5.1. OBJECTIFS

La notion de **SGBD actif** s'oppose à celle de SGBD passif, qui subit sans réagir des opérations de modification et interrogation de données.

Notion VIII.10 : SGBD actif (*Active DBMS*)

SGBD capable de réagir à des événements afin de contrôler l'intégrité, gérer des redondances, autoriser ou interdire des accès, alerter des utilisateurs, et plus généralement gérer le comportement réactif des applications.

La notion de **base de données active** va permettre de déplacer une partie de la sémantique des applications au sein du SGBD. Dans sa forme la plus simple, une BD active répercute les effets de mises à jour sur certaines tables vers d'autres tables. Un SGBD actif peut donc réagir par exemple lors d'opérations illicites, mais aussi lors d'opérations licites, parfois à un instant donné, et cela sous certaines conditions. Comment réagit-il ? Il pourra déclencher une opération subséquente à un événement donné (par exemple, une mise à jour), interdire une opération en annulant la transaction qui l'a demandée, ou encore envoyer un message à l'application, voire sur Internet.

5.2. TYPES DE RÈGLES

Les SGBD actifs ont intégré de manière procédurale les règles de production de l'intelligence artificielle. Une règle de production est une construction de la forme :

```
IF <Condition sur BD> THEN <Action sur BD>.
```

Une règle de production permet d'agir sur une base de données lorsque la condition de la règle est satisfaite : l'action est alors exécutée et change en général l'état de la base. Le système de contrôle choisit quelle règle appliquer, jusqu'à saturation d'une condition de terminaison, ou jusqu'au point de saturation où plus aucune règle ne peut modifier l'état de la base : on a alors atteint un point fixe.

Cette idée, qui est aussi à la source des bases de données déductives comme nous le verrons plus tard, a été reprise dans les SGBD relationnels en ajoutant aux règles un contrôle procédural : chaque règle sera appliquée suite à un **événement**. Les règles deviennent alors des **déclencheurs** ou **règles ECA** (Événement – Condition – Action).

Notion VIII.11 : Déclencheur (*Trigger*)

Règle dont l'évaluation de type procédural est déclenchée par un événement, généralement exprimée sous la forme d'un triplet Événement – Condition – Action : `WHEN` *<Événement>* `IF` *<Condition sur BD>* `THEN` *<Action sur BD>.*

Lorsque l'événement se produit, la condition est évaluée sur la base. Si elle est vraie, l'action est effectuée. Nous préciserons ultérieurement ce qu'est un événement, une condition et une action. Disons pour l'instant que l'événement est souvent une modification de la base, la condition un prédicat vrai ou faux, et l'action un programme de mise à jour. La condition est optionnelle ; sans condition, on obtient un déclencheur dégénéré événement-action (EA). Il est a remarquer que, dans ce dernier cas, la condition peut toujours être testée dans le programme constituant l'action, mais celui-ci est alors déclenché plus souvent et inutilement.

Le modèle d'exécution des déclencheurs est très variable dans les SGBD, mais il a été proposé une définition standard pour SQL [Cochrane96]. Malheureusement, les *triggers* apparaissent seulement au niveau de SQL3. Dans la suite, nous nous appuierons sur ce modèle, mais il faut savoir que les systèmes ne le respectent en général guère. Pour comprendre la sémantique des déclencheurs dans un SGBD actif, il faut distinguer la prise en compte des événements, la détermination des règles candidates à l'exécution, le choix d'une règle si plusieurs sont candidates, l'exécution d'une règle qui comporte l'évaluation de la condition puis l'exécution de l'action.

La façon dont les règles sont exécutées dans les SGBD actifs n'est pas standard. La sémantique d'une BD active est donc souvent difficile à percevoir. Pour cela, un SGBD actif se doit de répondre aux questions suivantes :

1. Quand prendre en compte un événement ? Ce peut être dès son apparition, ou seulement lorsqu'une règle n'est pas en cours d'exécution ; dans ce dernier cas, il ne sera pas possible d'interrompre l'exécution d'une règle pour en exécuter une plus prioritaire.
2. Quand exécuter les règles ? Ce peut être dès l'apparition de l'événement, ou plus tard lors d'un retour au système par exemple.
3. Comment choisir une règle lorsque plusieurs sont candidates à l'exécution ? Des mécanismes de priorité simples (par exemple, un niveau de priorité de 1 à 10) peuvent être mis en œuvre.
4. Quelle est la force du lien condition-action ? Autrement dit, doit-on exécuter l'action dès que la condition a été évaluée ou peut-on attendre ?

5. Lorsqu'une règle est exécutée et produit des événements provoquant le déclenchement d'autres règles, doit-on se dérouter ou attendre la fin de l'exécution de toutes les règles actives ou seulement de celle en cours d'exécution ?

Toutes ces questions ne sont pas indépendantes. Les réponses apportées fixent la sémantique du SGBD actif et conduisent à des résultats différents. Dans la suite, nous utilisons la sémantique décrite dans [Cochrane96], proposée pour SQL3. Par exemple, ce modèle procède en profondeur d'abord : il interrompt la règle en cours d'exécution à chaque nouvel événement.

5.3. COMPOSANTS D'UN SGBD ACTIF

Les SGBD actifs ne sont donc pas standard. Cependant, la figure VIII.7 présente une architecture typique pour un SGBD actif. Les composants essentiels sont les suivants :

1. L'**analyseur de règles** permet de saisir les règles en forme externe (souvent textuelle), de les analyser et de les ranger en format interne dans le dictionnaire de règles.
2. Le **moteur de règles** coordonne l'exécution des règles suite aux événements. Il reçoit les événements primitifs et composites et détermine les règles candidates à l'exécution. Une règle devient alors active. Parmi les règles actives, le moteur de règles choisi la plus prioritaire et lance sont exécution, qui comporte donc l'évaluation de la condition, puis l'exécution de l'action si la condition est satisfaite. Le moteur de règles gère aussi le contexte des règles actives, c'est-à-dire mémorise les variables qui doivent être maintenues et passées par exemple de la condition à l'action.
3. Le **moniteur d'événements** détecte les événements primitifs et composites. Il demande au moteur de règles l'exécution des règles déclenchées par ces événements.
4. Sur demande du moteur de règles, l'**évaluateur de conditions** évalue les conditions des règles actives, éventuellement en demandant l'exécution des recherches au SGBD. Il détermine si la condition est vraie et retourne cette information au moteur de règles.
5. L'**exécuteur d'actions** exécute les actions des règles actives dont les conditions ont été préalablement vérifiées. Il invoque pour ce faire le SGBD.
6. Le **dictionnaire de règles** est la partie de la méta-base du SGBD qui contient la définition des règles en format interne.

Deux approches sont possibles pour un SGBD actif [Llirbat97] : l'approche intégrée lie intimement les composants spécifiques du SGBD actif au SGBD, alors que l'approche sur-couche construit au-dessus d'un SGBD les composants nécessaires. L'approche intégrée est souvent plus efficace, mais elle ne permet pas de réaliser des

systèmes de déclencheurs portables, indépendants du SGBD natif (il en existe bien peu).

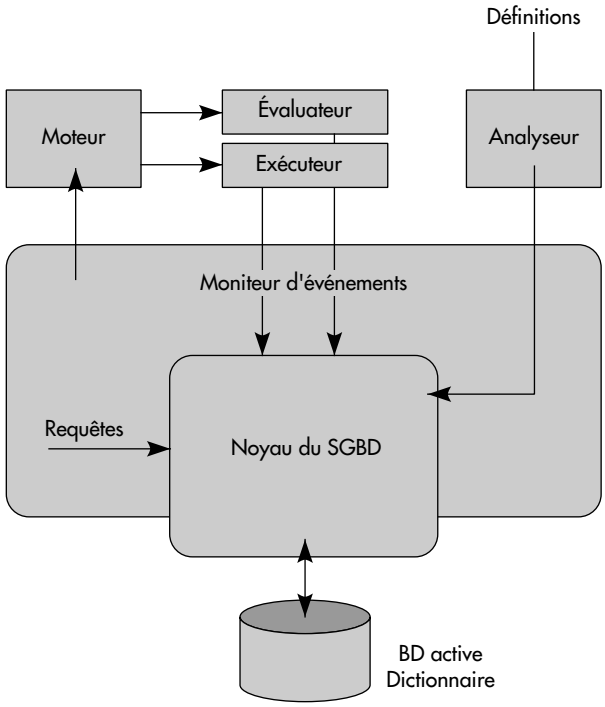


Figure VIII.7 : Architecture d'un SGBD actif

6. LA DÉFINITION DES DÉCLENCHEURS

Dans cette section, nous définissons plus précisément les éléments constituant un déclencheur. Puis, nous donnons la syntaxe permettant de définir les déclencheurs en SQL3.

6.1. LES ÉVÉNEMENTS

Notion VIII.12 : Événement (*Event*)

Signal instantané apparaissant à un point spécifique dans le temps, externe ou interne, détecté par le SGBD.

Un **événement** correspond donc à l'apparition d'un point d'intérêt dans le temps. La spécification d'un événement nécessite la définition d'un type. Un type d'événement peut correspondre au début (`BEFORE`) ou à la fin (`AFTER`) d'une recherche (`SELECT`), d'une mise à jour (`UPDATE, DELETE, INSERT`) ou d'une transaction (`BEGIN, COMMIT, ABORT`), à l'écoulement d'un délai, au passage d'une horodate, etc. Un type permet de définir un ensemble potentiellement infini d'instances d'événements, simplement appelé événement. Une instance d'événement se produit donc à un instant donné. Á un événement sont souvent associés des paramètres : une valeur d'une variable, un objet, une colonne de table, etc. Le **contexte d'un événement** est une structure de données nommée contenant les données nécessaires à l'évaluation des règles déclenchées par cet événement, en particulier les paramètres de l'événement. Parmi les événements, on distingue les événements simples (ou primitifs) et les événements composés.

6.1.1. Événement simple

Dans un SGBD, les événements simples (encore appelés primitifs) particulièrement considérés sont l'appel d'une modification de données (`BEFORE UPDATE`, `BEFORE INSERT`, `BEFORE DELETE`), la terminaison d'une modification de données (`AFTER UPDATE`, `AFTER INSERT`, `AFTER DELETE`), l'appel d'une recherche de données (`BEFORE SELECT`), la fin d'une recherche de données (`AFTER SELECT`), le début, la validation, l'annulation d'une transaction (`BEGIN`, `COMMIT`, `ABORT`), un événement temporel absolu (`AT TIMES` <Heure>) ou relatif (`IN TIMES` <Delta>), ou tout simplement un événement utilisateur (avant ou après exécution de procédure : `BEFORE` <procedure> ou `AFTER` <procedure>). Une typologie simplifiée des différents événements primitifs apparaît figure VIII.8. D'autres sont détectables, par exemple les erreurs.

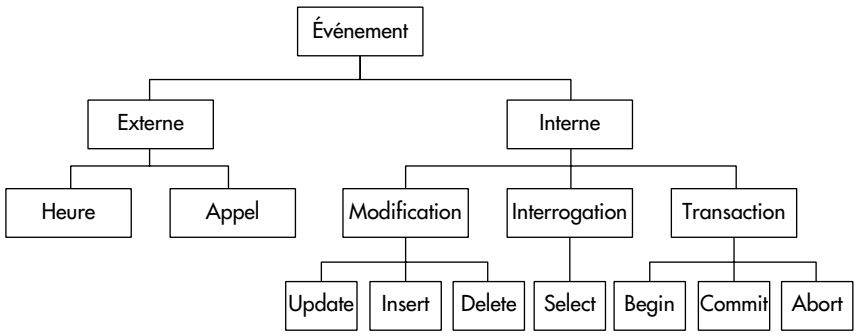


Figure VIII.8 : Typologie des événements primitifs

6.1.2. Événement composé

Un événement composé est une composition d'événements simples. Celle-ci est effectuée par des constructeurs spécifiques, tels le `OU` et le `ET` logique. On obtient ainsi des expressions logiques simples et parenthésables d'événements. (`A OU B`) se produit lorsque l'un ou l'autre des événements est vrai. Alors que les événements primitifs ne sont pas mémorisés après leur production, le problème devient plus difficile pour les événements composés. Par exemple, si A a été signalé, (`A OU B`) aussi. Que doit-on faire si B se produit ? Faut-il signaler à nouveau (`A OU B`) ? Le problème n'est pas simple et la solution retenue peut changer complètement la sémantique d'une application.

Au-delà des constructeurs logiques `ET` et `OU`, il est possible de faire intervenir des constructeurs temporels tels que la séquence (`A PUIS B`) qui est signalée si `B` suit `A`, la répétition (`N FOIS A`) déclenchée si A se produit N fois, ou la production ou la non production pendant un intervalle de temps (`A IN` <intervalle>, `A NOT IN` <Intervalle>). Tout ceci complique la gestion des événements, mais facilite la prise en compte de la logique temporelle des applications. On pourra par exemple écrire des événements du style `((1H AFTER UPDATE) OR (AFTER INSERT THEN BEFORE DELETE) AND (5 TIMES BEFORE INSERT) AND (COMMIT NOT IN [10H, 12H]))`, dont la signification est un peu complexe. Heureusement, peu de SGBD excepté quelques prototypes de recherche utilisent des événements composés [Gatziu92].

6.2. LA CONDITION

La **condition** est une expression logique de prédicats portant sur les variables de contexte d'exécution de la règle ou/et sur la base de données. Peu de systèmes permettent des requêtes complètes au niveau de la condition.

Notion VIII.13 : Condition de déclencheur (*Trigger condition*)

Qualification (aussi appelée condition de recherche, ou search condition) portant sur les variables de contexte et/ou la base et donnant en résultat une valeur booléenne.

Si l'on souhaite tester l'existence de tuples, on utilisera les prédicats `EXIST` et `NOT EXIST`, ou le compte de tuples résultats (`COUNT`). La condition est en général optionnelle au niveau des déclencheurs : sans condition, on a simplement une règle événement-action.

6.3. L'ACTION

L'**action** est une procédure exécutée lorsque la condition est vérifiée.

Notion VIII.14 : Action de déclencheur (*Trigger action*)

Procédure exécutée lorsque la règle est déclenchée.

Ce peut être une seule requête ou une séquence de requêtes SQL, une procédure stockée agissant sur la base écrite en L4G ou dans un L3G intégrant SQL (C/SQL par exemple), ou enfin une opération sur transaction (ABORT, COMMIT). L'action peut utiliser les paramètres du contexte. Elle peut être exécutée une seule fois suite à l'événement ou pour chaque ligne satisfaisant la condition, comme nous le verrons plus en détail ci-dessous.

6.4. EXPRESSION EN SQL3

Avec SQL3, tous les objets (règles, contraintes, etc.) sont nommés, donc en particulier les déclencheurs. Les événements sont simples et déclenchés par les requêtes de modification INSERT, UPDATE, et DELETE. L'événement peut être déclenché soit avant exécution de la modification (BEFORE), soit après (AFTER). Deux granularités sont possibles afin de contrôler l'évaluation de la condition et l'exécution éventuelle de l'action : les granularités ligne ou requête. Dans le premier cas, l'action est exécutée pour chaque ligne modifiée satisfaisant la condition, dans le second elle est exécutée une seule fois pour l'événement. L'action peut référencer les valeurs avant et après mise à jour (clause REFERENCING... AS...). Elle peut même remplacer complètement la modification dont l'appel a provoqué l'événement (clause INSTEAD OF).

6.4.1. Syntaxe

La figure VIII.9 donne la syntaxe résumée de la requête de création de déclencheur en SQL3. La sémantique est explicitée dans le paragraphe qui suit.

```
CREATE TRIGGER <nom>
// événement (avec paramètres)
{BEFORE | AFTER | INSTEAD OF}
{INSERT | DELETE | UPDATE [OF <liste de colonnes>]}
ON <table> [ORDER <valeur>]
[REFERENCING{NEW|OLD|NEW_TABLE|OLD_TABLE} AS <nom>]...
// condition
(WHEN (<condition de recherche SQL>)
// action
<Procédure SQL3>
// granularité
[FOR EACH {ROW | STATEMENT}])
```

Figure VIII.9 : Syntaxe de la commande de création de déclencheur

6.4.2. Sémantique

L'événement de déclenchement est une mise à jour (`UPDATE`), une insertion (`INSERT`) ou une suppression (`DELETE`) dans une table. En cas de mise à jour, un paramètre optionnel permet de préciser une liste de colonnes afin de réduire la portée de l'événement. L'événement n'est pas instantané et a un effet. Afin de préciser l'instant d'activation de la règle, trois options sont disponibles : avant (`BEFORE`), après (`AFTER`) la modification, ou à la place de (`INSTEAD OF`). On revient donc là à des événements instantanés. L'option « avant » est très utile pour contrôler les paramètres et les données d'entrées avant une modification. L'option « après » permet plutôt de déclencher des traitements applicatifs consécutifs à une mise à jour. L'option « à la place de » permet de remplacer un ordre par un autre, par exemple pour des raisons de sécurité.

Un ordre optionnel est spécifié pour préciser les priorités lorsque plusieurs déclencheurs sont définis sur une même table. La déclaration `REFERENCING NEW AS` <nom> définit une variable de nom <nom> contenant la valeur de la dernière ligne modifiée dans la base par l'événement. De même, `REFERENCING OLD AS` <nom> définit une variable de nom <nom> contenant la valeur de la même ligne avant l'événement. Des tables différentielles contenant les valeurs avant et après l'événement sont définies par les options `NEW_TABLE` et `OLD_TABLE`. Les déclencheurs sur `INSERT` ne peuvent que déclarer les nouvelles valeurs. Les déclencheurs sur `DELETE` ne peuvent que déclarer les anciennes.

Chaque déclencheur spécifie en option une action gardée par une condition. La condition est un critère de recherche SQL pouvant porter sur les variables de contexte ou sur la base via des requêtes imbriquées. L'action est une procédure contenant une séquence d'ordre SQL. Condition et action peuvent donc interroger la base de données et le contexte du déclencheur (par exemple, manipuler les variables et les tables différentielles). Condition et action lisent ou modifient les valeurs de la base avant mise à jour dans un déclencheur avant (`BEFORE`), après mise à jour dans un déclencheur après (`AFTER`).

La granularité d'un déclencheur est soit ligne (`FOR EACH ROW`), soit requête (`FOR EACH STATEMENT`). Dans le premier cas, il est exécuté pour chaque ligne modifiée (0 fois si aucune ligne n'est modifiée), alors que dans le second, il l'est une seule fois pour la requête.

6.5. QUELQUES EXEMPLES

6.5.1. Contrôle d'intégrité

Cet exemple porte sur la table des `VINS`. Le premier déclencheur de la figure VIII.10 contrôle l'existence d'un vin lors de l'ajout d'un abus. Le second cascade la suppression d'un vin, c'est-à-dire supprime les abus correspondant au vin supprimé.

```
// Ajout d'un abus
CREATE TRIGGER InsertAbus
BEFORE INSERT ON ABUS
REFERENCING NEW AS N
(WHEN (NOT EXIST (SELECT * FROM Vins WHERE NV=N.NV))
ABORT TRANSACTION
FOR EACH ROW) ;

// Suppression d'un vin
CREATE TRIGGER DeleteVins
BEFORE DELETE ON VINS
REFERENCING OLD AS O
(DELETE FROM ABUS WHERE NV = O.NV
FOR EACH ROW) ;
```

Figure VIII.10 : Contrôle d'intégrité référentielle

Le déclencheur de la figure VIII.11 est associé à la table des employés, de schéma :

`EMPLOYE (ID Int, Nom Varchar, Salaire Float).`

Il effectue un contrôle d'intégrité temporelle : le salaire ne peut que croître.

```
CREATE TRIGGER SalaireCroissant
BEFORE UPDATE OF Salaire ON EMPLOYE
REFERENCING OLD AS O, NEW AS N
(WHEN O.Salaire > N.Salaire
SIGNAL.SQLState '7005' (''Les salaires ne peuvent décroître')
FOR EACH ROW);
```

Figure VIII.11 : Contrôle d'intégrité temporelle

6.5.2. Mise à jour automatique de colonnes

Le déclencheur de la figure VIII.12 est déclenché lors de la mise à jour de la table

`PRODUITS (NP Int, NF Int, Coût Real, Auteur String, DateMaj Date).`

Il positionne les attributs Auteur et DateMaj aux valeurs courantes de la transaction pour lequel il est exécuté.

```
CREATE TRIGGER SetAuteurDate
BEFORE UPDATE ON PRODUITS
REFERENCING NEW_TABLE AS N
(UPDATE N
   SET N.Auteur = USER, N.DateMaj = CURRENT DATE);
```

Figure VIII.12 : Complément de mise à jour

Le déclencheur de la figure VIII.13 est aussi associé à la table PRODUITS. Il crée automatiquement la clé lors de l'insertion d'un tuple. Attention : si vous insérez plusieurs tuples ensemble, vous obtenez plusieurs fois la même clé ! Il faut alors écrire le programme en L4G et travailler sur chaque ligne.

```
CREATE TRIGGER SetCléVins
BEFORE INSERT ON PRODUITS
REFERENCING NEW_TABLE AS N
(UPDATE N
   SET N.NP = SELECT COUNT(*) +1 FROM PRODUITS);
```

Figure VIII.13 : Génération automatique de clé

6.5.3. Gestion de données agrégatives

Le déclencheur de la figure VIII.14 est associé à la table EMPLOYE vue ci-dessus permettant de gérer le salaire des employés. Il répercute les mises à jour du salaire sur la table de cumul des augmentations de schéma : CUMUL (ID int, Augmentation float).

```
CREATE TRIGGER CumulSal
AFTER UPDATE OF salaire ON EMPLOYE
REFERENCING OLD AS a, NEW AS n
(UPDATE CUMUL SET Augmentation = Augmentation +
   n.salaire – a.salaire WHERE ID = a.ID
FOR EACH ROW) ;
```

Figure VIII.14 : Gestion de données agrégative

7. EXÉCUTION DES DÉCLENCHEURS

Nous examinons ici brièvement les mécanismes nécessaires au sein d'un moteur de règles afin de cordonner l'exécution des règles, en plus bien sûr de la gestion du contexte des événements. Ces mécanismes sont difficiles du fait de la sémantique plutôt complexe des déclencheurs, basée sur une application jusqu'à saturation (point fixe) de règles de mise à jour. Nous utilisons ici la sémantique de référence de SQL3 [Cochrane96].

7.1. PROCÉDURE GÉNÉRALE

La procédure générale d'exécution d'une modification (UPDATE, INSERT ou DELETE) doit prendre en compte les déclencheurs et la vérification des contraintes d'intégrité. Les deux peuvent interagir. La procédure doit distinguer les actions à effectuer avant l'exécution de l'opération de modification ayant déclenché la règle de celle exécutée après. Il faut exécuter avant les déclencheurs avec option BEFORE et après ceux avec option AFTER. Les contrôles d'intégrité, généralement accomplis par des pré-tests, seront aussi effectués avant. Dans tous les cas, la modification doit être préparée afin de rendre accessibles les valeurs avant et après mise à jour. La figure VIII.15 résume l'enchaînement des contrôles lors de l'exécution d'une mise à jour. La figure VIII.16 montre comment est exécutée une règle, selon l'option. Cette procédure est au cœur du moteur de règles.

```
// Exécution d'une modification d'une relation R
Modification(R) {
Préparer les mises à jour de R dans R+ et R- ;
// Exécuter les déclencheur avant mise à jour (BEFORE)
For each "déclencheur BEFORE d" do {Exécuter(d.Règle)} ;

// Effectuer les contrôles d'intégrité, puis la mise à jour
If (Not Integre) then ABORT TRANSACTION ;
// effectuer la mise à jour
R = (R - R-) ∪ R+ ;

// Exécuter les déclencheurs après mise à jour (AFTER)
For each "déclencheur AFTER" do {Exécuter(d.Règle)} ;
} ;
```

Figure VIII.15 : Exécution d'une modification

```
// Exécution d'une règle WHEN Condition Action FOR EACH <Option>
Exécuter(Condition, Action, Option) {
// Appliquer à chaque tuple si option ligne
if Option = "FOR EACH ROW" then
   {For each t de ΔR = R+ ∪ R- do {
      if (Condition(t) = True) then Exécuter (Action(t)) ;}
if Option = "FOR EACH STATEMENT" then {
      if (Condition(ΔR) = True) then Exécuter (Action(ΔR) ;}
} ;
```

Figure VIII.16 : Exécution d'une règle

7.2. PRIORITÉS ET IMBRICATIONS

En fait, suite à un événement, plusieurs règles peuvent être candidates à l'exécution. Dans l'algorithme de la figure VIII.15, nous avons choisi d'exécuter les déclencheurs avant puis après dans un ordre quelconque (boucle `FOR EACH`). Malheureusement, l'ordre peut influer sur le résultat. Par exemple, un déclencheur peut défaire une mise à jour exécutée par un autre. Ceci ne se produirait pas si on exécutait les déclencheurs dans un ordre différent ! Il y a donc là un problème de sémantique.

Avec SQL3, il est prévu une priorité affectée soit par l'utilisateur (clause `ORDER`), soit selon l'ordre de définition des déclencheurs. Cette priorité doit donc être respectée : les boucles For each de la figure VIII.15 doivent choisir les déclencheurs en respectant les priorités.

Plus complexe, l'exécution d'une règle peut entraîner des mises à jour, qui peuvent à leur tour impliquer l'exécution de déclencheurs sur la même relation ou sur une autre. Dans ce cas, les contextes d'exécution des modifications sont empilés, comme lors de l'appel de sous-programmes. À chaque fin d'exécution d'une modification, un dépilage est nécessaire afin de revenir à la modification précédente. Deux problèmes se posent cependant, que nous examinons successivement.

Le premier problème tient aux mises à jour cumulatives. Si la modification porte sur la même relation R, de nouveaux tuples sont insérés dans R^+ et R^- par la modification imbriquée. On peut soit empiler les versions de R^+ et R^-, soit ne s'intéresser qu'à l'**effet net** de l'ensemble des mises à jour. La sémantique des déclencheurs sera alors différente.

Notion VIII.15 : Effet net (*Net effect*)

Impact cumulé d'un ensemble de mise à jour en termes de relations différentielles R^+ (tuples à insérer) et R^- (tuples à supprimer).

L'effet net est une notion importante aussi pour les transactions qui sont composées de plusieurs mises à jour. Par exemple, une transaction qui insère puis supprime un tuple a un effet net vide.

Le second problème concerne les risques de bouclage. En effet, il peut exister des boucles de déclencheurs qui ne s'arrêtent pas. En effet, une mise à jour peut impliquer une nouvelle mise à jour, et ainsi de suite. Le nombre de relations étant fini, la boucle reviendra forcément plusieurs fois sur une même relation. C'est un critère simple détectable par l'analyseur de déclencheurs. Cependant, ceci limite fortement l'usage des déclencheurs. Par exemple, on ne peut écrire un déclencheur qui en cas de baisse de votre salaire déclenche une autre règle pour vous donner une prime compensatrice ! Une solution plus sophistiquée consiste à détecter les boucles à l'exécution en limitant le nombre de déclencheurs activés lors d'une mise à jour. Une approche plus sophistiquée encore consiste à prendre en compte l'effet net d'un ensemble de déclencheurs successifs et à vérifier qu'il change de manière continue.

7.3. COUPLAGE À LA GESTION DE TRANSACTIONS

Jusque-là, nous avons supposé que les déclencheurs étaient exécutés lors de chaque modification, pour le compte de la transaction qui effectue la modification. En fait, différents types de liaisons avec les transactions sont possibles, notamment pour les déclencheurs après, pour les conditions et/ou les actions. On distingue :

1. Le **mode d'exécution** qui précise quand exécuter le déclencheur. Ce peut être immédiat, c'est-à-dire dès que l'événement se produit (`IMMEDIAT`), ou différé en fin de transaction (`DEFERRED`).
2. Le **mode de couplage** qui précise si le déclencheur doit être exécuté dans la même transaction (`COUPLED`) ou dans une transaction indépendante (`DECOUPLED`).
3. Le **mode de synchronisation** qui dans le cas d'une transaction indépendante précise si elle est synchrone (après) ou asynchrone (en parallèle) (`SYNCHRONOUS`) avec la transaction initiatrice.

Tout ceci complique la sémantique des déclencheurs. Dans le cas du mode différé, seul l'effet net de la transaction sera pris en compte par les déclencheurs. Le cas de transaction découplée asynchrone devient complexe puisque les deux transactions peuvent interagir via les mises à jour. L'une, pourquoi pas la transaction initiatrice, peut être reprise alors que l'autre ne l'est pas. Ainsi, le déclencheur peut être exécuté sans que la transaction déclenchante ne le soit ! Les SGBD actuels évitent ce mode de synchronisation et d'ailleurs plus radicalement le mode découplé.

8. CONCLUSION

Un système de règles supporte au minimum des événements simples de mise à jour. C'est le cas de SQL3 et des SGBD relationnels du commerce. Il est aussi possible de considérer les événements de recherche (`SELECT`) comme dans Illustra [Stonebraker90]. Seuls des systèmes de recherche gèrent des événements composés, avec `OU, ET`, séquences et intervalles de temps. On citera par exemple HIPAC [Dayal88] et SAMOS [Gatziu92].

Le standard SQL3 supporte des conditions et actions exécutées avant le traitement naturel de l'opération initiatrice, ou après, ou à sa place. Il permet aussi de différer l'exécution de certains déclencheurs en fin de transaction. Par contre, les déclencheurs sont exécutés pour le compte de la transaction déclenchante. Ils ne peuvent être découplés. Le découplage pose beaucoup de problèmes de sémantique et a été partiellement exploré dans des projets de recherche comme SAMOS.

Les déclencheurs sont très utiles dans les SGBD. Ils correspondent cependant à une vision procédurale des règles. De ce fait, la sémantique est souvent obscure. La conception de bases de données actives efficaces reste un problème difficile.

9. BIBLIOGRAPHIE

[Agrawal91] Agrawal R., Cochrane R.J., Linsay B., « On Maintaining Priorities in a Production Rule System », *Proc. 17th Int. Conf. on Very Large Data Bases*, Barcelona, Spain, Morgan Kaufman Ed., p. 479-487, Sept. 1991.

Cet article rapporte sur l'expérimentation du mécanisme de priorité entre règles implémenté à IBM dans le projet Starburst. Il montre l'intérêt du mécanisme.

[Bernstein80] Bernstein P., Blaustein B., Clarke E..M., « Fast Maintenance of semantic Integrity Assertions Using Redundant Aggregate Data », *Proc. 6th Int. Conf. on Very Large Data Bases*, Montreal, Canada, Morgan Kaufman Ed., Oct. 1991.

Les auteurs furent les premiers à proposer la maintenance d'agrégats redondants pour faciliter la vérification des contraintes d'intégrité lors des mises à jour. Depuis, ces techniques se sont répandues sous la forme de vues concrètes.

[Ceri90] Ceri S., Widom J., « Deriving Production Rules for Constraint Maintenance », *Proc. 16th Intl. Conf. on Very Large Data Bases*, Brisbane, Australia, Morgan Kaufman Ed., p. 566-577, Aug. 1990.

Les auteurs proposent des algorithmes pour générer des déclencheurs permettant la vérification automatique de règles d'intégrité lors des mises à jour. De tels algorithmes pourraient être intégrés à un compilateur de définitions de contraintes d'intégrité.

[Cochrane96] Cocherane R., Pirahesh H., Mattos N., Integrating triggers and Declarative Constraints in SQL Database Systems, *Proc. 16th Intl. Conf. on Very Large Data Bases*, Brisbane, Australia, Morgan Kaufman Ed., p. 566-577, Aug. 1990.

L'article de référence pour la sémantique des déclencheurs en SQL3. Après un rappel des contraintes d'intégrité existant en SQL2, l'article montre comment on exécute les déclencheurs en absence de contraintes, puis les interactions entre ces deux mécanismes. Il définit ensuite une sémantique de point fixe pour les déclencheurs.

[Date81] Date C.J., « Referential Integrity », *Proc. 7th Intl. Conf. on Very Large Data Bases*, Cannes, France, IEEE Ed., Sept. 1981.

L'article de base qui a introduit les contraintes référentielles. Celles-ci étaient intégrées au relationnel pour répondre aux attaques de perte de sémantique du modèle par rapport aux modèles type réseau ou entité-association.

[Dayal88] Dayal U., Blaunstein B., Buchmann A., Chakravavarthy S., Hsu M., Ladin R., McCarthy D., Rosenthal A., Sarin S., Carey M. Livny M., Jauhari J., « The HiPAC Project : Combining Active databases and Timing Constraints », *SIGMOD Record V° 17, n° 1*, Mars 1988.

Un des premiers projets à étudier un système de déclencheurs avec contraintes temporelles. Le système HiPAC fut un précurseur en matière de déclencheurs.

[Dittrich95] K.R., Gatziu S., Geppert A., « The Active Database Management System Manifesto », *Proc. 2nd Int. Workshop on Rules in Databas Systems*, Athens, Greece, Sept. 1995.

Ce manifesto se veut l'équivalent pour les bases de données actives du manifesto des bases de données objet. Il définit précisément les fonctionnalités que doit supporter un SGBD actif.

[Eswaran75] Eswaran K.P., Chamberlin D.D., « Functional Specifications of a Subsystem for Database Integrity », *Proc. 1st Intl. Conf. on Very Large Data Bases*, Framingham, Mass., p. 48-67, Sept. 1975.

La description détaillée du premier sous-système d'intégrité réalisé. Ce travail a été effectué dans le cadre du System R à IBM.

[Eswaran76] Eswaran K.P., Chamberlin D.D., « Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System », *IBM Research report RJ 1820*, IBM Research Lab., San José, California, Août 76.

La description détaillée du premier sous-système de déclencheurs réalisé. Ce travail a été effectué dans le cadre du System R à IBM.

[Gardarin79] Gardarin G., Melkanoff M., « Proving Consistency of Database Transactions », *Proc. 5th Int. Conf. on Very Large Data Bases*, Rio de Janeiro, Brésil, Sept. 1979.

Cet article propose une méthode pour générer automatiquement des pré-tests dans des programmes écrits en PASCAL/SQL. La méthode est basée sur une technique de preuve d'assertions en utilisant la sémantique de Hoare.

[Hammer78] Hammer M., Sarin S., « Efficient Monitoring of Database Assertions », *Proc. ACM SIGMOD Int. Conf. On Management of Data*, 1978.

Cet article propose des techniques de simplification de contraintes d'intégrité basées sur l'étude logique de ces contraintes.

[Horowitz92] Horowith B., « A Runtime Excecution Model for referential Integrity Maintenance », *Proc. 8th Intl. Conf. on Data Engineering*, Tempe, Arizona, p. 546-556, 1992.

Les auteurs décrivent un prototype réalisant efficacement la vérification des contraintes référentielles lors de l'exécution des mises à jour.

[Llirbat97] Llirbat F., Fabret F., Simon E., « Eliminating Costly Redundant Computations from SQL Trigger Executions » *Proc. ACM SIGMOD Int. Conf.* on Management of data, Tucson, Arizona, p. 428-439, Juin 1997.

Cet article développe une nouvelle technique d'optimisation des déclencheurs en SQL. Cette technique, au cœur de la thèse de F. Llirbat, permet d'extraire les invariants dans les boucles et de les mémoriser afin d'éviter les recalculs. Le papier développe aussi un modèle pour décrire les programmes, les déclencheurs, et leurs interactions.

[Simon87] Simon E., Valduriez P., « Design and Analysis of a Relational Integrity Subsystem », *MCC Technical report Number DB-015-87*, Jan. 1987.

Ce rapport résume la thèse d'Éric Simon (Paris VI, 1987) et décrit le système de contrôle d'intégrité du SGBD SABRE réalisé à l'INRIA. Ce système est basé sur des post-tests différentiels.

[Stonebraker75] Stonebraker M., « Implementation of integrity Constraints and Views by Query Modification », *Proc. ACM SIGMOD*, San José, California, 1975.

Cet article présente le premier sous-système de gestion de vues et d'intégrité d'Ingres, basé sur la modification de questions. Celle-ci est utilisée afin de générer des pré-tests intégrés aux requêtes modifiées.

[Stonebraker90] Stonebraker M., Jhingran A., Goh J., Potamianos S., « On Rules, Procedures, Caching, and Views in Database Systems », *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, New Jersey, p. 281-290, Juin 1990.

Suite aux expériences d'Ingres et Postgres, les auteurs commentent les mécanismes de gestion de vues, caches, procédures et règles dans les SGBD. Ils proposent d'intégrer tous ces composants autour du mécanisme de contrôle de concurrence, en étendant les types de verrous possibles. Les idées originales développées furent implémentées au cœur de Postgres.

[Widom91] Widom J, Cochrane R., Lindsay B., « Implementing Set-oriented Production Rules as an Extension to Starburst », *Proc. 17th Int. Conf. on Very Large Data Bases*, Barcelona, Spain, Morgan Kaufman Ed., p. 275-285, Sept. 1991.

Cet article décrit l'implémentation des règles réalisée dans Starburst au centre de recherche d'IBM en Californie. Ce prototype fut à la source du système de triggers aujourd'hui supporté par DB2.

[Widom96] Widom J., Ceri S., *Active Database Systems: Triggers and Rules for Advanced Database Processing*, Morgan-Kaufmann, San Fransisco, California, 1996.

Ce livre de synthèse sur les bases de données actives présente les concepts de base, différents prototypes et de multiples expériences réalisées sur le sujet.

Chapitre IX

LA GESTION DES VUES

1. INTRODUCTION

Pourquoi des **vues** ? Elles permettent de réaliser, dans le monde des SGBD relationnels, le niveau externe des SGBD selon l'architecture ANSI/SPARC. Rappelons que le niveau externe propose à l'utilisateur une perception de la base plus proche de ses besoins, en termes de structures et de formats de données. De manière générale, les vues garantissent une meilleure indépendance logique des programmes par rapport aux données. En effet, le programme restera invariant aux modifications de schéma s'il accède à la base via une vue qui l'isole de celle-ci. Lors des modifications du schéma de la base, l'administrateur modifiera seulement la définition des vues, et les programmes d'application pourront continuer à travailler sans modification. Les vues ont aussi un rôle de sécurité : l'utilisateur ne peut accéder qu'aux données des vues auxquelles il a droit d'accès ; ainsi, les données en dehors de la vue sont protégées. De manière détournée, les vues permettent aussi certains contrôles d'intégrité lorsqu'elles sont utilisées pour les mises à jour : on dit alors qu'on effectue une mise à jour au travers d'une vue. De telles mises à jour sont très contrôlées, comme nous le verrons ci-dessous.

Ces dernières années, les vues ont trouvé de nouvelles applications. Elles permettent de définir des tables virtuelles correspondant aux besoins des programmes d'application en termes de données. Dans le monde du client-serveur, les vues constituent un élément essentiel d'optimisation de performance. Par exemple, la définition d'une vue

jointure de deux tables, ou résultant d'un calcul d'agrégat, évite au client les risques de lire les tuples des tables et de faire le calcul de jointure ou d'agrégat sur le client : seules les données résultant du calcul de la vue sont exportées sur le site client. On laisse ainsi faire au serveur les calculs de jointure, agrégat, etc., qu'il sait en principe bien faire. Dans le monde des entrepôts de données et du décisionnel, les vues peuvent être concrétisées. Elles permettent de réaliser par avance des cumuls ou synthèses plus sophistiqués de quantités extraites de la base selon plusieurs dimensions. Des mécanismes de mises à jour des vues concrètes lors des mises à jour des relations de base sont alors développés afin d'éviter le recalcul des cumuls.

Les vues ont donc une importance croissante dans les bases de données. En pratique, ce sont des relations virtuelles définies par des questions. Cette définition est stockée dans la métabase. Les vues sont interrogées comme des relations normales. Idéalement, elles devraient pouvoir être mises à jour comme des relations normales. Les vues concrètes sont calculées sur disques lors de leurs créations. Des mécanismes de mise à jour différentiels permettent le report efficace des mises à jour des relations de base. Toutes ces techniques sont aujourd'hui bien maîtrisées ; nous allons les présenter ci-dessous.

Afin d'illustrer les mécanismes de vues, nous utiliserons tout d'abord la base de données viticole classique composée des relations :

BUVEURS (NB, NOM, PRENOM, ADRESSE, TYPE)
VINS (NV, CRU, REGION, MILLESIME, DEGRE)
ABUS (NV, NB, DATE, QUANTITÉ).

décrivant respectivement les buveurs, les vins, et les consommations de vins quotidiennes. Comme habituellement, les clés des relations sont soulignées. Des vues typiques sont les vins de Bordeaux, les gros buveurs, les quantités de vins bues par crus, etc.

Pour des exemples plus avancés, intégrant notamment des agrégats complexes, nous utiliserons la base MAGASINS suivante :

VENTES(NUMV, NUMPRO, NUMFOU, DATE, QUANTITÉ, PRIX)
PRODUITS (NUMPRO, NOM, MARQUE, TYPE,PRIX)
FOURNISSEURS (NUMFOU, NOM, VILLE, RÉGION, TELEPHONE)

Des vues typiques sont les quantités de produits vendus par fournisseurs et par mois, les évolutions des quantités commandées par région, etc. La relation Ventes décrit les ventes journalières de produits. NUMPRO est la clé de produits et NUMFOU celle de fournisseurs. Dans une telle base de données, les faits de base sont les ventes, alors que produits et fournisseurs constituent des dimensions permettant d'explorer les ventes selon une ville ou une région par exemple.

Ce chapitre est organisé comme suit. Après cette introduction, la section 2 expose plus formellement le concept de vue, détaille le langage de définition et présente quelques exemples simples de vues. La section 3 développe les mécanismes d'interro-

gation de vues. La section 4 pose le problème de la mise à jour des vues et isole les cas simples tolérés par SQL. La section 5 traite des vues concrètes, notamment en vue des applications décisionnelles. En particulier, les techniques de report des mises à jour depuis les relations de base sur des vues concrètes avec agrégats sont étudiées. La conclusion évoque quelques autres extensions possibles du mécanisme de gestion des vues.

2. DÉFINITION DES VUES

Dans cette partie, nous définissons tout d'abord précisément ce qu'est une **vue**, puis nous introduisons la syntaxe SQL pour définir une vue.

Notion IX.1 : Vue (*View*)

Une ou plusieurs tables virtuelles dont le schéma et le contenu sont dérivés de la base réelle par un ensemble de questions.

Une vue est donc un ensemble de relations déduites d'une base de données, par composition des relations de la base. Le schéma de la vue est un schéma externe au sens ANSI/SPARC. Dans la norme SQL, la notion de vue a été réduite à une seule relation déduite. Une vue est donc finalement une table virtuelle calculable par une question.

La syntaxe générale de la commande SQL1 de création de vue est :

```
CREATE VIEW <NOM DE VUE> [ (LISTE D'ATTRIBUT)]
AS <QUESTION>
[WITH CHECK OPTION]
```

Le nom de vue est le nom de la table virtuelle correspondant à la vue, la liste des attributs définit les colonnes de la table virtuelle, la question permet de calculer les tuples peuplant la table virtuelle. Les colonnes du `SELECT` sont appliquées sur celles de la vue. Si les colonnes de la vue ne sont pas spécifiées, celle-ci hérite directement des colonnes du `SELECT` constituant la question.

La clause `WITH CHECK OPTION` permet de spécifier que les tuples insérés ou mis à jour via la vue doivent satisfaire aux conditions de la question définissant la vue. Ces conditions seront vérifiées après la mise à jour : le SGBD testera que les tuples insérés ou modifiés figurent bien parmi la réponse à la question, donc dans la vue. Ceci garantit que les tuples insérés ou modifiés via la vue lui appartiennent bien. Dans le cas contraire, la mise à jour est rejetée si la clause `WITH CHECK OPTION` est présente. Par exemple, si la vue possède des attributs d'une seule table et la question un critère de jointure avec une autre table, les tuples insérés dans la table correspondant à

la vue devront joindre avec ceux de l'autre table. Ceci peut être utilisé pour forcer la vérification d'une contrainte référentielle lors d'une insertion via une vue. Nous étudierons plus en détail la justification de cette clause dans la partie traitant des mises à jour au travers de vues.

La suppression d'une vue s'effectue simplement par la commande :

```
DROP <NOM DE VUE>.
```

Cette commande permet de supprimer la définition de vue dans la métabase (aussi appelée catalogue) de la base. En principe, sauf comme nous le verrons ci-dessous, dans le cas des vues concrètes, une vue n'a pas d'existence physique : c'est une fenêtre dynamique (et déformante), non matérialisée sur les disques, par laquelle un utilisateur accède à la base. La destruction de vue n'a donc pas à se soucier de détruire des tuples.

Voici maintenant quelques exemples de définition de vues. Soit la base de données Viticole dont le schéma a été rappelé en introduction.

(V1) Les vins de Bordeaux :

```
CREATE VIEW VINSBORDEAUX (NV, CRU, MILL, DEGRÉ)
AS SELECT NV, CRU, MILLÉSIME, DEGRÉ
FROM VINS
WHERE RÉGION = "BORDELAIS".
```

Cette vue est simplement construite par une restriction suivie d'une projection de la table Vins. Chaque tuple de la vue est dérivé d'un tuple de la table Vins.

(V2) Les gros buveurs :

```
CREATE VIEW GROSBUVEURS
AS SELECT NB, NOM, PRÉNOM, ADRESSE
FROM BUVEURS B, ABUS A
WHERE B.NB = A.NB AND A.QUANTITÉ > 10
WITH CHECK OPTION
```

Un tuple figure dans la vue `GrosBuveurs` si le buveur correspondant a commis au moins un abus en quantité supérieure à 10 verres. C'est là une définition très large des gros buveurs. La définition de vue comporte une restriction et une jointure. La clause `WITH CHECK OPTION` précise que lors d'une insertion d'un buveur via la vue, on doit vérifier qu'il s'agit bien d'un gros buveur, c'est-à-dire que le buveur a déjà commis un abus de quantité supérieure à 10. Notez que ceci spécifie une règle d'intégrité référentielle à l'envers pour les gros buveurs, ce qui est paradoxal.

(V3) Les quantités de vins bues par crus :

```
CREATE VIEW VINSBUS (CRU, MILL, DEGRÉ, TOTAL)
AS SELECT CRU, MILLÉSIME, DEGRÉ, SUM(QUANTITÉ)
FROM VINS V, ABUS A
WHERE V.NV = A.NV
GROUP BY CRU.
```

Cette vue fait appel à une jointure (suivant une contrainte référentielle) suivie d'un calcul d'agrégat (la somme). Un tuple de la table `Vins` donne plusieurs tuples lors de la jointure (autant qu'il y a d'abus) ; ceux-ci sont ensuite regroupés selon le cru.

Pour la base `MAGASINS`, nous définirons seulement une vue (V4) documentant la table `VENTES` selon les « dimensions » `PRODUITS` et `FOURNISSEURS`, résultant de jointures sur clé de ces tables :

```
CREATE VIEW VENTEDOC (NUMV,NOMPRO,MARQUE,NOMFOU,VILLE,RÉGION, DATE,
        QUANTITÉ, PRIX) AS
SELECT V.NUMV,P.NOM,P.MARQUE,F.NOM,F.VILLE,F.RÉGION,V.DATE,
V.QUANTITÉ, V.PRIX
FROM VENTES V, PRODUITS P, FOURNISSEURS F
WHERE V.NUMPRO=P.NUMRO AND V.NUMFOU=F.NUMFOU
```

Nous verrons des vues plus complexes, notamment avec des agrégats ci-dessous.

La suppression des vues ci-dessus s'effectuera par les commandes :

```
DROP VINSBORDEAUX.
DROP GROSBUVEURS.
DROP VINBUS.
DROP VENTESDOC.
```

3. INTERROGATION AU TRAVERS DE VUES

Du point de vue de l'utilisateur, l'interrogation au travers d'une vue s'effectue comme pour une table normale. Le rôle du serveur est alors d'enrichir la question avec la définition de vue qu'il retrouve dans la métabase. Pus formellement, la définition d'une vue V sur des relations R1, R2...Rn est une fonction V = F(R1, R2...Rn), où F est une expression de l'algèbre relationnelle étendue avec des agrégats. F est donc une expression de restriction, projection, jointure, union, différence et agrégats. Une question Q est exprimée sur la vue en SQL. Il s'agit aussi d'une fonction calculant la réponse R = Q(V), où Q est aussi une expression de l'algèbre relationnelle étendue. Calculer R à partir de la base revient donc à remplacer V dans la requête R = Q(V) par sa définition. On obtient alors la fonction composée :

R = Q(F(R1, R2...Rn)).

Ce mécanisme est illustré figure IX.1. La requête résultante peut alors être passée à l'optimiseur et le plan d'exécution correspondant peut être calculé. C'est ainsi que fonctionnent les SGBD : ils génèrent la requête composition de la définition de vue et de la requête usager, et traitent ensuite cette requête plus complexe, mais évaluable sur les relations de la base.

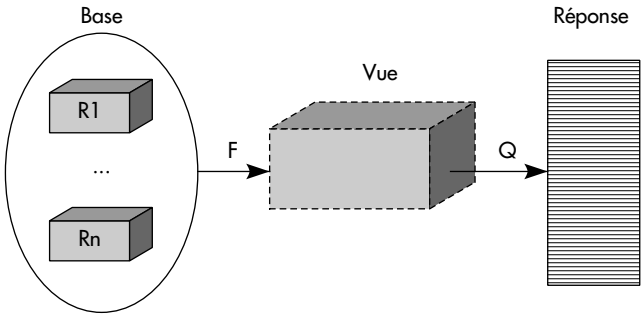


Figure IX.1 : Composition d'une question Q avec la définition de vue F

Deux techniques sont possibles pour réaliser la composition de la vue et de la requête utilisateur : la transformation de la requête source appelée **modification de question**, ou la transformation de l'arbre relationnelle, parfois appelée **concaténation d'arbre**.

Notion IX.2 : Modification de question (*Query modification*)

Mécanisme consistant à modifier une question en remplaçant certaines vues du `FROM` *par les relations de base sources de ces vues et en enrichissant les conditions de la clause* `WHERE` *pour obtenir le résultat de la question initiale.*

La modification de question est une technique inventée dans le projet Ingres à Berkeley [Stonebraker75]. Elle permet de remplacer les vues par leurs définitions lors des recherches ou d'ajouter des prédicats afin de vérifier des propriétés avant d'exécuter une requête. La figure IX.2 illustre cette technique pour la recherche des gros buveurs habitant à Versailles. Cette technique peut aussi être utilisée pour les mises à jour afin d'enrichir le critère pour vérifier par exemple la non-violation de contraintes d'intégrité.

(1) Question

```
SELECT NOM, PRÉNOM
FROM GROSBUVEURS
WHERE ADRESSE LIKE "VERSAILLES".
```

(2) Définition de vue

```
CREATE VIEW GROSBUVEURS
AS SELECT NB, NOM, PRÉNOM, ADRESSE
FROM BUVEURS B, ABUS A
WHERE B.NB = A.NB AND A.QUANTITÉ > 10.
```

(3) Question modifiée

```
SELECT NOM, PRÉNOM
FROM BUVEURS B, ABUS A
WHERE B.ADRESSE LIKE "VERSAILLES" AND B.NB=A.NB AND A.QUANTITÉ>10.
```

Figure IX.2 : Exemple de modification de question

Notion IX.3 : Concaténation d'arbres (*Tree concatenation*)

Mécanisme consistant à remplacer un nœud pendant dans un arbre relationnel par un autre arbre calculant le nœud remplacé.

La concaténation d'arbres est une technique inventée dans le fameux système R à San-José [Astrahan76]. La définition de vue se retrouve dans la métabase sous la forme d'un arbre relationnel. L'arbre résultant de l'analyse de la question est simplement connecté à celui définissant la vue. L'ensemble constitue un arbre qui représente la question enrichie, qui est alors passée à l'optimiseur. La figure IX.3 illustre ce mécanisme pour la recherche des gros buveurs habitant à Versailles.

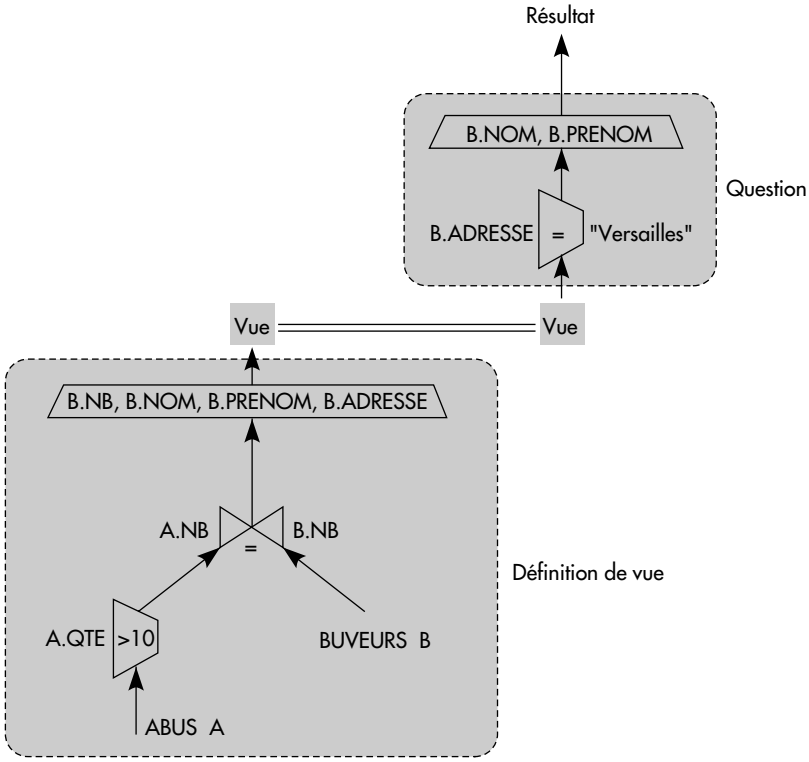


Figure IX.3 : Exemple de concaténation d'arbres

4. MISE A JOUR AU TRAVERS DE VUES

Dans cette section, nous examinons, d'un point de vue pratique puis théorique, le problème des mises à jour au travers des vues.

4.1. VUE METTABLE À JOUR

Le problème est de traduire une mise à jour (ou une insertion, ou une suppression) portant sur une vue en mise à jour sur les relations de la base. Toute vue n'est pas **mettable à jour**.

Notion IX.4 : Vue mettable à jour (*Updatable view*)

Vue comportant suffisamment d'attributs pour permettre un report des mises à jour dans la base sans ambiguïté.

De manière plus fine, une vue peut être mettable à jour en insertion, en suppression, ou en modification. Par exemple, la vue V1 définissant les vins de Bordeaux est totalement mettable à jour, c'est-à-dire que toute opération `INSERT`, `DELETE` ou `UPDATE` est reportable sur la base. Ajoutons à la vue V2 définissant les gros buveurs la quantité bue (`QUANTITÉ`) après le `SELECT`. Ceci pose problème lors d'une insertion : comment générer le numéro de vins (`NV`) et la date (`DATE`) de l'abus qui doit obligatoirement être inséré puisque `NB`, `NV`, `DATE` est clé de `ABUS` ? Si l'on obligeait l'existence de l'abus avant d'insérer le buveur il n'y aurait pas de problème. La clause `WITH CHECK OPTION` permet justement de vérifier l'existence de tels tuples. Malheureusement, sa présence simultanée à la contrainte référentielle `ABUS.NB REFERENCE BUVEURS` est impossible ! La suppression et la modification de tuples existants ne pose pas non plus de problème. La vue V3 est encore plus difficile à mettre à jour : il est impossible de déterminer les quantités élémentaires à partir de la somme ! En résumé, l'utilisation de certaines vues en mises à jour est problématique.

4.2. APPROCHE PRATIQUE

En pratique, la plupart des systèmes résolvent le problème en restreignant les possibilités de mise à jour à des vues monotable : seuls les attributs d'une table de la base doivent apparaître dans la vue. En imposant de plus que la clé de la table de la base soit présente, il est possible de définir une stratégie de mise à jour simple. Lors d'une insertion, on insère simplement les nouveaux tuples dont la clé n'existe pas, avec des valeurs nulles pour les attributs inexistants. Lors d'une suppression, on supprime les tuples répondant au critère. Lors d'une modification, on modifie les tuples répondant

au critère. La définition de vue peut référencer d'autres tables qui permettent de préciser les tuples de la vue. En théorie, il faut vérifier que les tuples insérés, supprimés ou modifiés appartiennent bien à la vue. En pratique, SQL n'effectue cette vérification que si elle a été demandée lors de la définition de vue par la `clause WITH CHECK OPTION`.

Restreindre la mise à jour à des vues monotables est beaucoup trop fort. On peut simplement étendre, au moins en insertion et en suppression, aux vues multitables comportant les clés des tables participantes. Les attributs non documentés sont alors remplacés par des valeurs nulles lors des insertions. Cette solution pratique génère cependant des bases de données avec de nombreuses valeurs nulles. Elle sera souvent interdite dans les systèmes commercialisés. Ceux-ci sont donc loin de permettre toutes les mises à jour théoriquement possibles, comme le montre la figure IX.4.

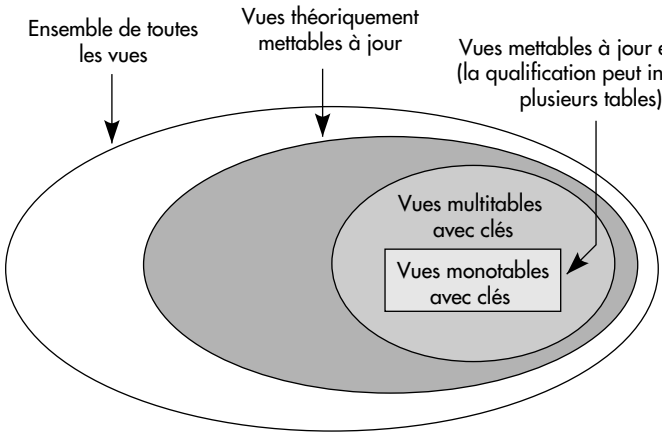


Figure IX.4 : Classification des vues selon les possibilités de mise à jour

4.3. APPROCHE THÉORIQUE

Le problème théorique est de définir une stratégie de report qui préserve la vue quelle que soit la mise à jour : si l'on calcule la vue et on lui applique la mise à jour, on doit obtenir le même résultat que si on applique les mises à jour à la base et on calcule ensuite la vue. L'équation `u(V(B)) = V(u'(B))` doit donc être vérifiée, en notant `B` la base, `v` le calcul de vue, `u` la mise à jour sur la vue et `u'` les mises à jour correspondantes sur la base. Autrement dit, le diagramme de la figure IX.5 doit commuter. D'autre part, il est évident qu'une bonne stratégie doit préserver le complément de la base, c'est-à-dire toutes les informations de la base qui ne figurent pas dans la vue [Bancilhon81]. Selon ces hypothèses raisonnables, le problème du calcul de u' n'a pas toujours une solution unique [Abiteboul91].

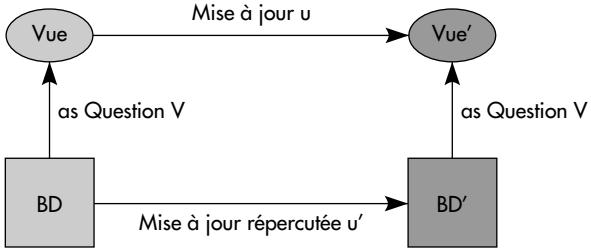


Figure IX.5 : Diagramme de mise à jour et dérivation de vue

Une approche intéressante a été proposée récemment dans [Bentayeb97]. L'idée est de définir l'inverse de chaque opération relationnelle. En effet, soit V = E(R1, R2...Rn) une définition de vue. E est une expression de l'algèbre relationnelle. Si l'on peut définir l'inverse de chaque opération de l'algèbre sur une relation, on pourra alors calculer R1 × R2... × Rn = E^{-1}(V). La répercussion d'une mise à jour de V se fera simplement en unifiant Ri avec la projection de E^{-1}(V) sur Ri. Le problème de l'inversion de l'algèbre relationnelle a aussi été étudié dans [Imielinski83]. Désignons par t un tuple de la vue V et par [t] une table composée de ce seul tuple. L'**image réciproque** d'un tuple t par E peut être vue comme un ensemble de tables T1, T2..., Tn telles que E (T1, T2..., Tn) = [t]. Les tables Ti peuvent posséder des attributs inconnus désignés par des variables x,y, etc. Elles ne doivent pas contenir de tuples inutiles pour composer t, donc pour tout tuple ti ∈ Ti, E (T1..., Ti-[ti]..., Tn) ≠ [t]. Insérer le tuple t dans la vue V a alors pour effet de modifier les relations de base Ri en les unifiant avec les relations Ti. L'unification est une opération difficile, pas toujours possible, explicitée dans [Bentayeb97] sous forme d'algorithmes pour l'insertion et la suppression.

En résumé, la mise à jour au travers de vue est un problème complexe, bien étudié en théorie, mais dont les solutions pratiques sont réduites. Certains SGBD permettent des mises à jour de vues multitables en laissant l'administrateur définir la stratégie de report.

5. VUES CONCRÈTES ET DÉCISIONNEL

Dans cette section, nous abordons le problème des vues concrètes. Celles-ci sont particulièrement intéressantes dans les **entrepôts de données** (*data warehouse*), où elles facilitent l'analyse de données (*OLAP*) pour l'aide à la décision.

5.1. DÉFINITION

Une vue est en principe une fenêtre dynamique sur une base de données, dont une partie est instanciée lors d'une question. La concrétisation de la vue par le serveur peut

être plus avantageuse si celle-ci est souvent utilisée et si les tables sources sont peu modifiées. Ainsi, certains serveurs supportent des **vues concrètes**.

Notion IX.5 : Vue concrète (*Concrete view*)

Table calculée à partir des tables de la base par une question et matérialisée sur disques par le SGBD.

Une vue concrète est calculée dès sa définition et mise à jour chaque fois qu'une transaction modifie la base sous-jacente. La mise à jour s'effectue si possible en différentiel, c'est-à-dire que seuls les tuples modifiés sont pris en compte pour calculer les modifications à apporter à la vue. Mais ceci n'est pas toujours possible. Dans le cas de vues définies par des sélections, projections et jointures (SPJ), le report différentiel des insertions est simple, celui des mises à jour de type suppression ou modification beaucoup plus difficile. Pour répercuter suppressions et mises à jour, il faut en général retrouver – au moins partiellement – les chaînes de dérivation qui ont permis de calculer les tuples de la vue concrète. Dans le cas général (vues avec différence, agrégat, etc.), les reports différentiels sont très difficiles, comme nous le verrons ci-dessous

Les vues concrètes sont définissables par une requête du type :

```
CREATE CONCRETE VIEW <SPÉCIFICATION DE VUE>.
```

Elles sont particulièrement intéressantes lorsqu'elles contiennent des agrégats, car elles permettent de mémoriser des résumés compacts des tables. Par exemple, il est possible de définir des vues concrètes avec agrégats de la table VENTES définie dans l'introduction :

```
VENTES(NUMV, NUMPRO, NUMFOU, DATE, QUANTITÉ, PRIX)
```

Les tables PRODUITS et FOURNISSEURS permettent de générer des exemples de vues avec jointures :

```
PRODUITS (NUMPRO, NOM, MARQUE, TYPE,PRIX)
FOURNISSEURS (NUMFOU, NOM, VILLE, RÉGION, TELEPHONE)
```

La vue suivante donne les ventes de produits par fournisseur et par jour :

```
CREATE CONCRETE VIEW VENTESPFD(NUMPRO,NUMFOU,DATE,COMPTE,QUANTOT)AS
SELECT NUMPRO,NUMFOU,DATE,COUNT(*)AS COMPTE,SUM(QUANTITÉ) AS QUANTOT
FROM VENTES
GROUP BY NUMPRO, NUMFOU, DATE.
```

La notation VENTESPFD signifie « ventes groupées par produits, fournisseurs et dates ». Une vue plus compacte éliminera par exemple les fournisseurs :

```
CREATE CONCRETE VIEW VENTESPD (NUMPRO,DATE,COMPTE,QUANTOT) AS
SELECT NUMPRO, DATE,COUNT(*)AS COMPTE,SUM(QUANTITÉ) AS QUANTOT
FROM VENTES
GROUP BY NUMPRO, DATE.
```

Notez que la vue VENTESPD peut être dérivée de la vue VENTESPFD par la définition suivante :

```
CREATE CONCRETE VIEW VENTESPD (NUMPRO, DATE, COMPTE,QUANTOT) AS
SELECT NUMPRO, DATE,SUM(COMPTE)AS COMPTE,SUM(QUANTOT) AS QUANTOT
FROM VENTESPFD
GROUP BY NUMFOU.
```

Il est aussi possible de dériver des vues concrètes avec agrégats par jointures avec les tables dimensions, par exemple en cumulant les ventes par région des fournisseurs :

```
CREATE CONCRETE VIEW VENTESPRD(NUMPRO,RÉGION,DATE,COMPTE,QUANTOT) AS
SELECT NUMPRO, DATE,COUNT(*)AS COMPTE,SUM(QUANTITÉ) AS QUANTOT
FROM VENTES V, FOURNISSEURS F
WHERE V.NUMFOU = F.NUMFOU
GROUP BY V.NUMPRO, F.RÉGION.
```

Toutes ces vues concrètes sont très utiles pour l'aide à la décision dans un contexte d'entrepôt de données, où l'on souhaite analyser les ventes selon différentes dimensions [Gray96].

5.2. STRATÉGIE DE REPORT

Les vues concrètes doivent être mises à jour lors de la mise à jour des relations de la base. Un problème important est de déterminer une stratégie de report efficace des mises à jour effectuées sur les relations de base. Bien sûr, il est possible de recalculer complètement la vue après chaque mise à jour, mais ceci est inefficace.

Une meilleure stratégie consiste à maintenir la vue concrète de manière différentielle, ou si l'on préfère incrémentale. Soit une vue V = F(R1, R2...Rn) définie sur les relations R1, R2... Rn. Des mises à jour sont effectuées par exemple sur la relation Ri. Soient des insertions de tuples notées Δ^+Ri et des suppressions notées Δ^-Ri. Une modification de tuples existants peut toujours se ramener à une suppression suivie d'une insertion. L'effet net des mises à jour sur Ri est donc Ri = (Ri – Δ^-Ri) $\cup$ Δ^+Ri. Le problème est de calculer l'effet net sur V, qui peut être exprimé par des tuples supprimés, puis par des tuples insérés comme suit : V = (V – Δ^-V) $\cup$ Δ^+V. Dans certains cas, les mises à jour à apporter sur V (Δ^-V, Δ^+V) peuvent être calculées à partir de celles apportées sur Ri (Δ^-Ri, Δ^+Ri). La vue V est alors qualifiée d'**auto-maintenable** [Tompa88, Gupta95]

Notion IX.6 : Vue auto-maintenable (*Self-maintenable view*)

Vue concrète contenant suffisamment d'informations pour être mise à jour à partir des mises à jour des relations de base, sans accès aux tuples des relations de la base.

Cette notion est très importante dans les architectures distribuées (client-serveur, entrepôts de données, copies), où la vue est matérialisée sur un site différent de celui

de la base. Elle est illustrée figure IX.6. Il est possible de distinguer l'auto-maintenabilité en insertion ou en suppression, c'est-à-dire lors des insertions ou suppressions dans une relation de base. L'auto-maintenabilité peut aussi être caractérisée pour chaque relation, et enfin généralisée au cas d'un groupe de vues [Huyn97].

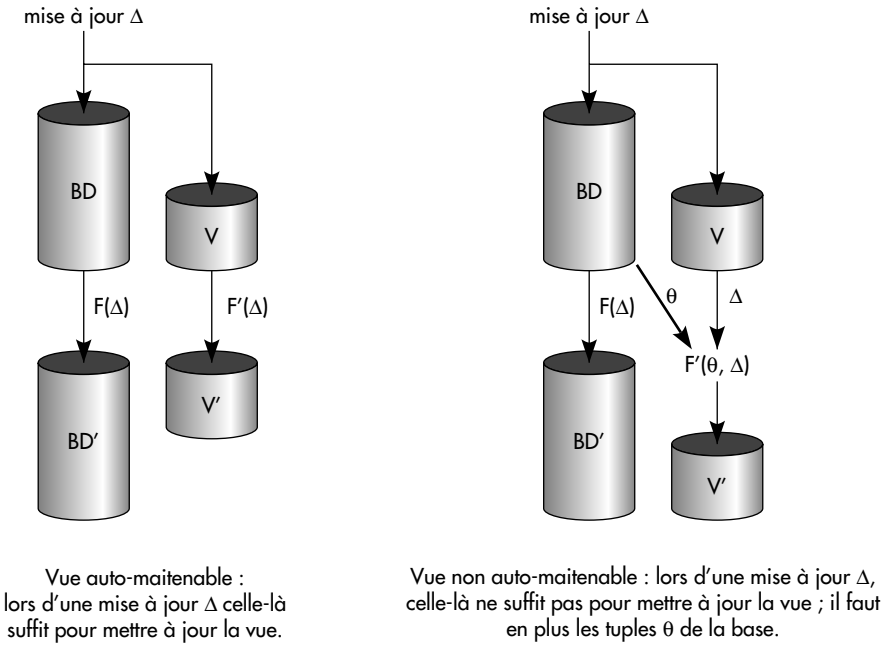


Figure IX.6 : Maintenance de vues avec ou sans accès à la base

En guise d'illustration, considérons la vue V1 des vins de Bordeaux définie ci-dessus comme une restriction de la table `Vins` : $V1 = \sigma_{REGION = \text{“Bordelais”}}$ (VINS). Cette vue est auto-maintenable. En effet, comme la restriction commute avec la différence, il est simple de calculer :

$\Delta^{+}V1 = \sigma_{REGION = \text{“Bordelais”}} (\Delta^{+}VINS)$.

$\Delta^{-}V1 = \sigma_{REGION = \text{“Bordelais”}} (\Delta^{-}VINS)$.

Par suite, les mises à jour à apporter $\Delta V1$ sont calculables sans accéder à la table VINS, mais seulement à partir des mises à jour de `VINS`. Malheureusement, dès qu'une vue comporte des projections avec élimination de doubles, des jointures ou des différences, elle n'est généralement plus auto-maintenable [Fabret94]. Par exemple, dans le cas de projections sans double, lorsque l'on enlève un tuple d'une table source, on ne sait plus s'il faut l'enlever dans la vue car sa projection peut provenir d'un autre tuple. De même, pour calculer le tuple à enlever dans une vue jointure lors d'une suppression d'un tuple dans une table, il faut accéder à l'autre table pour trouver les tuples jointures. De manière générale, savoir si une vue est auto-

maintenable s'avère difficile, et même très difficile lorsque la vue contient des agrégats [Gupta96]. Ce dernier cas est examiné dans le paragraphe qui suit.

Dans le cas d'une vue calculée par sélections, projections et jointures (SPJ), les problèmes essentiels semblent provenir des jointures. Toute vue comportant toutes les informations de la base nécessaires à son calcul (attributs de projection et de jointure, tuples) est **auto-maintenable en insertion** ; en effet, la nouvelle instance de la vue V' est :

$$V' = F(R1..., Ri \cup \Delta^{+}Ri..., Rn) .$$

Comme F ne contient pas de différence, on a :

$$V' = F(R1..., Ri..., Rn) \cup F(R1..., \Delta^{+}Ri..., Rn) = V \cup F(R1..., \Delta^{+}Ri..., Rn)$$

d'où l'on déduit :

$$\Delta V^{+} = F(R1..., \Delta^{+}Ri..., Rn).$$

Si F (R1..., Δ^{+}Ri..., Rn) est calculable à partir de V et Δ^{+}Ri, la vue est auto-maintenable en insertion. Tel est le cas à condition que V contienne les attributs et les tuples de R1...Rn nécessaires à son calcul. Les jointures externes seules ne perdent pas de tuples. Donc, une vue calculée par des jointures externes et projections est généralement auto-maintenable en insertion.

Pour l'**auto-maintenabilité en suppression**, le problème est plus difficile encore. On ne peut distribuer F par rapport à Ri – Δ^{-}Ri, ce qui rend le raisonnement précédent caduc. Cependant, toute vue contenant les attributs de R1..., Rn nécessaires à son calcul et au moins une clé de chaque relation de base est auto-maintenable. En effet, l'origine des tuples de la vue peut être identifiée exactement par les clés, ce qui permet de reporter les suppressions.

Des structures annexes associées à une vue peuvent aussi être maintenues [Colby96, Colby97]. Les reports sur la vue concrète peuvent être différés. La vue devient alors un **cliché** (*snapshot*) [Adiba80]. Beaucoup de SGBD supportent les clichés.

Notion IX.7 : Cliché (*Snapshot*)

Vue concrète d'une base de données mise à jour périodiquement.

La gestion de structures différentielles associées au cliché, par exemple les mises à jour sur chaque relation dont il est dérivé (les ΔRi), peut permettre un accès intégré à la vue concrète à jour et conduit au développement d'algorithmes de mise à jour spécialisés [Jagadish97].

5.3. LE CAS DES AGRÉGATS

Le cas de vues avec agrégats est particulièrement important pour les systèmes décisionnels, comme nous allons le voir ci-dessous. Le problème de l'auto-maintenabilité

des agrégats a été étudié dans [Gray96], qui propose de distinguer trois classes de fonctions : distributives, algébriques, et non régulières (*holistic*).

Les fonctions agrégatives distributives peuvent être calculées en partitionnant leurs entrées en ensembles disjoints, puis en appliquant la fonction à chacun, enfin en agrégeant les résultats partiels pour obtenir le résultat final. Une fonction distributive AGG vérifie donc la propriété :

```
AGG(v1,v2...vn) = AGG(AGG(v1...vi},AGG(vi+1..vn)).
```

Parmi les fonctions standard de calcul d'agrégats, `SUM`, `MIN` et `MAX` sont distributives. La fonction `COUNT`, qui compte le nombre d'éléments sans éliminer les doubles, peut être considérée comme distributive en l'interprétant comme une somme de comptes partiels égaux à 1 pour des singletons ; par contre, `COUNT DISTINCT` n'est pas distributive car se pose le problème des doubles.

Les fonctions agrégatives algébriques peuvent être exprimées comme fonctions algébriques de fonctions distributives. Par exemple, la moyenne `AVG` est une fonction algébrique, car `AVG(v1...vn) = SUM(v1...vn) / COUNT(v1...vn)`. Pour le problème de la maintenance des vues concrètes, les fonctions algébriques peuvent être remplacées par plusieurs fonctions distributives ; par exemple, une colonne `AVG` sera remplacée par deux colonnes `SUM` et `COUNT`.

Les fonctions non régulières ne peuvent être calculées par partitions. Des vues comportant de telles fonctions sont a priori non auto-maintenables.

La notion de vue auto-maintenable s'applique tout particulièrement aux vues résultant d'une agrégation d'une table de base. On parle alors d'**agrégats auto-maintenables**.

Notion IX.8 : Agrégats auto-maintenables (*Self-maintenable aggregate*)

Ensemble d'agrégats pouvant être calculés à partir des anciennes valeurs des fonctions d'agrégats et des mises à jour des données de base servant au calcul de la vue.

Comme pour les vues en général, on peut distinguer l'auto-maintenabilité en insertion et en suppression. [Gray96] a montré que toutes les fonctions d'agrégats distributives sont auto-maintenables en insertion. En général, ce n'est pas le cas en suppression.

Savoir quand éliminer un tuple de la vue pose problème. En introduisant un compteur du nombre de tuples source dans la base pour chaque tuple de la vue (`COUNT(*)`), il est facile de déterminer si un tuple doit être supprimé : on le supprime quand le compteur passe à 0. `COUNT(*)` est toujours auto-maintenable en insertion comme en suppression. Lorsque les valeurs nulles sont interdites dans l'attribut de base, `SUM` et `COUNT(*)` forment un ensemble auto-maintenables : `SUM` est maintenu par ajout ou retrait de la valeur, et `COUNT(*)` par ajout ou retrait de 1 ; `COUNT(*)` permet de savoir quand supprimer le tuple. En présence de valeurs nulles, il devient difficile de maintenir la somme. `MIN` et `MAX` sont aussi des fonctions non auto-maintenable en sup-

pression. En effet, si on supprime la valeur minimale ou maximale, on ne peut recalculer le nouveau minimum ou maximum qu'à partir des valeurs figurant dans la base.

Par exemple, la vue :

```
CREATE CONCRETE VIEW VENTESPFD (NUMPRO,NUMFOU,DATE,COMPTE,QUANTOT)AS
SELECT NUMPRO,NUMFOU,COUNT(*)AS COMPTE,SUM(QUANTITÉ) AS QUANTOT
FROM VENTES
GROUP BY NUMPRO, NUMFOU
```

est composée d'agrégats auto-maintenables en insertion et en suppression, donc auto-maintenables à condition que `Quantité` ne puisse être nulle (c'est-à-dire de valeur inconnue) dans l'entrée d'une suppression : il faut alors aller rechercher sa valeur dans la base pour enlever la véritable quantité à la somme.

Par contre, la vue :

```
CREATE CONCRETE VIEW VENTESPFD (NUMPRO,NUMFOU,DATE,COMPTE,QUANTOT)AS
SELECT NUMPRO,NUMFOU,COUNT(*)AS COMPTE,MIN(QUANTITÉ) AS QUANMIN
FROM VENTES
GROUP BY NUMPRO, NUMFOU
```

est composée d'agrégats auto-maintenables en insertion mais pas en suppression (problème avec le `MIN`).

6. CONCLUSION

La gestion de vues en interrogation est un problème bien compris dans les SGBD relationnels depuis la fin des années 70. Nous avons introduit ci-dessus ses principes de base. Le report des mises à jour des vues sur la base est un problème plus difficile encore. Des solutions pratiques et générales restent à inventer. L'utilisation de déclencheurs sur les vues peut être une solution pour définir les stratégies de report.

La matérialisation des vues a connu une nouvelle activité ces dernières années, avec l'apparition des entrepôts de données. Les vues avec agrégats sont particulièrement importantes pour gérer des données résumées, en particulier le fameux cube de données très utile en décisionnel. Les techniques sont maintenant connues. Il reste à les mettre en œuvre efficacement dans les systèmes, qui gère pour l'instant peu de redondances et souvent des organisations physiques ad hoc pour le multidimensionnel. Ceci n'est plus vrai dans les entrepôts de données, qui utilisent souvent les vues concrètes pour faciliter les calculs d'agrégats [Bello98].

Les vues connaissent aussi un nouveau développement dans le monde objet avec l'apparition de vues orientées objets au-dessus de bases de données relationnelles par exemple. Nous aborderons cet aspect dans le cadre des SGBD objet ou objet-relationnel.

7. BIBLIOGRAPHIE

[Abiteboul91] Abiteboul S., Hull R., Vianu V., *Foundations of Databases*, Addison-Wesley, 1995.

Comme son nom l'indique, ce livre traite des fondements des bases de données relationnelles et objet. Souvent fondé sur une approche logique ou algébrique, il reprend toute la théorie des bases de données, y compris le problème de la mise à jour au travers des vues, parfaitement bien analysé.

[Adiba80] Adiba M., Lindsay B., « Database Snapshots », *Intl. Conf. on Very Large Databases*, IEEE Ed., Montréal, Canada, Sept. 1980.

Cet article introduit la notion de cliché (snapshot) et montre son intérêt, notamment dans les bases de données réparties. Il décrit aussi la réalisation effectuée dans le fameux système R.*

[Adiba81] Adiba M., « Derived Relations : A Unified Mechanism for Views, Snapshots and Distributed Data », *Intl. Conf. on Very Large Databases,* IEEE Ed., Cannes, France, Sept. 1981.

L'auteur généralise les clichés aux relations dérivées, dont il montre l'intérêt dans les bases de données réparties. Il discute aussi les algorithmes de mise à jour.

[Astrahan76] Astrahan M. M. *et al*., « System R : Relational Approach to Database Management », *ACM TODS, V1, N2*, Juin 1976.

L'article de référence sur le fameux System R. Il décrit aussi le mécanisme de concaténation d'arbres implémenté pour la première fois dans ce système.

[Bancilhon81] Bancilhon F., Spyratos N., « Update Semantics and Relational Views », *ACM TODS, V4, N6*, Dec. 1981, p. 557-575.

Un article de référence en matière de mises à jour de vues. Les auteurs posent le problème en termes d'invariance de la vue suite aux mises à jour directes ou répercutées dans la base. Ils montrent qu'une condition suffisante de maintenabilité est la possibilité de définir des stratégies de report à compléments constants.

[Bello98] Bello R.G, Dias K., Downing A., Freenan J., Norcott D., Dun H., Witkowski A., Ziauddin M., « Materialized Views in Oracle », *Intl. Conf. on Very Large Databases,* Morgan & Kauffman Ed., New York, USA, Août 1998, p. 659-664.

Cet article explique la gestion des vues matérialisées dans Oracle 8. Celles-ci sont utilisées pour les entrepôts de données et la réplication. Elles peuvent être rafraîchies à la fin des transactions, sur demande ou périodiquement. Les rafraîchissements par lots sont optimisés. L'optimisation des requêtes prend en compte les vues concrètes à l'aide d'une technique de réécriture basée sur un modèle de coût.

[Bentayeb97] Bentayeb F., Laurent D., « Inversion de l'algèbre relationnelle et mises à jour », *13e Journées Bases de Données Avancées* (BDA 97), Ed. INRIA, Grenoble, 1997, p. 199-218.

Cet article propose une approche déterministe de la mise à jour au travers de vue, fondée sur la notion d'image réciproque. Des algorithmes d'inversion de l'algèbre sont proposés. Une stratégie de report avec stockage éventuel dans la vue concrétisée est étudiée.

[Chamberlin75] Chamberlin D., Gray J., Traiger I., « Views, Authorization and Locking in a Relational Database System », *Proc. National Computer Conf.*, 1975, p. 425-430.

Cet article détaille les mécanismes de vue, d'autorisation et de verrouillage implantée dans System R.

[Colby96] Colby L.S., Griffin T., Libkin L., Mumick I., RTrickey H., « Algorithms for Deferred View Maintenance », *Proc. ACM SIGMOD* Int. Conf. on Management of Data, Montreal, USA, Mai 1996.

[Colby97] Colby L.S., Kawaguchi A., Lieuwen D.F, Mimick I.S., Ross K., « Supporting Multiple View Maintenance Policies », *Proc. ACM SIGMOD* Int. Conf. on Management of Data, Tucson, USA, p. 405-416, Mai 1997.

Ces deux articles explorent différentes stratégies pour la maintenance de vues concrètes. Les auteurs proposent notamment des algorithmes qui garantissent la cohérence dans des cas de reports des mises à jour en temps différés, avec divers niveaux et groupes de vues.

[Dayal82] Dayal U., Bernstein P., « On the Correct Translation of Update Operations on Relational Views », *ACM TODS*, V8, N3, Dept. 1982.

Cet article discute des stratégies de report de mise à jour au travers de vues et définit des critères de correction.

[Fabret94] Fabret F., *Optimisation du calcul incrémental dans les langages de règles pour bases de données*, Thèse, Université de Versailles SQ, Décembre 1994.

Cette thèse présente une caractérisation des vues concrètes auto-maintenables dans les bases de données. Les vues sont définies par projection, jointure, union, différence et restriction. Cette thèse propose aussi des algorithmes afin de maintenir des réseaux de vues pour accélérer le calcul des requêtes dans une BD déductive.

[Gray96] Gray J., Bosworth A., Layman A., Pirahesh H., « Datacube : A relational Aggregation Operator Generalizing Group-by, Cross-tab, and Sub-totals », *IEEE Int. Conf. on Data Engineering*, p. 152-159, 1996.

Cet article introduit un nouvel opérateur d'agrégation, appelé Datacube, afin de calculer des agrégats multidimensionnels. Nous étudierons plus en détail les opérateurs des bases de données multidimensionnelles dans le cadre du chapitre sur les entrepôts de données.

[Gupta95] Gupta M., Mumick I.S., « Maintenance of Materialized Views : Problems, Techniques and Applications », *IEEE Data Engineering Bulletin, special Issue on Materialized Views and data Warehousing*, N18(2), Juin 1995.

Les auteurs introduisent les différents problèmes posés par la gestion de vues concrètes dans une base de données. Ils proposent quelques solutions et montrent quelques applications possibles pour la gestion d'entrepôts de données performants.

[Gupta96] Gupta A., Jagadish H.V., Mumick I.S., « Data Integration Using Self-Maintainable Views », *Intl. Conf. on Extended Database Technology (EDBT),* LCNS N1057, Avignon, France, 1996, p. 140-144.

Cet article définit précisément la notion de vue auto-maintenable. Les auteurs établissent des conditions suffisantes pour qu'une vue selection-projection-jointure soit auto-maintenable.

[Huyn97] Huyn N., « Multiple-View Self-Maintenance in Data Warehousing Environments » *Intl. Conf. on Very Large Databases,* Morgan & Kauffman Ed., Athens, Grèce, Août 1997, p. 26-35.

Cet article étudie le problème de l'auto-maintenabilité d'un ensemble de vues. Il propose des algorithmes qui permettent de tester si un groupe de vues est auto-maintenable par génération de requêtes SQL. De plus, dans le cas où la réponse est positive, les algorithmes génèrent les programmes de mises à jour.

[Jagadish97] Jagadish H.V., Narayan P.P.S., Seshadri S., Sudarshan S., Kanneganti R., « Incremental Organization for Data Recording and Warehousing », *Intl. Conf. on Very Large Databases*, Morgan & Kauffman Ed., Athens, Greece, Août 1997, p. 16-25.

Les auteurs proposent deux techniques, la première basée sur des séquences triées insérées dans un B-tree, la seconde sur une organisation par hachage, pour stocker les enregistrements lors de leur arrivée dans un entrepôt de données, avant leur intégration dans les vues matérialisées. Ces techniques précisément évaluées supportent des environnements concurrents avec reprises.

[Keller87] Keller M.A., « Comments on Bancilhon and Spyratos's Update Semantics and Relational Views », *ACM TODS*, V12, N3, Sept. 1987, p. 521-523.

Cet article montre que la condition suffisante de maintenabilité qu'est la possibilité de définir des stratégies de report à compléments constants est trop forte, et qu'elle peut conduire à interdire des stratégies intéressantes.

[Kerhervé86] Kerhervé B., « Vues Relationnelles : Implémentation dans un SGBD centralisé et distribué », *Thèse de doctorat, Université de Paris VI*, Mars 1986.

Cette thèse décrit l'implémentation de vues réalisée dans le SGBD SABRE à l'INRIA. Un mécanisme de gestion de vues concrètes par manipulation de compteurs au niveau des opérateurs relationnels est aussi proposé.

[Nicolas83] Nicolas J-M., Yazdanian K., « An Outline of BDGen : A Deductive DBMS », *Woldwide IFIP Congress*, Paris, 1983.

Cet article introduit le système déductif BDGen réalisé au CERT à Toulouse. Il maintient des vues définies par des règles en utilisant des compteurs de raisons de présence d'un tuple.

[Stonebraker74] Stonebraker M.R., « A Functional View of Data Independance », *ACM SIGMOD Workshop on Data Description, Access and Control*, ACM Ed., mai 1974.

Un des premiers articles de Mike Stonebraker, l'un des pères du système INGRES. Il plaide ici pour l'introduction de vues assurant l'indépendance logique.

[Stonebraker75] Stonebraker M., « Implémentation of Integrity Constraints and Views by Query Modification », *Proc. ACM-SIGMOD* Intl. Conf. On Management of data, San José, CA 1975.

Cet article propose de modifier les questions au niveau du source par la définition de vues pour répondre aux requêtes. La technique est formalisée avec le langage QUEL d'INGRES, où elle a été implémentée.

Chapitre X

OPTIMISATION DE QUESTIONS

1. INTRODUCTION

La plupart des SGBD relationnels offrent aujourd'hui des langages de manipulation basés sur SQL, non procéduraux et utilisant des opérateurs ensemblistes. Avec de tels langages, l'utilisateur définit les données qu'il veut visualiser sans fournir les algorithmes d'accès aux données. Le but des algorithmes d'optimisation de questions est justement de déterminer les algorithmes d'accès. Ces algorithmes sont aussi utiles pour les mises à jour, car une mise à jour est une question suivie d'un remplacement. Plus précisément, il s'agit d'élaborer un programme d'accès composé d'opérations de bas niveau attachées à des algorithmes efficaces de recherche dans les tables. Il est essentiel pour un système de déterminer des plans d'accès optimisés – ou au moins proches de l'optimum – pour les questions les plus fréquentes. Ce n'est pas là un problème facile, comme nous allons le voir dans ce chapitre.

Depuis les années 1975, l'optimisation de requêtes dans les bases de données relationnelles a reçu une attention considérable [Jarke84, Kim85, Graefe93]. Les systèmes ont beaucoup progressé. Alors que les premiers étaient très lents, capables seulement d'exécuter quelques requêtes par seconde sur les bases de données du benchmark TPC/A ou B [Gray91], ils supportent aujourd'hui des milliers de transactions par seconde. Bien sûr, cela n'est pas dû seulement à l'optimiseur, mais aussi aux progrès du matériel (les temps d'entrée-sortie disque restent cependant de l'ordre de la dizaine de millisecondes) et des méthodes d'accès. L'optimiseur est donc un des composants

essentiels du SGBD relationnel ; avec les nouveaux SGBD objet-relationnel ou objet, il devient encore plus complexe. Dans ce chapitre, nous nous consacrons au cas relationnel. Nous étudierons plus loin dans cet ouvrage le cas des nouveaux SGBD, où l'optimiseur doit être extensible, c'est-à-dire capable de supporter des opérateurs non prévus au départ.

Classiquement, un optimiseur transforme une requête exprimée dans un langage source (SQL) en un **plan d'exécution** composé d'une séquence d'opérations de bas niveau réalisant efficacement l'accès aux données. Le langage cible est donc constitué d'opérateurs de bas niveau, souvent dérivés de l'algèbre relationnelle complétée par des informations de niveau physique, permettant de déterminer l'algorithme choisi pour exécuter les opérateurs [Selinger79]. Ces informations associées à chaque opérateur, appelées **annotations**, dépendent bien sûr du schéma interne de la base (par exemple, de l'existence d'un index) et de la taille des tables. Elles complètent utilement l'algèbre pour choisir les meilleurs chemins d'accès aux données recherchées.

Au-delà de l'analyse de la question, l'optimisation de requêtes est souvent divisée en deux phases : l'**optimisation logique**, qui permet de réécrire la requête sous une forme canonique simplifiée et logiquement optimisée, c'est-à-dire sans prendre en compte les coûts d'accès aux données, et l'**optimisation physique**, qui effectue le choix des meilleurs algorithmes pour les opérateurs de bas niveau compte tenu de la taille des données et des chemins d'accès disponibles. L'optimisation logique reste au niveau de l'algèbre relationnelle, alors que l'optimisation physique permet en particulier d'élaborer les annotations. Les optimisations logique et physique ne sont malheureusement pas indépendantes ; les deux peuvent en effet changer l'ordre des opérateurs dans le plan d'exécution et modifier le coût du meilleur plan retenu.

Ce chapitre présente d'abord plus précisément les objectifs de l'optimisation et introduit les éléments de base. La section 3 est consacrée à l'étude des principales méthodes d'optimisation logique ; celles-ci recherchent en général la question la plus simple équivalente à la question posée et utilise une heuristique de restructuration algébrique pour élaborer une forme canonique. Nous abordons ensuite l'étude des différents algorithmes d'accès physique aux données : sélection, tri, jointure et calcul d'agrégats. Les variantes sans index et avec index des algorithmes sont discutées. Pour chaque algorithme, un coût approché est calculé en nombre d'entrées-sorties. En effet, l'optimisation physique nécessite un modèle de coût pour estimer le coût de chaque plan d'exécution afin de choisir le meilleur, ou au moins un proche du meilleur. Un modèle de coût complet est décrit au paragraphe 5. Enfin, le paragraphe 6 résume les stratégies essentielles permettant de retrouver un plan d'exécution proche de l'optimal. En conclusion, nous discutons des problèmes plus avancés en matière d'optimisation relationnelle, liés au parallélisme et à la répartition des données.

2. LES OBJECTIFS DE L'OPTIMISATION

Cette section propose deux exemples de bases de données, situe les objectifs de l'optimisation, et définit les concepts de base essentiels.

2.1. EXEMPLES DE BASES ET REQUÊTES

À titre d'illustration, nous utiliserons une extension de notre base favorite des buveurs, composée de cinq relations :

BUVEURS (NB, NOM, PRÉNOM, TYPE)
VINS (NV, CRU, MILLESIME, DEGRÉ)
ABUS (NB, NV, DATE, QUANTITÉ)
PRODUCTEURS (NP, NOM, REGION)
PRODUIT (NV, NP)

Les tables BUVEURS, VINS et PRODUCTEURS correspondent à des entités alors que ABUS et PRODUIT sont des tables représentant des associations. La table BUVEURS décrit des buveurs caractérisés par un numéro, un nom et un prénom. La table VINS contient des vins caractérisés par un numéro, un cru, un millésime et un degré. Les abus commis par les buveurs associant un numéro de buveur, un numéro de vin bu, une date et une quantité bue, sont enregistrés dans la table ABUS. La table PRODUCTEURS décrit des producteurs caractérisés par un numéro, un nom et une région. L'association entre un vin et le producteur qui l'a produit est mémorisée dans la table PRODUIT. On considérera par exemple la question « Quels sont les crus des vins produits par un producteur bordelais en 1976 ayant un degré inférieur ou égal à 14 ? ». Celle-ci s'écrit comme suit en SQL :

```
(Q1)  SELECT V.CRU
      FROM PRODUCTEURS P, VINS V, PRODUIT R
      WHERE V.MILLESIME = "1976" AND V.DEGRE ≤ 14
      AND P.REGION = "BORDELAIS" AND P.NP = R.NP
      AND R.NV = V.NV.
```

Pour diversifier un peu les exemples, nous considérerons aussi parfois la base de données dérivée du banc d'essai (*benchmark*) TPC-D [TPC95]. Ce banc d'essai est centré sur l'aide à la décision et comporte donc beaucoup de questions avec des agrégats sur une base plutôt complexe. C'est dans de telles conditions que l'optimisation de questions devient une fonction très importante. Le schéma simplifié de la base est le suivant :

COMMANDES(NUMCO, NUMCLI, ETAT, PRIX, DATE, PRIORITÉ, RESPONSABLE, COMMENTAIRE)

LIGNES(NUMCO, NUMLIGNE, NUMPRO, FOURNISSEUR, QUANTITÉ, PRIX, REMISE, TAXE, ÉTAT, DATELIVRAISON, MODELIVRAISON, INSTRUCTIONS, COMMENTAIRE)

PRODUITS(NUMPRO, NOM, FABRIQUANT, MARQUE, TYPE, FORME, CONTAINER, PRIX, COMMENTAIRE)

FOURNISSEURS(NUMFOU, NOM, ADRESSE, NUMPAYS, TELEPHONE, COMMENTAIRE)

PRODFOURN(NUMPRO, NUMFOU, DISPONIBLE, COUTLIVRAISON, COMMENTAIRE)

```
CLIENTS(NUMCLI, NOM, ADRESSE, NUMPAYS, TELEPHONE, SEGMENT, COMMENTAIRE)
PAYS(NUMPAYS, NOM, NUMCONT, COMMENTAIRE)
CONTINENTS(NUMCONT, NOM, COMMENTAIRE)
```

Cette base décrit des commandes composées d'un numéro absolu, d'un numéro de client ayant passé la commande, d'un état, d'un prix, d'une date, d'une priorité, d'un nom de responsable et d'un commentaire textuel de taille variable. Chaque ligne de commande est décrite par un tuple dans la table LIGNES. Celle-ci contient le numéro de la commande référencée, le numéro de la ligne et le numéro de produit qui référence un produit de la table PRODUITS. Les fournisseurs et les clients sont décrits par les attributs indiqués dont la signification est évidente, à l'exception de l'attribut SEGMENT qui précise le type de client. La table associative PROFOURN relie chaque fournisseur aux produits qu'il est capable de fournir ; elle contient pour chaque couple possible la quantité disponible, le coût de la livraison par unité, et un commentaire libre. Les tables PAYS et CONTINENTS sont liées entre elles par le numéro de continent ; elles permettent de connaître les pays des fournisseurs et des clients, ainsi que le continent d'appartenance de chaque pays.

La question suivante est une requête décisionnelle extraite de la vingtaine de requêtes du banc d'essai. Elle calcule les recettes réalisées par des ventes de fournisseurs à des clients appartenant au même pays, c'est-à-dire les recettes résultant de ventes internes à un pays, et ceci pour tous les pays d'Europe pendant une année commençant à la date D1.

```
(Q2)  SELECT   P.NOM, SUM(L.PRIX * (1-L.REMISE* QUANTITÉ) AS RECETTE
      FROM     CLIENTS C, COMMANDES O, LIGNES L, FOURNISSEURS F, PAYS P,
                  CONTINENTS T
      WHERE    C.NUMCLI = O.NUMCLI
      AND      O.NUMCOM = L.NUMCO
      AND      L.NUMFOU = F.NUMFOU
      AND      C.NUMPAYS = F.NUMPAYS
      AND      F.NUMPAYS = P.NUMPAYS
      AND      P.NUMCONT = T.NUMCONT
      AND      T.NOM = "EUROPE"
      AND      O.DATE ≥ $D1
      AND      O.DATE < $D1 + INTERVAL 1 YEAR
      GROUP BY P.NOM
      ORDER BY P.NOM, RECETTE DESC ;
```

Il s'agit là d'une requête complexe contenant six jointures, trois restrictions dont deux paramétrées par une date ($D1), un agrégat et un tri ! De telles requêtes sont courantes dans les environnements décisionnels.

2.2. REQUÊTES STATIQUES OU DYNAMIQUES

Certaines requêtes sont figées lors de la phase de développement et imbriquées dans des programmes d'application, donc exécutées aussi souvent que le programme qui

les contient. D'autres au contraire sont construites dynamiquement, par exemple saisies une seule fois pour un test ou une recherche *ad hoc*. Pour l'optimiseur, il est important de distinguer ces différents types de requêtes. Les premières répétitives sont appelées **requêtes statiques**, alors que les secondes sont qualifiées de **requêtes dynamiques**.

Notion X.1 : Requête statique (*Static Query*)

Requête SQL généralement intégrée à un programme d'application dont le code SQL est connu à l'avance et fixé, souvent exécutée plusieurs fois.

Ce premier type de requête gagne à être optimisé au mieux. En effet, l'optimisation est effectuée une seule fois, lors de la compilation du programme contenant la requête, pour des milliers voire des millions d'exécutions. La requête SQL peut être paramétrée, les valeurs constantes étant passées par des variables de programme. L'optimiseur doit donc être capable d'optimiser des requêtes paramétrées, par exemple contenant un prédicat `CRU = $x`, ou x est une variable de programme.

Notion X.2 : Requête dynamique (*Dynamic Query*)

Requête SQL généralement composée en interactif dont le code n'est pas connu à l'avance, souvent exécutée une seule fois.

Ce deuxième type de requêtes, aussi appelé requêtes *ad hoc* car correspondant à des requêtes souvent saisies en ligne sans une longue réflexion préalable, est exécuté une seule fois. On doit donc limiter le temps d'optimisation afin de ne pas trop pénaliser l'unique exécution de la requête.

2.3. ANALYSE DES REQUÊTES

Dans un premier temps, une question est analysée du point de vue syntaxique. L'existence des noms de relations et d'attributs cités est vérifiée par rapport au schéma. La correction de la qualification de la question est analysée. Aussi, la requête est mise dans une forme canonique. Par exemple, celle-ci peut être mise en forme normale conjonctive (`ET` de `OU`) ou disjonctive (`OU` de `ET`), selon qu'elle sera ultérieurement traitée par des opérateurs élémentaires supportant le `OU` (forme normale conjonctive), ou simplement comme plusieurs questions (forme normale disjonctive). Souvent, cette transformation est accomplie au niveau suivant. Finalement, ce premier traitement est analogue à l'analyse syntaxique d'expressions dans un langage de programmation.

À partir de la requête analysée, la plupart des systèmes génère un arbre d'opérations de l'algèbre relationnelle (projection, restriction, jointure, union, différence, intersec-

tion), éventuellement étendu avec des agrégats et des tris, appelé **arbre algébrique**, ou **arbre relationnel,** ou parfois aussi **arbre de traitement (***processing tree***).**

Notion X.3 : Arbre algébrique (*Algebraic tree*)

Arbre représentant une question dont les nœuds terminaux représentent les relations, les nœuds intermédiaires des opérations de l'algèbre relationnelle, le nœud racine le résultat d'une question, et les arcs les flux de données entre les opérations.

Dans l'arbre algébrique, chaque nœud est représenté par un graphisme particulier, déjà décrit lors de l'introduction des opérateurs algébriques. Nous rappelons les notations figure X.1, en introduisant une notation particulière pour les tris (simplement un rectangle avec le nom des attributs de tri à l'intérieur) et les agrégats, qui peuvent être vus comme un tri suivi d'un groupement avec calculs de fonctions pour chaque monotonie. Les agrégats sont ainsi représentés par un rectangle contenant les attributs de la clause GROUP BY, suivi d'un rectangle contenant les attributs résultats calculés.

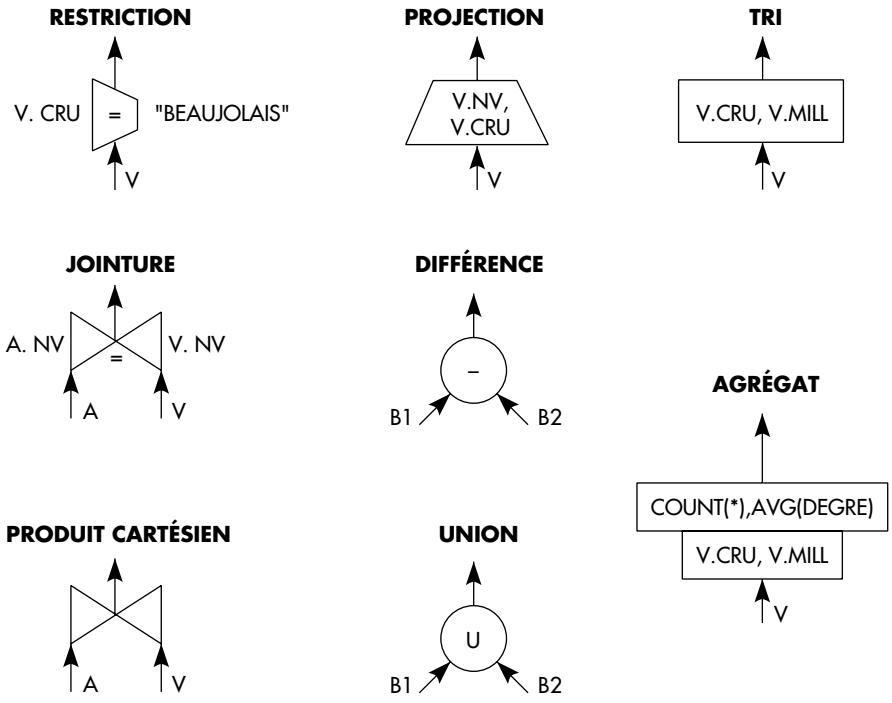


Figure X.1 : Représentation des opérateurs de l'algèbre étendue

Plusieurs arbres représentent une même question, selon l'ordre choisi pour les opérations. Une méthode de génération simple d'un arbre consiste à prendre les prédicats de qualification dans l'ordre où ils apparaissent et à leur associer l'opération relation-

nelle correspondante, puis à ajouter les agrégats, et enfin une projection finale pour obtenir le résultat avec un tri éventuel. Cette méthode permet par exemple de générer l'arbre représenté figure X.2 pour la question Q1, puis l'arbre de la figure X.3 pour la question Q2.

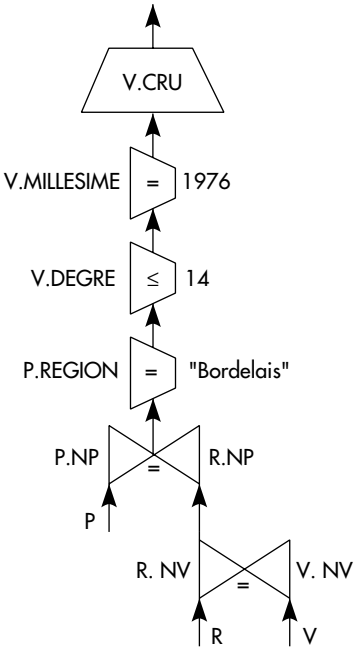


Figure X.2 : Un arbre algébrique pour la question Q1 sur la base des vins

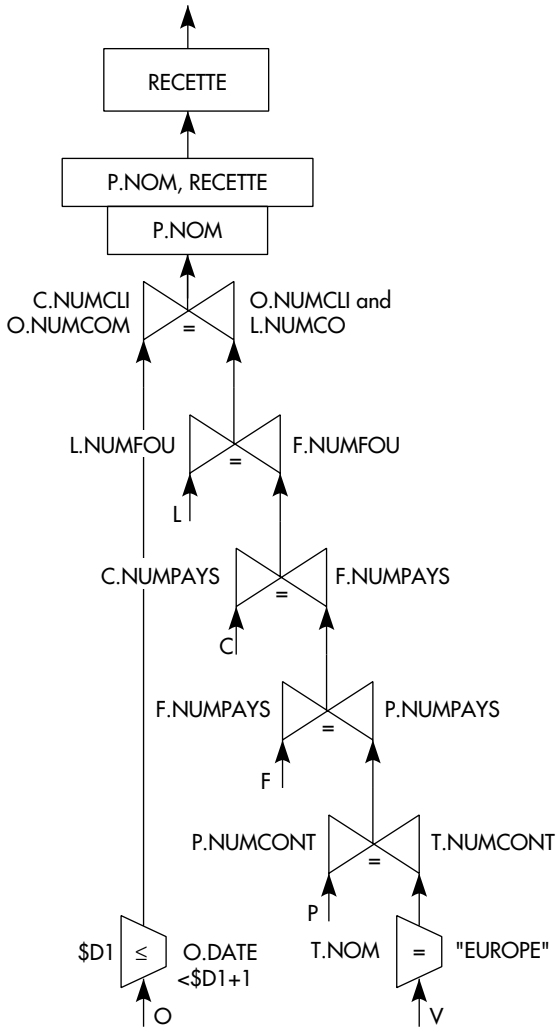


Figure X.3 : Un arbre algébrique pour la question Q2 sur la base TPC/D

À partir d'un arbre algébrique, il est possible de générer un plan d'exécution en parcourant l'arbre, des feuilles vers la racine. Une opération peut être exécutée par ensembles de tuples ou tuple à tuple, dès que ses opérandes sont disponibles. L'**exécu-**

tion ensembliste consiste à attendre la disponibilité des opérandes pour exécuter une opération.

Notion X.4 : Exécution ensembliste (*Set-oriented Execution*)

Mode d'exécution consistant à calculer l'ensemble des tuples des relations en entrée d'un opérateur avant d'évaluer cet opérateur.

Dans ce mode, si l'opération O1 n'utilise pas les résultats de l'opération O2, les opérations O1 et O2 peuvent être exécutées en parallèle. Ce mode d'exécution peut être intéressant pour des algorithmes capables de travailler efficacement sur des ensembles (basés sur le tri par exemple).

Plus souvent, on cherche à démarrer une opération dès qu'un tuple est disponible dans l'une des tables arguments. C'est le mode **tuple à tuple** ou **pipeline**.

Notion X.5 : Exécution pipline (*Pipline execution*)

Mode d'exécution consistant à démarrer une opération le plus tôt possible, si possible dès qu'un tuple est disponible pour au moins un opérande.

Pour les opérations unaires, il suffit qu'un tuple soit disponible. On peut ainsi exécuter deux opérations successives de restriction en pipeline, c'est-à-dire commencer la deuxième dès qu'un tuple (ou un groupe de tuples, par exemple une page) a été générée par la première. Les opérations binaires comme la différence peuvent nécessiter d'avoir à calculer complètement l'un des deux opérandes pour commencer. Le mode pipeline est donc seulement possible sur un des deux opérandes. Ceci peut dépendre de l'algorithme pour la jointure.

2.4. FONCTIONS D'UN OPTIMISEUR

Un optimiseur a donc pour objectif d'élaborer un **plan d'exécution** optimisé.

Notion X.6 : Plan d'exécution (*Execution plan*)

Programme éventuellement parallèle d'opérations élémentaires à exécuter pour évaluer la réponse à une requête.

Ceci s'effectue en général en deux phases, la **réécriture** et le **planning**, encore appelé ordonnancement [Haas89].

Notion X.7 : Réécriture (*Rewriting*)

Phase d'optimisation consistant à transformer logiquement la requête de sorte à obtenir une représentation canonique.

Cette phase comporte un aspect sémantique pouvant aller jusqu'à prendre en compte des règles d'intégrité, et un aspect syntaxique concernant le choix d'une forme canonique. Celle-ci comprend la mise sous forme normale des critères et l'affectation d'un ordre fixé pour les opérateurs algébriques.

La phase suivante est donc le **planning**, qui peut remettre en question certains choix effectués lors de la réécriture.

Notion X.8 : Planning (*Planning*)

Phase d'optimisation consistant à ordonner les opérateurs algébriques et choisir les algorithmes et mode d'exécution.

Alors que la première phase génère un arbre logique, la seconde ajoute les annotations et obtient un plan d'exécution. Elle s'appuie de préférence sur un modèle de coût. Les deux phases ne sont pas indépendantes, la première pouvant influer sur la seconde et biaiser le choix du meilleur plan. Cependant, on les distingue souvent pour simplifier.

On cherche bien sûr à optimiser les temps de réponse, c'est-à-dire à minimiser le temps nécessaire à l'exécution d'un arbre. Le problème est donc de générer un arbre optimal et de choisir les meilleurs algorithmes pour exécuter chaque opérateur et l'arbre dans son ensemble. Pour cela, il faut optimiser simultanément :

– le nombre d'entrées-sorties ;
– le parallélisme entre les opérations ;
– le temps de calcul nécessaire.

La fonction de coût doit si possible prendre en compte la taille des caches mémoires disponibles pour l'exécution. L'optimisation effectuée dépend en particulier de l'ordre des opérations apparaissant dans l'arbre algébrique utilisé et des algorithmes retenus. Il est donc essentiel d'établir des règles permettant de générer, à partir d'un arbre initial, tous les plans possibles afin de pouvoir ensuite choisir celui conduisant au coût minimal. En fait, le nombre d'arbres étant très grand, on est amené à définir des heuristiques pour déterminer un arbre proche de l'optimum.

Nous allons maintenant étudier les principales méthodes proposées pour accomplir tout d'abord la réécriture, puis le planning. Leur objectif essentiel est évidemment d'optimiser les temps de réponse aux requêtes.

3. L'OPTIMISATION LOGIQUE OU RÉÉCRITURE

La réécriture permet d'obtenir une représentation canonique de la requête, sous la forme d'un arbre algébrique dans lequel les opérations sont ordonnées et les critères mis sous forme normale. Elle peut comporter elle-même deux étapes : la **réécriture sémantique** qui transforme la question en prenant en compte les connaissances sémantiques sur les données (par exemple, les contraintes d'intégrité), et la **réécriture syntaxique** qui profite des propriétés simples de l'algèbre relationnelle pour ordonner les opérations.

3.1. ANALYSE ET RÉÉCRITURE SÉMANTIQUE

Ce type d'analyse a plusieurs objectifs, tels que la détermination de la correction de la question, la recherche de questions équivalentes par manipulation de la qualification ou à l'aide des contraintes d'intégrité. Ce dernier type d'optimisation est d'ailleurs rarement réalisé dans les systèmes. On peut aussi inclure dans la réécriture les modifications de questions nécessaires pour prendre en compte les vues, problème étudié dans le chapitre traitant des vues.

3.1.1. Graphes support de l'analyse

Plusieurs types de graphes sont utilisés pour effectuer l'analyse sémantique d'une question. Tout d'abord, le **graphe de connexion des relations** a été introduit dans INGRES [Wong76].

Notion X.9 : Graphe de connexion des relations (*Relation connection graph*)

Graphe dans lequel: (a) un sommet est associé à chaque occurrence de relation, (b) une jointure est représentée par un arc entre les deux nœuds représentant les relations jointes, (c) une restriction est représentée par une boucle sur la relation à laquelle elle s'applique, (d) la projection finale est représentée par un arc d'un nœud relation vers un nœud singulier résultat.

Avec SQL, chaque instance de relation dans la clause FROM correspond à un nœud. Les arcs sont valués par la condition qu'ils représentent. Ainsi, le graphe de connexion des relations de la question Q1 est représenté figure X.4. Notez la difficulté à représenter des questions avec des disjonctions (ou) de jointures qui peuvent être modélisées par plusieurs graphes.

Plusieurs variantes d'un tel graphe ont été proposées, en particulier le **graphe des jointures** [Berstein79] où seules les jointures sont matérialisées par un arc entre les nœuds relations. Le nœud singulier figurant le résultat, les arcs représentant les pro-

jections finales, et les boucles symbolisant les restrictions sont omis. Ce graphe simplifié peut être utilisé pour diverses manipulations sémantiques sur les jointures, mais aussi pour ordonner les jointures ; il est alors couplé à un modèle de coût.

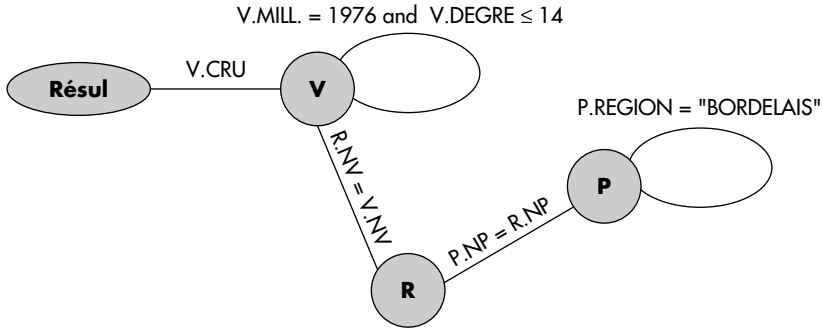


Figure X.4 : Graphe de connexion des relations de la question Q1

Le **graphe de connexion des attributs** peut être défini comme suit [Hevner79].

Notion X.10 : Graphe de connexion des attributs (*Attribute connection graph*)

Graphe dans lequel : (a) un sommet est associé à chaque référence d'attribut ou de constante, (b) une jointure est représentée par un arc entre les attributs participants, (c) une restriction est représentée par un arc entre un attribut et une constante.

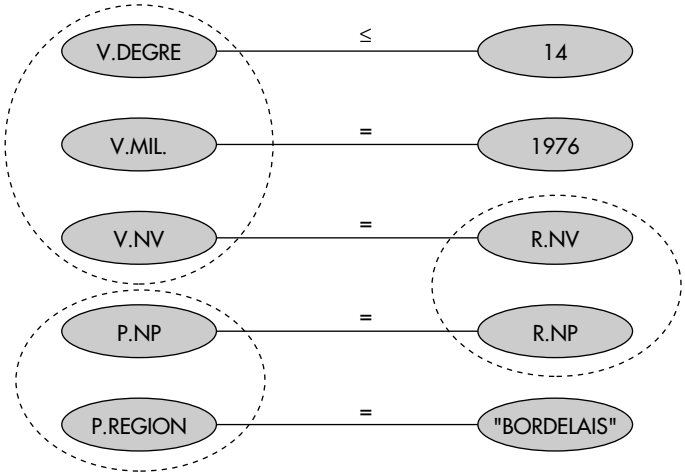


Figure X.5 : Graphe de connexion des attributs de la question Q1

La figure X.5 représente le graphe de connexion des attributs de la question Q1. Notez que ce graphe perd la notion de tuple, chaque attribut étant représenté individuellement.

Il est possible d'introduire des hyper-nœuds (c'est-à-dire des groupes de nœuds) pour visualiser les attributs appartenant à un même tuple, comme représentés figure X.5. Un tel graphe permet de découvrir les critères contenant une contradiction : il possède alors un cycle qui ne peut être satisfait par des constantes. Il peut aussi permettre de découvrir des questions équivalentes à la question posée par transitivité (voir ci-dessous).

3.1.2. Correction de la question

La notion de **question correcte** est à préciser. Deux catégories d'incorrection sont à distinguer :

1. Une question peut être **mal formulée** car certaines parties apparaissent inutiles dans la question ; c'est sans doute que l'utilisateur a oublié une jointure dans la question.
2. Une question peut présenter une qualification **contradictoire**, qui ne peut être satisfaite par aucun tuple ; ainsi par exemple la question « crus des vins de degré supérieur à 14 et inférieur à 12 ».

Deux résultats importants ont été établis afin d'éliminer les questions incorrectes, dans le cas des questions conjonctives (sans ou) avec des opérateurs de comparaisons $=, <, >, \leq, \geq$:

1. Une question est généralement **mal formulée** si son graphe de connexion des relations n'est pas connexe [Wong76]. En effet, tout sous-graphe non relié au nœud résultat ne participe pas à ce résultat. Il peut cependant s'agir d'un filtre qui rend la réponse vide si aucun tuple de la base ne le satisfait.
2. Une question est **contradictoire** si son graphe de connexion des attributs complété par des arcs de comparaison entre constantes présente un cycle non satisfiable [Rosenkrantz80]. En effet, dans ce cas aucun tuple ne peut satisfaire les prédicats du cycle, par exemple du style AGE < 40 et AGE >50.

3.1.3. Questions équivalentes par transitivité

La notion de **questions équivalentes** mérite une définition plus précise [Aho79].

Notion X.11 : Questions équivalentes (*Equivalent queries*)

Deux questions sont équivalentes si et seulement si elles donnent le même résultat pour toute extension possible de la base de données.

Ainsi, quels que soient les tuples dans la base (obéissant évidemment aux contraintes d'intégrité), deux questions équivalentes exécutées au même instant donneront le même résultat.

Tout d'abord, des questions équivalentes peuvent être générées par l'étude des propriétés de transitivité du graphe de connexion des attributs. Si l'on considère ce der-

nier, tout couple d'attributs (x, y) reliés par un chemin x –> y dont les arcs sont étiquetés par des égalités doit vérifier x = y. Il est alors possible de générer la fermeture transitive de ce graphe. Tous les sous-graphes ayant même fermeture transitive génèrent des questions équivalentes, bien qu'exprimées différemment. Ils correspondent en effet à des contraintes équivalentes sur le contenu de la base. Par exemple, considérons la question Q3 : « Noms des buveurs ayant bu un Bordeaux de degré supérieur ou égal à 14 ». Elle peut s'exprimer comme suit :

```
(Q3)  SELECT B.NOM
      FROM BUVEURS B, ABUS A, PRODUIT R, PRODUCTEURS P, VINS V
      WHERE B.NB = A.NB AND A.NV = R.NV AND R.NV = V.NV
      AND R.NP = P.NP AND P.REGION = "BORDELAIS"
      AND V.DEGRE ≥ 14 ;
```

où B, A, R, P, V désignent respectivement les relations BUVEURS, ABUS, PRODUIT, PRODUCTEURS et VINS. Le graphe de connexion des attributs réduit aux jointures de Q3 est représenté figure X.6. Sa fermeture transitive apparaît en pointillés. Un graphe ayant même fermeture transitive est représenté figure X.7. Ainsi la question Q3 peut être posée par la formulation résultant du graphe de la figure X.7 avec les mêmes variables, comme suit :

```
(Q'3) SELECT B.NOM
      FROM BUVEURS B, ABUS A, PRODUIT R, PRODUCTEURS P, VINS V
      WHERE B.NB = A.NB AND A.NV = R.NV AND A.NV = V.NV
      AND R.NP = P.NP AND P.REGION = "BORDELAIS"
      AND V.DEGRE ≥14 ;
```

La différence réside dans le choix des jointures, le prédicat R.NV = V.NV étant remplacé par A.NV = V.NV. D'autres expressions sont possibles. Le problème consiste bien sûr à choisir celle qui pourra être évaluée le plus rapidement. Le problème est plus difficile si l'on considère des inégalités, mais il peut être résolu en étendant le graphe de connexion des attributs [Rosenkrantz80].

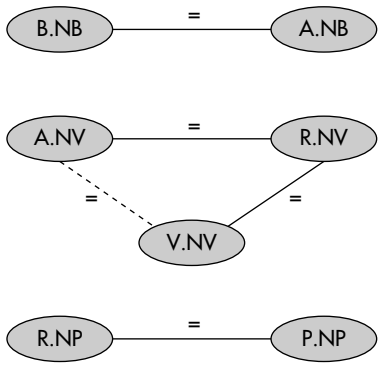


Figure X.6 : Graphe de connexion des attributs de jointure de la question Q3

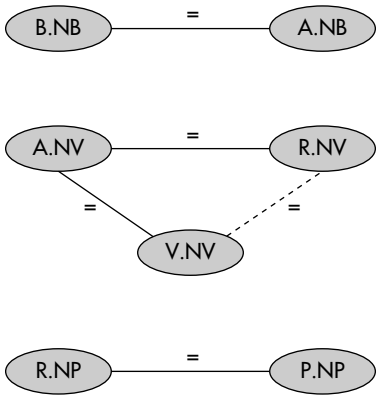


Figure X.7 : Graphe ayant même fermeture transitive

3.1.4. Questions équivalentes par intégrité

Une autre manière de générer des questions équivalentes consiste à utiliser les contraintes d'intégrité [Chakravarthy90, King81, Finance94]. Le problème peut être posé comme suit. Étant donnée une question de qualification Q et un ensemble de contraintes d'intégrité I1, I2 … In, si Q est contradictoire à une contrainte, la question a une réponse vide. Sinon, il suffit d'évaluer la « meilleure » qualification Q′ impliquant Q sous les contraintes I1, I2... In, c'est-à-dire telle que I1 ∧ I2 ∧ …In ∧ Q′ ⇒ Q.

Le problème est un problème de déduction que nous illustrerons simplement par un exemple. Soit la contrainte exprimant que tous les Bordeaux sont de degré supérieur à 15 (où P désigne un tuple de PRODUCTEURS, R un tuple de PRODUIT et V désigne un tuple de VINS) :

```
P.REGION = "BORDELAIS" AND P.NP = R.NP AND R.NV = V.NV
AND V.DEGRE > 15.
```

Alors, la qualification de la question Q1 est contradictoire avec cette contrainte et par conséquence sa réponse est vide. En revanche, cette contrainte permet de simplifier la question Q3 car la condition V.DEGRE ≥ 14 est inutile et peut donc être supprimée. Il n'est pas sûr que cette simplification réduise le temps d'exécution.

L'optimisation sémantique a été particulièrement développée dernièrement dans le contexte des bases de données objet, où elle peut conduire à des optimisations importantes. Les contraintes d'équivalence, d'implication, d'unicité de clé et d'inclusion sont particulièrement utiles pour transformer et simplifier les requêtes.

3.2. RÉÉCRITURES SYNTAXIQUES ET RESTRUCTURATIONS ALGÉBRIQUES

Nous introduisons ici une technique de base pour transformer les arbres algébriques et changer l'ordre des opérations. Cette technique résulte des propriétés de commutativité et d'associativité des opérateurs de l'algèbre. Elle provient d'une traduction dans le contexte de l'algèbre relationnelle des techniques de réécriture des expressions arithmétiques. Aujourd'hui, la plupart des optimiseurs mixent les restructurations algébriques avec la phase de planning, basée sur un modèle de coût, que nous décrivons plus loin. Cependant, pour la commodité de la présentation, nous introduisons ci-dessous les règles de transformation des arbres, puis un algorithme simple de restructuration algébrique, et aussi des heuristiques plus complexes de décomposition appliquées parfois par certains optimiseurs [Stonebraker76, Gardarin84].

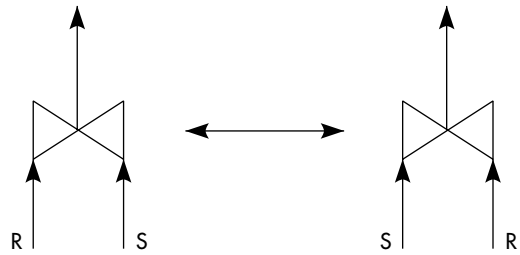
3.2.1. Règles de transformation des arbres

Les règles suivantes ont pour la première fois été précisées dans [Smith75]. Elles sont bien développées dans [Ullman88]. Elles dépendent évidemment des opérations relationnelles prises en compte. Nous considérons ici l'ensemble des opérations de restriction, jointure, union, intersection et différence. Nous représentons figure X.8 neuf règles sous forme de figures donnant deux patrons d'arbres équivalents. Ces règles sont très classiques. Ce sont les suivantes :

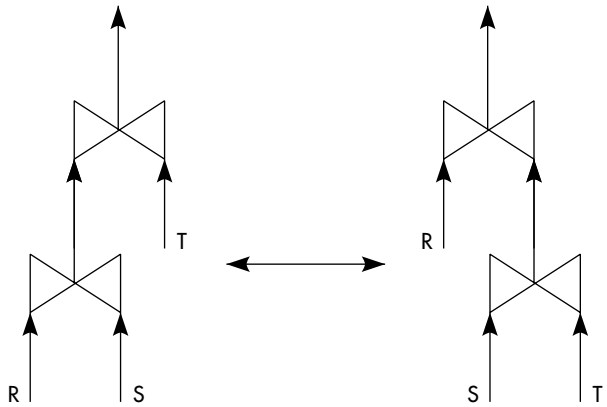
1. commutativité des jointures ;
2. associativité des jointures ;
3. fusion des projections ;
4. regroupement des restrictions ;
5. quasi-commutativité des restrictions et des projections ;
6. quasi-commutativité des restrictions et des jointures ;
7. commutativité des restrictions avec les unions, intersections ou différences ;
8. quasi-commutativité des projections et des jointures ;
9. commutativité des projections avec les unions.

Notez que dans toutes les règles où figurent des jointures, celles-ci peuvent être remplacées par des produits cartésiens qui obéissent aux mêmes règles. En effet, le produit cartésien n'est jamais qu'une jointure sans critère. La quasi-commutativité signifie qu'il y a en général commutativité, mais que quelques précautions sont nécessaires pour éviter par exemple la perte d'attributs, ou qu'une opération commutée ne soit effectuée que sur l'une des relations.

1. Commutativité des jointures



2. Associativité des jointures



3. Fusion des projections

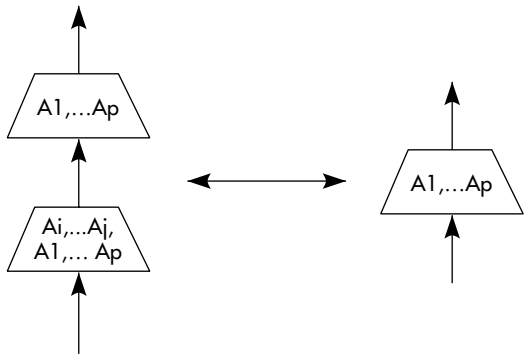
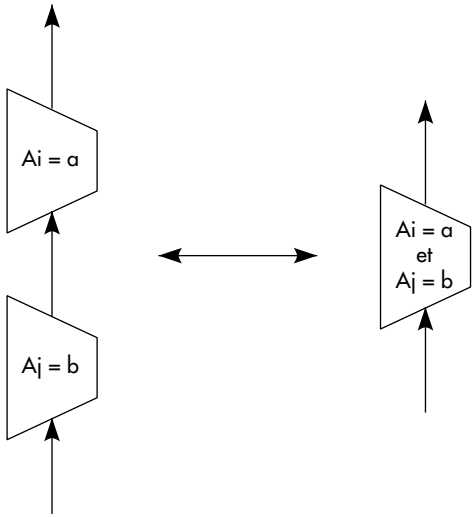


Figure X.8 (début) : Règles de restructurations algébriques

4. Regroupement des restrictions



5. Quasi-commutativité des restrictions et projections (l'attribut de restriction doit être conservé par la projection)

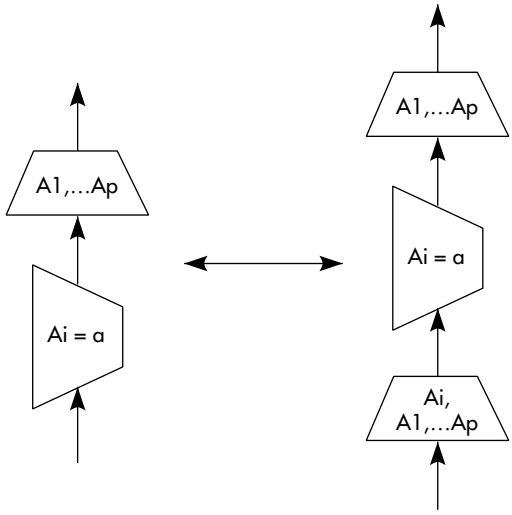
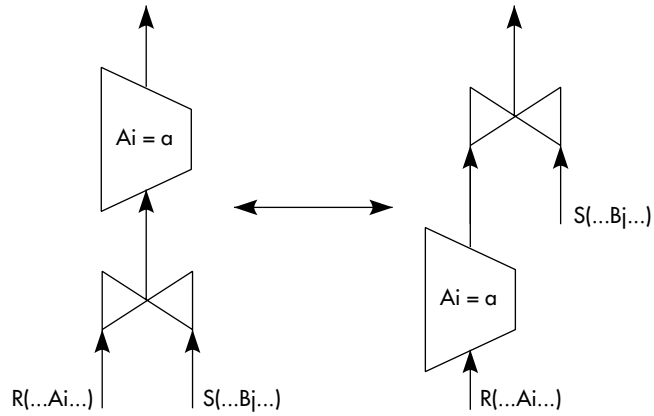


Figure X.8 (suite) : Règles de restructurations algébriques

6. Quasi-commutativité des restrictions et des jointures (la restriction est appliquée sur la relation contenant l'attribut restreint)



7. Commutativité des restrictions avec les unions, intersections ou différences

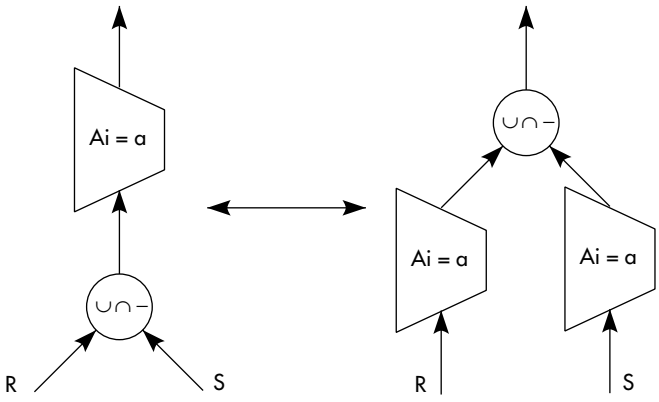
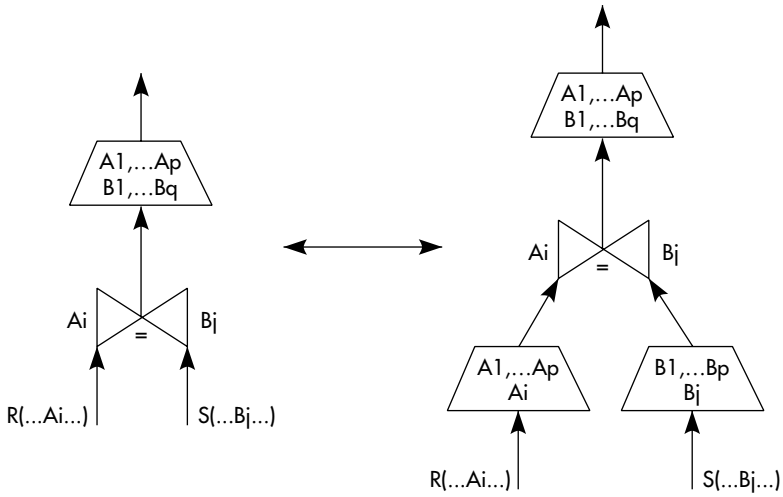


Figure X.8 (suite) : Règles de restructurations algébriques

8. Quasi-commutativité des projections et jointures (la projection ne doit pas perdre les attributs de jointure)



9. Commutativité des projections avec les unions

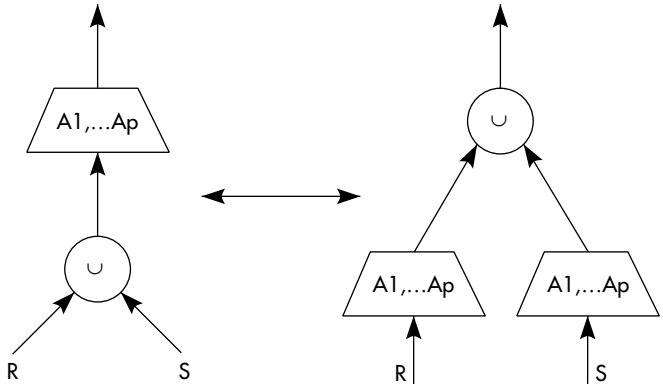


Figure X.8 (fin) : Règles de restructurations algébriques

3.2.2. Ordonnancement par descente des opérations unaires

Une première optimisation simple consiste à exécuter tout d'abord les opérations unaires (restriction, projection), puis les opérations binaires (jointure, produit cartésien, union, intersection, différence). En effet, les opérations unaires sont des réducteurs de la taille des relations, alors qu'il n'en est pas ainsi de certaines opérations

binaires qui ont tendance à accroître la taille des résultats par rapport aux relations arguments. Par exemple, une jointure peut générer une très grande relation. Ceci se produit lorsque de nombreux tuples de la première relation se compose à de nombreux de la seconde. À la limite, elle peut se comporter comme un produit cartésien si tous les couples de tuples des deux relations vérifient le critère de jointure.

Aussi, afin de ne considérer que les arbres à flux de données minimal et de réduire ainsi le nombre d'entrées-sorties à effectuer, on est conduit à descendre les restrictions et projections. De plus, quand deux projections successives portent sur une même relation, il est nécessaire de les regrouper afin d'éviter un double accès à la relation. De même pour les restrictions. Ces principes précédents conduisent à l'algorithme d'optimisaton suivant :

1. Séparer les restrictions comportant plusieurs prédicats à l'aide de la règle 4 appliquée de la droite vers la gauche.
2. Descendre les restrictions aussi bas que possible à l'aide des règles 5, 6, et 7.
3. Regrouper les restrictions successives portant sur une même relation à l'aide de la règle 4 appliquée cette fois de la gauche vers la droite.
4. Descendre les projections aussi bas que possible à l'aide des règles 8 et 9.
5. Regrouper les projections successives à partir de la règle 3 et éliminer d'éventuelles projections inutiles qui auraient pu apparaître (projection sur tous les attributs d'une relation).

Pour simplifier les arbres et se rapprocher des opérateurs physiques réellement implémentés dans les systèmes, une restriction suivie par une projection est notée par un unique **opérateur de sélection** (restriction + projection) ; celui-ci permet d'éviter deux passes sur une même relation pour faire d'abord la restriction puis la projection. Il en est de même pour une jointure suivie par une projection : on parle alors d'opérateur de **jointure-projection**.

À titre d'illustration, l'algorithme de descente des restrictions et des projections a été appliqué à l'arbre de la figure X.2. On aboutit à l'arbre représenté figure X.9. Cet arbre n'est pas forcément optimal pour au moins deux raisons : la descente des restrictions et des projections n'est qu'une heuristique et l'ordre des opérations binaires n'a pas été optimisé.

3.3. ORDONNANCEMENT PAR DÉCOMPOSITION

La décomposition est une stratégie d'ordonnancement qui fut appliquée dans le système INGRES [Wong76]. Elle est basée sur deux techniques de transformation de questions appelées **détachement** et **substitution**. Appliqué récursivement, le détachement permet d'ordonner les opérations selon l'ordre sélection, semi-jointure, et enfin jointure. La substitution consiste simplement à évaluer une question portant sur une

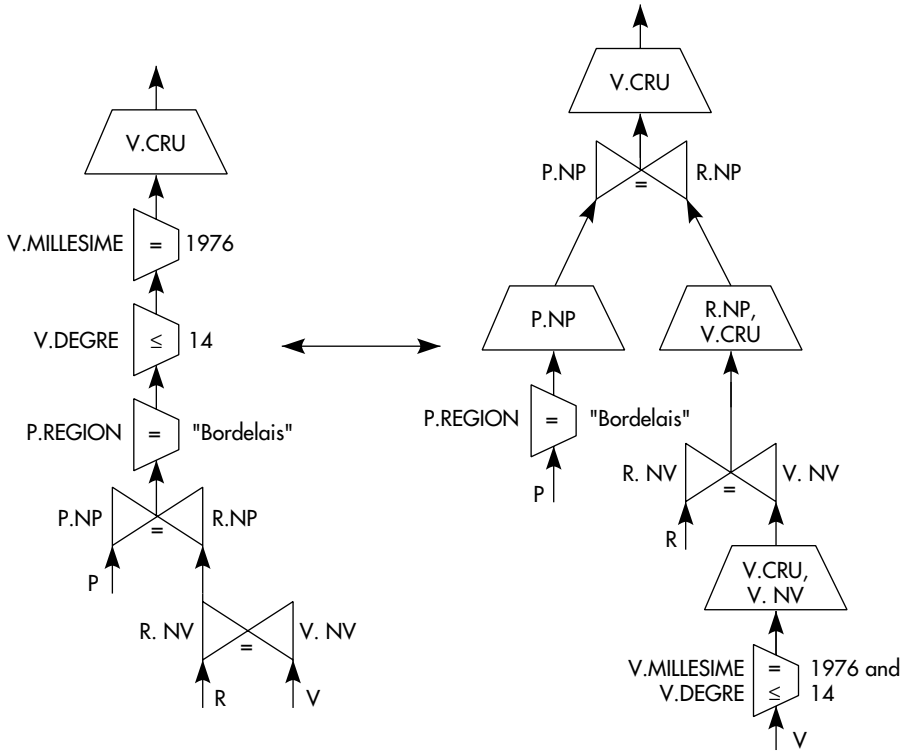


Figure X.9 : Optimisation d'un arbre par descente des restrictions et des projections

relation en l'appliquant sur chacun des tuples de la relation. Cette technique revient donc au balayage et nous n'insisterons pas dessus. Le détachement permet par contre un premier ordonnancement des jointures.

3.3.1. Le détachement de sous-questions

Le détachement permet de décomposer une question `Q` en deux questions successives `Q1` et `Q2`, ayant en commun une variable unique résultat de `Q1`. C'est une transformation de question consistant à diviser une question en deux sous-questions successives ayant une seule table commune. De manière plus formelle, la situation générale où le détachement est possible est la suivante. Soit une question `QT` de la forme :

```
(QT)  SELECT T(X1, X2 …, Xm)
      FROM R1 X1, R2 X2..., Rn Xn
      WHERE B2(X1, X2 …, Xm)
      AND B1(Xm, Xm+1 …, Xn).
```

B1 et B2 sont des qualifications incluant respectivement les variables X_1,X_2… X_m et X_m, X_{m+1}… X_n, alors que T est un résultat de projection à partir des variables X_1, X_2… X_m.

Une telle question peut être décomposée en deux questions, q1 suivie de q2, par détachement :

```
(q1)  SELECT K(Xm) INTO R'm
      FROM RM Xm, Rm+1Xm+1... Rn Xn
      WHERE B1(Xm, Xm+1, … Xn)
```

$K(X_m)$ contient les informations de X_m nécessaires à la deuxième sous-question :

```
(q2)  SELECT T(X1, X2, … Xm)
      FROM R1 X1, R2 X2... R'm Xm
      WHERE B2(X1, X2, … Xm).
```

Le détachement consiste en fait à transformer une question en deux questions imbriquées q1 et q2 qui peuvent être écrites directement en SQL comme suit :

```
SELECT T(X1, X2, … Xm)
FROM R1 X1, R2 X2... Rm X'm
WHERE B2(X1, X2, … Xm) AND K(X'm) IN
   SELECT K(Xm)
   FROM RM Xm, Rm+1 Xm+1... Rn Xn
   WHERE B1(X m, Xm+1, … Xn).
```

Les questions n'ayant pas de variables communes, il est possible d'exécuter la question interne q1, puis la question externe q2.

Une question dans laquelle aucun détachement n'est possible est dite **irréductible**. On peut montrer qu'une question est irréductible lorsque le graphe de connexion des relations ne peut être divisé en deux classes connexes par élimination d'un arc [Wong76].

3.3.2. Différents cas de détachement

Comme indiqué plus haut, certaines techniques d'optimisation de questions cherchent à exécuter tout d'abord les opérations qui réduisent la taille des relations figurant dans la question. Une telle opération correspond souvent à une sélection (restriction suivie de projection), mais parfois aussi à une **semi-jointure** [Bernstein81].

Notion X.12 : Semi-jointure (*Semi-join*)

La semi-jointure de la relation R par la relation S, notée $R \ltimes S$, est la jointure de R et S projetée sur des attributs de R.

Autrement dit, la semi-jointure de R par S est composée de tuples (ou partie de tuples) de R qui joignent avec S. On peut aussi voir la semi-jointure comme une généralisation de la restriction : cette opération restreint les tuples de R par les valeurs qui appa-

raissent dans le (ou les) attribut(s) de jointure de S (par la projection de S sur ce (ou ces) attributs(s)). La semi-jointure R $\ltimes$ S est donc une opération qui réduit la taille de R, tout comme une restriction.

Le détachement permet de faire apparaître d'une part les restrictions, d'autre part les semi-jointures. Ce sont les deux seuls cas de détachement possibles. Nous allons illustrer ces détachements sur la question Q1 déjà vue ci-dessus :

```
SELECT DISTINCT V.CRU
FROM PRODUCTEURS P, VINS V, PRODUIT R
WHERE V.MILLESIME = 1976 AND V.DEGRE ≤ 14
AND P.REGION = "BORDELAIS" AND P.NP = R.NP
AND R.NV = V.NV.
```

Le graphe de connexion des relations de cette question est représenté figure X.4.

Tout d'abord, les deux sélections :

```
(q11) SELECT V.NV INTO V1
      FROMS VINS V
      WHERE V.DEGRE ≤ 14 AND V.MILLESIME = 1976
```

et

```
(q12) SELECT P.NP INTO P1
      FROMS PRODUCTEURS P
      WHERE P.REGION = "BORDELAIS"
```

peuvent être détachées.

En faisant maintenant varier `V` sur la relation `V1` résultat de `Q11` et `P` sur la relation `P1` résultat de `Q12`, il reste à exécuter la question :

```
(q1') SELECT DISTINCT V.CRU
      FROM P1 P, V1 V, PRODUIT R
      WHERE P.NP = R.NP
      AND R.NV = V.NV.
```

L'arc du graphe des relations correspondant à la jointure `P.NP = R.NP` peut ensuite être enlevé, ce qui correspond au détachement de la sous-question :

```
(q13) SELECT R.NV INTO R1
      FROM P1 P, PRODUIT R
      WHERE P.NP = R.NP.
```

`q13` est bien une semi-jointure. Finalement, en faisant varier `R` sur la relation R1 résultat de `q13`, il ne reste plus qu'à exécuter une dernière semi-jointure :

```
(Q14) SELECT DISTINCT V.CRU
      FROM V1 V, R1 R
      WHERE V.NV = R.NV
```

`Q1` a été décomposée en une suite de sélection et semi-jointures ((`q11 | q12`) ; `q13` ; `q14`). `q11` et `q12` peuvent être exécutées en parallèle, puis `q13` peut l'être, et enfin `q14`.

Toutes les questions ne sont pas ainsi décomposables en sélections suivies par des semi-jointures. Il est toujours possible de détacher les sélections. Par contre [Bernstein81] a montré que seules les questions dont les graphes de connexion après élimination des boucles (détachement des sélections) sont des arbres peuvent être décomposées en séquences de semi-jointures (d'ailleurs pas uniques). Les questions irréductibles présentent donc des cycles dans le graphe de connexion des relations. Cependant, certaines questions peuvent paraître irréductibles alors qu'il existe des questions équivalentes décomposables par détachement [Bernstein81].

Il existe donc des questions qui ne peuvent être ramenées à des questions mono-variables par détachement : celles qui présentent des cycles dans le graphe de connexion des attributs. En fait, ces questions retrouvent des résultats à partir de plusieurs relations et comportent donc de vraies jointures, non transformables en semi-jointures. Par exemple, la question `Q4` qui recherche les couples (noms de buveurs, noms de producteurs) tels que le buveur ait bu un vin du producteur :

```
(Q4)  SELECT B.NOM, P.NOM
      FROM BUVEURS B, ABUS A, PRODUIT R, PRODUCTEUR P
      WHERE B.NB = A.NB AND A.NV = R.NV AND R.NP = P.NP
```

est une question irréductible. Son graphe de connexion des relations présente en effet un cycle, comme le montre la figure X.10.

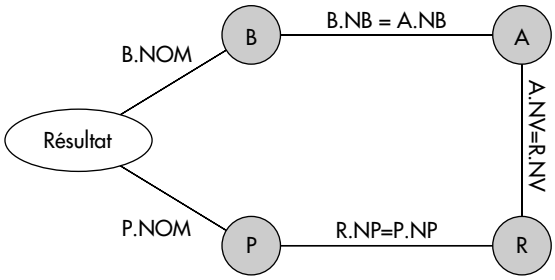


Figure X.10 : Graphe de connexion de relations avec cycle

Afin de résoudre les questions irréductibles, le système INGRES de l'Université de Berkeley appliquait une méthode très simple : une des relations était balayée séquentiellement, par substitution des tuples successifs à la variable. Pour chaque tuple ainsi obtenu, la sous-question générée par substitution de la variable par le tuple lu était traitée par détachement. En cas de génération d'une nouvelle sous-question irréductible, la substitution était à nouveau appliquée. L'algorithme était donc récursif [Wong76]. Des méthodes de jointures beaucoup plus sophistiquées sont aujourd'hui disponibles.

3.4. BILAN DES MÉTHODES D'OPTIMISATION LOGIQUE

Les méthodes de réécriture travaillant au niveau des opérateurs logiques sont suffisantes pour traiter les problèmes de correction de requêtes. Du point de vue optimisation, elles permettent d'ordonner les opérateurs à l'aide d'heuristiques simples. Les unions peuvent être accomplies avant les jointures. L'intersection et la différence ne sont que des cas particuliers de jointures.

La décomposition est une heuristique d'ordonnancement sophistiquée qui consiste à exécuter tout d'abord les sélections, puis à ordonner les jointures de sorte à faire apparaître en premier les semi-jointures possibles. Puisque les semi-jointures réduisent les tailles des relations, on peut ainsi espérer réduire les tailles des résultats intermédiaires et donc le nombre d'entrées-sorties. Cette heuristique apparaît ainsi comme supérieure à l'ordonnancement par restructuration algébrique qui n'ordonne pas du tout les jointures.

Le vrai problème est celui d'ordonner les jointures, et plus généralement les opérations binaires. La réécriture est une première approche qui permet un certain ordonnancement et ne nécessite aucune estimation de la taille des résultats des jointures. Cette approche est tout à fait insuffisante pour obtenir un plan d'exécution de coût minimal, car elle ne permet ni d'ordonner les opérations de manière optimale, ni de choisir les algorithmes optimaux pour chaque opération.

4. LES OPÉRATEURS PHYSIQUES

Avant d'aborder l'optimisation physique, il est nécessaire de comprendre les algorithmes exécutant l'accès aux tables. Ceux-ci réalisent les opérateurs relationnels et sont au centre du moteur du SGBD responsable de l'exécution des plans. En conséquence, leur optimisation est importante. Dans la suite, nous nous concentrons sur la sélection, le tri, la jointure et les calculs d'agrégats. Les autres opérations ensemblistes peuvent être effectuées de manière analogue à la jointure.

4.1. OPÉRATEUR DE SÉLECTION

Le placement des données dans les fichiers est effectué dans le but d'optimiser les sélections les plus fréquentes, et aussi les jointures. La présence d'index sur les attributs référencés dans le prédicat argument de la sélection change radicalement l'algorithme de sélection. Cependant, il existe des cas dans lesquels l'utilisation d'index est pénalisante, par exemple si la sélectivité du prédicat est mauvaise (plus de 20 % des

tuples satisfont le critère). De plus, aucun index n'est peut être spécifié. En résumé, on doit donc considérer deux algorithmes différents : la sélection par balayage séquentiel et par utilisation d'index.

4.1.1. Sélection sans index

Le **balayage séquentiel** (*sequential scan*) nécessite de comparer chaque tuple de la relation opérande avec le critère. Si celui-ci est vrai, les attributs utiles sont conservés en résultat. La procédure effectue donc à la fois restriction et projection.

Le critère peut être compilé sous la forme d'un automate afin d'accélérer le temps de parcours du tuple. L'automate peut être vu comme une table ayant en colonnes les codes caractères et en lignes les états successifs. Il permet pour chaque caractère lu d'appliquer une action et de déterminer l'état suivant de l'automate. L'état suivant est soit continuation, soit échec, soit succès. En cas de continuation, le caractère suivant est traité. Sinon, le tuple est retenu en cas de succès, ou ignoré en cas d'échec. La figure X.11 illustre l'automate qui peut être utilisé pour traiter le critère `COULEUR = "ROUGE" OU COULEUR = "ROSE"`. Des filtres matériels ont été réalisés selon ce principe afin d'effectuer la sélection en parallèle à l'entrée-sortie disque. Les SGBD effectuent plutôt le filtrage en mémoire après avoir lu une page du fichier en zone cache.

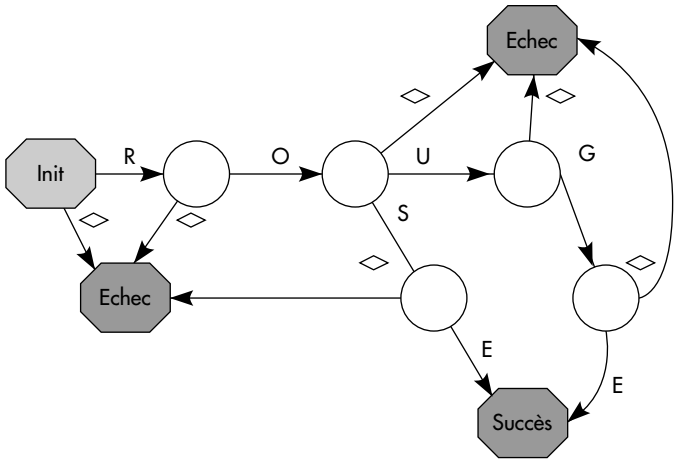


Figure X.11 : Exemple d'automate
(<> signifie tout autre caractère que ceux cités)

Un paramètre important pour la sélection sans index est le fait que le fichier soit trié ou non sur l'attribut de restriction. Dans le cas où le fichier n'est pas trié, le balayage séquentiel (BS) complet du fichier est nécessaire. Le nombre d'entrées-sorties nécessaire est alors le nombre de pages du fichier : Coût(BS) = Page(R).

Dans le cas où le fichier est trié sur l'attribut testé, une recherche dichotomique (RD) est possible : le bloc médian du fichier sera d'abord lu, puis selon que la valeur cherchée de l'attribut est inférieure ou supérieure à celle figurant dans le premier tuple du bloc, on procédera récursivement par dichotomie avec la première ou la seconde partie du fichier. Quoi qu'il en soit, la recherche sans index est longue et conduit à lire toutes les pages du fichier si celui-ci n'est pas trié, ou $Log_2(Page(R))$ pages si le fichier est trié sur l'attribut de sélection : $Coût(RD) = Log_2(Page(R))$.

4.1.2. Sélection avec index de type arbre-B

Un **index** associe les valeurs d'un attribut avec la liste des adresses de tuples ayant cette valeur. Il est généralement organisé comme un arbre-B et peut être plaçant, c'est-à-dire que les tuples sont placés dans les pages selon l'ordre croissant des valeurs de clé dans l'index. Il s'agit alors d'un arbre-B+ ; celui-ci peut être parcouru séquentiellement par ordre croissant des clés. Dans les bases de données relationnelles, l'index primaire est souvent plaçant, alors que les index secondaires ne le sont pas.

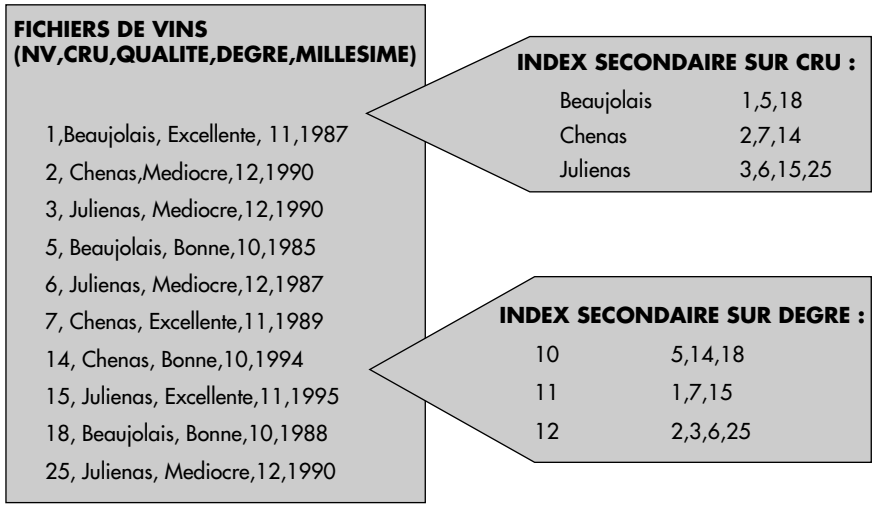
Voici maintenant le principe d'évaluation d'un critère dont certains attributs sont indexés. On choisit les index intéressants (par exemple, ceux correspondant à une sélectivité du sous-critère supérieure à 60%). Pour chacun, on construit une liste d'adresses de tuples en mémoire. Puis on procède aux intersections (critère ET) et aux unions (critère OU) des listes d'adresses. On obtient ainsi la liste des adresses de tuples qui satisfont aux sous-critères indexés. On peut ensuite accéder à ces tuples et vérifier pour chacun le reste du critère par un test en mémoire.

Certains systèmes retiennent seulement le meilleur index et vérifie tout le reste du critère en lisant les tuples sélectionnés par le premier index. Si un index est plaçant, on a souvent intérêt à l'utiliser comme critère d'accès. Un cas dégénéré est le cas de placement par hachage. Alors, seul le sous-critère référençant l'attribut de hachage est en général retenu et le reste est vérifié en mémoire.

La figure X.12 illustre une table VINS placée en séquentielle et possédant deux index sur CRU et DEGRE. Pour simplifier, le numéro de vin a été porté à la place des adresses relatives dans les entrées des index secondaires. Aucun index n'est plaçant. Le traitement du critère `(CRU = "CHENAS") AND (MILLESIME ≥ 1986) AND (DEGRE = 12)` conduit à considérer les deux index. On retient les index sur CRU et DEGRE, et l'on effectue la vérification du millésime par accès au fichier. D'où le plan d'accès indiqué figure X.12.

En général, le choix des meilleurs chemins d'accès (index ou fonction de hachage) n'est pas évident. Par exemple, on pourrait croire que les index sont inutiles dans les cas de comparateurs inférieur (<) ou supérieur (>). Il n'en est rien au moins pour les index plaçants : l'accès à l'index permet de trouver un point d'entrée dans le fichier que l'on peut ensuite balayer en avant ou en arrière en profitant du fait qu'il est trié.

Les index non plaçants peuvent parfois être utiles avec des comparateurs supérieurs ou inférieurs, mais ceci est plus rare car ils provoquent des accès désordonnés au fichier. Tout cela montre l'intérêt d'un modèle de coût qui permet seul de faire un choix motivé, comme nous le verrons ci-dessous.




```
PLAN D'ACCÉS
(table VINS critère (CRU = "CHENAS") AND (MILLESIME ≥ 1986)
   AND (DEGRE = 12))
{  C = LIRE (index CRU entrée CHENAS)
   D = LIRE (index DEGRE entrée 12)
   L = UNION (C, D)
   Pour chaque l de L faire
   {  Tuple = LIRE (table VINS adresse l)
      if Tuple.MILLESIME ≥ 1986 alors
      résultat = UNION (résultat, Tuple)}
   }
}
```

Figure X.12 : Exemple de sélection via index

4.2. OPÉRATEUR DE TRI

Le tri est un opérateur important car il est utilisé pour les requêtes demandant une présentation de résultats triés, mais aussi pour éliminer les doubles, effectuer certaines jointures et calculer des agrégats.

Le tri de données qui ne tiennent pas en mémoire est appelé **tri externe**. Des algorithmes de tri externes ont été proposés bien avant l'avénement des bases de données relationnelles [Knuth73]. On distingue les algorithmes par tri-fusion (*sort – merge*) et les algorithmes distributifs. Les algorithmes par tri-fusion créent des monotonies en mémoire. Par exemple, si (B+1) pages sont disponibles, des monotonies de B pages sont créées, sauf la dernière qui comporte en général moins de pages. Le nombre de monotonies créées correspond au nombre de pages de la relation R à créer divisé par B en nombre entier, plus la dernière :

$$Nmon = 1 + ENT(Page(R)/B).$$

Pour constituer une monotonie, au fur et à mesure des lectures d'enregistrements, un index trié est créé en mémoire. Lorsque les B pages ont été lues, les enregistrements sont écrits par ordre croissant des clés dans un fichier qui constitue une monotonie. La constitution de toutes les monotonies nécessite de lire et d'écrire la relation, donc 2*Page(R) entrées-sorties.

Dans une deuxième phase, les monotonies sont fusionnées. Comme seulement B pages en mémoire sont disponibles pour lire les monotonies, il faut éventuellement plusieurs passes pour créer une monotonie unique. Le nombre de passes nécessaires est $LOG_B(Nmon)$. Chaque passe lit et écrit toutes les pages de la relation. D'où le nombre total d'entrées-sorties pour un tri-fusion :

$$Coût(TF) = 2*Page(R)*(1+LOG_B(1+ ENT(Page(R)/B))).$$

Lorsque Page(R) est grand, un calcul approché donne :

$$Coût(TF) = 2*Page(R)*(1+LOG_B(Page(R)/B))),$$

On obtient :

$$Coût(TF) = 2*Page(R)*LOG_B(Page(R)).$$

4.3. OPÉRATEUR DE JOINTURE

Comme avec les sélections, il est possible de distinguer deux types d'algorithmes selon la présence d'index sur les attributs de jointure ou non. Dans la suite, nous développons les principes des algorithmes de jointure sans index puis avec index. Nous considérons l'équi-jointure de deux relations R1 et R2 avec un critère du type R1.A = R2.B, où A et B sont respectivement des attributs de R1 et R2. Dans le cas d'inéqui-jointure (par exemple, R1.A > R2.B), les algorithmes doivent être adaptés, à l'exception des boucles imbriquées qui fonctionne avec tout comparateur. Nous supposons de plus que R1 a un nombre de tuples inférieur à celui de R2.

4.3.1. Jointure sans index

En l'absence d'index, il existe trois algorithmes fondamentaux : les boucles imbriquées, le tri-fusion, et le hachage [Blasgen76, Valduriez84, DeWitt84]. Les algorithmes hybrides sont souvent les plus efficaces [Fushimi86, DeWitt92].

4.3.1.1. Boucles imbriquées

L'algorithme des **boucles imbriquées** est le plus simple. Il consiste à lire séquentiellement la première relation R1 et à comparer chaque tuple lu avec chaque tuple de la deuxième R2. R1 est appelée relation externe et Ré relation interne. Pour chaque tuple de R1, on est donc amené à lire chaque tuple de R2. L'algorithme est schématisé figure X.13. L'opérateur + permet de concaténer les deux tuples opérandes. Le test d'égalité des attributs Tuple1.A et Tuple2.B doit être remplacé par la comparaison des attributs en cas d'inéqui-jointure.

Les entrées-sorties s'effectuant par page, en notant Page(R) le nombre de pages d'une relation R, on obtient un coût en entrées-sorties de :

Coût(BI) = Page(R1) + Page(R1) * Page(R2).

Si la mémoire cache permet de mémoriser (B+1) pages, il est possible de garder B pages de R1 en mémoire et de lire R2 seulement Page(R1)/B fois.

La formule devient alors :

Coût(BI) = Page(R1) + Page(R1)* Page(R2)/B.

Ceci souligne l'intérêt d'une grande mémoire cache qui permet de réduire le nombre de passes.

```
Boucles_Imbriquées (R1,A, R2,B){
Pour chaque page B1 de R1 faire{
   Lire (R1,B1) ;
   Pour chaque page B2 de R2 faire{
      Lire (R2,B2) ;
      Pour chaque tuple Tuple1 de B1 faire
         Pour chaque tuple Tuple2 de B2 faire{
               si Tuple1.A = Tuple2.B alors
               ECRIRE(Résultat, Tuple1 + Tuple2) ;
         }
      }
   }
}
```

Figure X.13 : Algorithme des boucles imbriquées

4.3.1.2. Tri-fusion

L'algorithme par tri-fusion effectue le tri des deux relations sur l'attribut respectif de jointure. Ensuite, la fusion des tuples ayant même valeur est effectuée. L'algorithme est schématisé figure X.14. En supposant des tris de N pages en 2N LOG N comme ci-dessus, le coût en entrées-sorties de l'algorithme est :

Coût(JT) = 2*Page(R1)*LOG(Page(R1)) + 2*Page(R2)*LOG(Page(R2)) + Page(R1) + Page(R2).

Les logarithmes (LOG) sont à base 2 pour des tris binaires, et à base B pour des tris avec (B+1) pages en mémoire cache. L'algorithme est en général beaucoup plus efficace que le précédent. Son efficacité dépend cependant de la taille mémoire disponible. Soulignons aussi que si l'une des relations est déjà triée sur l'algorithme de jointure, il n'est pas nécessaire de refaire le tri.

```
Tri_Fusion (R1,A, R2,B){
      R1T = TRIER (R1,A) ;
      R2T = TRIER (R2,B) ;
      Résultat = FUSIONNER (R1T,R2T) }
```

Figure X.14 : Algorithme de tri-fusion

4.3.1.3. Partition

Il s'agit d'une méthode par hachage qui peut aisément être combinée avec l'une des précédentes. L'algorithme par partition consiste à découper les deux relations en partitions en appliquant une même fonction de hachage h sur les attributs de jointure A et B. Dans la mesure du possible, les partitions générées sont gardées en mémoire. On est alors ramené à joindre les paquets de même rang par l'un des algorithmes précédents. En effet, pour pouvoir être joints, deux tuples doivent donner la même valeur par la fonction de hachage appliquée aux attributs de jointure, car ceux-ci doivent être égaux.

L'algorithme peut être amélioré par la gestion d'un tableau de bits de dimension N en mémoire [Babb79] (N aussi grand que possible). L'astuce consiste à hacher la première relation simultanément avec la fonction h pour créer les partitions et une autre fonction g distribuant uniformément les valeurs de A sur [1,N]. La fonction g permet de maintenir un tableau de bits servant de filtre pour la deuxième relation. À chaque tuple haché de R1, le bit g(B) est mis à 1 dans le tableau de bits. Lorsqu'on partitionne la deuxième relation, on effectue pour chaque tuple de R2 un test sur le bit g(B) du tableau. S'il est à 0, on peut éliminer ce tuple qui n'a aucune chance de jointure (aucun tuple de R1 ne donne la même valeur par g). Le principe de l'algorithme par hachage est représenté figure X.15 : seuls les paquets de même rang, donc de la diagonale sont joints.

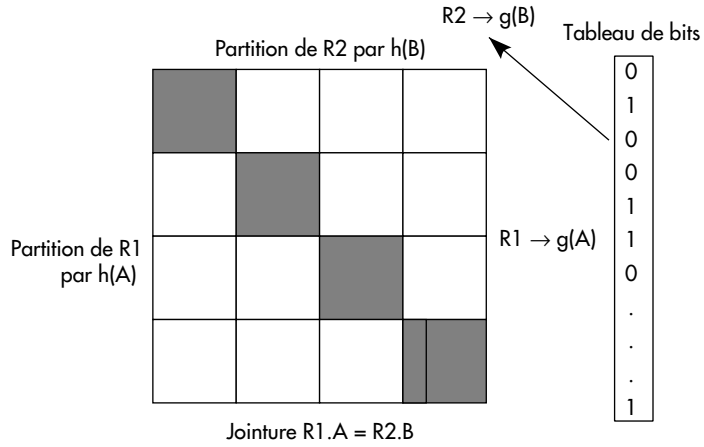


Figure X.15 : Illustration de l'algorithme par partition

4.3.1.4. Hachage

L'algorithme par hachage le plus simple consiste à hacher seulement la première relation dans un fichier haché si possible gardé en mémoire. Ensuite, on balaye la deuxième relation, et pour chaque tuple lu on tente d'accéder au fichier haché en testant l'existence d'enregistrement de clé identique dans le fichier haché (fonction PROBE). Tant que l'on trouve des enregistrements de clé similaire, on compose la jointure résultat. Cet algorithme est schématisé figure X.16.

```
Hachage (R1,A, R2,B) {
   Pour chaque page B1 de R1 faire {// hacher R1 sur A
      Lire (R1,B1) ;
      Pour chaque tuple Tuple1 de B1 faire{
         p = h(Tuple1,A) ;
         ECRIRE (Fichier_Haché, Paquet p, Tuple1) ;}}
   Pour chaque page B2 de R2 faire {// Parcourir R2
      Lire (R2,B2) ;
      Pour chaque tuple Tuple2 de B2 faire   {
         Tant que PROBE(Fichier_Haché,Tuple2) faire {
               // Joindre si succès
               ACCEDER (Fichier_Haché, Tuple1) ;
               si Tuple1.A = Tuple2.B alors
                     ECRIRE(Résultat,Tuple1 + Tuple2) ;
               }
         }
      }
   }
```

Figure X.16 : Algorithme par hachage

Le coût de l'algorithme par hachage dépend beaucoup de la taille mémoire disponible. Au mieux, il suffit de lire les deux relations. Au pire, on est obligé de faire plusieurs passes pour hacher la première relation, le nombre de passes étant le nombre de paquets de la table hachée divisé par le nombre de paquets disponibles en mémoire. On peut estimer ce nombre par Page(R1)/B, B+1 étant toujours le nombre de pages disponibles en mémoire. On obtient donc Page(R1)*Page(R1)/B entrées-sorties pour hacher R1. Il faut ensuite lire R2. Pour chaque tuple de R2, on va accéder au fichier haché. Notons Tuple(R) le nombre de tuples d'une relation R. Pour l'accès au fichier haché, on obtient un coût maximal de Tuple(R2) entrées-sorties, en négligeant l'effet de la mémoire cache. L'utilisation additionnelle d'un tableau de bits permet de diminuer Tuple(R2) d'un facteur difficile à évaluer, noté f avec $f < 1$, qui dépend de la sélectivité de la jointure. Ce facteur peut aussi intégrer l'effet de la mémoire cache. On obtient donc au total :

$$\text{Coût(JH)} = \text{Page(R1)*Page(R1)/B} + \text{Page(R2)} + \text{f*Tuple(R2)}$$

Ceci est difficilement comparable avec les formules précédentes, mais très bon si B est grand et la sélectivité de la jointure faible (f petit).

4.3.1.5. Table de hachage

Une variante souvent utilisée de l'algorithme précédent consiste à construire en mémoire une table de hachage pour la plus petite relation. Cette table remplace le fichier haché et conserve seulement l'adresse des enregistrements, et pour chacun d'eux la valeur de l'attribut de jointure. La deuxième phase consiste à lire chaque tuple de la deuxième relation et à tester s'il existe des tuples ayant même valeur pour l'attribut de jointure dans la table de hachage. L'algorithme comporte donc une phase de construction (*Build*) de la table de hachage et une phase de tests (*Probe*) pour les tuples de la deuxième relation. La table de hachage mémorise pour chaque valeur de la fonction de hachage v la liste des adresses de tuples tels que H(A) = v, ainsi que la valeur de l'attribut A pour chacun des tuples. La table de hachage est généralement suffisamment petite pour tenir en mémoire. Le coût de l'algorithme est donc :

$$\text{Coût(TH)} = \text{Page(R1)} + \text{Page(R2)} + \text{Tuple(R1} \rtimes \text{R2)}.$$

Le pseudo-code de cet algorithme est représenté figure X.17. L'algorithme présente aussi l'avantage de permettre l'évaluation des requêtes en pipeline : dès que la table de hachage sur R1 a été construite, le pipeline peut être lancé lors du parcours séquentiel de R2. Nous considérons donc cet algorithme comme l'un des plus intéressants. Il peut être couplé avec la construction de partition vue ci-dessus lorsque la table de hachage devient trop importante.

```
TableHachage (R1,A, R2,B) {
Pour i = 1, H faire TableH[i] = null ; // nettoyer la table de hachage
Pour chaque page B1 de R1 faire // construire la table de hachage {
   Lire (R1,B1) ;
   Pour chaque tuple Tuple1 de B1 faire Build(TableH,Tuple1, A) ;
   } ;
// Parcourir R2 et tester chaque tuple
Pour chaque page B2 de R2 faire {
   Lire (R2,B2) ;
   Pour chaque tuple Tuple2 de B2 faire {
      Tant que PROBE(TableH,Tuple2, B) faire {
         // Joindre si succès
         ACCEDER (TableH, AdresseTuple1) ; // obtenir adr.
         LIRE (AdresseTuple1, Tuple1) ; // Lire le tuple
         ECRIRE(Résultat,Tuple1||Tuple2) ; // Ecrire résultat
         }
      }
   }
}
```

Figure X.17 : Algorithme par table de hachage

4.3.2. Jointure avec index de type arbre-B

Lorsque l'une des relations est indexée sur l'attribut de jointure (par exemple R1 sur A), il suffit de balayer la deuxième relation et d'accéder au fichier indexé pour chaque tuple. En cas de succès, on procède à la concaténation pour composer le tuple résultat. L'algorithme est analogue à la deuxième phase de l'algorithme par hachage en remplaçant le fichier haché par le fichier indexé. Le coût est de l'ordre de 3*Tuple(R2), en supposant trois accès en moyenne pour trouver un article dans le fichier indexé. Si R2 est grande, le coût peut être prohibitif et on peut avoir intérêt à créer un index sur R2 pour se ramener au cas suivant.

Lorsque les deux relations sont indexées sur les attributs de jointure, il suffit de fusionner les deux index [Valduriez87]. Les couples d'adresses de tuples joignant sont alors directement obtenus. L'algorithme est peu coûteux en entrées-sorties (lecture des index et des tuples résultats). Par contre, la mise à jour de ces index peut être pénalisante.

4.4. LE CALCUL DES AGRÉGATS

Les algorithmes permettant de calculer les agrégats sont basés soit sur le tri, soit sur une combinaison du hachage et du tri. Un tri sur les attributs de groupement (argument du

`GROUP BY` de SQL) permet de créer des monotonies correspondant à chaque valeur de ces attributs. Pour chaque monotonie, on applique les fonctions d'agrégat (`COUNT`, `SUM`, `MIN`, `MAX`, etc.) demandées. Un hachage préalable sur l'attribut de groupement permet de ramener le problème à un calcul d'agrégat dans chaque partition. L'approche est donc ici analogue à celle de la jointure par partition et tri présentée ci-dessus.

La présence d'index sur les attributs de groupement simplifie les calculs d'agrégats en permettant de créer les monotonies à partir de l'index. La présence d'un index sur l'attribut cible argument de la fonction est intéressante pour les cas minimum (`MIN`), et maximum (`MAX`). La première clé de l'index donne la valeur minimale et la dernière la valeur maximale. Il s'agit alors de partitionner l'index en fonction des attributs de groupement, ce qui peut s'effectuer en une passe de la relation si l'index tient en mémoire. Les meilleures optimisations d'agrégats, très importantes dans les systèmes décisionnels, passent par la mémorisation des calculs, par exemple dans des vues concrètes. Ces problèmes sont étudiés dans le chapitre consacré aux vues.

5. L'ESTIMATION DU COÛT D'UN PLAN D'EXÉCUTION

La restructuration algébrique est insuffisante car elle n'ordonne pas les opérations binaires. De plus, l'application d'une sélection initiale peut faire perdre un index qui serait utile pour exécuter une jointure. Afin d'aller au-delà, une solution pour choisir le meilleur plan d'exécution consiste à les générer tous, à estimer le coût d'exécution de chacun et à choisir celui de moindre coût. Malheureusement, une telle stratégie se heurte à plusieurs difficultés.

5.1. NOMBRE ET TYPES DE PLANS D'EXÉCUTION

Tout d'abord, le nombre de plans d'exécution possible est très grand. Cet inconvénient peut être évité en éliminant tous les plans qui font appel à l'opération de produit cartésien comme dans le fameux système R d'IBM ; en effet, la commutation de jointures peut générer des produits cartésiens qui en général multiplient les tailles des relations à manipuler. On a donc tout intérêt à éviter les plans d'exécution contenant des produits cartésiens. On peut aussi ne pas considérer les **arbres ramifiés**, mais seulement ceux contenant des jointures d'une relation de base avec une relation intermédiaire produite par les jointures successives. On parle alors d'**arbre linéaire droit** ou **gauche** (cf. figure X.18).

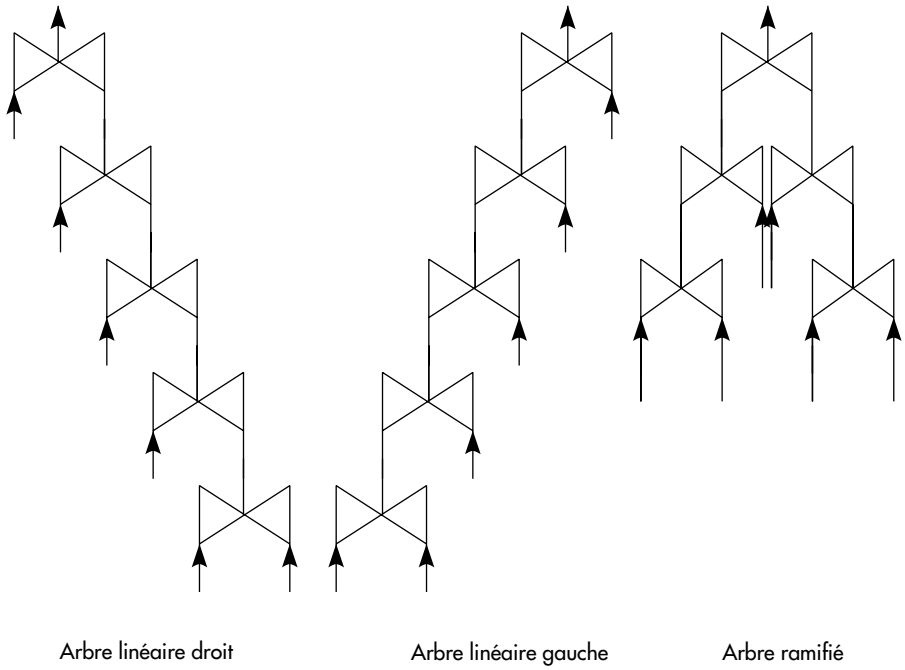


Figure X.18 : Arbres ramifié ou linéaires

Une autre possibilité encore plus restrictive est d'éliminer tous les plans qui n'effectuent pas les sélections dès que possible. Ces heuristiques éliminant souvent des plans intéressants, on préfère aujourd'hui mettre en œuvre une stratégie d'exploration de l'espace des plans plus sophistiquée pour trouver un plan proche de l'optimal, comme nous le verrons ci-dessous.

Le nombre de plans devient vite très grand. En effet, considérons simplement le cas d'une requête portant sur n relations et effectuant donc (n-1) jointures. Soit P(n) le nombre de plans. Si l'on ajoute une relation, pour chaque jointure il existe deux positions possibles pour glisser la nouvelle jointure (à droite ou à gauche), et ce de deux manières possibles (nouvelle relation à droite ou à gauche) ; pour la jointure au sommet de l'arbre, il est aussi possible d'ajouter la nouvelle jointure au-dessus en positionnant la nouvelle relation à droite ou à gauche. Le nombre de jointures de n relations étant (n-1), il est possible de calculer :

$P(n+1) = 4*(n-1) *P(n)+2*P(n)$ avec $P(2) = 1$.

D'où l'on déduit encore :

$P(n+1) = 2*(2n-1)*P(n)$ avec $P(2) = 1$.

Il s'ensuit le nombre de plans possibles :

$P(n+1) = (2n) ! / n !$

Un rapide calcul conduit au tableau suivant donnant le nombre de plans possibles P(n) pour n relations :

n	P(n)
2	2
3	12
4	120
5	1680
6	30240
7	665280
8	17297280
9	518918400
10	1,7643E+10
11	6,7044E+11
12	2,8159E+13

On voit ainsi que le nombre de plans devient rapidement très grand. Ce calcul ne prend pas en compte le fait que plusieurs algorithmes sont possibles pour chaque opérateur de jointure. Si a algorithmes sont disponibles, le nombre de plans pour n relations doit être multiplié par a^{n-1}. Ceci devient rapidement un nombre considérable.

Le calcul du coût d'un plan peut être effectué récursivement à partir de la racine N de l'arbre en appliquant la formule suivante, dans laquelle $Fils_i$ désigne les fils du nœud N :

$$\text{COUT}(PT) = \text{COUT}(N) + \Sigma\ \text{COUT}(\text{FILS}_I).$$

Le problème est donc de calculer le coût d'un nœud qui représente une opération relationnelle. La difficulté est que l'estimation du coût d'une opération nécessite, au-delà de l'algorithme appliqué, la connaissance de la taille des relations d'entrée et de sortie. Il faut donc estimer la taille des résultats intermédiaires de toutes les opérations de l'arbre considéré. Le choix du meilleur algorithme pour un opérateur nécessite d'autre part la prise en compte des chemins d'accès aux relations (index, placement, lien…) qui change directement ces coûts. Nous allons ci-dessous examiner les résultats concernant ces deux problèmes.

5.2. ESTIMATION DE LA TAILLE DES RÉSULTATS INTERMÉDIAIRES

L'estimation de la taille des résultats est basée sur un modèle de distribution des valeurs des attributs dans chacune des relations, et éventuellement des corrélations entre les valeurs d'attributs. Le modèle doit être conservatif, c'est-à-dire que pour toute opération de l'algèbre relationnelle, il faut être capable, à partir des paramètres décrivant les relations arguments, de calculer les mêmes paramètres pour la relation

résultat. Le modèle le plus simple est celui qui suppose l'uniformité de la distribution des valeurs et l'indépendance des attributs [Selinger79]. De telles hypothèses sont utilisées dans tous les optimiseurs de systèmes en l'absence d'informations plus précises sur la distribution des valeurs d'attributs. Un tel modèle nécessite de connaître au minimum :

– le nombre de valeurs d'un attribut A noté `Tuple(A)`;
– les valeurs minimale et maximale d'un attribut `A` notées `MIN(A)` et `MAX(A)`;
– le nombre de valeurs distinctes de chaque attribut A noté `NDIST(A)`;
– le nombre de tuples de chaque relation `R` noté `Tuple(R)`.

Le nombre de tuples d'une restriction est alors calculé par la formule :

```
TUPLE (σ(R)) = P(CRITERE) * TUPLE(R)
```

où p(critère) désigne la probabilité que le critère soit vérifié, appelée **facteur de sélectivité** du prédicat de restriction.

Notion X.13 : Facteur de sélectivité (*Selectivity factor*)

Coefficient associé à une opération sur une table représentant la proportion de tuples de la table satisfaisant la condition de sélection.

Avec une hypothèse de distribution uniforme des valeurs d'attributs, le facteur de sélectivité peut être calculé comme suit :

```
p(A=valeur) = 1/NDIST(A)
p(A>valeur) = (MAX(A) – valeur)/(MAX(A) – MIN(A))
p(A<valeur) = (valeur – MIN(A))/(MAX(A) – MIN(A))
p(A IN liste) = (1/NDIST(A)) * Tuple(liste)
p(P et Q) = p(P) * p(Q)
p(P ou Q) = p(P) + p(Q) – p(P) * p(Q)
p(not P) = 1 – p(P)
```

Le nombre de tuples d'une projection sur un groupe d'attributs X est plus simplement obtenu par la formule :

```
TUPLE(Π(R)) = (1-d) * TUPLE(R)
```

avec d = probabilité de doubles. La probabilité de doubles peut être estimée en fonction du nombre de valeurs distinctes des attributs composant X ; une borne supérieure peut être obtenue par la formule :

```
d =  (1-NDIST(A1)/Tuple(R))*(1-NDIST(A2)/Tuple(R)) …
     * (1-NDIST(Ap)/Tuple(R)
```

où A1, A2, …. An désigne les attributs retenus dans la projection.

La taille d'une jointure est plus difficile à évaluer en général. On pourra utiliser la formule :

Tuple(R1 ⋈ R2) = p * Tuple(R1) * Tuple(R2)

où p, compris entre 0 et 1, est appelé facteur de sélectivité de la jointure. En supposant une équi-jointure sur un critère `R1.A = R2.B` et une indépendance des attributs A et B, le nombre de tuples de R2 qui joignent avec un tuple de `R1` donné est au plus `Tuple(R2)/NDIST(B)`. Pour tous les tuples de `R1`, on obtient donc `Tuple(R1)*Tuple(R2)/NDIST(B)` tuples dans la jointure. On en déduit la formule `p = 1/NDIST(B)`. Mais en renversant l'ordre des relations, on aurait déduit la formule `p = 1/NDIST(A)`. Ceci est dû au fait que certains tuples de `R1` (et de `R2`) ne participent pas à la jointure, alors qu'ils ont été comptés dans chacune des formules. Une estimation plus raisonnable demanderait d'éliminer les tuples ne participant pas à la jointure. Il faudrait pour cela prendre en compte le nombre de valeurs effectives du domaine commun de `A` et `B`, nombre en général inconnu. On utilisera plutôt une borne supérieure de p donnée par la formule

```
p = 1/ MIN(NDIST(A),NDIST(B)).
```

Les estimations de tailles des relations et attributs doivent être mémorisées au niveau de la métabase. Elles devraient théoriquement être mises à jour à chaque modification de la base. La plupart des systèmes ne mettent à jour les statistiques que lors de l'exécution d'une commande spécifique (`RUNSTAT`) sur la base. Afin d'affiner les estimations, il est possible de mémoriser des histogrammes de distribution des valeurs d'attributs dans la métabase. Ces valeurs sont calculées par des requêtes comportant des agrégats. Les formules précédentes sont alors appliquées pour chaque fraction de l'histogramme qui permet d'estimer le nombre de valeurs distinctes (`NDIST`) ou non (Tuple) dans une plage donnée.

5.3. PRISE EN COMPTE DES ALGORITHMES D'ACCÈS

Chaque opérateur est exécuté en appliquant si possible le meilleur algorithme d'accès aux données. Le coût dominant d'un opérateur est le coût en entrées-sorties. Celui-ci dépend grandement des chemins d'accès disponibles, comme nous l'avons vu ci-dessus, lors de l'étude des opérateurs d'accès. Par exemple, en désignant toujours par Tuple(R) le nombre de tuple de la relation R, par Page(R) son nombre de pages, par (B+1) le nombre de pages disponibles en mémoire cache, nous avons pour la jointure par boucles imbriquées (BI) dans le cas sans index :

Coût(BI(R1,R2)) = Page(R1) + [Page(R1)/B]*Page(R2).

S' il existe un index sur l'attribut de jointure de R2, le coût devient :

Coût(BI(R1,R2)) = Page(R1) + Tuple(R1)*I(R2)

où I(R2) désigne le nombre moyen d'entrées-sorties pour accéder au tuple de `R2` via l'index. I(R2) dépend du type d'index, de sa hauteur, de la mémoire centrale disponible pour l'index, et aussi du fait que R1 soit triée sur l'attribut de jointure ou non. En général, `I(R2)` varie entre `0.5` et la hauteur de l'index, `2` étant une valeur moyenne classique.

Au-delà des coûts en entrées-sorties, il faut aussi estimer les coûts de calcul, souvent négligés. Dans Système R [Selinger79], la détermination du plan d'exécution est effectuée en affectant un coût à chaque plan candidat calculé par la formule suivante :

```
COUT = NOMBRE D'ACCES PAGE + W*NOMBRE D'APPELS INTERNES.
```

Le nombre d'appels internes donne une estimation du temps de calcul nécessaire à l'exécution du plan, alors que le nombre d'accès page évalue le nombre d'entrées-sorties nécessaires à l'exécution du plan. W est un facteur ajustable d'importance relative du temps unité centrale et du temps entrées-sorties. Le nombre d'accès page est estimé par évaluation d'un facteur de sélectivité `F` pour chacun des prédicats associés à une opération de l'algèbre relationnelle. Le nombre d'accès pour l'opération est calculé en fonction de `F` et de la taille des relations opérandes. Le calcul de `F` tient compte de la présence ou non d'index ; ainsi, par exemple, le facteur de sélectivité d'une sélection du type `Attribut = valeur` est estimé par :

```
F = 1/ Nombre d'entrées dans l'index
```

s'il existe un index et à `1/10` sinon.

6. LA RECHERCHE DU MEILLEUR PLAN

L'optimisation physique permet de prendre en compte le modèle de coût afin de déterminer le meilleur plan, ou un plan proche de celui-là. Elle part d'un arbre en forme canonique, par exemple composé de sélections dont les critères sont sous forme normale conjonctive, puis de jointures, unions, intersections et différences, et enfin d'agrégats suivis encore d'éventuelles sélections/projections finales, certaines de ces opérations pouvant être absentes. Elle transforme cet arbre en un plan d'accès en choisissant les algorithmes pour chaque opérateur, et en modifiant éventuellement l'arbre afin de profiter de propriétés physiques de la base (index par exemple).

Comme vu ci-dessus, le nombre de plans d'exécution possible peut être très grand pour des questions complexes. L'ensemble des plans possible est appelé **espace des plans**. Afin d'éviter d'explorer exhaustivement cet espace de recherche, c'est-à-dire d'explorer tous les plans, les optimiseurs modernes sont construits comme des générateurs de plans couplés à une **stratégie de recherche** découlant des techniques d'optimisation de la recherche opérationnelle [Swami88].

Notion X.14 : Stratégie de recherche (*Search strategy*)

Tactique utilisée par l'optimiseur pour explorer l'espace des plans d'exécution afin de déterminer un plan de coût proche du minimum possible.

Un **optimiseur est fermé** lorsque tous les opérateurs et algorithmes d'accès, ainsi que toutes les règles de permutation de ces opérateurs, sont connus à la construction du système. Les systèmes relationnels purs ont généralement des optimiseurs fermés. La stratégie de recherche dans les optimiseurs fermés est basée sur un algorithme déterministe, qui construit une solution pas à pas soit en appliquant une heuristique ou une méthode d'évaluation progressive permettant d'exhiber le meilleur plan, de type programmation dynamique. Nous étudions dans cette section quelques algorithmes de cette classe.

6.1. ALGORITHME DE SÉLECTIVITÉ MINIMALE

Cet algorithme très simple consiste à compléter les optimisations logiques présentées ci-dessus, telles par exemple la descente des restrictions et projections, par un ordonnancement des jointures par ordre de sélectivité croissante. L'algorithme construit un arbre linéaire gauche en partant de la plus petite relation, et en joignant à chaque niveau avec la relation restante selon le plus petit facteur de sélectivité de jointure. Ainsi, les relations intermédiaires sont globalement gardées près du plus petit possible. Cet algorithme, représenté figure X.19 généralise la méthode de décomposition d'Ingres étudiée ci-dessus. Une variante peut consister à produire à chaque fois la relation de plus petite cardinalité. Pour chaque opération, l'algorithme de coût minimal est ensuite sélectionné. Ces algorithmes restent simples, mais sont loin de déterminer le meilleur plan.

```
Algorithme MinSel {
   Rels = liste des relations à joindre ;
   p = plus petite relation ;
   Tant que Rels non vide {
      R = relation de selectivité minimum de Rels ;
      p = join(R,p) ;
      Rels = Rels – R ;} ;
   Retourner(p) ;}
```

Figure X.19 : Algorithme d'optimisation par sélectivité minimale

6.2. PROGRAMMATION DYNAMIQUE (DP)

La programmation dynamique est une méthode très connue en recherche opérationnelle, dont le principe est que toute sous-politique d'une politique optimale est optimale. Ainsi, si deux sous-plans équivalents du plan optimal cherché sont produits, il suffit de garder le meilleur et de le compléter pour atteindre le plan optimal. Cette technique, employée dans le système R [Selinger79] pour des plans sans produit cartésien, permet de construire progressivement le plan, par ajouts successifs d'opérateurs. L'algorithme commence par créer tous les plans mono-relation, et construit des plans de plus en plus larges étape par étape. A chaque étape, on étend les plans pro-

duits à l'étape précédente en ajoutant un opérateur. Quand un nouveau plan est généré, la collection de plans produits est consultée pour retrouver un plan équivalent. Si un tel plan est trouvé, alors les coûts des deux plans sont comparés et seul celui de coût minimal est conservé. La figure X.20 décrit de manière plus précise l'algorithme connu sous le nom DP (*Dynamic Programming*). Cet algorithme assez simple devient très coûteux pour un nombre important de relations, si l'on considère tous les arbres possibles. La complexité est de l'ordre de 2^N, où N est le nombre de relations. Il n'est donc que difficilement applicable au-delà d'une dizaine de relations [Swami88].

```
Algorithme DP {
   PlanOuverts = liste de tous les plans mono-relation possible ;
   Eliminer tous les plans équivalents exceptés les moins coûteux ;
   Tant qu'il existe p ∈ PlanOuverts {
      Enlever p de PlanOuverts ;
      PlanOptimal = p ;
      Pour chaque opérateur o permettant d'étendre p {
         Etendre le plan p en ajoutant cet opérateur o ;
         Calculer le coût du nouveau plan ;
         Insérer le nouveau plan dans PlanOuverts ;}
   Eliminer tous les plans équivalents exceptés les moins coûteux ;}
   Retourner(PlanOptimal) ;} ;
```

Figure X.20 : Algorithme d'optimisation par programmation dynamique

6.3. PROGRAMMATION DYNAMIQUE INVERSE

Une variante de l'algorithme précédent consiste à calculer récursivement la meilleure jointure possible parmi N relations, et ce selon tous les ordres possibles. Bien que récursif, l'algorithme est analogue au précédent mais procède en profondeur plutôt qu'en largeur. Il a l'avantage de construire plus rapidement des résultats et peut ainsi être arrêté si le temps consacré à l'optimisation est dépassé. La figure X.21 illustre cet algorithme dans le cas des jointures.

```
Algorithme RDP (Rels) {
   PlanOuverts = {} ;
   Coût(MeilleurPlan) = ∞ ;
   Pour chaque R in Rels{
      RestRels = Rels – R ;
      si RestRels = {} alors p = R sinon
         p = R ⋈ RDP(RestRels) ;
      Insérer p dans PlanOuverts ;
      si coût(p) ≤ coût(MeilleurPlan) alors MeilleurPlan = p ;
    Eliminer tous les plans équivalents exceptés les moins coûteux ;}
   Retourner (MeilleurPlan) ;};
```

Figure X.21 : Algorithme d'optimisation par programmation dynamique inverse

6.4. LES STRATÉGIES DE RECHERCHE ALÉATOIRES

Les différentes stratégies de recherche sont classées selon une hiérarchie de spécialisation figure X.22. On distingue les **stratégies énumératives** des **stratégies aléatoires.** Les premières énumèrent systématiquement des plans possibles. La **stratégie exhaustive** les énumère tous. La **stratégie par augmentation** les construit progressivement en partant par exemple de la projection finale et en introduisant progressivement les relations, par exemple par ordre de taille croissante. Elle évitera en général les produits cartésiens et appliquera dès que possible restriction et projection.

Les **stratégies aléatoires** [Lanzelotte91] explorent aléatoirement l'espace des plans. L'**amélioration itérative** tire au hasard n plans (par exemple en permutant l'ordre des relations) et essaie de trouver pour chacun d'eux le plan optimal le plus proche (par exemple par descente des restrictions et projections, et choix des meilleurs index). L'optimum des plans localement optimum est finalement sélectionné. La stratégie du **recuit simulé** procède à partir d'un plan que l'on tente d'optimiser en appliquant des transformations successives. Les transformations retenues améliorent ce plan exceptées quelques-unes introduites afin d'explorer un espace plus large avec une probabilité variant comme la température du système (en $e^{-1/T}$, où T est la température). La température est de fait une variable dont la valeur est très élevée au départ et qui diminue au fur et à mesure des transformations, de sorte à ne plus accepter de transformation détériorant le coût quand le système est gelé. Cette stratégie, bien connue en recherche opérationnelle, permet de converger vers un plan proche de l'optimum, en évitant les minimums locaux. On citera enfin les **stratégies génétiques** qui visent à fusionner deux plans pour en obtenir un troisième.

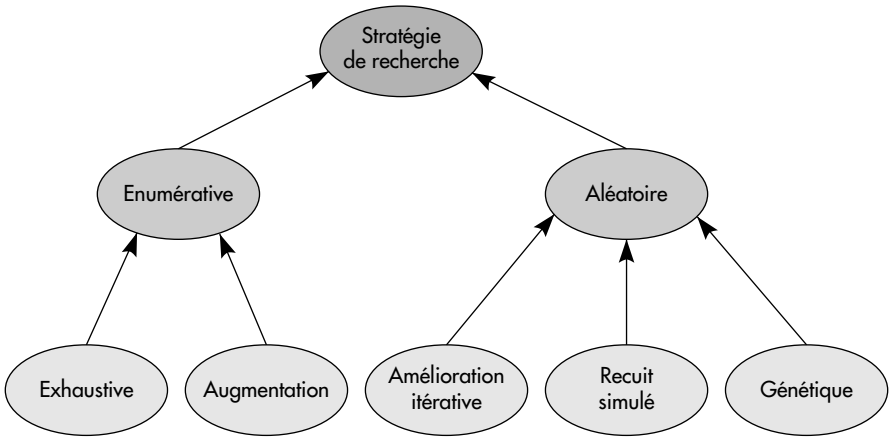


Figure X.22 : Principales stratégies de recherche

Toutes ces stratégies sont très intéressantes pour des systèmes extensibles, objet ou objet-relationnel. En effet, dans un tel contexte le nombre de plans d'exécution

devient très grand. Les choix sont donc très difficiles et les stratégies de type programmation dynamique deviennent d'application difficile. Voilà pourquoi nous étudierons les stratégies aléatoires plus en détail dans la partie III de cet ouvrage.

7. CONCLUSION

Dans ce chapitre, nous avons étudié les algorithmes et méthodes de base pour l'évaluation efficace des requêtes dans les SGBD relationnelles. L'étude a été réalisée à partir de requêtes exprimées en algèbre relationnelle. Pour réduire la complexité, nous avons décomposé le problème de l'optimisation en une phase logique fondée sur les techniques de réécriture et une phase physique basée sur un modèle de coût. De nos jours, la réalisation d'un optimiseur performant nécessite d'intégrer ces deux phases.

Nous nous sommes plus concentrés sur l'optimisation des requêtes avec jointures car celles-ci sont très fréquentes et souvent coûteuses dans les bases de données relationnelles. Nous n'avons qu'esquissé l'optimisation des agrégats. Nous reviendrons sur ce problème dans les perspectives pour l'aide à la décision. Nous n'avons aussi qu'esquissé l'étude des stratégies de recherche aléatoires, qui permettent de trouver rapidement un plan satisfaisant dans des espaces de recherche très vastes. Nous reviendrons sur cet aspect dans le cadre des optimiseurs extensibles, mis en œuvre dans les SGBD objet et objet-relationnel, au moins en théorie. Dans ce contexte, les plans sont encore plus nombreux que dans les SGBD relationnels, et la stratégie devient un composant essentiel.

Les techniques abordées se généralisent au contexte des bases de données réparties et parallèles. Les techniques de base de l'approche multiprocesseurs consistent à diviser pour régner : les tables sont partitionnées sur plusieurs serveurs parallèles ou répartis, les requêtes sont divisées en sous-requêtes par l'évaluateur de requêtes. Le problème supplémentaire est d'optimiser les transferts entre serveurs. Ceux-ci s'effectuent par le biais d'une mémoire commune, d'un bus partagé ou d'un réseau, selon le contexte. Le modèle de coût doit prendre en compte ces transferts, et les algorithmes doivent être adaptés. Les fondements restent cependant identiques à ceux présentés dans ce chapitre.

8. BIBLIOGRAPHIE

[Aho79] Aho A.V., Sagiv Y., Ullman J.D., « Equivalences among Relational Expressions », *SIAM Journal of Computing*, vol. 8, n° 2, p. 218-246, Juin 1979.

Cet article définit formellement l'équivalence d'expressions algébriques et donne des conditions d'équivalence.

[Astrahan76] Astrahan M.M. *et al.*, « System R : Relational Approach to Database Management », *ACM TODS*, vol. 1, n° 2, Juin 1976.

Cet article de synthèse présente le fameux système R réalisé à San José, au centre de recherche d'IBM. Il décrit tous les aspects, en particulier l'optimiseur de requêtes, intéressant pour ce chapitre.

[Babb79] Babb E., « Implementing a Relational Database by Means of Specialized Hadware », *ACM TODS*, vol. 4, n° 1, Mars 1979.

Auteur de la machine bases de données CAFS réalisée à ICL en Angleterre, E. Babb introduit en particulier les tableaux de bits afin d'accélérer les jointures par hachage.

[Bernstein81] Bernstein P.A., Chiu D.W., « Using Semijoins to Solve Relational Queries », *Journal of the ACM*, vol. 28, n° 1, p. 25-40, Janvier 1981.

Cet article présente une étude approfondie de l'usage des semi-jointures pour résoudre les questions relationnelles. De multiples résultats basés sur l'étude du graphe des relations pour décomposer une requête en semi-jointures sont établis.

[Blasgen76] Blasgen M.W., Eswaran K.P., « On the Evaluation of Queries in Relational Systems », *IBM Systems Journal*, vol. 16, p. 363-377, 1976.

Se fondant sur le développement et les expériences conduites autour de système R, les auteurs montrent que différents algorithmes de jointures doivent être considérés pour évaluer efficacement les requêtes dans les SGBD relationnels.

[Chakravarthy90] Chakravarthy U.S., Grant J., Minker J., « Logic Based Approach to Semantic Query Optimization », *ACM Transactions on Database Systems*, vol. 15, n° 2, p. 162-207, Juin 1990.

Cet article démontre l'apport de contraintes d'intégrité exprimées en logique du premier ordre pour l'optimisation de requêtes. Il propose un langage d'expression de contraintes et des algorithmes de base pour optimiser les requêtes.

[DeWitt84] DeWitt D., Katz R., Olken F., Shapiro L., Stonebraker M., Wood D., « Implementation Techniques for Main Memory Databases », *Proc. ACM SIGMOD* Int. Conf. on Management of data, Boston, Mass., p. 1-8, Juin 1984.

Les auteurs proposent différentes techniques pour la gestion de bases de données en mémoire. L'algorithme de hachage « Build and Probe » consiste à construire une table hachée en mémoire à partir de la première relation, puis à tester chaque tuple de la seconde relation contre cette table en appliquant la même fonction de hachage. Chaque entrée de la table contient un pointeur vers le tuple et la valeur de l'attribut de jointure, ce qui permet de tester le prédicat de jointure et de retrouver le tuple efficacement.

[DeWitt86] DeWitt D., Gerber R., Graefe G., Heytens M., Kumar K., Mualikrishna M., « Gamma – A high performance Dataflow Database Machine », *Proc. 12th Intl. Conf. on Very Large Data Bases*, Kyoto, Japan, Morgan Kaufman Ed., Septembre 1986.

La machine GAMMA réalisée à l'Université de Madison explore les techniques de flot de données pour paralléliser les opérateurs de jointure. Les algorithmes de type pipeline ont été expérimentés pour la première fois sur un réseau local en boucle.

[DeWitt92] DeWitt D., Naughton J., Schneider D., Seshadri S., « Practical Skew Handling in Parallel Joins » *Proc. 18th Intl. Conf. on Very Large Data Bases*, Sydney, Australia, Morgan Kaufman Ed., Septembre 1992.

Cet article étudie les effets des distributions biaisées de valeurs d'attributs de jointure sur les algorithmes de jointure. Les algorithmes hybrides semblent les plus résistants.

[Finance94] Finance B., Gardarin G., « A Rule-based Query Optimizer with Adaptable Search Strategies », *Data and Knowledge Engineering*, North-Holland Ed., vol. 3, n° 2, 1994.

Les auteurs décrivent l'optimiseur extensible réalisé dans le cadre du projet Esprit EDS. Cet optimiseur est extensible grâce à un langage de règles puissant et dispose de différentes stratégies de recherche paramétrables.

[Fushimi86] Fushimi S., Kitsuregawa M., Tanaka H., « An Overview of the System Software of a Parallel Relational Database Machine : GRACE », *Proc. 12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, Morgan Kaufman Ed., Septembre 1986.

Les auteurs présentent la machine bases de données GRACE. Celle-ci se distingue par son algorithme de jointure parallèle basé sur une approche mixte, combinant hachage et tri.

[Gardarin84] Gardarin G., Jean-Noël M., Kerhervé B., Mermet D., Pasquer F., Simon E., « Sabrina : un Système de Gestion de Bases de Données Relationnelles issu de la Recherche », *Revue TSI*, Dunod AFCET Ed., vol. 5, n° 6, 1986.

Cet article décrit le SGBD SABRE réalisé à l'INRIA de 1980 à 1984, puis industrialisé et commercialisé par la société INFOSYS de 1984 à 1990. Ce SGBD disposait d'un optimiseur basé sur des heuristiques sophistiquées.

[Graefe93] Graefe G., « Query Evaluation Techniques for Large Databases », *ACM Computer Surveys*, vol. 25, n° 2, p. 73-170, Juin 1993.

Un des plus récents articles de synthèse sur l'optimisation de requêtes. L'auteur présente une vue d'ensemble des techniques d'optimisation de requêtes, en mettant en avant leur comportement vis-à-vis de grandes bases de données.

[Gray91] Gray J., *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd edition, Morgan Kaufman, San Mateo, CA, 1991.

Le livre de référence du Transaction Processing Council (TPC). Il présente les « benchmarks » de base, notamment TPC/A, TPC/B, TPC/C, et les conditions précises d'exécution de ces bancs d'essais.

[Haas89] Haas M.L., Freytag J.C., Lohman G.M., Pirahesh H., « Extensible Query Processing in Starbust », *Proc. ACM SIGMOD Intl. Conf.* On Management of Data, p. 377-388, 1989.

Cet article présente l'optimiseur de Starbust, un système extensible réalisé à IBM. Cet optimiseur se décompose clairement en deux phases, la réécriture et le planning. Il est à la base des nouvelles versions de l'optimiseur de DB2.

[Hevner79] Hevner A.R., Yao B., « Query Processing in Distributed Database Systems », *IEEE Transactions on Software Engineering*, vol. 5, n° 3, p. 177-187, Mai 1979.

Cet article présente une synthèse des techniques d'optimisation de requêtes connues à cette date dans les BD réparties. Il développe un modèle de coût incluant le trafic réseau.

[Jarke84] Jarke M., Koch J., « Query Optimization in Database Systems » *ACM Computing Surveys*, vol. 16, n° 2, p. 111-152, Juin 1984.

Un des premiers articles de synthèse sur l'optimisation de requêtes dans les bases de données relationnelles. L'article suppose les questions exprimées en calcul relationnel de tuples. Il donne un cadre général pour l'évaluation de requêtes et couvre les algorithmes de réécriture logique, les méthodes d'optimisation physique, les modèles de coût, et plus généralement les principales techniques connues à cette époque.

[Kim85] Kim Won, Reiner S., Batory D., *Query Processing in Database Systems*, Springer-Verlag Ed., 1985.

Ce livre de synthèse est composé d'une collection d'articles. À partir d'un chapitre de synthèse introductif, les auteurs développent différents aspects spécifiques : bases de données réparties, hétérogènes, objets complexes, optimisation multi-requêtes, machines bases de données, modèle de coût, etc.

[King81] King J.J., « QUIST : A System for Semantic Query Optimization in Relational Data Bases », *Proc. 7th Intl. Conf. on Very Large Data Bases*, Cannes, France, IEEE Ed., p. 510-517, Sept. 1981.

Cet article décrit l'un des premiers systèmes capables de prendre en compte les contraintes d'intégrité pour l'optimisation de requêtes.

[Knuth73] Knuth D.E., *The Art of Computer Programming, Volume 3 : Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

Le fameux livre de Knuth consacré aux algorithmes et structures de données pour les recherches et les tris.

[Rosenkrantz80] Rosenkrantz D.J., Hunt H.B., « Processing Conjunctive Predicates and Queries », *Proc. 6th Intl. Conf. on Very Large Data Bases*, Montréal, Canada, IEEE Ed., Septembre 1980.

Cet article présente des techniques permettant de déterminer des questions contradictoires, équivalentes, ou incluses l'une dans l'autre. Les méthodes sont basées sur un graphe d'attribut normalisé où les arcs sont valués par des constantes. Un arc allant d'un nœud x à un nœud y correspond à l'inégalité $x \leq y+c$. Une égalité est modélisée par deux arcs. Par exemple, une question est contradictoire s'il existe un cycle dont la somme des valuations est négative.

[Selinger79] Selinger P., « Access Path Selection in a Relational Database Management System », *ACM SIGMOD Intl. Conf. On Management of Data,* Boston, Mai 1979.

Cet article décrit les algorithmes de sélection de chemin d'accès dans le système R. Il introduit le modèle de coût basé sur des distributions uniformes décrit section 5, et la méthode de sélection du meilleur plan basée sur la programmation dynamique.

[Smith75] Smith J.M., Chang P.Y., « Optimizing the performance of a Relational Algebra Database Interface », *Comm. ACM*, vol. 18, n° 10, p. 68-579, 1975.

Les auteurs introduisent l'ensemble des règles de restructuration algébrique permettant d'optimiser logiquement les expressions de l'algèbre relationnelle.

[Stonebraker76] Stonebraker M., Wong E., Kreps P., Held G., « The Design and Implementation of Ingres », *ACM TODS*, vol. 1, n° 3, Septembre 1976.

Cet article de synthèse présente le fameux système INGRES réalisé à l'université de Berkeley. Il décrit tous ses aspects, en particulier l'optimiseur de requêtes basé sur la décomposition de requêtes.

[Stonebraker87] Stonebraker M., « The Design of the Postgres Storage System », *13e Intl. Conf. on Very Large Data Bases*, Morgan Kaufman Ed., Brighton, Angleterre, 1987.

Postgres signifie « après Ingres ». M. Stonebraker a construit ce système à Berkeley après avoir réalisé et commercialisé le système Ingres. Postgres est original car fortement basé sur les concepts d'événements et déclencheurs, mais aussi par ses capacités à intégrer de nouveaux types de données pris en compte par un optimiseur extensible. Postgres est devenu plus tard le SGBD Illustra racheté par Informix en 1995.

[Swami88] Swami A., Gupta A., « Optimization of Large Join Queries », *ACM SIGMOD Intl. Conf.,* Chicago, 1988.

Cet article étudie les stratégies de recherche du meilleur plan pour des requêtes comportant plus d'une dizaine de jointures. Des stratégies aléatoires telles que l'amélioration itérative et le recuit simulé sont comparées.

[TPC95] Transaction Processing Council, *Benchmark TPC/D*, San Fransisco, CA, 1995.

La définition du TPC/D telle que proposée par le TPC.

[Ullman88] Ullman J.D., *Principles of Database and Knowledge-base Systems*, volumes I (631 pages) et II (400 pages), Computer Science Press, 1988.

Deux volumes très complets sur les bases de données, avec une approche plutôt fondamentale. Jeffrey Ullman détaille tous les aspects des bases de données, depuis les méthodes d'accès aux modèles objets en passant par le modèle logique. Ces ouvrages sont finalement très centrés sur une approche par la logique aux bases de données. Les principaux algorithmes d'accès et d'optimisation de requêtes sont détaillés dans un chapitre plutôt formel.

[Valduriez84] Valduriez P., Gardarin G., « Join and Semi-Join Algorithms for a Multiprocessor Database Machine », *ACM TODS*, vol. 9, n° 1, 1984.

Cet article introduit et compare différents algorithmes de jointure et semi-jointure pour machines multiprocesseurs. Il montre qu'aucune d'entre-elles n'est dominante, mais que chacune a son domaine d'application selon la taille des relations et la sélectivité des jointures.

[Valduriez87] Valduriez P., « Join Indices », *ACM TODS*, vol. 12, n° 2, Juin 1987.

L'auteur introduit les index de jointure, des index qui mémorisent la jointure pré-calculée entre deux tables. Chaque entrée de l'index est un couple de pointeurs référençant deux tuples participant à la jointure, l'un appartenant à la première table, l'autre à la seconde. De tels index sont très efficaces en interrogation. Le problème de ce type d'index est la mise à jour.

[Wong76] Wong E., Youssefi K., « Decomposition – A Strategy for Query Processing », *ACM TODS*, vol. 1, n° 3, Septembre 1976.

Cet article présente la méthode de décomposition telle qu'implémentée dans le système Ingres. On a compris plus tard que la méthode permettait de détacher les semi-jointures en plus des sélections.

Partie 3

L'OBJET ET L'OBJET-RELATIONNEL

XI – Le modèle objet (*The object model*)

XII – Le standard de l'ODMG
(*OQL, ODL and OML languages*)

XIII – L'objet-relationnel et SQL3
(*Object-relational DB and SQL3*)

XIV – L'optimisation de requêtes objet
(*Object query optiomization*)

Chapitre XI

LE MODÈLE OBJET

1. INTRODUCTION

Les modèles à objets, encore appelés modèles orientés objets ou simplement modèles objet, sont issus des réseaux sémantiques et des langages de programmation orientés objets. Ils regroupent les concepts essentiels pour modéliser de manière progressive des objets complexes encapsulés par des opérations de manipulation associées. Ils visent à permettre la réutilisation de structures et d'opérations pour construire des entités plus complexes. Ci-dessous, nous définissons les concepts qui nous semblent importants dans les modèles de données à objets. Ces concepts sont ceux retenus par l'OMG – l'organisme de normalisation de l'objet en général –, dans son modèle objet de référence [OMG91]. Certains sont obligatoires, d'autres optionnels. Ce modèle est proche du modèle de classe de C++ [Stroustrup86, Lippman91], qui peut être vu comme une implémentation des types abstraits de données. Il est aussi très proche du modèle objet du langage Java [Arnold96].

Bien que C++ et Java soient aujourd'hui les langages de programmation d'applications sur lesquels s'appuient la plupart des bases de données à objets, il existe d'autres langages orientés objet. Historiquement, Simula a été le premier langage orienté objet ; il est toujours un peu utilisé. Simula a introduit le concept de classe qui regroupe au sein d'une même entité la structure de données et les fonctions de services qui la gèrent. Simula avait été développé pour la simulation et disposait notamment d'outils génériques dont un échéancier intéressants. Smalltalk [Goldeberg83] est

né à la fin des années 70, en reprenant des concepts de Simula. C'est un langage orienté objet populaire, souvent interprété et disposant d'environnements interactifs intéressants. Dans le monde des bases de données, il a été utilisé par les concepteurs de GemStone [Maier86] comme langage de base pour définir les structures d'objets et programmer les applications.

Divers dialectes Lisp sont orientés objet, si bien que l'approche bases de données à objets est née à partir de Lisp. Orion [WonKim88] fut historiquement le premier SGBD à objets construit à partir d'un Lisp objet. Aujourd'hui, et malgré quelques détours vers des langages spécifiques [Lécluse89], la plupart des SGBD à objets sont basés sur C++ et s'orientent de plus en plus vers Java. Ils permettent de gérer des classes d'objets persistants. Ce choix est effectué essentiellement pour des raisons de performance et de popularité du langage C++, qui est une extension du langage C ; C++ est d'ailleurs traduit en C par un pré-compilateur. Quelques SGBDO supportent Smalltalk. Java est le langage objet d'avenir, supporté par la plupart des SGBDO. La très grande portabilité et la sécurité du code intermédiaire généré – le fameux *byte-code* facile a distribuer –, en font un langage de choix pour les applications client-serveur et *web* à plusieurs niveaux de code applicatif.

Ce chapitre se propose d'introduire les concepts de la modélisation objet et les opérations de base pour manipuler des objets persistants. Après cette introduction, la section 2 introduit les principes des modèles à objets en les illustrant par un modèle de référence proche de celui de l'OMG. La section 3 définit plus précisément ce qu'est un SGBDO et discute des principales techniques de gestion de la persistance proposées dans les systèmes. La section 4 propose une algèbre pour objets complexes définie sous forme de classes et permettant de manipuler des collections d'objets à un niveau intermédiaire entre un langage SQL étendu et un langage de programmation navigationnel de type C++ persistant. La conclusion résume les points abordés et introduit les problèmes essentiels à résoudre pour réaliser un SGBDO.

2. LE MODÈLE OBJET DE RÉFÉRENCE

2.1. MODÉLISATION DES OBJETS

Les modèles de données à objets ont été créés pour modéliser directement les entités du monde réel avec un comportement et un état. Le concept essentiel est bien sûr celui d'**objet**. Il n'est pas simple à définir car composite, c'est-à-dire intégrant plusieurs aspects. Dans un modèle objet, toute entité du monde réel est un objet, et réciproquement, tout objet représente une entité du monde réel.

Notion XI.1 : Objet (*Object*)

Abstraction informatique d'une entité du monde réel caractérisée par une identité, un état et un comportement.

Un objet est donc une instance d'entité. Il possède une identité qui permet de le repérer. Par exemple, un véhicule particulier V1 est un objet. Un tel véhicule est caractérisé par un état constitué d'un numéro, une marque, un type, un moteur, un nombre de kilomètres parcourus, etc. Il a aussi un comportement composé d'un ensemble d'opérations permettant d'agir dessus, par exemple créer(), démarrer(), rouler(), stopper(), détruire(). Chaque opération a bien sûr des paramètres que nous ne précisons pas pour l'instant.

L'objet de type véhicule d'identité V1 peut être représenté comme un groupe de valeurs nommées avec un comportement associé, par exemple :

V1 {
Numéro: 812 RH 94, Marque: Renault, Type: Clio, Moteur: M1 ;
créer(), démarrer(), rouler(), stopper(), détruire() }.

Une personne est aussi un objet caractérisé par un nom, un prénom, un âge, une voiture habituellement utilisée, etc. Elle a un comportement composé d'un ensemble d'opérations { naître(), vieillir() , conduire(), mourir() }. L'objet de type personne d'identité P1 peut être décrit comme suit :

P1 {
Nom: Dupont, Prénom: Jules, Age: 24, Voiture: V1 ;
naître(), vieillir(), conduire(), mourir() }.

Un objet peut être très simple et composé seulement d'une identité et d'une valeur (par exemple, un entier E1 {Valeur: 212}). Il peut aussi être très complexe et lui-même composé d'autres objets. Par exemple, un avion est composé de deux moteurs, de deux ailes et d'une carlingue, qui sont eux-mêmes des objets complexes.

À travers ces exemples, deux concepts importants apparaissent associés à la notion d'objet. Tout d'abord, un objet possède un **identifiant** qui matérialise son identité. Ainsi, deux objets ayant les mêmes valeurs, mais un identifiant différent, sont considérés comme différents. Un objet peut changer de valeur, mais pas d'identifiant (sinon, on change d'objet).

Notion XI.2 : Identifiant d'objet (*Object Identifier*)

Référence système unique et invariante attribuée à un objet lors de sa création permettant de le désigner et de le retrouver tout au long de sa vie.

L'**identité d'objet** [Khoshafian86] est un concept important : c'est la propriété d'un objet qui le distingue logiquement et physiquement des autres objets. Un identifiant d'objet est en général une adresse logique invariante. L'attribution d'identifiants

internes invariants dans les bases de données à objets s'oppose aux bases de données relationnelles dans lesquelles les données (tuples) ne sont identifiés que par des valeurs (clés). Deux objets O1 et O2 sont identiques (on note O1 == O2) s'ils ont le même identifiant ; il n'y a alors en fait qu'un objet désigné par deux pointeurs O1 et O2. L'identité d'objet est donc l'égalité de pointeurs. Au contraire, deux objets sont égaux (on note O1 = O2) s'ils ont même valeur. O1 == O2 implique O1 = O2, l'inverse étant faux.

L'identité d'objet apporte une plus grande facilité pour modéliser des objets complexes ; en particulier, un objet peut référencer un autre objet. Ainsi, le véhicule V1 référence le moteur M1 et la personne P1 référence le véhicule V1. Les graphes peuvent être directement modélisés (voir figure XI.1). Le **partage référentiel** d'un sous-objet commun par deux objets devient possible sans duplication de données. Par exemple, une personne P2 { Nom: Dupont; Prénom: Juliette; Age: 22; Voiture: V1 } peut référencer le même véhicule que la personne P1.

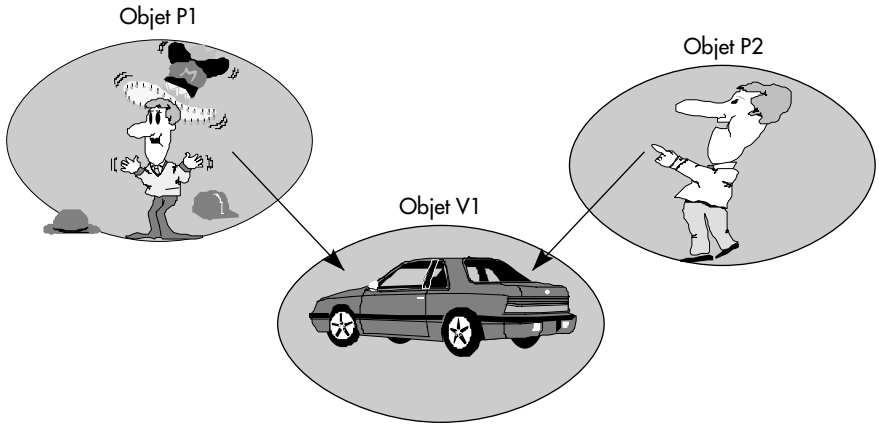


Figure XI.1 : Partage référentiel d'un objet par deux autres objets

Comme le montre ces exemples, en plus d'un identifiant, un objet possède des **attributs** aussi appelés **variables d'instance**. Un attribut mémorise une valeur ou une référence précisant une caractéristique d'un objet. La valeur peut être élémentaire (un entier, un réel ou un texte) ou complexe (une structure à valeurs multiples). La référence correspond à un identifiant d'un autre objet. Elle permet de pointer vers un autre objet avec des pointeurs invariants.

Notion XI.3 : Attribut (*Attribute*)

Caractéristique d'un objet désignée par un nom permettant de mémoriser une ou plusieurs valeurs, ou un ou plusieurs identifiants d'objets.

2.2. ENCAPSULATION DES OBJETS

Au-delà d'une structure statique permettant de modéliser des objets et des liens entre objets, les modèles à objets permettent d'encapsuler les structures des objets par des **opérations**, parfois appelées **méthodes** (en Smalltalk) ou **fonctions membres** (en C++).

Notion XI.4 : Opération (*Operation*)

Modélisation d'une action applicable sur un objet, caractérisée par un en-tête appelé signature définissant son nom, ses paramètres d'appel et ses paramètres de retour.

Le terme méthode sera aussi employé pour désigner une opération. Issu de Smalltalk, ce concept a cependant une connotation plus proche de l'implémentation, c'est-à-dire que lorsqu'on dit méthode, on pense aussi au code constituant l'implémentation. Quoi qu'il en soit, un objet est manipulé par les méthodes qui l'encapsulent et accédé via celles-ci : le **principe d'encapsulation** hérité des types abstraits cache les structures de données (les attributs) et le code des méthodes en ne laissant visible que les opérations exportées, appelées opérations **publiques**. Par opposition, les opérations non exportées sont qualifiées de **privées**. Elles ne sont accessibles que par des méthodes associées à l'objet. L'encapsulation est un concept fondamental qui permet de cacher un groupe de données et un groupe de procédures associées en les fusionnant et en ne laissant visible que **l'interface** composée des attributs et des opérations publics.

Notion XI.5 : Interface d'objet (*Object Interface*)

Ensemble des signatures des opérations, y compris les lectures et écritures d'attributs publics, qui sont applicables depuis l'extérieur sur un objet.

L'interface d'un objet contient donc toutes les opérations publiques que peuvent utiliser les clients de l'objet. Pour éviter de modifier les clients, une interface ne doit pas être changée fréquemment : elle peut être enrichie par de nouvelles opérations, mais il faut éviter de changer les signatures des opérations existantes. En conséquence, un objet exporte une interface qui constitue un véritable contrat avec les utilisateurs. Les attributs privés (non visibles à l'extérieur) et le code des opérations peuvent être modifiés, mais changer des opérations de l'interface nécessite une reprise des clients.

Ce principe facilite la programmation modulaire et l'indépendance des programmes à l'implémentation des objets. Par exemple, il est possible de développer une structure de données sous la forme d'un tableau, permettant de mémoriser le contenu d'un écran. Cette structure peut être encapsulée dans des opérations de signatures fixées permettant d'afficher, de redimensionner, de saisir des caractères. Le changement du tableau en liste nécessitera de changer le code des opérations, mais pas l'interface, et donc pas les clients.

En résumé, les opérations et attributs publics constituent l'interface d'un objet. Ceux-ci sont les seuls accessibles à l'extérieur de l'implémentation de l'objet. Le code qui constitue cette implémentation peut être structuré. Certaines opérations sont alors invisibles à l'extérieur de l'objet : elles sont appelées **opérations privées**. Par exemple, la saisie d'un texte peut s'effectuer par plusieurs appels d'une méthode saisissant une ligne : la méthode SaisirLigne restera invisible au monde extérieur et seule la méthode SaisirTexte pourra être invoquée. Dans la suite et afin de simplifier, nous supposerons que toutes les méthodes d'un objet sont publiques. Si nous ne le précisons pas, les attributs sont publiques. Mettre des attributs publics ne respecte pas très bien le principe d'encapsulation qui consiste à cacher les propriétés servant à l'implémentation. Briser ainsi l'encapsulation conduit à des difficultés si l'on veut changer par exemple le type d'un attribut : il faut alors prévenir les clients qui doivent être modifiés !

Notez qu'un attribut d'un objet peut être lu par une fonction appliquée à l'objet délivrant la valeur de l'attribut. Nous noterons cette fonction GetAttribut, où Attribut est le nom de l'attribut considéré. Ainsi, l'attribut Nom définit une fonction GetNom qui, appliquée à une personne P, délivre un texte (son nom). La propriété Voiture peut aussi être lue par une fonction GetVoiture qui, appliquée à une personne, délivre l'identifiant de l'objet constituant son véhicule habituel. Un attribut peut aussi être écrit par une fonction particulière que nous noterons PutAttribut. En résumé, tout attribut définit implicitement deux méthodes (écriture : Put, et lecture : Get) qui peuvent être privées ou publiques, selon que l'attribut est visible ou non à l'extérieur. Le formalisme unificateur des fonctions est très puissant, car il permet de raccrocher à une même théorie (les types abstraits) les propriétés mémorisées (attributs) et calculées (fonctions) d'un objet.

2.3. DÉFINITION DES TYPES D'OBJETS

Le concept de **type de données abstrait** est bien connu dans les langages de programmation [Guttag77]. Un type de données abstrait peut être vu comme un ensemble de fonctions qui cache la représentation d'un objet et contraint les interactions de l'objet avec les autres objets. Ainsi, un type de données abstrait englobe une représentation cachée d'un objet et une collection de fonctions abstraites visibles à l'extérieur. La définition d'un type de données abstrait correspond à la définition d'une ou plusieurs interfaces, comme vu ci-dessus. L'emploi de types abstraits favorise la modularité des programmes car la modification de la structure d'un objet n'affecte pas les objets extérieurs qui le manipulent.

Dans les systèmes à objets, les types abstraits (et donc les interfaces) doivent être implémentés. En général, un type abstrait est implémenté sous la forme d'un moule permettant de définir les attributs et les opérations communs associés aux objets créés selon ce moule. Un tel moule est appelé **classe**.

Notion XI.6 : Classe (*Class*)

Implémentation d'une ou plusieurs interfaces sous la forme d'un moule permettant de spécifier un ensemble de propriétés d'objets (attributs et opérations) et de créer des objets possédant ces propriétés.

Notez que certains systèmes à objets séparent la fonction de création des objets de la classe : on obtient alors des classes qui sont simplement des implémentations de types abstraits et l'on ajoute des **usines à objets** (*Object factory*) pour créer les objets. Chaque classe doit alors avoir son usine, ce qui complique le modèle.

Une classe spécifie donc la structure et le comportement communs des objets qu'elle permet de créer. Au-delà du nouveau type de données abstrait ajouté à l'environnement par une définition de classe, une classe supporte une implémentation : c'est le code des opérations. Elle donne aussi naissance à une famille d'objets : on parle alors de l'**extension de la classe**. Cette extension est une collection d'objets ayant mêmes structure et comportement. La classe est donc un peu l'analogue de la table dans les bases de données relationnelles, bien qu'une table ne permette que de modéliser la structure commune des tuples qui la composent.

En résumé, le concept de classe est plutôt complexe. Par abus de langage, le mot classe désigne généralement une intention (le type abstrait), mais aussi parfois une extension (la collection des objets membres de la classe), d'autre fois une implémentation (la structure des objets et le code des opérations). La spécification progressive des classes d'objets composant une base de données à objets permet de modéliser le comportement commun des objets de manière modulaire et extensible. Elle permet aussi de spécifier les collections contenant les objets ou extensions de classes. La plupart des systèmes distinguent l'intention de l'extension, une classe (ou un type selon le vocabulaire) pouvant avoir plusieurs extensions.

Du point de vue de la représentation d'une définition de classe, nous utiliserons la notation préconisée par UML [Rational97] et illustrée figure XI.2. Nous avons à gauche le cadre générique servant à représenter une classe, à droite le cas de la classe Vin. Chaque attribut a un nom, de même que chaque opération. Un attribut est typé. Les opérations peuvent avoir des paramètres, et un type dans le cas de fonctions. Un attribut ou une opération publics sont précédés de + ; de même pour une opération.

Nous pouvons maintenant illustrer la notion de classes par quelques exemples. La syntaxe choisie pour les définitions de classe est proche de C++, chaque attribut ou méthode ayant un type suivi d'un nom et d'éventuels paramètres. Notre premier exemple (voir Figure XI.3) vise à modéliser le plan géométrique sous forme de points. Pour créer et rassembler les points créés, nous définissons une classe comportant deux attributs, `X` et `Y`, de type flottant. Une méthode sans paramètre Distance permet de calculer la distance du point à l'origine ; cette méthode retourne un flottant. Une méthode `Translater` permet de translater un point d'un vecteur fourni en para-

mètre et ne retourne aucun paramètre (void en C++). Une autre méthode `Afficher` permet de générer un point lumineux sur un écran ; elle ne retourne aucun paramètre.

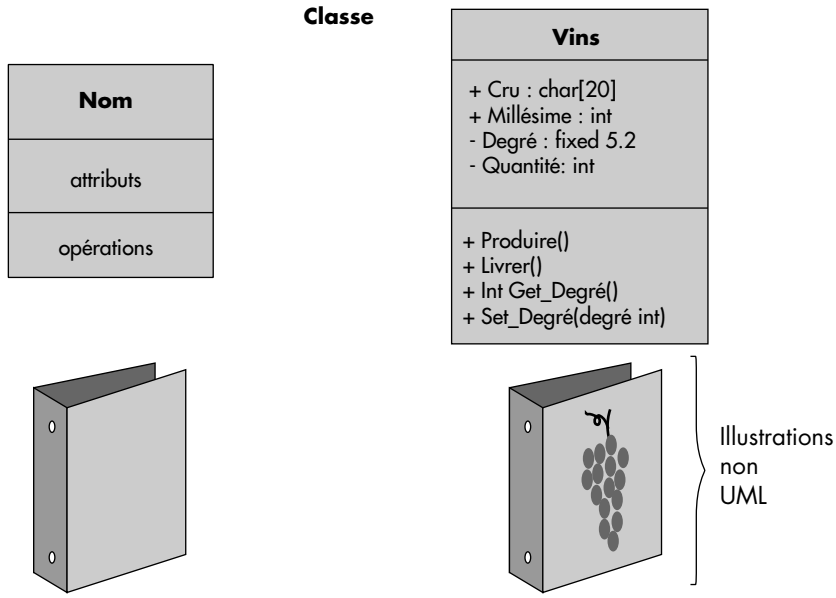


Figure XI.2 : Représentation d'une classe en UML

```
Class Point {
   Float X;
   Float Y;
   Float Distance();
   Void Translater(Float DX, Float DY);
   Void Afficher(); };
```

Figure XI.3 : La classe point

La figure XI.4 permet de définir les classes `Vin, Personne` et `Véhicule` dont quelques objets ont été vus ci-dessus. Les mots clés `Public` et `Private` permettent de préciser si les propriétés sont exportées ou non. Par défaut, il restent cachés dans la classe, donc privés. Notez que comme en C++, nous définissons une référence par le type de l'objet pointé suivi d'une étoile (par exemple `Véhicule*`). Deux méthodes sont attachées à la classe Personne : `Vieillir` qui permet d'incrémenter de 1 l'âge d'une personne et rend le nouvel âge, `Conduire` qui permet de changer le véhicule associé et ne retourne aucun paramètre. Notez que la classe `Véhicule` référence d'autres classes (`Constructeur, Propulseur`) non définies ici.

```
Class Vin {
   Fixed 5.2 degré ;
   Int quantité ;
   Public :
   Char cru [20] ;
   Int millésime ;
   void Produire()
   void Livrer()
   Int Get_Degré()
   void Set_Degré(Int degré) } ;

Class Personne {
   String Nom;
   String Prénom;
   Int Age;
   Véhicule* Voiture;
   Public :
   Int Vieillir();
   Void Conduire(Véhicule V); };


Class Véhicule {
   Public :
   String Numéro;
   Constructeur* Marque;
   String Type;
   Propulseur* Moteur; };
```

Figure XI.4 : Les classes Vin, Personne et Véhicule

Une opération définie dans le corps d'une classe s'applique à un objet particulier de la classe. Par exemple, pour translater un point référencé par `P` d'un vecteur unité, on écrira `P→Translater(1,1)`. Pour calculer la distance de P à l'origine dans une variable d, on peut écrire `d = P→Distance()`. Certaines méthodes peuvent s'appliquer à plusieurs objets de classes différentes : une telle méthode est dite **multi-classe**. Elle est en général affectée à une classe particulière, l'autre étant un paramètre. La possibilité de supporter des méthodes multiclasses est essentielle pour modéliser des associations entre classes sans introduire de classes intermédiaires artificielles.

2.4. LIENS D'HÉRITAGE ENTRE CLASSES

Afin d'éviter la répétition de toutes les propriétés pour chaque classe d'objets et de modéliser la relation « est-un » entre objets, il est possible de définir de nouvelles classes par affinage de classes plus générales. Cette possibilité est importante pour faciliter la définition des classes d'objets. Le principe est d'affiner une classe plus

générale pour obtenir une classe plus spécifique. On peut aussi procéder par mise en facteur des propriétés communes à différentes classes afin de définir une classe plus générale. La notion de **généralisation** ainsi introduite est empruntée aux modèles sémantiques de données [Hammer81, Bouzeghoub85].

Notion XI.7 : Généralisation (*Generalization*)

Lien hiérarchique entre deux classes spécifiant que les objets de la classe supérieure sont plus généraux que ceux de la classe inférieure.

La classe inférieure est appelée **sous-classe** ou **classe dérivée**. La classe supérieure est appelée **super-classe** ou **classe de base**. Le parcours du lien de la super-classe vers la sous-classe correspond à une **spécialisation**, qui est donc l'inverse de la généralisation. Une sous-classe reprend les propriétés (attributs et méthodes) des classes plus générales. Cette faculté est appelée **héritage**.

Notion XI.8 : Héritage (*Inheritance*)

Transmission automatique des propriétés d'une classe de base vers une sous-classe.

Il existe différentes sémantiques de la généralisation et de l'héritage, qui sont deux concepts associés [Cardelli84, Lécluse89]. La plus courante consiste à dire que tout élément d'une sous-classe est élément de la classe plus générale : il hérite à ce titre des propriétés de la classe supérieure. La relation de généralisation est alors une relation d'inclusion. Bien que reprenant les propriétés de la classe supérieure, la classe inférieure possède en général des propriétés supplémentaires : des méthodes ou attributs sont ajoutés.

Le concept de généralisation permet de définir un **graphe de généralisation** entre classes. Les nœuds du graphe sont les classes et un arc relie une classe C2 à une classe C1 si C1 est une généralisation de C2. Le graphe dont les arcs sont orientés en sens inverse est appelé **graphe d'héritage**. La figure XI.5 illustre un graphe de généralisation entre les classes `Personne`, `Employé`, `Cadre`, `NonCadre` et `Buveur`. Les définitions de classes correspondantes dans un langage proche de C++ apparaissent figure XI.6. Les super-classes d'une classe sont définies après la déclaration de la classe suivie de « : ». À chaque classe est associé un groupe de propriétés défini au niveau de la classe. Les classes de niveaux inférieurs héritent donc des propriétés des classes de niveaux supérieurs.

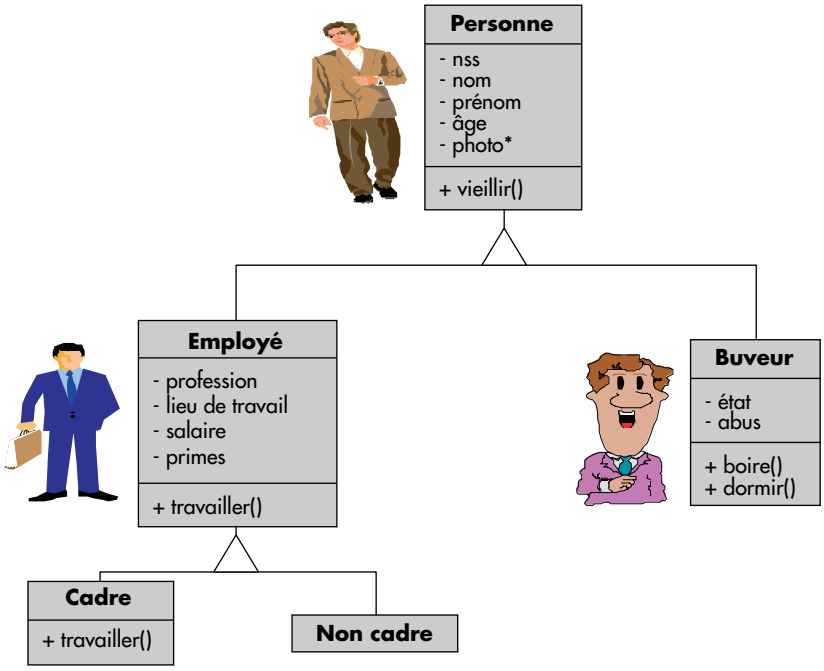


Figure XI.5 : Exemple de graphe de généralisation

```
Class Personne {
   Int nss ;
   String nom ;
   String prenom ;
   Int age ;
   Image* photo ;
Public :
   Int vieillir() ; }

Class Employé : Personne {
   String profession ;
   String lieudetravail ;
   Double salaire ;
   Double primes ;
Public :
   void travailler() ; }

Class Buveur : Personne {
   Enum etat (Normal, Ivre) ;
   List <consommation> abus ;
Public :
   Int boire() ;
   void dormir() ; }
```

```
Class Cadre : Employé {
Public :
   void travailler() ; }
Class NonCadre : Employé {
   }
```

Figure XI.6 : Définition des classes de la figure précédente

Afin d'augmenter la puissance de modélisation du modèle, il est souhaitable qu'une classe puisse hériter des propriétés de plusieurs autres classes. L'**héritage multiple** permet à une classe de posséder plusieurs super-classes immédiates.

Notion XI.9 : Héritage multiple (*Multiple Inheritance*)

Type d'héritage dans lequel une classe dérivée hérite de plusieurs classes de niveau immédiatement supérieur.

Dans ce cas, la sous-classe hérite des propriétés et opérations de toutes ses super-classes. L'héritage multiple permet de définir des classes intersection comme dans la figure XI.7, où les EmployésBuveurs héritent à la fois des Buveurs et des Employés. Ils héritent certes d'un seul nom provenant de la racine Personne.

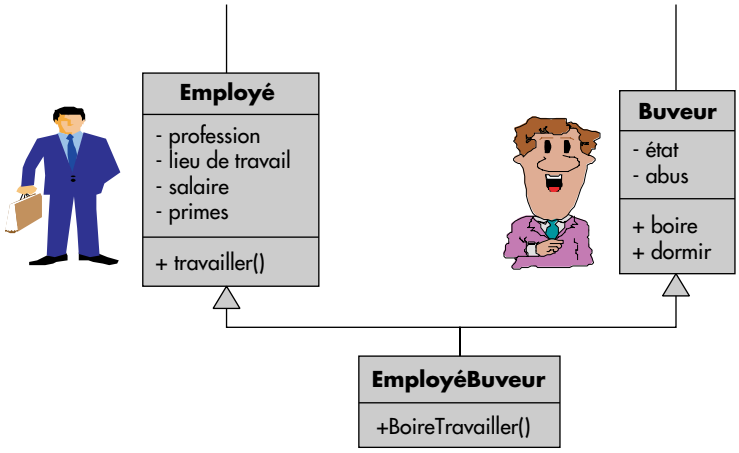


Figure XI.7 : Exemple d'héritage multiple

Des **conflits de noms** d'attributs ou d'opérations peuvent survenir en cas d'héritage multiple. Plusieurs solutions sont possibles. En particulier, un choix peut être effectué par l'utilisateur de manière statique à la définition, ou de manière dynamique pour chaque objet. Il est aussi possible de préfixer les noms de propriétés ou méthodes héritées par le nom de la classe où elles ont été définies, cela bien sûr dans le seul cas

d'ambiguïté de noms. Enfin, un choix automatique peut être fait par le système, par exemple le premier à gauche. Dans l'exemple de la figure XI.8, les triangles rectangles isocèles vont hériter de trois fonctions de calcul de surface. Deux d'entres elles sont identiques et proviennent de l'opération `surface` associée aux triangles. Il faut pouvoir choisir parmi les deux restantes, soit en conservant les deux en les renommant `polygone_surface` et `triangle_surface`, soit en en sélectionnant une, par exemple la première à gauche, donc celle des polygones droits.

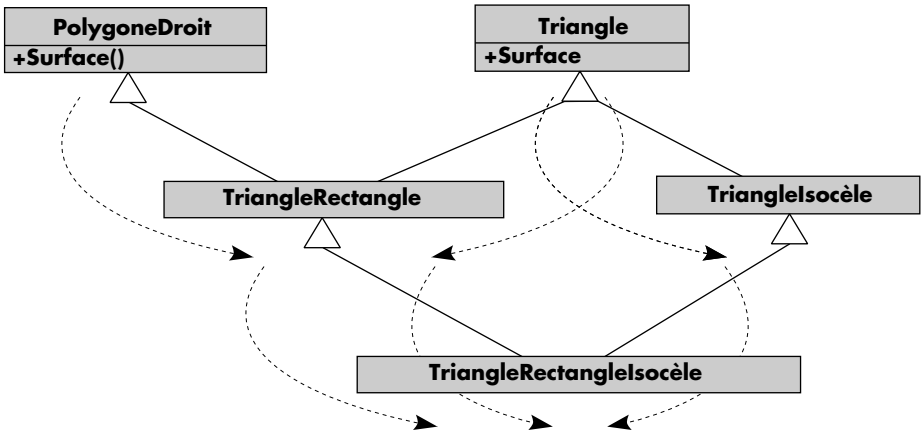


Figure XI.8 : Exemple de conflit de noms

Dire qu'une classe C1 est plus générale qu'une classe C2 permet de définir une relation d'ordre C2 $\subseteq$ C1, qui représente le fait que les instances de C2 sont incluses dans celles de C1. Cette relation d'ordre correspond donc à une inclusion au niveau des ensembles d'objets. La relation d'ordre ainsi définie permet de considérer le treillis des classes. Ce treillis possède en général un plus grand élément qui est la classe dénommée `Objet` (*Object*) ; cette classe apparaît alors comme la racine de la hiérarchie d'héritage qui fournit toutes les méthodes de base de manipulation des objets, méthodes héritées par toutes les autres classes. La hiérarchie des types et l'héritage de propriétés ont été formellement étudiés dans le contexte des types abstraits sous forme de sigma-algèbres [Guogen78], et plus récemment dans le contexte de la programmation objet [Castagna96].

2.5. POLYMORPHISME, REDÉFINITION ET SURCHARGE

Lors de la définition d'une hiérarchie de classes, certaines propriétés peuvent être spécifiées différemment pour chaque sous-classe. L'idée est de modéliser dans une sous-

classe une fonctionnalité similaire de même signature d'opération, mais avec un code différent. On parle alors de **redéfinition**.

Notion XI.10 : Redéfinition (*Overriding*)

Spécification d'une méthode existante dans une super-classe au niveau d'une sous-classe, avec une implémentation différente.

Par exemple, une méthode globale telle que `Travailler` peut être spécifiée au niveau de la classe `Employé`, puis redéfinie de manière spécifique au niveau de la classe `Cadre`. En effet, la méthode `Travailler` de la classe `Employé` pourra être redéfinie par un code correspondant à un travail de direction au niveau de la classe `Cadre`. Il est même possible de ne pas définir le code de l'opération au niveau d'une classe et d'imposer de le définir au niveau des sous-classes : on parle alors de **méthode virtuelle** au niveau de la super-classe.

Pour une même classe, il est aussi possible de définir plusieurs implémentations pour une même opération. Chacune d'elle sera alors distinguée par des paramètres différents. Ceci correspond à la notion de **surcharge**.

Notion XI.11 : Surchage (*Overloading*)

Possibilité de définir plusieurs codes pour une même opération d'une classe, le code approprié étant sélectionné selon le type des paramètres fournis lors d'un appel.

La possibilité de surcharge rend nécessaire le choix du code d'une méthode en fonction de ses arguments. Cette possibilité est d'ailleurs généralisée : une opération d'un nom donné peut avoir différentes signatures; un code est alors attaché à chaque signature. Par exemple, au niveau de la classe `Personne`, il est possible de définir plusieurs méthodes `Vieillir`, l'une sans paramètre ajoutant un à l'âge, l'autre avec un paramètre indiquant le nombre d'années à ajouter. Chaque signature aura alors un code spécifique. Cette possibilité est souvent utilisée pour gérer des valeurs par défaut de paramètres.

Redéfinition et surcharge sont deux formes de **polymorphisme**. Cette fonctionnalité importante peut être définie comme suit :

Notion XI.12 : Polymorphisme (*Polymorphism*)

Faculté pour une opération d'avoir différentes signatures avec un code spécifique attaché à chaque signature.

Le polymorphisme permet donc à une même opération de s'appliquer à des objets de différentes classes ou à des objets d'une même classe. Dans ce dernier cas, les paramètres de l'opération doivent être de types différents. En particulier, certains peuvent ne pas exister.

Par exemple, la méthode `Travailler()` peut être appliquée à un objet de type `Cadre` ou `NonCadre` sans argument. Une méthode `Travailler(Int Durée)` pourra être définie au niveau des `NonCadre`. Lors d'un appel de la méthode `Travailler` du type `P→Travailler()` ou `P→Travailler(10)` (P référence une personne particulière), un code différent sera exécuté, selon que P est un cadre ou non, et selon qu'un paramètre est passé ou non pour un non cadre.

Le polymorphisme (étymologiquement, la faculté de posséder plusieurs formes) est une fonctionnalité évoluée qui implique en général le choix du code de la méthode à l'exécution (lorsque le type des paramètres est connu). Ce mécanisme est appelé **liaison dynamique** (en anglais, *dynamic binding*).

2.6. DÉFINITION DES COLLECTIONS D'OBJETS

Certains attributs peuvent être multivalués. Par exemple, un livre peut avoir plusieurs auteurs. Un texte est composé de plusieurs paragraphes. Il est donc souhaitable de pouvoir regrouper plusieurs objets pour former un seul attribut. Ceci s'effectue généralement au moyen de classes génériques permettant de supporter des **collections** d'objets.

Notion XI.13 : Collection (*Collection*)

Container typé désigné par un nom, contenant des éléments multiples organisés selon une structure particulière, auxquels on accède par des opérations spécifiques au type du container.

Les principales collections sont :

– l'**ensemble** (Set) qui permet de définir des collections non ordonnées sans double ;

– le **sac** (Bag) qui permet de définir des collections non ordonnées avec doubles;

– la **liste** (List) qui permet de définir des collections ordonnées avec doubles ;

– le **tableau** (Array) qui permet de définir des collections ordonnées et indexées.

D'autres sont utilisées, par exemple la pile, le tableau insérable où il est possible d'insérer un élément à une position donnée, etc. Les éléments rangés dans les collections sont en général des objets ou des valeurs. Par exemple {10, 20, 30, 75} est un ensemble d'entiers ; <O1, O5, O9> où les Oi représentent des objets est une liste d'objets.

Dans les systèmes objet, la notion de collection est souvent réalisée par des **classes paramétrées**, encore appelées **classes génériques** ou **patterns de classes**.

Notion XI.14 : Classe paramétrée (*Template*)

Classe avec paramètres typés et ayant différentes implémentations selon le type de ces paramètres.

La génération des implémentations selon le type des paramètres est généralement à la charge du compilateur du langage objet de définition de classe paramétrée. L'intérêt d'utiliser des classes paramétrées est la facilité de réutilisation, dans des contextes différents selon les paramètres. L'inconvénient est souvent l'accroissement de la taille du code généré.

Une classe paramétrée est donc une classe possédant un ou plusieurs paramètres ; dans le cas des collections, le paramètre sera souvent le type des objets de la collection. La collection doit donc être homogène. Elle apparaît à son tour comme une classe. Par exemple `List<X>` et `Set<X>` sont des collections, respectivement des listes et des ensembles d'objets de type X. `Set<Int>` est un ensemble d'entiers alors que `List<Vin>` est une liste de vins. `List<X>` et `Set<X>` sont des classes paramétrées. `List<Vin>` étant une classe, `Set<List<Vin>>` est un type de collection valide. Les collections permettent donc de construire des objets complexes par imbrications successives de classes. Par exemple, { <V1, V5, V7>, <V3, V2, V1>, <V4, V7>, <>} est une collection de type `Set<List<Vin>>`.

Dans la plupart des modèles à objets, les collections sont des classes d'objets génériques qui offrent des méthodes d'accès spécifiques (par exemple, le parcours d'une liste ou d'un ensemble). Elles peuvent être organisées en hiérarchie de généralisation. La figure XI.9 illustre une bibliothèque de classes génériques de type collections avec, pour chacune, des méthodes caractéristiques [Gardarin94]. Outre les méthodes d'insertion, suppression et obtention d'un élément, la classe Collection offre des méthodes de second ordre travaillant sur plusieurs objets d'une collection ; `Count` compte le nombre d'objets, `Apply` applique une fonction passée en paramètre à chaque objet et `Aggregate` calcule une fonction d'agrégat passée en paramètre (par exemple, somme ou moyenne). `Search` recherche un élément donné, par exemple par une clé. Les opérations des sous-classes sont classiques. Par exemple, `First` et `Next` permettent le parcours de liste, `Tail` extrait la sous-liste après l'élément courant et `Append` ajoute un élément. D'autres fonctions peuvent être envisagées.

On écrira par exemple `Array<Véhicule>` pour spécifier un tableau de Véhicules de dimension variable. Notez que la plupart des langages orientés objets supportent des tableaux de dimension fixe par la notation C[N], où C est une classe et N un entier. Par exemple, `Véhicule[4]` définit un tableau de 4 véhicules. Comme cela a déjà été dit, il est possible d'imbriquer des collections. L'exemple de la figure XI.10 montre comment on peut gérer des listes de tableau afin de structurer intelligemment un texte. Celui-ci est composé d'une liste de paragraphes, chacun ayant un thème, des détails structurés sous forme d'une liste de phrases, et une conclusion. Thème et conclusion sont des phrases. Une phrase est une liste de mots modélisés par des tableaux de caractères.

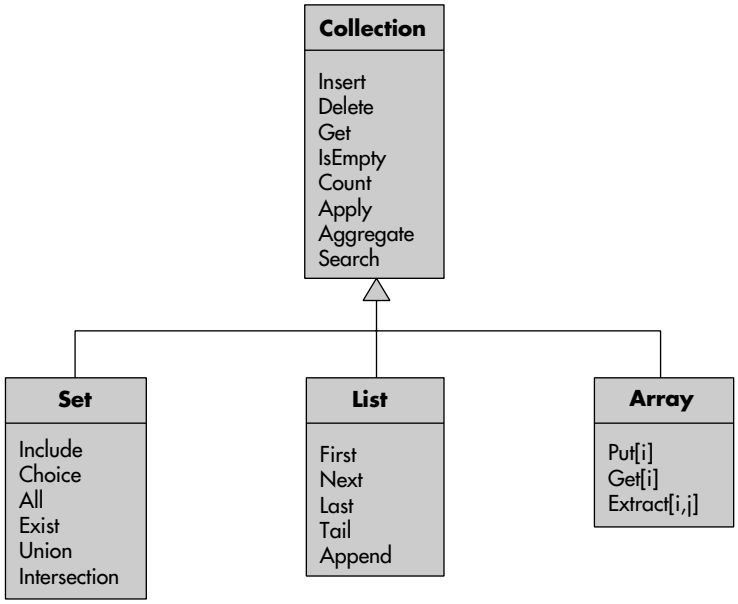


Figure XI.9 : Exemple de bibliothèque de collections avec opérations

```
class Phrase {
   List <Array <char>> Séquence; };
class Paragraphe {
   Phrase Thème;
   List <Phrase> Détails;
   Phrase Conclusion; };
class Texte {
   List <Paragraphe*> Contenu; };
```

Figure XI.10 : Utilisation de collections imbriquées

L'existence de collections imbriquées traduit le fait que certains objets sont inclus dans d'autres. Il s'agit en fait d'une représentation de la relation d'**agrégation** entre classes.

Notion XI.15 : Agrégation (*Aggregation*)

Association entre deux classes exprimant que les objets de la classe cible sont des composants de ceux de la classe source.

L'agrégation traduit la relation « fait partie de » ; par exemple, les caractères font partie de la phrase. Les collections permettent d'implémenter les agrégations multiva-

luées. L'agrégation peut être implémentée par une référence (par exemple `List<Paragraphe*>`) ou par une imbrication, selon que l'on désire ou non partager les objets inclus. La représentation des agrégations par un graphe permet de visualiser le processus de construction des objets complexes. La figure XI.11 représente le graphe d'agrégation, encore appelé graphe de composition, de la classe Texte. Les cardinalités multiples sont mentionnées par des étoiles (*), conformément à la notation UML [Rational97].

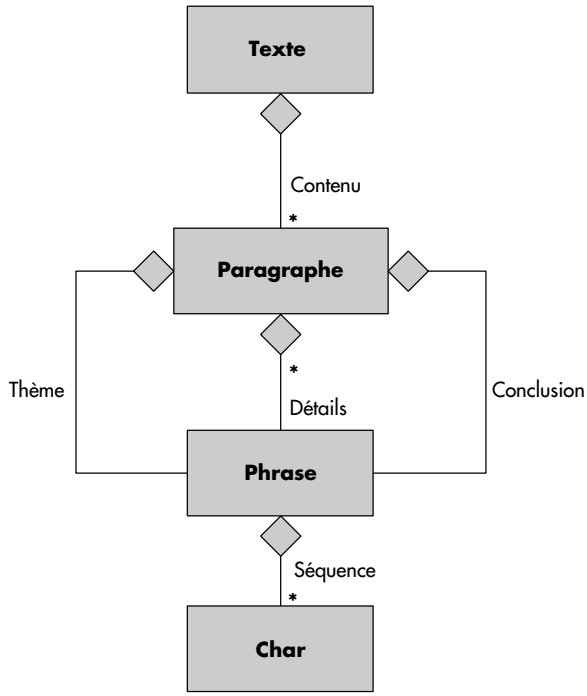


Figure XI.11 : Graphe d'agrégation de la classe Texte

2.7. PRISE EN COMPTE DE LA DYNAMIQUE

Dans les environnements à objets, les objets communiquent entre eux par des **messages** comportant le nom d'une méthode et ses paramètres. Un objet reçoit un message et réagit à un message. L'envoi de messages est en fait une implémentation flexible et contrôlée de l'appel de procédure par valeur classique des langages de programmation. C'est un élément important des langages objet qui permet de réaliser le polymorphisme lors de l'interprétation du message.

Notion XI.16 : Message (*Message*)

Bloc de paramètres composé d'un objet récepteur, d'un nom d'opérations et de paramètres, permettant par envoi l'invocation de l'opération publique de l'objet récepteur.

Ainsi, un objet réagit à un message en sélectionnant le code de la méthode associée selon le nom de la méthode et le type des paramètres, réalisant ainsi le polymorphisme. Il retourne les paramètres résultats de l'opération. Une base de données à objets ou un programme structuré en objets apparaît donc comme un ensemble d'objets vivants qui communiquent par des messages. Un dispatcher de messages fait transiter les messages entre les objets et assure leur bonne délivrance. Un nouveau graphe peut être construit dynamiquement, le **graphe des appels de méthodes**, qui permet de visualiser quelle méthode a appelé quelle autre méthode.

Un message peut être envoyé à un objet désigné par son identifiant en utilisant le **sélecteur de propriété** dénoté →, déjà vu ci-dessus. Par exemple, un point défini comme figure XI.3, référencé par P, pourra recevoir les messages :

P→Distance();
P→Translater(10,10);
P→Afficher().

Le dispatcher du message activera le code de l'opération sélectionnée en effectuant le passage de paramètre par valeur.

2.8. SCHÉMA DE BASES DE DONNÉES À OBJETS

À partir des outils et concepts introduits ci-dessus, une base de données à objets peut être décrite. La description s'effectue par spécification d'un **schéma de classes** composé de définitions de classes, chacune d'elles comportant des attributs (éventuellement organisés en collections) et des opérations. Parmi les classes, certaines sont plus générales que d'autres, comme spécifié par le graphe de généralisation. La figure XI.12 représente le schéma partiel d'une base de données à objets dont d'autres éléments ont été introduits ci-dessus. Le graphe de référence entre classes correspondant est représenté figure XI.14

Notion XI.17 : Schéma de BD objet (*Object DB Schema*)

Description d'une base de données à objets particulière incluant les définitions de classes, d'attributs et d'opérations ainsi que les liens entre classes.

Les agrégations, cas particuliers d'associations, sont bien sûr représentées. Un schéma de BD objet est donc un schéma objet général. Il pourra en plus inclure les directions de traversée des associations, afin de faciliter le passage à des références entre objets au niveau de l'implémentation. La figure XI.12 représente un schéma de BD objet sous une forme proche de la notation UML ; la figure XI.13 en donne une vue selon une syntaxe proche de C++.

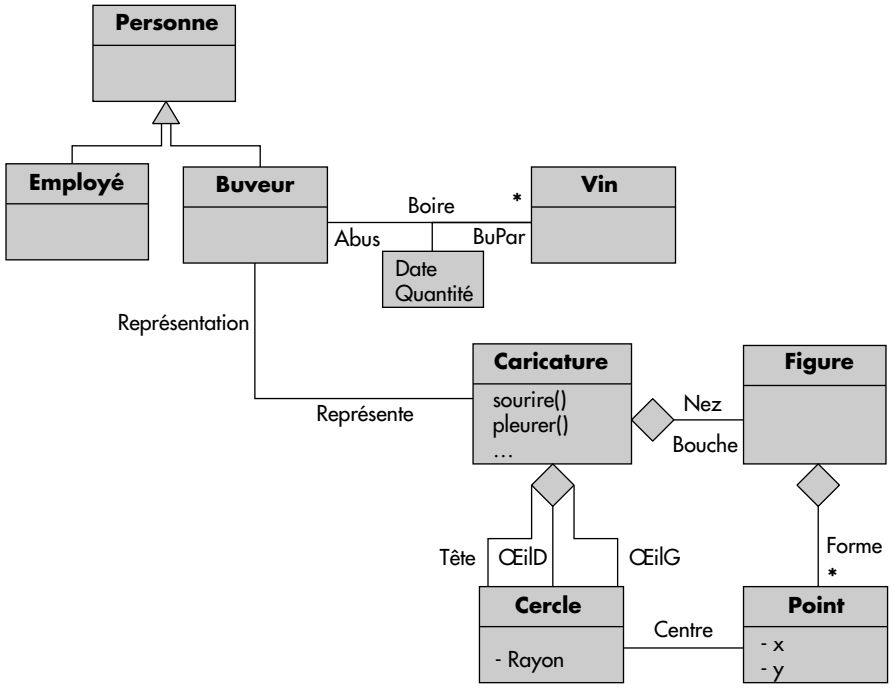


Figure XI.12 : Schéma graphique partiel de la base exemple

```
Class Vin {
public:
   Char Cru[20] ;
   Int Mill ;
   Double Degré ; }

Class Buveur {
private:
   Enum Etat (Normal, Ivre) ;
public:
   Char Nom[15] ;
   Char Prénom[15] ;
   List <Date Jour, Vin* Boisson, Int Quantité> Abus ;
   Caricature* Représentation ;
   // opération sur buveur
   void Boire (Date d, Vin* v, Int q) ;
   void Dormir (Int durée); }

Class Caricature {
   Cercle* Tête ;
   Cercle* OeilD ;
   Cercle* OeillG ;
   Figure* Nez ;
```

```
   Figure* Bouche ;
   // operations sur personnage
   void Sourire () ;
   void Pleurer () ;
   Graphic Dessiner (Int Echelle) ; }

Class Cercle {
   Point Centre ;
   Double Rayon ; }

Class Figure {
   List <Point> Forme ; }
```

Figure XI.13 : Définition C++ d'une partie de la base exemple

Soulignons qu'un schéma de classes modélise à la fois la structure (les attributs des classes) et le comportement (les méthodes) des objets. La définition complète d'une base de données à objets nécessite donc un langage de programmation afin de spécifier le code des programmes. Ce langage est un langage orienté objet, du type Smalltalk, ou de plus en plus souvent C++ ou Java.

3. LA PERSISTANCE DES OBJETS

Après avoir précisé le concept de base de données à objets, cette section étudie les diverses techniques utilisées pour réaliser la persistance des objets nécessaire à l'approche base de données. La dernière partie montre comment on navigue dans une base de données à objets depuis un langage orienté objet tel C++.

3.1. QU'EST-CE-QU'UNE BD À OBJETS ?

La notion de bases de données à objets s'est précisée au début des années 90. Nous avons introduit ci-dessus les concepts essentiels de la modélisation orientée objet, tels qu'ils sont implémentés dans des langages objet Smalltalk, C++ ou Java. À partir de là, le concept de base de données à objets s'élabore en ajoutant la persistance. La notion de système de base de données à objets étant au départ très confuse, Atkinson, Bancilhon, Dewitt, Ditrich, Maier et Zdonick ont proposé de la clarifier dans une publication à la conférence DOOD (*Deductive and Object-Oriented Databases*) en 1989, intitulée « *The Object-Oriented Manifesto* » [Atkinson89].

Pour mériter le nom de SGBD objet (SGBDO), un système doit d'abord supporter les fonctionnalités d'un SGBD. Pour ce faire, il doit obligatoirement assurer :

– La **persistance des objets**. Tout objet doit pouvoir persister sur disque au-delà du programme qui le crée. Un objet peut être **persistant** ou **transient**. Dans le deuxième cas, sa durée de vie est au plus égale à celle du programme qui le crée ; il s'agit d'un objet temporaire qui reste en mémoire.

Notion XI.18 : Objet persistant (*Persistent object*)

Objet stocké dans la base de données dont la durée de vie est supérieure au programme qui le crée.

Notion XI.19 : Objet transient (*Transient object*)

Objet restant en mémoire, dont la durée de vie ne dépasse pas celle du programme qui le crée.

– La **concurrence d'accès**. La base d'objets doit pouvoir être partagée simultanément par les transactions qui la consultent et la modifient ; les blocages doivent être minimaux afin d'assurer la cohérence de la base.
– La **fiabilité des objets**. Les objets doivent être restaurés en cas de panne d'un programme dans l'état où ils étaient avant la panne. Les transactions doivent être atomiques, c'est-à-dire totalement exécutées ou pas du tout.
– La **facilité d'interrogation**. Il doit être possible de retrouver un objet à partir de valeurs de ses propriétés, qu'il s'agisse de valeurs d'attributs, de résultats de méthodes appliquées à l'objet ou de liens avec les objets référencés ou référençants.

Le manifesto propose en outre des fonctionnalités bases de données optionnelles :

– La **distribution des objets**. Cette facilité permet de gérer des objets sur différents sites, en particulier sur un serveur et des clients.
– Les **modèles de transaction évolués**. Il s'agit de supporter des transactions imbriquées, c'est-à-dire elles-mêmes décomposées en sous-transactions qui peuvent être totalement reprises.
– Les **versions d'objets**. La gestion de versions permet de revenir à un état antérieur de l'objet avant modification. À partir d'un objet, plusieurs versions peuvent être créées par des modifications successives ou parallèles. On aboutit ainsi à un graphe des versions d'un objet qui peut être géré par le système. Il est alors possible de remonter aux versions précédentes à partir des dernières versions et vice versa. Si plusieurs versions sont créées en parallèle, une fusion avec possibilité de choisir certaines modifications est ultérieurement nécessaire. Un objet pouvant posséder des versions est appelé **objet versionnable**.

Notion XI.20 : Objet à versions (*Versionnable object*)

Objet dont l'historique des instances créées (successivement ou simultanément) est gardé dans la base sous forme de versions consultables et modifiables.

Outre les fonctionnalités orientées bases de données définies ci-dessus, le manifesto prescrit les fonctionnalités orientées objets que doit supporter un SGBDO. Sont obligatoires :

– Le **support d'objets atomiques et complexes**. Il s'agit de supporter des objets avec des attributs références et des collections imbriquées.
– L'**identité d'objets**. Tout objet doit avoir un identifiant système invariant, permettant de le retrouver sur disque et en mémoire.
– L'**héritage simple**. Une classe doit pouvoir être une spécialisation d'une autre classe et hériter de celle-ci.
– Le **polymorphisme**. Le code d'une méthode doit être choisi en fonction de ses paramètres instanciés.

Deux fonctionnalités sont optionnelles :

– L'**héritage multiple**. Il permet qu'une sous-classe soit la spécialisation directe de deux sur-classes ou plus. Elle hérite alors de toutes ses sur-classes.
– Les **messages d'exception**. Il s'agit d'un mécanisme de traitement d'erreur analogue à celui introduit en C++ ou en Java. Lorsqu'une erreur survient dans une méthode, un message d'exception est levé. Il peut être repris par un traitement d'erreur inséré par exemple dans un bloc de reprise, par une syntaxe du type : `try <traitement normal> catch <exception : traitement d'erreur>`.

En résumé, le manifesto essaie de définir précisément ce qu'est une base de données à objets. S'il a apporté en son temps une clarification, il manque aujourd'hui de précision, si bien qu'un SGBD objet-relationnel, c'est-à-dire un SGBD relationnel étendu avec des types abstraits, peut souvent être classé comme un SGBDO. Il présente aussi un peu d'arbitraire dans la sélection de fonctionnalités, notamment au niveau des options.

3.2. GESTION DE LA PERSISTANCE

Un modèle de données orienté objet permet de définir les types des objets. Dans les environnements de programmation, les objets doivent être construits et détruits en mémoire par deux fonctions spécifiques, appelées **constructeur** et **destructeur**.

Notion XI.21 : Constructeur d'objet (*Object constructor*)

Fonction associée à une classe permettant la création et l'initialisation d'un objet en mémoire.

Notion XI.22 : Destructeur d'objet (*Object destructor*)

Fonction associée à une classe permettant la destruction d'un objet en mémoire.

En C++ ou en Java, le constructeur d'un objet est une fonction membre de la classe. Il fait en général appel à une fonction de réservation de mémoire et à des fonctions d'initialisation. Les constructeurs sont normalement définis par le programmeur, mais C++ et Java insèrent un constructeur minimal dans les classes qui n'en possèdent pas. Le nom du constructeur est le nom de la classe. Par exemple, un point origine pourra être défini comme suit :

```
Point Origine(0,0).
```

Le compilateur génère alors automatiquement l'appel au constructeur Point(0,0) lors de la rencontre de cette déclaration.

En C++, le destructeur est une fonction membre notée par le nom de la classe précédé de ~ ou appelé explicitement par `delete`. Par exemple, la destruction de l'origine s'effectue par:

```
~Point(0,0).
```

Le destructeur libère la place mémoire associée à l'objet. Il peut être fourni par le programmeur. Certains langages orientés objet tels Java et Smalltalk sont munis d'un ramasse-miettes détruisant automatiquement les objets non référencés, si bien que le programmeur n'a pas à se soucier d'appeler le destructeur d'objets.

Le problème qui se pose dans les SGBDO est d'assurer la persistance des objets sur disques pour pouvoir les retrouver ultérieurement. En effet, constructeur et destructeur d'objets ne font que construire et détruire les objets en mémoire. Une solution couramment retenue pour sauvegarder les objets sur disques consiste à donner un nom à chaque objet persistant et à fournir une fonction permettant de faire persister un objet préalablement construit en mémoire. Cette fonction peut être intégrée ou non au constructeur d'objet. Sa signature pourra être la suivante :

```
// Rendre persistant et nommer un objet pointé.
Oid = Persist(<Nom>, <Ref>);
```

Nom est le nom attribué à l'objet qui permettra ultérieurement de le retrouver (dans le même programme ou dans un autre programme). Ref est la référence en mémoire de l'objet. Oid est l'identifiant attribué à l'objet dans la base par le SGBDO.

Un objet persistant pourra ensuite être retrouvé à partir de son nom, puis monté en mémoire à partir de son identifiant d'objet (adresse invariante sur disque). On trouvera donc deux fonctions plus ou moins cachées au programmeur dans les SGBD à objets :

```
// Retrouver l'oid d'un objet persistant à partir du nom.
Oid = Lookup(<Nom>);

// Activer un objet persistant désigné par son oid.
Ref = Activate(<Oid>);
```

Un objet actif en mémoire pourra être désactivé (réécriture sur disque et libération de la mémoire) par une fonction du style:

```
// Désactiver un objet persistant actif.
Oid = DesActivate(<Ref>);
```

Finalement, il sera aussi possible de rendre non persistant (détruire) dans la base un objet persistant à partir de son identifiant d'objet ou de son nom, en utilisant l'une des fonctions suivantes :

```
// Supprimer un objet persistant désigné par son nom
Void UnPersist(<Nom>);

// Supprimer un objet persistant désigné par son identifiant
Void UnPersist(<Oid>);
```

Une telle bibliothèque de fonction peut être utilisée directement par le programmeur dans un langage orienté objet comme C++ ou Java pour gérer manuellement la persistance des objets. De plus, des fonctions de gestion de transactions devront être intégrées afin d'assurer les écritures disques, la gestion de concurrence et de fiabilité. Les écritures peuvent être explicites par une fonction `Put` ou implicites lors de la validation, en fin de transaction. Un tel système est voisin de celui offert par Objective-C pour la persistance des objets dans des fichiers. Il constitue un niveau minimal de fonctionnalités nécessaire à la gestion de la persistance que nous qualifierons de **persistance manuelle**. Ce niveau de fonctions peut être caché au programmeur du SGBDO par l'une des techniques étudiées ci-dessous.

Dans tous les cas, un problème qui se pose lors de l'écriture d'un objet dans la base est la sauvegarde des pointeurs vers les autres objets persistants. Comme lors de l'activation d'un objet persistant, il faut restaurer les pointeurs en mémoire vers les autres objets actifs. Des solutions sont proposées par chacune des techniques de persistance décrites ci-dessous. Notez que les systèmes implémentent parfois des techniques de persistance mixtes, qui empruntent un peu aux deux décrites ci-dessous.

3.3. PERSISTANCE PAR HÉRITAGE

La **persistance par héritage** permet de cacher plus ou moins complètement au programmeur les mouvements d'objets entre la base et la mémoire. L'idée est de profiter de l'héritage pour assurer la persistance automatiquement. Le système offre alors une classe racine des objets persistants, nommée par exemple `PObject` (voir figure XI.14). Cette classe intègre au constructeur et destructeur d'objets des appels aux fonctions `Persist` et `Unpersist` vues ci-dessus. Ainsi, tout objet appartenant à une classe qui hérite de `PObject` est persistant. En effet, il hérite du constructeur d'objet qui le rend persistant. Il sera détruit sur disque par le destructeur, qui fait appel à la fonction `Unpersist`. La qualité d'être persistant ou non dépend alors du type de l'objet : un objet est persistant si et seulement s'il est du type d'une sous-classe de `PObject`. On dit alors que la persistance n'est pas orthogonale au type.

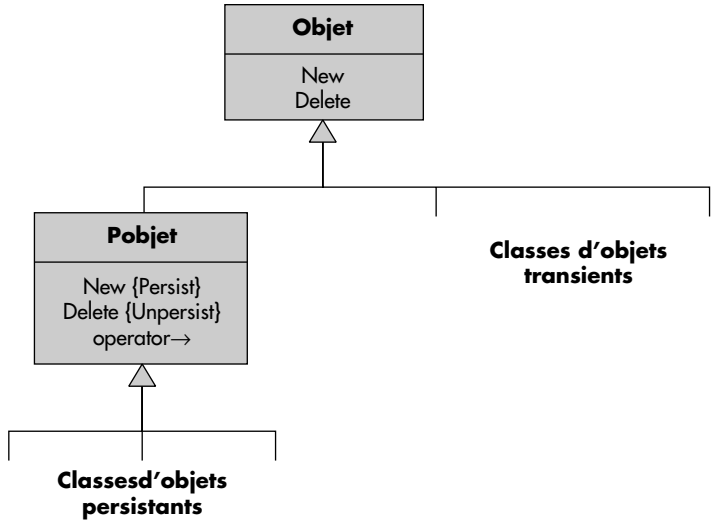


Figure XI.14 : Persistance par héritage

Outre les constructeurs et destructeurs destinés à gérer l'objet sur disque, la technique de persistance par héritage surcharge en général le parcours de référence (opérateur → en C++). Cela permet d'activer automatiquement un objet pointé par un objet déjà actif lors du premier parcours de la référence, en utilisant une technique de mutation de pointeur, comme nous le verrons ci-dessous.

En résumé, nous définirons la persistance par héritage comme suit :

Notion XI.23 : Persistance par héritage (*Persistance by inheritance*)

Technique permettant de définir la qualité d'un objet à être persistant par héritage d'une classe racine de persistance, rendant invisible l'activation et la désactivation des objets.

Cette technique présente l'avantage d'être simple à réaliser. Cependant, elle n'assure pas l'orthogonalité de la persistance aux types de données, si bien que tout type ne peut pas persister. Si l'on veut avoir dans une même classe des objets persistants et transients (par exemple des personnes persistantes et des personnes transientes), on est conduit à dupliquer les classes. Ceci peut être évité en marquant les objets non persistants d'une classe persistante simplement par un booléen. La performance de la technique de persistance par héritage est discutée, la surcharge des opérateurs (constructeurs et parcours de références) pouvant être coûteuse.

3.4. PERSISTANCE PAR RÉFÉRENCE

Une autre technique possible pour cacher les mouvements d'objets est la **persistance par référence**. L'idée est que tout objet ou variable peut être une racine de persistance

à condition d'être déclaré comme tel, puis que tout objet pointé par un objet persistant est persistant (voir figure XI.15). En général, les objets persistants sont les objets nommés, nom et persistance étant déclarés dans la même commande de création d'objet persistant. Cela conduit à ajouter un mot clé « persistant » ou « db » au langage de programmation (par exemple C++) et donc nécessite un précompilateur qui génère les appels aux fonctions `Persist`, `Unpersist`, `Activate`, etc. Par exemple, un employé pourra être créé persistant par la déclaration :

```
Employe* emp = new persistant Employe("Toto");
```

De même, une simple variable x pourra être rendue persistante par la déclaration :

```
persistant int x;
```

Au vu de ces déclarations, le précompilateur générera les commandes de persistance et de recherche nécessaires. Tout objet racine de persistance (donc déclaré persistant) sera répertorié dans un catalogue où l'on retrouvera son nom et son oid.

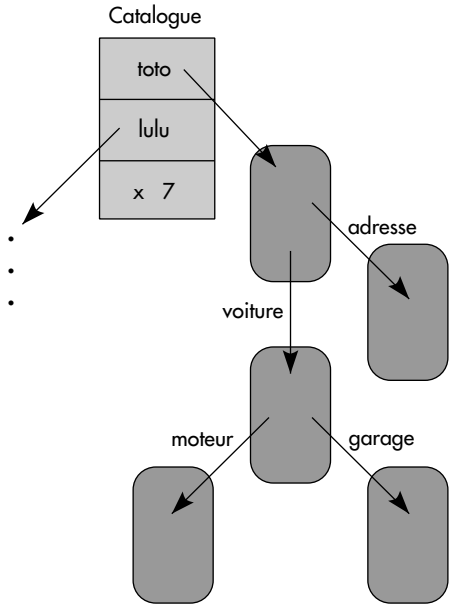


Figure XI.15 : Persistance par référence

Tout objet référencé par un objet persistant sera persistant. Là encore, le pré-compilateur devra assurer la génération des appels aux fonctions de persistance lors des assignations ou des parcours de références. Les références devront aussi être rendues persistantes par l'une des techniques que nous allons étudier ci-dessous.

En résumé, nous définirons la persistance par référence comme suit :

Notion XI.24 : Persistance par référence (*Persistence by reference*)

Technique permettant de définir la qualité d'un objet à être persistant par attribution d'un nom (pour les racines de persistance) ou par le fait qu'il soit référencé par un objet persistant.

Cette technique présente l'avantage de l'orthogonalité de la persistance au type de données, si bien que toute donnée peut être rendue persistante. Elle permet de gérer des graphes d'objets persistants. Ainsi, un arbre d'objet est rendu persistant simplement en nommant la racine. Il est rendu transient en supprimant ce nom.

3.5. NAVIGATION DANS UNE BASE D'OBJETS

Quelle que soit la technique de persistance, les objets persistants se référencent, et il faut pouvoir retrouver un objet puis naviguer vers les objets référencés.

Notion XI.25 : Navigation entre objets (*Object navigation*)

Parcours dans une base d'objets par suivi de pointeurs entre objets.

Dans les langages objet comme C++, la navigation s'effectue en mémoire par simple décodage de pointeurs. Dans une base d'objets, les choses ne sont pas aussi simples : un objet pointant sur un autre objet est en effet écrit dans la base, puis relu par un autre programme plus tard. L'objet pointé n'est alors plus présent en mémoire (voir figure XI.16). Le problème soulevé est donc de mémoriser de manière persistante les chaînages d'objets sur disques, puis de les décoder en mémoire de manière efficace.

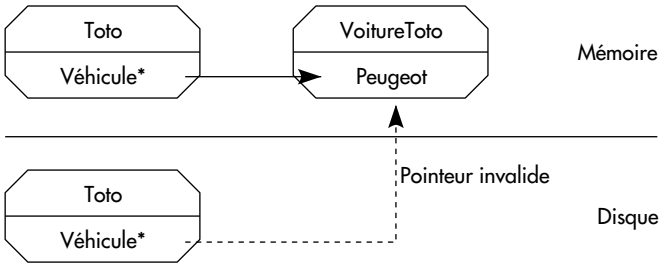


Figure XI.16 : Nécessité de mettre à jour les pointeurs en mémoire

En résumé, il faut pouvoir utiliser des identifiants d'objets comme des pointeurs sur disques et des adresses en mémoire. Le passage d'un type de pointeur à l'autre est appelé **mutation de pointeurs**.

Notion XI.26 : Mutation de pointeurs (*Pointer swizzling*)

Transformation consistant à passer de pointeurs disques à des pointeurs mémoire lors de la première navigation en mémoire via un objet et inversement lors de la dernière.

La mutation de pointeurs disques sous forme d'identifiants à pointeurs en mémoire peut s'effectuer par différentes techniques : utilisation de doubles pointeurs ou utilisation de mémoire virtuelle.

L'**utilisation de doubles pointeurs** consiste à remplacer les références mémoire par des couples `<oid-ref>` (voir figure XI.17). Lors du chargement d'un objet, toutes les parties ref des couples sont mises à 0. Lors du parcours d'une référence (opérateur →), si la partie `ref` est à 0, l'objet référencé est activé à partir de son oid et la référence est ensuite chargée avec l'adresse de l'objet en mémoire. Lors des accès suivants, l'adresse mémoire est directement utilisée. Ainsi, un objet référencé par un objet actif (par exemple retrouvé par `lookup`) est activé automatiquement lors de la première traversée du pointeur référençant.

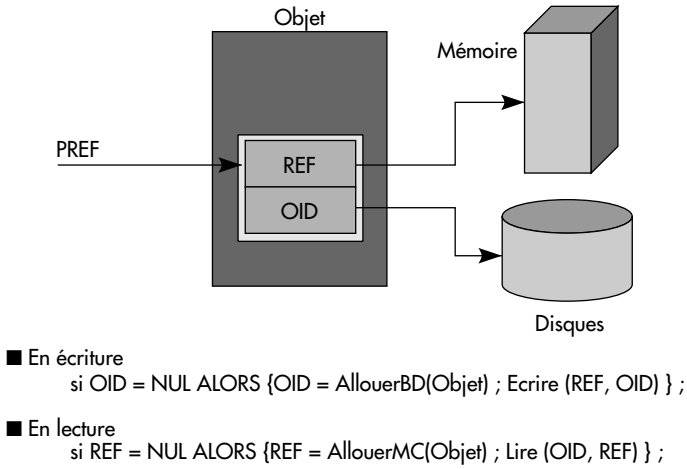


Figure XI.17 : L'utilisation de doubles pointeurs

La **technique de mémoire virtuelle** consiste à avoir les mêmes adresses en mémoire centrale et sur disques. Elle a été proposée et brevetée par Object Design, le fabricant du SGBDO ObjectStore. Une référence est une adresse mémoire virtuelle, cette dernière contenant une image exacte de la base de données (ou d'une partition de la base). Lors de la lecture d'un objet, l'adresse mémoire virtuelle composant toute référence est forcée sur une page manquante si l'objet référencé n'est pas en mémoire. La page manquante est réservée en mémoire et marquée inaccessible en lecture et en écriture. Ainsi, lors de la traversée du pointeur, une violation mémoire virtuelle en lecture est déclenchée et récupérée par le SGBDO (voir figure XI.18). Celui-ci retrouve alors la page de l'objet sur disques dans ces tables, lit cette page dans la page manquante, et rend la page accessible en lecture. Celle-ci peut alors être accédée comme une page normale et l'objet peut être lu. En cas de mise à jour, la violation de page est aussi récupérée pour mémoriser la nécessité d'écrire la page en fin de transaction à la validation, puis la page est rendue accessible en écriture.

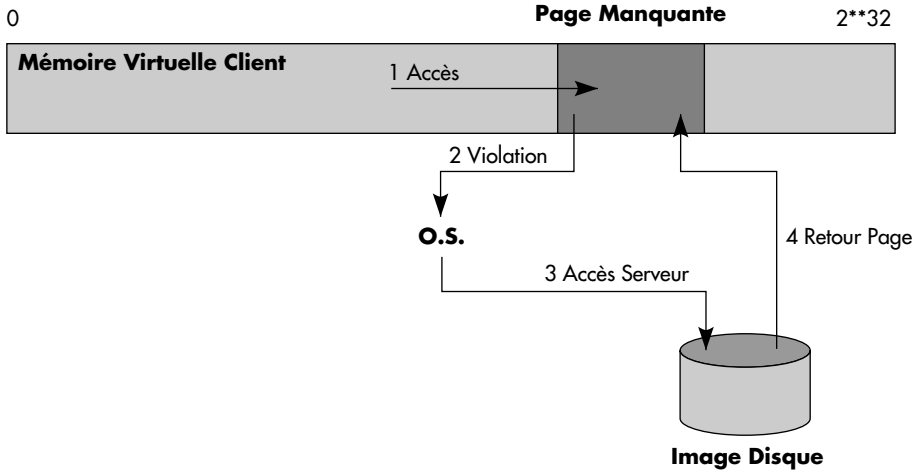


Figure XI.18 : La technique de mémoire virtuelle

Cette technique est parfois appelée **mémoire à un seul niveau** (*Single Level Store*) ; en effet, l'utilisateur travaille directement sur l'image de la base en mémoire. Elle est très efficace lorsqu'un même objet est traversé plusieurs fois et que les objets parcourus sont groupés dans une même page. Cependant, la dépendance du programme à la structure de la mémoire virtuelle peut constituer un inconvénient encore mal mesuré aujourd'hui.

4. ALGÈBRE POUR OBJETS COMPLEXES

Les algèbres pour objets complexes résultent d'extensions de l'algèbre relationnelle aux objets complexes. Elles permettent d'exprimer les questions sur une bases de données objet comme des expressions d'opérations élémentaires. Afin d'illustrer la conception orientée objet, nous proposons une structuration de l'algèbre en graphe d'objets, chaque opération étant un objet d'une classe.

4.1. EXPRESSIONS DE CHEMINS ET DE MÉTHODES

L'algèbre relationnelle permet normalement de référencer les attributs des relations dans les expressions résultats et les expressions de sélection. Ces valeurs de résultats ou de sélections sont extraites des tuples par des **expressions valuables** qui figurent

en arguments des opérations de l'algèbre. La notion d'expression valuable recouvre traditionnellement les références aux attributs, les constantes et les fonctions arithmétiques en relationnel. Par exemple, dans une table Produits, il est possible de sélectionner le prix toutes taxes comprises des produits par l'expression valuable PRIX*(1+TVA).

Dans un environnement objet, il est nécessaire de pouvoir appliquer des opérations sur les objets et aussi de pouvoir référencer les identifiants, par exemple afin de parcourir les associations. On arrive ainsi naturellement à généraliser la notion d'expression valuable. Une première généralisation telle que proposée ici a été effectuée par [Zaniolo85]. Deux types d'expressions valuables nouvelles sont nécessaires : les **expressions de chemins** et les **expressions de méthodes**.

Notion XI.27 : Expression de chemin (*Path Expression*)

Séquence d'attributs de la forme $A_1.A_2...A_n$ telle que chaque attribut A_i de la classe Ci référence un objet de la classe C_{i+1} dont le suivant est membre, à l'exception du dernier.

Une expression de chemin permet d'effectuer un parcours dans le graphe des associations de classes. Sur la base de la figure XI.12, Représente.Tête.Centre.x est une expression de chemins : partant d'un buveur B, elle permet d'atteindre un réel représentant l'abscisse du cente de la tête, en traversant les classes Caricature Cercle et Point. De telles expressions sont à la fois utilisables en algèbre d'objets complexes et en SQL étendu, comme nous le verrons dans les chapitres suivants. Dès que l'on rencontre un chemin multivalué, la traversée devient ambiguë ; par suite, la plupart des langages interdisent les **expressions de chemins multivaluées** du style Abus.cru sur la base de la figure XI.12, car bien sûr un buveur boit plusieurs vins.

Notion XI.28 : Expression de méthodes (*Method Expression*)

Séquence d'appels de méthodes de la forme $M_1.M_2...M_n$, avec d'éventuels paramètres pour certaines méthodes M_i de la forme $M_i(P_1, P_2...P_j)$.

Une expression de méthodes permet en principe d'appliquer des méthodes à un objet, l'objet sélectionné étant l'argument distingué permettant l'envoi du message correspondant à la méthode. Le polymorphisme doit être mis en œuvre pour sélectionner le bon code de la méthode. Par exemple, `Travailler(10)` est une expression de méthode dont le code sera différent selon que l'employé est un cadre ou non (voir figure XI.12).

Il est possible de généraliser les expressions de méthodes afin de les appliquer aussi à des valeurs : on obtient alors des **expressions fonctionnelles** de la forme $F_i(P_1, P_2...P_j)$ s'appliquant sur des éléments (objets ou valeurs) de type Ti et retournant des éléments de type Ti+1 sur lesquels on peut à nouveau appliquer des opérations définies sur le type Ti+1. Une expression de chemins est alors un cas particulier où la

fonction consiste à traverser le pointeur. Il est aussi possible de généraliser aux expressions fonctionnelles multivaluées, en tolérant en résultat d'une fonction une collection. En résumé, la figure XI.19 présente les différents types d'expressions valuables dans un environnement objet.

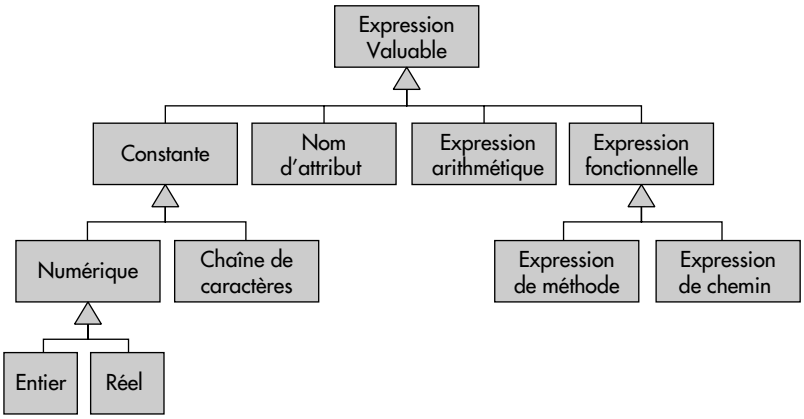
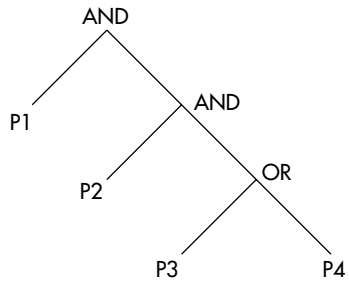


Figure XI.19 : Différents types d'expressions valuables

À partir des expressions valuables généralisées, il est possible de spécifier une qualification de restriction ou jointure généralisée, capable de traiter des objets complexes. Une telle qualification peut être vue comme un arbre ET-OU de prédicats élémentaires (voir figure XI.20). Un prédicat élémentaire est de la forme :

```
<Prédicat élémentaire> ::= <Expression valuable>
[<Comparateur> <Expression valuable>].
```

Un comparateur est choisi parmi {=, <, >, ≥, ≤, ≠}. Le comparateur et la deuxième expression valuable ne sont pas nécessaires si la première expression valuable est de type booléen (par exemple, une méthode booléenne telle Contains).



((P3 OU P4) AND P2) AND P1

Figure XI.20 : Graphe ET-OU de prédicats élémentaires

4.2. GROUPAGE ET DÉGROUPAGE DE RELATIONS

Outre les opérations classiques de l'algèbre relationnelle, les algèbres d'objets complexes généralisent les opérations issues des agrégats et introduites dans les modèles non en première forme normale, c'est-à-dire avec des attributs multivalués ensemblistes. Les opérations de groupement comportent le **groupage** (*nest* en anglais) et le **dégroupage** (*unnest*). Ces deux opérations ont à l'origine été définies pour des relations comme suit (nous les étendrons aux objets plus loin) :

Notion XI.29 : Groupage (*Nest*)

Opération transformant une relation en créant pour chaque valeur des attributs de groupement un ensemble de valeurs des attributs groupés.

Cette opération, notée ν dans certaines extensions de l'algèbre relationnelle, fait donc apparaître des attributs à valeur dans des ensembles. Elle est illustrée figure XI.21. Une définition plus générale pourrait consister à grouper la relation selon un schéma hiérarchique des attributs : on pourrait ainsi faire plusieurs groupages en une seule opération. Dans le monde objet, cette opération peut être appliquée à une collection pour générer une nouvelle collection, avec pour chaque objet obtenu par groupage, attribution d'un nouvel identifiant.

L'opération de dégroupage est l'opération inverse (voir figure XI.21).

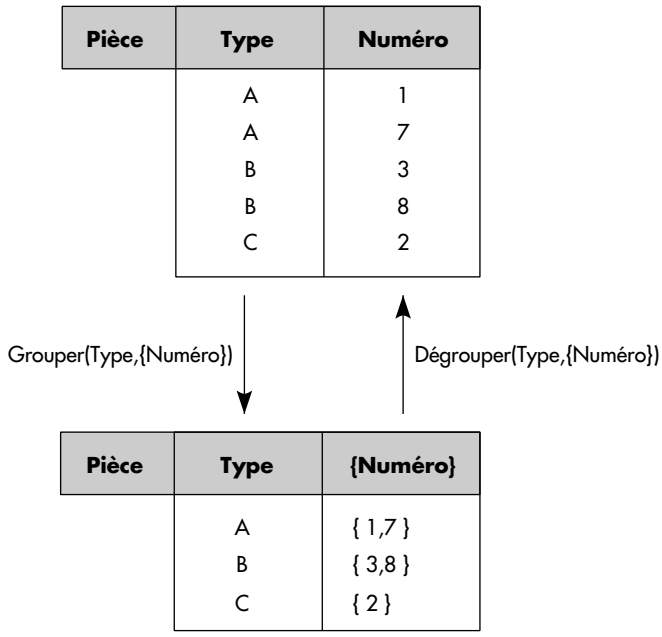


Figure XI.21 : Exemple d'opérations de groupage et dégroupage

Attention il n'est pas toujours vrai qu'un dégroupage après un groupage donne la relation initiale, en particulier si cette dernière a des doubles.

Notion XI.30 : Dégroupage (*Unnest*)

Opération transformant une relation à attributs groupés en relation plate, créant pour cela un tuple pour chaque valeur du groupe en dupliquant les valeurs des autres attributs.

4.3. L'ALGÈBRE D'ENCORE

Cette algèbre a été introduite dans le cadre du projet de SGBDO Encore par Shaw et Zdonik [Shaw90]. Elle apporte les concepts objets au sein d'une algèbre proche de l'algèbre relationnelle. Cette algèbre supporte les types abstraits et les identifiants d'objets. Les opérations accèdent des collections typées d'objets en invoquant l'interface publique du type. Pour cela, l'algèbre utilise des expressions d'opérations notées f_i et des prédicats construits à partir de ces expressions notés p.

Les opérations sont les suivantes :

– La **sélection d'objets** dans une collection d'entrée par un prédicat p est définie comme suit : Select(Collection, p) = {s | (s ∈ Collection) ∧ p(s)}. Le résultat est donc un ensemble d'identifiants d'objets indiquant ceux qui satisfont au prédicat p.

– L'**image d'une collection** par une expression fonctionnelle f de type T est définie par : Image(Collection, f : T) = {f(s) | s ∈ Collection}. Il s'agit donc de l'ensemble des objets résultants de l'application de f à ceux de la collection.

– La **projection d'une collection** par une famille d'expressions fonctionnelles f1, f2…fn sur un tuple <A1, A2…An> est définie par : Project(Collection,<(A1, f1)...,(An, fn)>) = {<A1 : f1(s)...,An : fn(s)> | (s ∈ Collection)}. Chaque objet donne donc naissance à un tuple, alors que chaque objet donnait naissance à élément simple dans le cas de l'image.

– Le **groupage d'une collection** de tuples sur un attribut Ai est défini par Nest(Collection,Ai) = {<A1 : s.A1..., Ai : t...,An : s.An> | ∀r ∃s (r ∈t ∧ s ∈ Collection ∧ s.Ai = r)}. L'opération remplace la collection d'entrée par un ensemble de tuples similaires pour les attributs autres que Ai, mais groupant dans un ensemble les valeurs de Ai correspondant aux tuples identiques pour les autres attributs. Cette opération introduit dans le monde objet le groupage relationnel.

– Le **dégroupage d'une collection** de tuples sur un attribut Ai est défini par : UnNest(Collection,Ai) = {<A1 : s.A1...,Ai : t...,An : s.An> | s ∈ InputCollection ∧ t ∈ Ai}. C'est l'opération inverse du groupage.

– L'**aplatissage d'une collection** est utilisé pour restructurer des collections de collections. Il est défini par : Flatten(Collection) = {r | ∃ t ∈ Collection ∧ r ∈ t}.
– La **jointure de collections** sur un prédicat p est une transposition simple de la jointure par valeur du relationnel : OJoin(Collection1, Collection2,A1,A2,p) = {<A1 : s, A2 : r> | s ∈ Collection1 ∧ r ∈ Collection2 ∧ p(s,r)}.

Afin d'éliminer les doubles, une opération d'**élimination de duplicata** est introduite. Elle est notée DupEliminate (Collection, e). e est un paramètre permettant de définir l'égalité d'objets dans la collection. Ce peut être par exemple l'égalité d'identifiants ou de valeurs. Coallesce (Collection, A_i, e) est une variante permettant d'éliminer les doubles dans un attribut A_i d'un tuple avec attribut multivalué.

Les opérations ensemblistes classiques d'**union**, **intersection** et **différence** de collections sont à considérer. Elles nécessitent la définition du type d'égalité d'objets considéré, sur identifiant ou sur valeur.

4.4. UNE ALGÈBRE SOUS FORME DE CLASSES

Examinons maintenant l'algèbre LORA dérivée de celle utilisée dans un SGBD objet-relationnel construit à l'INRIA à la fin des années 80 [Gardarin89]. Cette algèbre s'appuie aussi sur des expressions valuables permettant de construire des qualifications et prédicats spécifiés figure XI.22.

```
Class Qualification { // Connexion logique de prédicats ET de OU
   Connecteur Node; // Connecteur logique AND, OR, NIL
   Predicat* Primaire; // Prédicat élémentaire
   Qualification* QualifDroite; // Reste de la qualification
};

Class Predicat { // Critère de comparaison élémentaire
   Expression* Droite; // Expression droite
   Comparateur Comp; // Comparateur =, <, ≤, >, ≥, ≠
   Expression* Gauche; // Expression gauche
};
```

Figure XI.22 : Spécifications en C++ des qualifications

L'algèbre distingue les opérations par valeur et par référence, dont l'argument est en général un ou plusieurs identifiants. Les opérations sont classées en opérations de recherche, opérations ensemblistes, opérations de groupement et enfin opérations de mise à jour. La figure XI.23 représente la hiérarchie de généralisation des opérations de l'algèbre proposée.

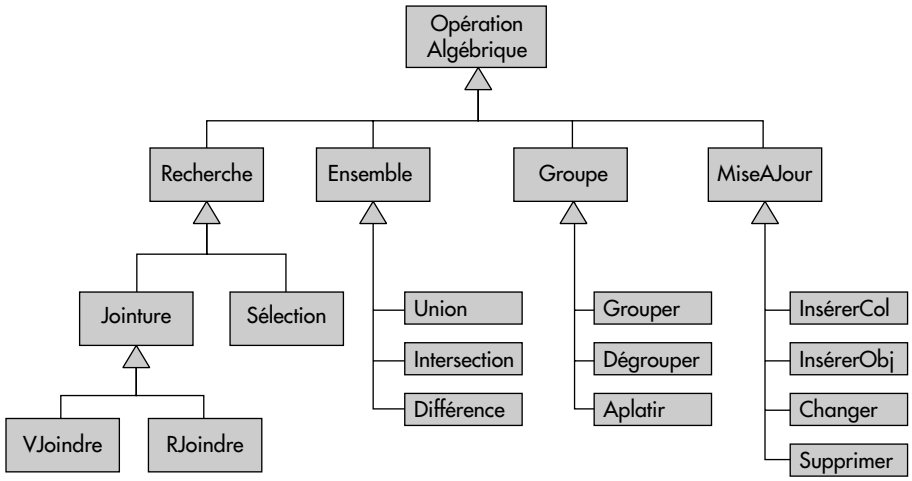


Figure XI.23 : Les différentes opérations de l'algèbre

4.4.1. Les opérations de recherche

Nous distinguons les opérations élémentaires de recherche suivantes :

– **Sélection** (`Filter`) correspond à l'application d'une qualification aux objets d'une collection et à la projection à l'aide d'expressions construites à partir des attributs en résultat ; elle équivaut à `Project` ou `Image` de l'algèbre Encore, qui sont ici regroupées en une seule opération.

– **Jointure par valeur** (`VJoindre`) permet de filtrer le produit cartésien de deux collections à l'aide d'une expression de qualification classique. Elle est réalisée par l'`Ojoin` de l'algèbre Encore.

– **Jointure par référence** (`RJoindre`), où le critère est une expression de chemin mono– ou multivaluée, aussi réalisable par `Ojoin` avec Encore.

L'opération générique de recherche permet d'effectuer plusieurs opérations élémentaires à partir d'une qualification générale et d'une expression de résultats (liste d'expressions). La figure XI.24 donne une spécification indicative en C++ de la classe Opération et des sous-classes définissant les opérations de recherche. Chaque sous-classe d'opérations de recherche correspond à un type particulier de qualification.

```
class Opération {
   // Options de tri et dédoublement des résultats:
   LIST(Field*) SortInfo; // Champs de tri.
   Boolean Ascendant; // Ascendant si Vrai.
   Boolean Duplicata; // Garder les doubles si vrai.
};
```

```
class Recherche : public Opération {
   // *** output = SEARCH (input-list, qualif, resul [,sort-dup])
   Qualification* Critère; // Qualification générale.
   LIST(Expression) Résultat; // Expressions sélectionnées.
};

class Selection : public Recherche {
   // *** output = FILTER (input-list, qualif, resul [,sort-dup])
   // Le critère est une qualification simple ou de méthodes};

class Jointure : public recherche { } ;
class RJoindre : public Jointure {
// *** result = RJOINDRE (input-list, qualif-ref, resul [, sort-dup])
// Le critère est une expression de chemins référence.
};

class VJoindre : public Jointure {
// *** result = VJOINDRE(input, input, qualif-join, resul [,sort-dup])
// Le critère est un prédicat de jointure.
};
```

Figure XI.24 : La classe des opérations de recherche

4.4.2. Les opérations ensemblistes

Les opérations ensemblistes permettent de réaliser l'union, l'intersection et la différence de deux ensembles d'objets (en principe des instances de classes). Les objets peuvent être repérés par leurs identifiants (identité d'objets) ou par leurs valeurs (égalité d'objets). Selon que les opérations comparent les identifiants ou les valeurs, on obtient deux résultats en général différents. Il faut donc spécifier l'égalité d'objets utilisée en paramètre des opérations ensemblistes, ce qui est fait au niveau de la classe Assembler (voir Figure XI.25).

```
class Ensemble : public Operation {
   Egalité Enum {identifiant; valeur}; // Type d'égalité
   Field* Identifiant; // Référence au champ identifiant.
};

class Union : public Ensemble{
   // *** output = UNION (input-list, [,sort-dup])
};
class Intersect : public Ensemble{
   // *** output = INTERSECT (input-list [,sort-dup])
};
class Difference : public Ensemble{
   // *** output = DIFFERENCE (input-list [,sort-dup])
};
```

Figure XI.25 : Les classes des opérations ensemblistes

4.4.3. Les opérations de mise à jour

Les opérations de mise à jour comportent les insertions, les suppressions et les modifications. Nous distinguons deux types d'insertion : insertion à partir d'une collection temporaire (InsérerCol) et insertion d'une liste d'objets fournis par valeurs (InsérerObj). Ces deux opérations sont définies figure XI.26.

```
class InsertCol : public MiseAJour{
   // *** input1 = INSERT_OBJ (input1, input2 [, sort-dup])
   // La 2e collection contient les objets à insérer dans la première.
   // La 1e collection est une collection de base.
   // La 2e est une collection calculée.
};

class InsertObj : public MiseàJour{
   // *** input = INSERT_VAL (input, Object_list [,sort-dup])
   LIST(Expression) Objet[MaxObj]; // Objets à insérer.
};
```

Figure XI.26 : Les classes des opérations d'insertion

Les suppressions s'effectuent à partir d'une collection calculée qui contient les identifiants des tuples à supprimer dans l'autre collection. La suppression est équivalente à une différence sur identifiant d'objets. Elle est définie figure XI.27.

```
class Supprimer : public MiseàJour{
// *** input1 = SUPPRIMER (input1, input2, identifiant [, sort-dup])
// La 2e collection contient les objets à supprimer de la première.
// La 1e collection est une collection de base.
// La 2e collection est une collection calculée.
Field* identifiant; // Référence au champ identifiant.
};
```

Figure XI.27 : La classe des opérations de suppression

Les modifications s'effectuent aussi à partir d'une extension de classe calculée qui contient les identifiants de tuples à modifier dans une classe de base et les éléments pour calculer les nouvelles valeurs des champs modifiés. Ces éléments sont exprimés, pour chaque attribut, sous la forme d'une expression de calcul (par exemple A = A *1,1 pour une augmentation de 10 % de A). La figure XI.28 définit plus précisément l'opération Changer.

```
class Changer : public MiseAJour{
   // *** input1 = UPDATE (input1, input2 [, sort-dup])
   // La 2e collection contient les objets à changer dans la première.
   // La 1e collection est une collection de base.
   // La 2e collection est une collection calculée.
   Field* identifiant; // Référence au champ identifiant.
   LIST(Field*) Achanger; // Champs à modifier.
   LIST(Expression) Calcul; // Mode de calcul des modifs.
};
```

Figure XI.28 : La classe des opérations de modification

4.4.4. Les opérations de groupe

Les opérations restant à spécifier sont celles de groupage correspondant aux Nest, Unest et Flatten de l'algèbre Encore. La figure XI.29 spécifie plus précisément les opérations de groupage, dégroupage et aplatissage en termes de classes C++.

```
class Grouper : public Groupe {
   // *** output = NEST (input, nest_exp, result-expression)
   // Le groupage est effectué sur un seul groupe d'attributs.
   // Une expression résultat est applicable aux champs groupés.
   // L'expression résultat doit s'appliquer à une collection.
   LIST(Field*) Base; // Attributs pour le partitionnement.
   LIST(Field*) Groupé; // Attributs à grouper.
   LIST(Expression) Résultat; // Résultats à fournir.
};

class Degrouper : public Groupe {
   // *** output = UNNEST (input, unnest_exp, result-expression)
   // Le dégroupage est effectué sur un seul groupe d'attributs.
   // Une expression résultat est applicable aux champs dégroupés.
   LIST(Field*) Base; // Attributs à dupliquer.
   LIST(Field*) Dégroupé; // Attributs à dégrouper.
   LIST(Expression) Résultat; // Résultats à fournir.
};

class Aplatir : public Groupe {
   // *** output = FLATTEN (input, flatten_exp)
   // La restructuration est faite sur un seul attribut
   Field* Aplatit; // Attributs à désimbriquer de 1 niveau.
} ;
```

Figure XI.29 : Les classes d'opérations de groupage et dégroupage

4.4.5. Arbres d'opérations algébriques

Finalement, comme avec le modèle relationnel, une question peut être représentée par une expression d'opérations de l'algèbre d'objets complexes. L'expression peut être visualisée sous la forme d'un arbre. Pour être plus général, en particulier pour permettre le partage de sous-arbre et le support de boucles, il est souvent permis à chaque opération d'avoir plusieurs flots d'entrées et surtout plusieurs flots de sorties. Un flot correspond à une collection temporaire qui peut être matérialisée ou non, selon la stratégie d'évaluation du système.

Un exemple simple de graphe d'opérations est illustré figure XI.31. Il représente une question portant sur une base de données qui décrit des véhicules référençant des constructeurs automobiles référençant eux-mêmes leurs employés directeurs de divisions (voir figure XI.30). Soit la question « rechercher les numéros des véhicules de couleur rouge dont le fabriquant est de Paris et a un directeur au moins de moins de 50 ans ». Le graphe associé est un graphe possible parmi les nombreux graphes d'opérations permettant d'exécuter cette question. Il est très proche d'un graphe relationnel. Nous étudierons son optimisation au chapitre sur l'optimisation de requêtes objet.

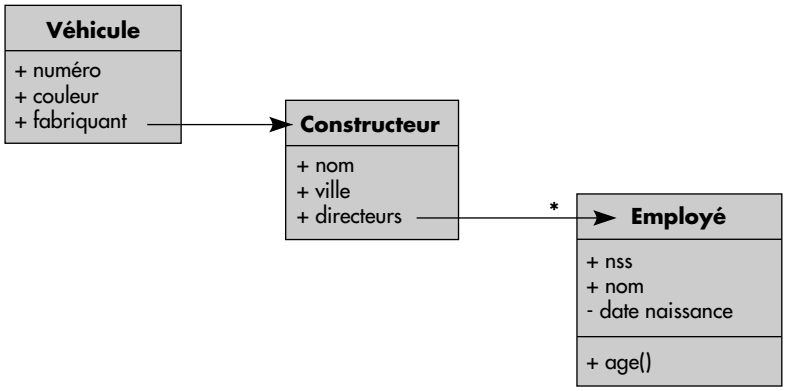


Figure XI.30 : Schéma de la base Véhicule, Constructeur, Employé

Pour terminer la modélisation de l'algèbre en C++, il faut maintenant spécifier la classe dont les graphes d'expressions algébriques sont les instances. La figure XI.32 modélise un graphe d'opérations par deux classes en C++, l'une correspondant aux nœuds (Nœud) et l'autre aux arcs (Flot). Nous avons choisi de représenter chaque arc par un objet de la classe Flot, cela afin de pouvoir étiqueter les arcs, par exemple par un volume de données estimées circulant sur l'arc. Cela permettrait d'approcher les volumes de données traités et les coûts d'opérations, comme nous le verrons au chapitre sur l'optimisation de requêtes.

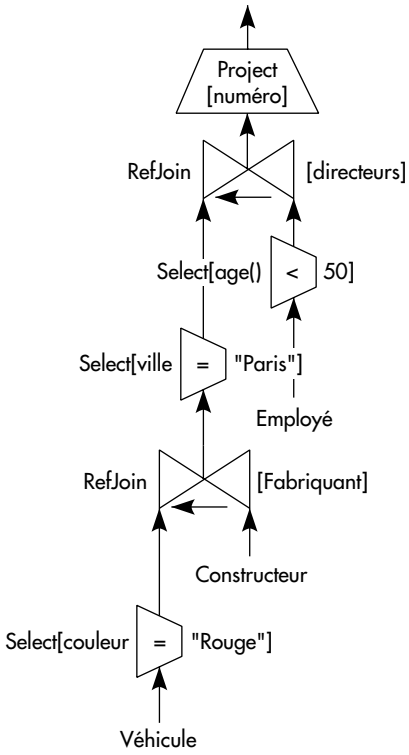


Figure XI.31 : Exemple de graphe d'opérations

```
Class Nœud {
   Operation* Oper; // Operation à effectuer en ce nœud.
   Flot* Input[MaxIn]; // collections d'entrée (fils).
   Flot* Output[MaxOut]; // collections de sortie (parents).
};

Class Flot {
   Int TailleEst; // Taille du flux estimée.
} ;
```

Figure XI.32 : Classes modélisant un graphe d'opérations

5. CONCLUSION

Dans ce chapitre, nous avons présenté les concepts de modélisation introduits par l'orientation objet, les techniques de gestion de la persistance des objets et une algèbre d'objets complexes. Ces concepts et techniques constituent l'essentiel des fonctionnalités aujourd'hui offertes par les SGBDO par le biais du langage de définition d'objets, du langage de requêtes et du langage de manipulation d'objets. Au-delà, il est important de mentionner que les SGBDO offrent pour la plupart des environnements de développements visuels élaborés. Ceux-ci permettent de parcourir les graphes de généralisation et de composition (agrégation et association) de classes, mais aussi de visualiser les objets, voire de créer de nouvelles classes et de programmer des méthodes. Ces éditeurs évolués (en anglais, *browsers*) sont un des attraits importants des SGBDO : en exploitation par exemple, ils autorisent la vision des classes d'objets sous forme d'icônes clicables pour déclencher les méthodes associées. Ces méthodes peuvent d'ailleurs être des opérations de recherche ou de mise à jour proches de celles de l'algèbre d'objets.

Outre ceux étudiés dans ce chapitre, les bases de données à objets soulèvent de nombreux problèmes difficiles. Les plus cruciaux sont sans doute ceux d'architecture et de performance. Quelle architecture client-serveur retenir ? Faut-il plutôt des serveurs d'objets ou de pages [Dewitt90] ? Comment optimiser les performances, en gérant des caches d'objets, en utilisant des techniques de type mémoire virtuelle d'objets, en développant des méthodes de regroupement des objets souvent accédés ensemble, etc. ? Du point de vue du langage de requêtes étendu aux objets, nous allons voir que deux propositions de standards s'opposent quelque peu. Les techniques d'optimisation sont encore mal maîtrisées en pratique. Un autre problème important est celui posé par les modifications de schémas : comment éviter de décharger la base et recompiler les méthodes, par exemple lors d'ajout de super-classes ou de suppression de sous-classes ? Une solution est sans doute la gestion de versions d'objets et de schémas [WonKim90]. Les problèmes de concurrence en présence de transactions longues (une transaction de conception peut durer plusieurs heures) sont eux aussi cruciaux. Tous ces problèmes (notre liste n'est malheureusement pas exhaustive) seront étudiés dans les chapitres qui suivent.

Sous sa forme pure ou sous forme intégrée au relationnel, l'objet constitue sans doute la voie d'avenir pour les bases de données, comme pour bien d'autres domaines. Des standards de représentation se développent au niveau des applications selon les techniques de modélisation orientée objet, en particulier les standards XML, SGML, EXPRESS et CMIS/CMIP respectivement pour le WEB, la gestion de données techniques, la CAO et l'administration de réseaux. Ces langages de modélisation de documents ou de composants semblent bien adaptés aux bases de données à objets. L'approche objet reste cependant limitée, car elle nécessite de bien connaître les objets pour en définir le type. Fondé sur un typage fort, l'objet est peu adapté pour aborder les applications à données faiblement structurées, comme le Web. Des extensions sont nécessaires.

6. BIBLIOGRAPHIE

[Abiteboul95] Abiteboul S., *Foundations of databases*, Addison-Wesley, Reading, Mass., 1995.

Ce livre présente les fondements théoriques des bases de données en faisant le lien avec des domaines de recherche connexes, tels que la logique et la complexité. Il fait le point sur les problèmes avancés des bases de données, notamment sur le couplage des modèles déductifs et objets.

[Abiteboul87] Abiteboul S., Beeri C., « On the Power of Languages for the Manipulation of Complex Objects », *International Workshop on Theory and Applications of Nested Relations and Complex Objects*, 1987, aussi rapport INRIA n° 846, Paris, mai 1988.

Cet article présente une vue d'ensemble théorique mais progressive des algèbres pour objets complexes. Il discute de la puissance des langages résultants.

[Arnold96] Arnold K., Gosling J., *Le Langage Java*, International Thomson Publishing France, Traduction de *The Java Programming Language* par S. Chaumette et A. Miniussi, Addison Wesley Pub., 1996.

Ce livre est la référence sur le langage Java, par les inventeurs. Il inclut une brève introduction au langage et une présentation détaillée des commandes, constructions et bibliothèques. Sont couverts les aspects définition de classes et d'interfaces, traitement des exceptions, multitâche, package, classes systèmes et bibliothèques.

[Atkinson89] Atkinson M., Bancilhon F., DeWitt D., Dittrich K., Maier D., Zdonick S., « The Object-Oriented Database System Manifesto », *Deductive and Object-Oriented Databases Int. Conf.*, Kyoto, Japan, 1989.

Le fameux manifesto pour les bases de données pures objet. Les caractéristiques obligatoires et optionnelles des SGBDO sont précisées comme vu ci-dessus.

[Banerjee87] Banerjee J., Kim W., Kim H.J., Korth. H.F., « Semantics and Implementation of Schema Evolution in Object-Oriented Databases », *ACM SIGMOD Int. Conf.*, San Fransisco, Cal., 1987.

Cet article pose les problèmes de modification de schémas dans les bases de données objets: suppression ou addition d'attributs, de méthodes, de surclasses, etc. Les solutions retenues dans ORION, qui permettent une grande souplesse à condition de respecter des règles précises (par exemple, pas de cycle de généralisation), sont présentées.

[Bouzeghoub85] Bouzeghoub M., Gardarin G., Métais E., « SECSI: An Expert System for Database Design », *11th Very Large Data Base International Conference*, Morgan Kaufman Pub., Stockolm, Suède, 1985.

Cet article décrit le système SECSI basé sur un modèle sémantique appelé MORSE. MORSE supporte l'agrégation, la généralisation, l'association et

l'instanciation. SECSI est construit selon une architecture système expert. C'est un outil d'aide à la conception de bases de données relationnelles qui transforme le modèle sémantique en relations normalisées. Le modèle sémantique est élaboré à partir de langages quasi naturels, graphiques ou de commande.

[Bouzeghoub91] Bouzeghoub M., Métais E., « Semantic Modelling of Object Oriented Databases », *17th Very Large Database International Conference*, Morgan Kaufman Pub., Barcelone, Espagne, août 1991.

Cet article propose une méthodologie de conception de bases de données à objets, fondée sur un réseau sémantique. L'application est spécifiée par un langage de haut niveau bâti autour d'un modèle sémantique et permet de définir des contraintes d'intégrité et des règles de comportements. Cette approche est intégrée dans la version objet du système d'aide à la conception SECSI.

[CACM91] Communication of the ACM, « Special Issue on Next-Generation Database Systems », *Communication of the ACM*, vol. 34, n° 10, octobre 1991.

Ce numéro spécial des CACM présente une synthèse des évolutions des SGBD vers une nouvelle génération. Les produits O2 commercialisé par O2 Technology, ObjectStore commercialisé par Object Design, GemStone commercialisé par Servio, et les prototypes Postgres de l'universté de Californie Berkeley et Starbust du centre de recherche d'IBM à Alamaden sont décrits en détail.

[Cardelli84] Cardelli L., Wegner P., « On Understanding Types, Data, Abstraction, and Polymorphism », *ACM Computing Surveys*, vol. 17, n° 4, décembre 1985.

Vaste article de synthèse sur le typage, les abstractions de type et le polymorphisme. Les conditions de typage sûr, c'est-à-dire vérifiable à la compilation, sont étudiées.

[Castagna96] Castagna G., *Object-Oriented Programming – A Unified Foundation*, 366p., Birkhaäuser, Boston, 1997.

Ce livre développe une théorie de l'orientation objet, plus spécialement du polymorphisme, qui couvre les méthodes multiclasses. Il apporte un nouvel éclairage au problème du typage des paramètres des méthodes dans les cas de surcharge et redéfinition. En clair, la nouvelle théorie en vogue dans le monde objet.

[Cattell91] Cattell R.G., « The Engineering Database Benchmark », in [Gray91].

Article présentant les résultats du premier benchmark comparant bases de données à objets et relationnelles. Les résultats démontrent la supériorité des SGBD à objets pour les parcours de graphes.

[Cluet90] Cluet S., Delobel C., Lécluse C., Richard P., « RELOOP, an Algebra Based Query Language for an Object-Oriented Database System », *Data & Knowledge Engineering,* vol. 5, n° 4, octobre 90.

Une présentation du langage d'interrogation du système O2. La sémantique du langage est basée sur une algèbre étendue. Bien que possédant une syntaxe

particulière, le langage est d'un point de vue sémantique proche d'un SQL étendu supportant des objets complexes.

[Delobel91] Delobel C., Lécluse Ch., Richard Ph., *Bases de données : des systèmes relationnels aux systèmes à objets*, 460 pages, InterÉditions, Paris, 1991.

Une étude très complète de l'évolution des SGBD, des systèmes relationnels aux systèmes à objets, en passant par les systèmes extensibles. Une attention particulière est portée sur les langages de programmation de bases de données. Le livre décrit également en détail le système O2, son langage CO2 et les techniques d'implémentation sous-jacentes. Un livre en français.

[Dewitt90] DeWitt D., Futtersack P., Maier D., Velez F., « A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems », *16th Very Large Database International Conference*, Morgan Kaufman Pub., Brisbane, Australie, août 1990.

Une étude comparative de trois architectures (serveur d'objets, de pages et de fichiers) client-serveur pour les SGBDO. L'analyse démontre sommairement que l'approche serveur de pages est plus performante sous certaines conditions dans un contexte mono-utilisateur.

[Gardarin89] Gardarin G., Kiernan J., Cheiney J.P., Pastre D., « Managing Complex Objects in an Extensible Relational DBMS », *15th Very Large Databases International Conference*, Morgan & Kaufman Ed., Amsterdam, p. 55-65, Août 1989.

Cet article présente le SGBD Sabrina réalisé à l'INRIA de 1988 à 1990, qui fut un des premiers SGBD relationnels à intégrer l'objet. Les objets étaient définis comme des types abstraits dont les opérations étaient programmées en LeLisp ou en C. Ils étaient intégrés au relationnel comme valeurs de domaines des tables. Ce SGBD fut commercialisé par une start-up qui fut malheureusement plus soutenue après 1988, les pouvoirs publics préférant une démarche objet pure.

[Gardarin94] Gardarin G., Nowak M., Valduriez P., « Flora : A Functional-Style Language for Object and Relational Algebra », *5th DEXA (Database and Expert System Application) Intl. Conf.*, Athens, in LNCS n° 856, p. 37-46, Sept. 1994.

FLORA est un langage fonctionnel permettant d'écrire des plans d'exécution résultant de la compilation de requêtes objet. Il est du même niveau qu'une algèbre d'objets complexes, mais basé sur une approche fonctionnelle. FLORA manipule une riche bibliothèque de collections.

[Goldberg83] Goldberg A., Robson D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

Le livre de référence de Smalltalk, par les inventeurs du langage.

[Gray91] Gray J. Ed., *The Benchmark Handbook*, Morgan & Kaufman Pub., San Mateo, 1991.

Le livre de base sur les mesures de performances des SGBD. Composé de différents articles, il présente les principaux benchmarks de SGBD, en particulier le

fameux benchmark TPC qui permet d'échantillonner les performances des SGBD en transactions par seconde. Les conditions exactes du benchmark définies par le « Transaction Processing Council » sont précisées. Les benchmarks de l'université du Madisson, AS3AP et Catell pour les bases de données à objets, sont aussi présentés.

[Guogen78] Guogen J., « An Intial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types », *Current Trends in Programming Methodology*, vol. 4, Prentice-Hall 1978, L. Yeh Ed., 1978.

Cet article propose une formalisation des types abstraits de données comme des sigma-algèbres multisortes. Toutes les fonctions sont modélisées comme des transformations de sortes. Cela permet de spécifier chaque fonction par des axiomes. Par exemple, une pile p munie des fonctions PUSH et POP doit obéir à l'axiome PUSH(POP(p)) = p. Un ensemble d'axiomes permet de spécifier complétement un type abstrait. Un des articles de base sur la théorie des types abstraits de données.

[Guttag77] Guttag J., « Abstract Data Types and the Development of Data Structures », *Comm. of ACM*, Vol.20, n° 6, juin 1977.

Cet article montre comment on peut spécifier l'implémentation de types abstraits en fonction d'autres types abstraits et comment ces implémentations peuvent être cachées à l'utilisateur (principe d'encapsulation). Des idées sur le contrôle des spécifications et des implémentations sont aussi proposées. Cet article plutôt pratique développe les principes des types abstraits.

[Hammer81] Hammer M., McLeod D., « Database Description with SDM: A Semantic Database Model », *ACM TODS*, Vol.6, n° 3, septembre 1981.

Un des premiers modèles sémantiques proposés en bases de données ; les notions d'agrégation, généralisation, abstraction sont notamment introduites dans un modèle à base de graphe. Ce modèle a été implémenté sur les systèmes UNISYS, mais le SGBD résultant n'a malheureusement pas eu un grand succès, bien qu'il fût un précurseur des SGBD à objets.

[Hose91] Hose D., Fitch J., « Using C++ and Operator Overloading to Interface with SQL Databases », *The C++ Journal*, vol. 1, n° 4, 1991.

Cet article présente une intégration de SQL à C++. Des classes Base, Table, Colonne et Curseur sont définies. Les requêtes sont formulées dans une syntaxe proche de C++, mais aussi conforme à SQL. Elles sont traduites en requêtes soumises au SGBD interfacé. Le produit résultant nommé CommonBase s'interface avec SYBASE, ORACLE, INGRES, etc.

[Khoshafian86] Khoshafian S., Copeland G., « Object Identity », *OOPSLA Intl. Conf.*, Portland, Oregon, 1986, also in [Zdonik90].

Une discussion de l'identité d'objets : un identifiant est un repère qui distingue un objet d'un autre objet, indépendamment de son état. Les différents types d'égalité et le partage de sous-objets sont introduits dans cet article de référence.

[Lécluse89] Lécluse C., Richard Ph., « The O2 Database Programming Language », *15th Very Large Data Bases International Conference*, Morgan Kaufman Pub., Amsterdam, Pays-Bas, août 1989.

La description du langage du système O2 réalisé à l'INRIA au sein du GIP Altaïr. Ce langage est une extension orientée objet de C distinguant objets et des valeurs. Le langage permet d'introduire des classes avec méthodes, des constructeurs d'objets complexes (tuples et ensembles), de la généralisation et des facilités de filtrage itératif de données. Le système O2 est aujourd'hui commercialisé par O2 Technology.

[Lippman91] Lippman B. S., *C++ Primer*, 2e édition, 614 pages, Addison-Wesley, 1991.

Un excellent livre sur C++. Le langage est présenté sous tous ses aspects.

[Maier86] Maier D. *et. al.*, « Development of an object-Oriented DBMS », *1st Int. Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, Oct. 1986.

Un des premiers articles décrivant l'implémentation d'un SGBDO, le SGBD GemStone. Celui-ci est construit à partir d'une interface Smalltalk (Gem) connectée à un serveur d'objets (Stone).

[OMG91] Object Management Group & X Open Group, « The Common Object Request Broker: Architecture and Specification », *OMG Document n° 91, 12.1*, Revision 1.1, octobre 1991.

Présentation de l'architecture CORBA de l'OMG. Cette architecture distribuée permet d'envoyer des requêtes à des objets distants et de recevoir les réponses avec des interfaces et un modèle d'objets en voie de standardisation, en s'abstrayant de l'implémentation des objets. CORBA vise à assurer l'interopérabilité entre les environnements orientés objet. Les plus grandes compagnies industrielles soutiennent CORBA. Aujourd'hui, plus de 800 fournisseurs de logiciels et utilisateurs adhèrent à l'OMG.

[Rational97] Rational Software, *The Unified Modeling Language UML 1.1*, Reference Manual, Release 1.1, Boston, aussi disponible auprès de l'OMG, 1997.

Le document de référence d'UML. Les spécifications d'UML, le langage graphique universel de modélisation de l'objet, sont disponibles à l'adresse Internet www.rational.com/uml/ sous forme de plusieurs documents HTML ou PDF. Nous en donnons un résumé dans le chapitre sur la conception des bases de données.

[Shaw90] Shaw G., Zdonik B.S., « A Query Algebra for Object-Oriented Databases », *Proc. of the 6th International Conf. On Data Engineering*, IEEE Ed., p. 154-162, 1990.

Cet article propose une algèbre d'objet pour interroger les bases de données objet. Cette algèbre a été implémentée dans le projet ENCORE, et est restée connue sous ce nom. Nous l'avons décrite ci-dessus.

[Stoustrup86] Stoustrup B., *The C++ Programming Language*, New York, Addison-Wesley, 1986.

Le livre de référence de C++ par son inventeur. Stoustrup a créé C++ pour modéliser des problèmes de réseaux de télécommunications.

[WonKim88] Won Kim *et. al.*, « Features of the ORION Object-Oriented Database System », dans le livre « *Object-Oriented Concepts, Applications and Databases* », W. Kim et Lochovsjy Ed., Addison-Wesley, 1988.

Une description du système ORION, le SGBDO qui a popularisé l'approche bases de données à objets. Développé à MCC dès 1985, ORION est un SGBDO très complet. Initialement basé sur Lisp, le produit, commercialisé aujourd'hui par Itasca Systems, a évolué vers C et C++.

[WonKim89] Won Kim, « A Model of Queries for Object-Oriented Database », *Very Large Database International Conference*, Morgan Kaufman Pub., Amsterdam, Pays-Bas, août 1989.

Cet article présente les méthodes d'optimisation utilisées dans le système ORION pour le langage d'interrogation. La technique retenue est très proche de la restructuration d'arbre, considérant en plus les jointures par références et les parcours de chemins.

[WonKim90] Won Kim, *Introduction to Object-Oriented Databases*, 235 pages, The MIT Press, 1990.

Ce livre décrit les différentes techniques des SGBD à objets. Il s'inspire fortement du système ORION. Plus particulièrement, les problèmes de modèle orienté objet, de modification de schéma, de langage SQL étendu aux objets, de structures de stockage, de gestion de transactions, d'optimisation de requêtes et d'architecture sont abordés. Une bonne référence sur les bases de données à objets.

[Zaniolo85] Zaniolo C., « The Representation and Deductive Retrieval of Complex Objects », *11th Very Large Data Bases International Conference*, Morgan Kaufman Pub., Stockholm, Suède, août 1985.

Cet article présente une extension de l'algèbre relationnelle aux fonctions permettant de retrouver des objets complexes. Des opérateurs déductifs de type point fixe sont aussi intégrés.

[Zdonik90] Zdonik S., Maier D., *Readings in Object-Oriented Database Systems*, Morgan Kaufman Pub., San Mateo, California, 1990.

Une sélection d'articles sur les bases de données à objets.

Chapitre XII

LE STANDARD DE L'ODMG : ODL, OQL ET OML

1. INTRODUCTION

Depuis 1988, une dizaine de petites sociétés commercialisent des SGBDO, avec un succès encore limité. Elles se heurtent au problème de la portabilité des applications. Il existe maintenant des langages de programmation objet ayant une bonne portabilité comme C++, ou vraiment portables comme Java, qui a été conçu pour être porté. Mais porter des applications accédant à des bases de données exige la portabilité des interfaces d'accès. Ceci est possible en relationnel, avec SQL et des *middlewares* universels comme ODBC ou JDBC pour Java. Ceci était très difficile en objet devant l'absence de standards.

Ainsi, afin de définir des interfaces portables, s'est constitué l'*Object Database Management Group* (**ODMG**), formé au départ par cinq vendeurs de SGBDO. L'ODMG vise à réaliser pour les bases de données objets l'équivalent de la norme SQL, ou au moins d'un projet de norme. Deux versions du standard proposé ont été publiées assez rapidement : l'une en 1993 [Odmg93], l'autre présentée dans ce chapitre, en 1997 [Odmg97]. Un des buts des SGBDO, et donc de l'ODMG, est d'éviter le problème soulevé par les SGBD classiques où deux systèmes cohabitent lors de l'interfaçage avec un langage : celui du SGBD et celui du langage. Pour permettre une

utilisation directe des types des langages objet, l'ODMG a choisi de définir un modèle abstrait de définition de bases de données objet, mis en œuvre par un langage appelé ODL (*Object Definition Language*). Ce modèle est ensuite adapté à un langage objet particulier : l'ODMG propose un standard d'intégration en C++, Smalltalk, et Java. Un langage d'interrogation pour ce modèle est proposé : il s'agit d'OQL (*Object Query Language*), pour beaucoup issu du langage de requête du système O2 réalisé à l'INRIA [Bancilhon92, Adiba93]. OQL est aussi intégrable dans un langage de programmation objet.

Ce chapitre présente le standard de l'ODMG, en l'illustrant par des exemples. Après cette introduction, la section 2 précise le contexte et l'architecture d'un SGBDO conforme à l'ODMG. La section 3 développe le modèle abstrait et le langage ODL. La section 4 présente un exemple de base et de schéma en ODL. La section 5 aborde le langage OQL à travers des exemples et des syntaxes types de requêtes constituant des exemples génériques, appelés profils. La section 6 se consacre à l'intégration dans un langage de programmation ; le cas de Java est détaillé. La section 7 conclut ce chapitre en montrant les limites du standard de l'ODMG.

2. CONTEXTE

Dans cette section, nous présentons le contexte général du « standard » : les auteurs, le contenu de la proposition et l'architecture d'un système conforme à l'ODMG.

2.1. L' ODMG (*OBJECT DATABASE MANAGEMENT GROUP*)

Apparus vers 1986, les SGBD objet n'avaient pas connu le succès escompté cinq ans après leur naissance. Une des difficultés majeures provenait de l'absence de standards. Alors que les applications des bases de données relationnelles pouvaient prétendre à une très bonne portabilité assurée par le standard SQL, du poste de travail au calculateur central sur toute machine et tout système d'exploitation, les SGBD objet présentaient chacun une interface spécifique, avec des langages parfois exotiques. Un groupe de travail fut donc fondé en septembre 1991 à l'initiative de SUN par cinq constructeurs de SGBD objet : O2 Technology, Objectivity, Object Design, Ontos et Versant. Ce groupe prit rapidement le nom de ODMG (*Object Database Management Group*) et trouva un président neutre chez SUN en la personne de Rick Cattell, auteur de différent bancs d'essai sur les bases de données objet. Le groupe publia un premier livre intitulé *The Object Database Management Standard*, connu sous le nom

ODMG'93. En fait, il ne s'agit pas d'un standard avalisé par les organismes de normalisation, mais plutôt d'une proposition d'un groupe de pression représentant des vendeurs de SGBDO.

Le groupe a continué à travailler et s'est enrichi de représentants de POET Software, UniSQL, IBEX et Gemstone Systems, ainsi que de multiples gourous et observateurs externes. Une nouvelle version du livre a été publiée en 1997, sous le titre **ODMG 2.0** ; elle comporte dix auteurs. Le groupe est maintenant bien établi et collabore avec l'OMG et l'ANSI, notamment sur l'intégration à CORBA et à SQL3. Les constructeurs participants s'engagent à suivre les spécifications de l'ODMG, malheureusement sans dates précises. Un des échecs majeurs du groupe est sans doute l'absence de systèmes conformes aux nouvelles spécifications. O2, qui est le système le plus proche, ne répond pas exactement aux fonctionnalités requises [Chaudhri98].

2.2. CONTENU DE LA PROPOSITION

La proposition décrit les interfaces externes d'un SGBDO utilisées pour réaliser des applications. Le SGBDO se fonde sur une adaptation du modèle objet de l'OMG, modèle de référence étudié au chapitre précédent. Il comporte un langage de définition des interfaces des objets persistants dérivés de l'IDL de l'OMG et appelé **ODL** (*Object Definition Language*).

Notion XII.1 : ODL (*Object Definition Language*)

Langage de définition de schéma des bases de données objet proposé par l'ODMG.

Une définition peut aussi être effectuée directement dans l'un des langages supportés. La partie la plus intéressante de la proposition est le langage **OQL** (*Object Query Language*).

Notion XII.2 : OQL (*Object Query Language*)

Langage d'interrogation de bases de données objets proposé par l'ODMG, basé sur des requêtes SELECT proches de celles de SQL.

Une intégration est proposée avec les langages objets C++, Smalltalk et Java. Celle-ci précise les conversions de types effectuées et permet la manipulation des objets gérés par le SGBD depuis le langage. Elle est appelée **OML** (*Object Manipulation Language*).

Notion XII.3 : OML (*Object Manipulation Language*)

Langage de manipulation intégré à un langage de programmation objet permettant la navigation, l'interrogation et la mise à jour de collections d'objets persistants, dont l'OMG propose trois variantes : OML C++, OML Smalltalk et OML Java.

La figure XII.1 illustre les différentes interfaces proposées par le standard ODMG. Ce sont celles permettant de réaliser des applications autour d'un SGBDO. Sont-elles suffisantes pour assurer la portabilité ? Probablement non, car les interfaces graphiques sont aussi importantes et souvent spécifiques du SGBDO. Quoi qu'il en soit, le respect de ces interfaces par les produits améliorerait de beaucoup la portabilité des applications.

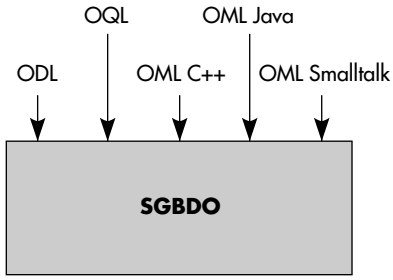


Figure XII.1 : Interfaces d'accès à un SGBDO

2.3. ARCHITECTURE

La figure XII.2 illustre l'architecture typique d'un SGBDO conforme à l'ODMG. Autour d'un noyau gérant la persistance des objets, l'attribution des identifiants, les méthodes d'accès, et les aspects transactionnels, gravitent trois composants : le préprocesseur ODL permet de compiler les définitions d'objets et de générer les données de la métabase ; le composant langage OML spécifique à chaque langage de programmation permet de manipuler les objets conformes aux définitions depuis un langage de programmation tel C++, Smalltalk ou Java ; le composant OQL comporte un analyseur et un optimiseur du langage OQL capables de générer des plans d'exécution exécutables par le noyau. Au-dessus de ces trois composants, différents outils interactifs permettent une utilisation facile des bases ; ce sont par exemple un éditeur de classes pour éditer les schémas ODL, un manipulateur d'objets pour naviguer dans la base (les deux réunis constituent souvent le *browser*), une bibliothèque d'objets graphiques, des débogueurs et éditeurs pour les langages de programmation, etc.

Du côté interface avec les langages de programmation, le schéma préconisé est basé sur un système de type unique entre le langage et le SGBDO, chaque type du modèle objet du SGBDO (en principe celui de l'ODMG) étant traduit directement dans un type correspondant du langage. Un principe de base est aussi de ne nécessiter aucune modification du compilateur du langage. Les déclarations de classes persistantes peuvent être écrites en ODL, ou directement dans le langage de programmation (PL ODL). Un précompilateur permet de charger la métabase du SGBDO et de générer la définition pour le langage de programmation. Les programmes enrichis avec les définitions de classes d'objets persistants sont compilés normalement. Le binaire résultant

est lié à la bibliothèque d'accès au SGBD lors de l'édition de liens, ce qui permet la génération d'un exécutable capable d'accéder à la base. La figure XII.3 illustre le processus d'obtention d'un exécutable.

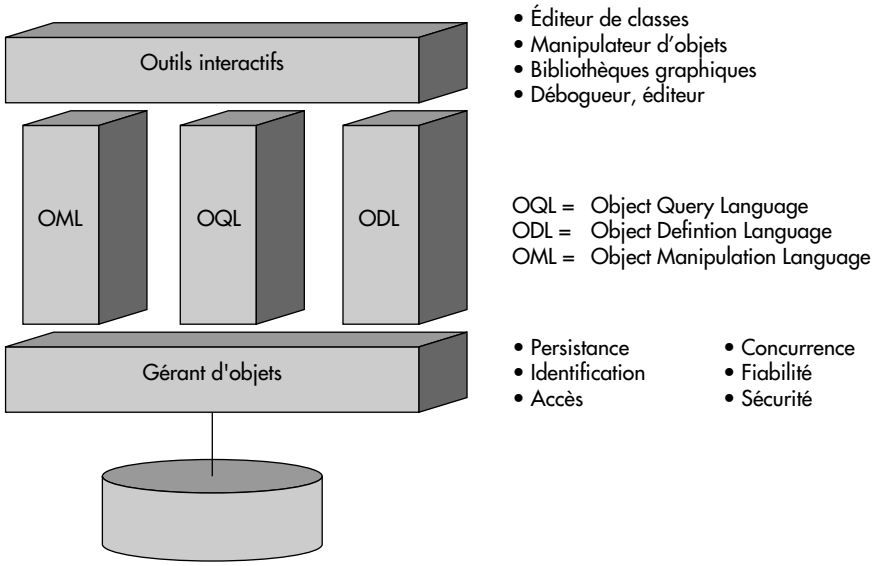


Figure XII.2 : Architecture type d'un SGBDO conforme à l'ODMG

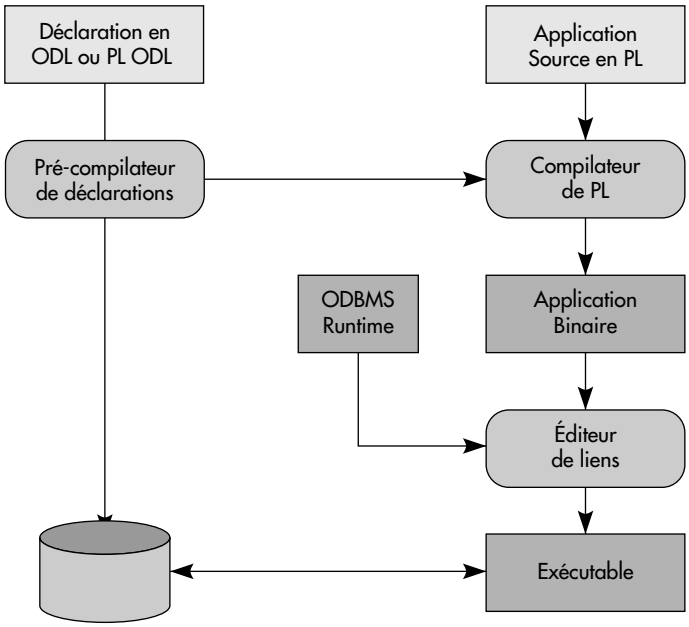


Figure XII.3 : Lien avec les langages de programmation

3. LE MODÈLE DE L'ODMG

Nous décrivons maintenant le modèle abstrait proposé pour la définition des schémas des bases de données objet.

3.1. VUE GÉNÉRALE ET CONCEPTS DE BASE

L'OMG – organisme de normalisation de l'objet composé de plus de 800 membres, à ne pas confondre avec l'ODMG – a proposé un modèle standard pour les objets permettant de définir les interfaces visibles par les clients. Le modèle de l'ODMG est une extension du modèle de l'OMG et un candidat pour un profil BD de ce dernier. Il est mis en œuvre à l'aide du langage ODL qui permet de spécifier les schémas de bases de données, alors que le modèle de l'OMG est supporté par le langage IDL (*Interface Definition Language*). Les bases de données objets nécessitent des adaptations ou extensions ; ODL se veut l'adaptation d'IDL aux bases de données.

Une extension principale est tout d'abord la nécessité de considérer un niveau d'abstraction permettant de manipuler des états abstraits pour les objets. Ce sont ces états qui sont mémorisés plus ou moins directement dans la base pour les objets persistants. En outre, les objets peuvent être groupés en collections beaucoup plus variées que les seules séquences de l'OMG. Ils peuvent aussi être associés par des associations. Tout cela conduit à un modèle et à un langage de définition associé (ODL) beaucoup plus complexe que celui de l'OMG. ODL reste cependant un langage de niveau conceptuel, supportant un modèle objet abstrait, qui peut être implémenté dans différents langages, en particulier C++, Smalltalk ou Java. Chaque construction ODL a donc une implémentation correspondante pour chacun de ces langages. Au contraire d'ODL, OML s'appuie sur une implémentation et est donc spécifique à un langage de programmation objet.

Le modèle de l'OMG comporte des types d'objets avec identifiants et des types de valeurs, ou littéraux. Outre les interfaces qui permettent de spécifier des comportements abstraits, ODL permet de spécifier des classes qui définissent, en plus d'un comportement abstrait, un état abstrait pour des objets d'un même type. On aboutit donc à trois types de définitions d'objets ou valeurs, précisés ci-dessous ; c'est un peu complexe, voire confus, mais nécessaire pour permettre l'implémentation des spécifications de comportement ou d'état en C++, Smalltalk ou Java.

Notion XII.4 : Définition d'interface (*Interface definiton*)

Spécification du comportement observable par les utilisateurs (ou d'une partie de celui-là) pour un type d'objets.

Nous définissons par exemple figure XII.4 une interface calculateur modélisant une machine à calculer.

```
INTERFACE CALCULATEUR {
   CLEAR();
   FLOAT ADD(IN FLOAT OPERAND);
   FLOAT SUBSTRACT (IN FLOAT OPERAND);
   FLOAT DIVIDE(IN FLOAT DIVISOR);
   FLOAT MULTIPLY (IN FLOAT MULTIPLIER);
   FLOAT TOTAL();}
```

Figure XII.4 : Définition de l'interface d'un calculateur

Notion XII.5 : Définition de classe (*Class definition*)

Spécification du comportement et d'un état observables par les utilisateurs pour un type d'objets.

Une classe implémente ainsi une ou plusieurs interfaces. En plus d'un comportement, une classe définit un état abstrait. Pour mémoriser les états abstraits de ses instances, une classe possède aussi une **extension de classe**.

Notion XII.6 : Extension de classe (*class extent*)

Collection caractérisée par un nom contenant les objets créés dans la classe.

Le comportement abstrait pourra être hérité d'une interface. On voit donc qu'une classe donne une spécification d'une ou plusieurs interfaces en précisant quelques éléments d'implémentation, en particulier l'état des objets (abstrait car indépendant de tout langage) et l'extension qui va les contenir. Dans certains cas complexes, une classe peut d'ailleurs avoir plusieurs extensions. Il est aussi possible de préciser une clé au niveau d'une extension de classe : comme en relationnel, il s'agit d'un attribut dont la valeur détermine un objet unique dans l'extension. La figure XII.5 illustre une définition de classe incorporant l'interface calculateur.

```
CLASS ORDINATEUR (EXTENT ORDINATEURS KEY ID) : CALCULATEUR {
   ATTRIBUTE SHORT ID ;
   ATTRIBUTE FLOAT ACCUMULATEUR;
   VOID START();
   VOID STOP(); }.
```

Figure XII.5 : Un exemple de classe

Notez qu'une interface peut aussi comporter des attributs abstraits, mais ceux-ci sont vus comme des raccourcis d'opérations, une pour lire l'attribut, l'autre pour l'écrire. Une interface n'a en principe pas d'extension.

Interfaces et classes sont des spécifications de types. Il est aussi possible de spécifier des types de valeurs : ceux-ci sont appelés des littéraux.

Notion XII.7 : Définition de littéral (*Literal définition*)

Spécification d'un type de valeur correspondant à un état abstrait, sans comportement.

Les littéraux correspondent aux types de base tels entier, réel, chaîne de caractères, mais aussi aux structures. Un exemple de littéral est un nombre complexe : struct complex {float re; float im}. Les littéraux sont directement implémentés comme des types de valeurs en C++. Dans les autres langages purs objets (Smalltalk ou Java), ce seront des objets.

En résumé, l'ODMG propose donc un système de types sophistiqué, capable d'être facilement mappé en C++, Smalltalk ou Java. Il y a donc des types, des interfaces, des classes et des littéraux. L'ensemble forme la hiérarchie de spécialisation représentée figure XII.6.

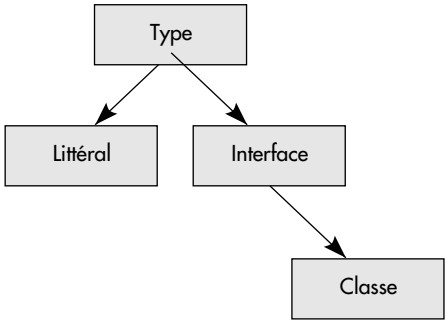


Figure XII.6 : Classification des définitions de type

3.2. HÉRITAGE DE COMPORTEMENT ET DE STRUCTURE

Comme vu ci-dessus, une classe peut hériter d'une interface : il s'agit d'un héritage de comportement. Toutes les opérations de l'interface seront alors définies pour les objets de la classe. Au niveau de l'implémentation, la classe fournira le code des opérations.

En plus de cette relation d'héritage de comportement d'interface vers une classe, l'ODMG propose un héritage de structure abstraite, c'est-à-dire de l'état des objets, cette fois de classe à classe. Cette relation d'héritage d'état est notée **`EXTENDS` (à ne** pas confondre avec `EXTENT` !). Par exemple, nous étendrons la classe `ORDINATEUR` comme indiqué figure XII.7.

Soulignons que l'héritage de comportement (noté :) peut être multiple : une classe peut implémenter plusieurs interfaces, mais elle ne peut dériver que d'une seule autre classe. En cas de conflits de noms, c'est à l'utilisateur qu'il incombe de distinguer les noms.

```
CLASS ORDINATEURAREGISTRE EXTENDS ORDINATEUR: CALCULATEUR {
   ATTRIBUTE FLOAT REGISTRE;
   VOID RTOA();
   VOID ATOR (); }.
```

Figure XII.7 : Exemple d'héritage de structure et de comportement

3.3. LES OBJETS INSTANCES DE CLASSES

Les objets sont donc regroupés selon des types. Comme dans tout modèle objet, ils sont identifiés par des OID. Ceux-ci sont des chaînes binaires de format spécifique de chaque implémentation, gérés par le SGBD OO pour distinguer les objets. Un OID permet de retrouver l'objet qu'il identifie pendant toute sa durée de vie. Il reste donc invariant pendant la vie de l'objet.

Classiquement, les objets sont persistants ou transients : les objets persistants sont les objets de la base, les autres restant en mémoire. Les objets persistants peuvent être nommés : les noms sont donnés par les utilisateurs et sont uniques dans une base de données.

Les objets peuvent être atomiques, structurés, ou collections, c'est-à-dire composés de collections d'objets ou de littéraux. Les objets atomiques sont spécifiés par l'utilisateur par des définitions de classes comportant des attributs, des associations et des opérations. Les objets structurés sont prédéfinis et obéissent à des interfaces spécifiques.

3.4. PROPRIÉTÉS COMMUNES DES OBJETS

Les objets sont créés par une opération new() définie au niveau d'une interface ObjectFactory implémentée par le SGBDO (voir figure XII.8). Ils héritent d'un ensemble d'opérations implémentées par le SGBDO pour le verrouillage – qui peut être bloquant si l'objet est occupé (opération `LOCK`) ou non (opération `TRY_LOCK`) –, la comparaison d'identifiants, la copie d'objet avec génération d'un nouvel objet et la suppression (voir figure XII.8).

```
INTERFACE OBJECTFACTORY {
   OBJECT NEW(); // CRÉATION D'UN NOUVEL OBJET
};
INTERFACE OBJECT {
   ENUM LOCK_TYPE{READ, WRITE, UPGRADE}; // TYPES DE VERROUS
   EXCEPTION LOCKNOTGRANTED(); // ERREUR VERROU REFUSÉ
   VOID LOCK(IN LOCK_TYPE MODE) RAISES (LOCKNOTGRANTED); //VERROUILLAGE
      BLOQUANT
   BOOLEAN TRY_LOCK(IN LOCK_TYPE MODE);// VERROUILLAGE NON BLOQUANT
   BOOLEAN SAME_AS(IN OBJECT ANOBJECT); // COMPARAISON D'IDENTIFIANTS
      D'OBJETS
   OBJECT COPY(); // COPIE AVEC GENERATION D'UN NOUVEL OBJET
   VOID DELETE()}; // SUPPRESSION D'UN OBJET
```

Figure XII.8 : Interface commune des objets

3.5. LES OBJETS STRUCTURÉS

Les objets structurés s'inspirent des types SQL2 utilisés pour la gestion du temps. L'intérêt de définir ces types comme des objets est de permettre de spécifier leur comportement sous forme d'interfaces. Des structures correspondantes sont aussi fournies dans les types de base proposés par l'ODMG. Les objets seuls offrent des opérations standard. Voici les types d'objets structurés supportés :

– `Date` représente un objet date par une structure (mois, jour, an) munie de toutes les opérations de manipulation classique des dates.
– `Interval` représente un objet durée par une structure (jour, heure, minute, seconde) munie de toutes les opérations de manipulation nécessaires, telles que l'addition, la soustraction, le produit, la division, les tests d'égalité, etc.
– `Time` représente les heures avec zones de temps, en différentiel par rapport au méridien de Greenwich ; l'unité est la milliseconde.
– `Timestamp` encapsule à la fois une date et un temps. Un objet timestamp permet donc une référence temporelle absolue en millisecondes.

3.6. LES COLLECTIONS

L'ODMG préconise le support de collections homogènes classiques de type SET<t>, BAG<t>, LIST <t> et ARRAY<t>. Une collection un peu moins classique est DICTIONARY<t,v>, qui est une collection de doublets <clé-valeur>. Toutes les collections héritent d'une interface commune `COLLECTION`, résumée figure XII.9. Celle-ci permet de créer des collections d'une taille initiale donnée par le biais d'une « ObjectFactory », puis de manipuler directement les collections pour récupérer leurs

propriétés (taille, vide ou non, ordre ou non, doubles permis ou non, appartenance d'un élément), pour insérer ou supprimer un élément, pour parcourir la collection par le biais d'un itérateur mono– ou bidirectionnel.

```
INTERFACE COLLECTIONFACTORY : OBJECTFACTORY {
   COLLECTION NEW_OF_SIZE(IN LONG SIZE)
};
INTERFACE COLLECTION : OBJECT {
   EXCEPTION INVALIDCOLLECTIONTYPE(), ELEMENTNOTFOUND(ANY ELEMENT);
   UNSIGNED LONG CARDINALITY();
   BOOLEAN IS_EMPTY(),IS_ORDERED(),ALLOWS_DUPLICATES(),
      CONTAINS_ELEMENT(IN ANY ELEMENT);
   VOID INSERT_ELEMENT(IN ANY ELEMENT);
   VOID REMOVE_ELEMENT(IN ANY ELEMENT) RAISES(ELEMENTNOTFOUND);
   ITERATOR CREATE_ITERATOR(IN BOOLEAN STABLE);
   BIDIRECTIONALTERATOR CREATE_BIDIRECTIONAL_ITERATOR()RAISES(INVALID
      COLLECTIONTYPE);
};
```

Figure XII.9 : Interface commune aux collections

Comme son nom l'indique, un itérateur permet d'itérer sur les éléments. Pour cela, il fournit une interface résumée figure XII.10. Un itérateur bidirectionnel permet d'aller en avant, mais aussi en arrière.

```
INTERFACE ITERATOR {
   VOID RESET() ; // INITIALISATION AU DÉBUT
   ANY GET_ELEMENT() RAISES(NOMOREELEMENTS); // OBTENTION ÉLÉMENT
   VOID NEXT_POSITION RAISES(NOMOREELEMENTS); // AVANCE POSITION
   REPLACE_ELEMENT (IN ANY ELEMENT) RAISES(INVALIDCOLLECTIONTYPE) ;
...
};
```

Figure XII.10 : Éléments d'interface pour les itérateurs

Chaque collection possède en plus des interfaces spécifiques classiques. La figure XII.11 présente les opérations spécifiques aux dictionnaires. Il s'agit de la gestion de doublets <clé-valeur>, la clé étant en principe unique. Elle correspond à un mot d'entrée dans le dictionnaire, la valeur étant sa définition (ou son synonyme). L'utilisateur peut lier une clé à une valeur (ce qui revient à insérer ce doublet), délier la clé (la supprimer), rechercher la valeur associée à une clé, et tester si une clé figure dans le dictionnaire. Les dictionnaires peuvent permettre par exemple de gérer des répertoires de noms d'objets, etc.

```
INTERFACE DICTIONARY : COLLECTION
   EXCEPTION KEYNOTFOUND(ANY KEY);
   VOID BIND(IN ANY KEY, IN ANY VALUE); // INSERTION
   VOID UNBIND (IN ANY KEY)RAISE(KEYNOTFOUND); // SUPPRESSION
   ANY LOOKUP (IN ANY KEY)RAISE(KEYNOTFOUND); // RECHERCHE
   BOOLEAN CONTAINS_KEY(IN ANY KEY) ;
};
```

Figure XII.11 : Interface spécifique aux dictionnaires

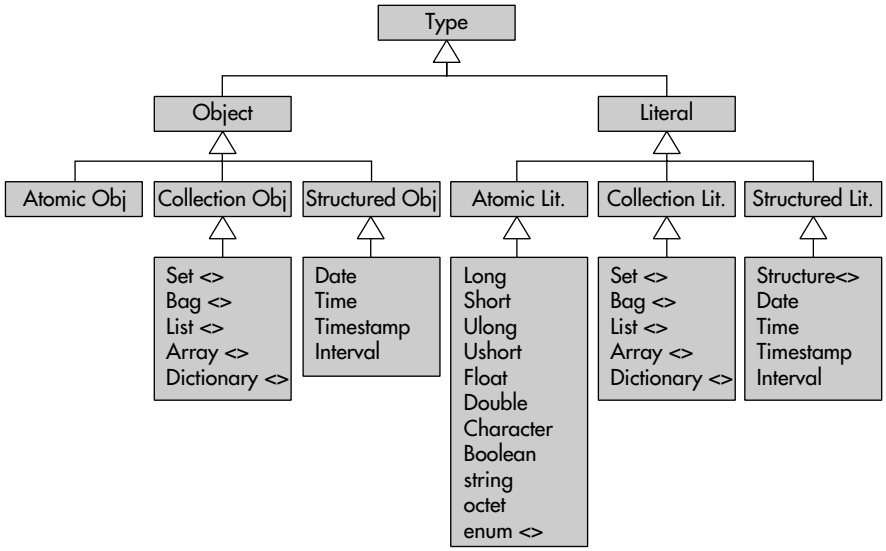


Figure XII.12 : La hiérarchie des types ODMG

3.7. LES ATTRIBUTS

Les attributs permettent de modéliser les états abstraits des objets. Un attribut est une propriété permettant de mémoriser un littéral ou un objet. Il peut être vu comme une définition abrégée de deux opérations : Set_value et Get_value. Un attribut possède un nom et un type qui précise ses valeurs légales. Il n'est pas forcément implémenté, mais peut être calculé.

3.8. LES ASSOCIATIONS (*RELATIONSHIPS*)

Les associations permettent de compléter la modélisation des états des objets. L'ODMG préconise le support d'associations binaires, bidirectionnelles de cardinalité

(1:1), (1:N), ou (N:M), sans données. Une association de A vers B définit deux chemins de traversée inverses, A->B et B->A. Chaque chemin doit être défini en ODL au niveau du type d'objet source par une clause `RELATIONSHIP`. L'association pointe vers un seul objet cible ou vers une collection. Elle porte un nom et son inverse doit être déclaré. Pour la gestion, le SGBDO doit fournir des opérations sur associations telles que `Add_member`, `Remove_member`, `Traverse` et `Create_iterator_for`. De fait, les associations sont simplement des déclarations abstraites d'attributs couplés, valués par une valeur simple ou une collection, contenant des identifiants d'objets réciproques. La figure XII.13 illustre l'association classique entre `VINS` et `BUVEURS`, mais sans données.

```
INTERFACE BUVEURS {
   ...
   RELATIONSHIP LIST<VINS> BOIRE INVERSE VINS::EST_BU_PAR;
   ...
};
INTERFACE VINS {
   ...
   RELATIONSHIP SET<BUVEURS> EST_BU_PAR INVERSE BUVEURS::BOIRE;
   ...
};
```

Figure XII.13 : Exemple de définition d'association

Lorsqu'une association est mise à jour, le SGBDO est responsable du maintien de l'intégrité : il doit insérer ou supprimer les références dans les deux sens si l'option inverse a été déclarée. Il ne doit pas y avoir de référence pointant sur des objets inexistants. Ce n'est pas le cas pour un attribut valué par un objet, d'où l'intérêt d'utiliser des associations plutôt que de définir directement les attributs supports. Par exemple, la déclaration au niveau des buveurs d'un attribut par la clause `ATTRIBUT BOIRE LIST<VINS>` n'implique pas l'existence d'un chemin inverse ni la gestion de l'intégrité référentielle par le SGBDO. Cependant, les associations de l'ODMG restent assez pauvres puisqu'elles ne peuvent avoir de données associées.

3.9. LES OPÉRATIONS

Classiquement, les opérations permettent de définir le comportement abstrait d'un type d'objets. Elles possèdent les propriétés habituelles des opérations sur objet : le nom de l'opération, le nom des conditions d'erreurs (après le mot clé `RAISES`), le nom et le type des arguments (après le mot clé `IN`), le nom et le type des paramètres retournés (après le mot clé `OUT`). L'ensemble constitue la signature de l'opération. La figure XII.14 illustre la définition de l'opération `boire` dans l'interface `buveurs`.

```
INTERFACE BUVEURS {
   ...
   INT BOIRE(IN VINS V, IN INT QTE) RAISES(NOVINS); //SIGNATURE DE
      L'OPÉRATION
...
};
```

Figure XII.14 : Un exemple de définition d'opération

3.10. MÉTA-MODÈLE DU MODELE ODMG

Dans sa nouvelle version 2.0, l'ODMG spécifie un méta-modèle objet, c'est-à-dire un modèle objet décrivant son modèle. Ce modèle, plutôt complexe, permet de définir le schéma d'un référentiel objet capable de gérer des schémas de bases de données objet conformes à l'ODMG. Nous en donnons une vue très simplifiée figure XII.15. Il faudrait tout d'abord ajouter les concepts de littéraux, type et interface dont classe hérite. D'autres concepts sont nécessaires, comme MetaObject, Scope, Module, Exceptions. Tout cela est défini dans [ODMG97].

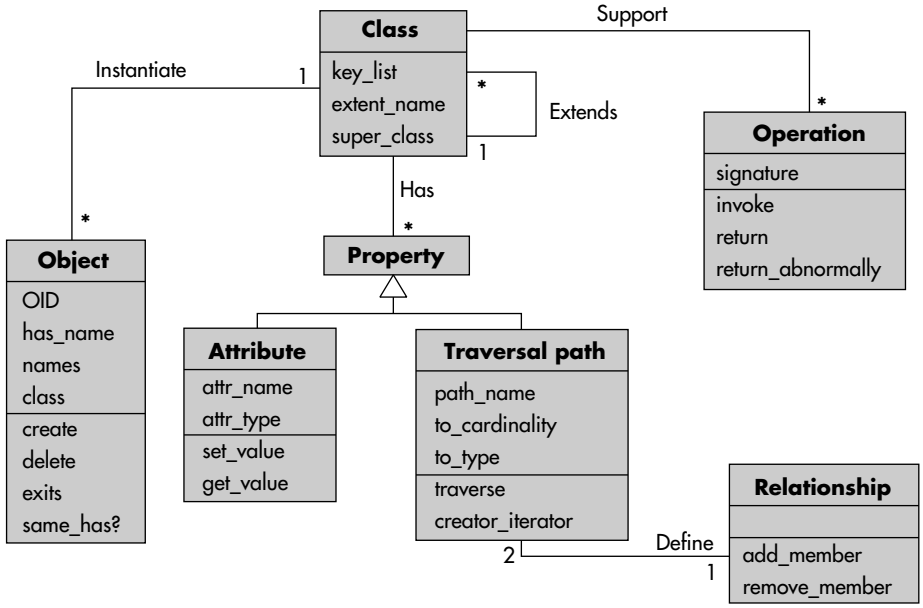


Figure XII.15 : Vue simplifiée du méta-modèle de l'ODMG

4. EXEMPLE DE SCHÉMA ODL

À titre d'illustration, la figure XII.17 définit le schéma d'une base de données objet dont le modèle est représenté figure XII.16. La base décrit simplement des situations du monde réel : des personnes possèdent des voitures, et habitent dans des appartements. Parmi elles, certaines sont employées, d'autres sont buveurs. Parmi les employés il y a des buveurs. Les buveurs boivent des vins.

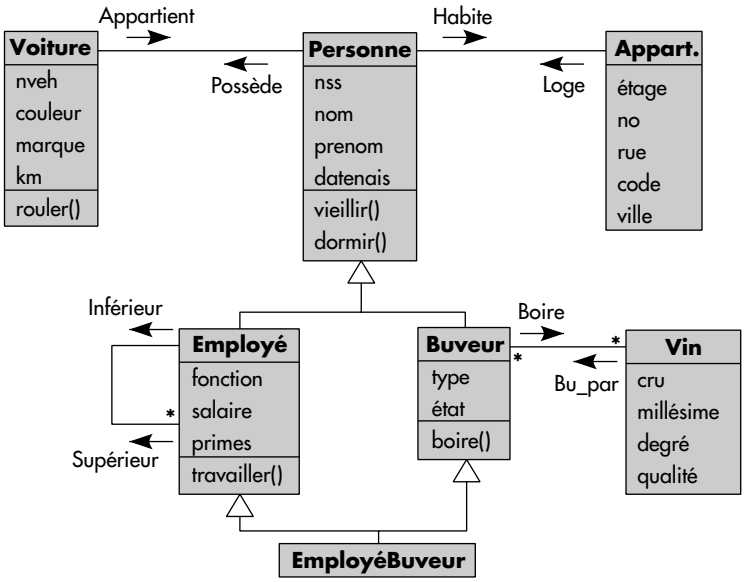


Figure XII.16 : Schéma graphique de la base exemple (en UML)

```
Class Voiture (extent voitures key nveh) { // classe avec extension
   attribute string nveh;
   attribute string couleur;
   attribute string marque;
   attribute short km;
   relationship Personne Appartient inverse Personne::Possede;
   short rouler(in short distance); };

Interface Personne { // interface abstraite pour implémentation dans classe
   attribute string nss ;
   attribute string nom ;
   attribute string prenom ;
   attribute date datenaissance;
   relationship Appart habite inverse Appart::loge; // relationship
      avec inverse
   relationship Voiture Possede inverse Voiture::Appartient;
```

```
   short vieillir();
   void dormir()
   short age(); };

class Employé : Personne(extent Employés key nss) { //classe avec extension
   attribute enum fonct{ingénieur, secrétaire, analyste, programmeur}
      fonction;
   attribute float salaire ;
   attribute list<float> primes ; //attribut multi-valué
   relationship Set<Employé> inferieur inverse supérieur;
   relationship Employé supérieur inverse inférieur;
   void travailler(); };

class Buveur : Personne(extent buveurs key nss) { // classe avec extension
   attribute typebuv{petit,moyen,gros} type;
   attribute etabuv{normal,ivre} etat;
   relationship list<Vin> boire inverse vin::bu_par;
   void boire(in Vin v); // paramètre d'entrée v };

class Appart (extent Apparts) { // classe avec extension
   attribute struct adresse (short etage, unsigned short numero, string rue,
   unsigned short code, string ville);
   relationship Set<personne> loge inverse Personne::habite; };

class Vin (extent Vins) { // classe avec extension
   attribute string cru;
   attribute string millesime;
   attribute string degre;
   attribute string qualite;
   relationship list<Buveur> bu_par inverse Vin::boire; };

class EmployéBuveur extends Employé { // classe sans extension hérite
   de employé
   attribute typebuv{petit,moyen,gros} type;
   attribute etabuv{normal,ivre} etat;
   relationship list<Vin> boire inverse vin::bu_par;
   void boire(in Vin v); // paramètre d'entrée v
};
```

Figure XII.17 : Schéma en ODL de la base exemple

Une décision à prendre lors de la définition est de choisir entre interfaces et classes. Certaines classes peuvent avoir des extensions. Nous avons choisi d'implémenter les extensions de `Voiture`, `Vin`, `Appartement`, `Buveur` et `Employé`. `Personne` devient une interface. L'héritage multiple n'étant pas possible au niveau des classes, les employés buveurs sont des employés avec les propriétés de buveurs répétées. On aurait pu éviter cette répétition en définissant une interface Buveurs, puis une classe CBuveurs implémentant cette interface. Nous ne l'avons pas fait pour ne pas perturber le lecteur. Le bon choix est probablement de définir des interfaces pour tout ce qui est visible à l'extérieur, puis de réaliser ces interfaces par des classes.

5. LE LANGAGE OQL

Cette section détaille le langage de requêtes OQL et introduit des profils de requêtes typiques.

5.1. VUE GÉNÉRALE

Le langage OQL a été défini à partir d'une première proposition issue du système O2 développé à l'INRIA. Les objectifs des auteurs étaient les suivants :

– permettre un accès facile à une base objet via un langage interactif autonome, mais aussi depuis un langage objet par intégration dans C++, Smalltalk, et plus tard Java ;

– offrir un accès non procédural pour permetttre des optimisations automatiques (ordonnancement, index…) ;

– garder une syntaxe proche de SQL au moins pour les questions exprimables en SQL ;

– rester conforme au modèle de l'ODMG, en permettant l'interrogation de toutes les collections d'objets – extensions de classes ou autres collections imbriquées ou non –, le parcours d'association, l'invocation d'opérations, la manipulation d'identifiants, le support de l'héritage et du polymorphisme, etc.

– permettre de créer des résultats littéraux, objets, collections…

– supporter des mises à jour limitées via les opérations sur objets, ce qui garantit le respect de l'encapsulation.

Ces objectifs sont-ils atteints ? Beaucoup le sont, mais il n'est pas sûr que le langage soit simple, au moins pour les habitués de SQL. La compatibilité avec SQL reste faible, la sémantique étant généralement différente, par exemple pour le traitement des valeurs nulles. Le langage est aussi faiblement assertionnel, en ce sens que les parcours de chemins doivent être explicitement exprimés. Pour le reste, le langage est très riche, plutôt élégant, et il existe une définition formelle, certes assez difficile à lire. La syntaxe est compacte.

Le langage est construit de manière fonctionnelle. Une question est une expression fonctionnelle qui s'applique sur un littéral, un objet ou une collection d'objets, et retourne un littéral, un objet ou une collection d'objets. Les expressions fonctionnelles peuvent être composées afin de constituer des requêtes plus complexes. Le langage est aussi fortement typé, chaque expression ayant un type qui peut être dérivé de la structure de l'expression et du schéma de la base. La correction de la composition des types doit être vérifiée par le compilateur OQL. Du point de vue du typage, OQL est peu permissif, et ne fait guère de conversions automatiques de types, à la différence de SQL. Syntaxiquement, beaucoup de questions sont correctes, mais erronées du

point de vue typage. OQL n'est pas seulement un langage d'interrogation, mais bien un langage de requêtes avec mises à jour. En effet, il est possible de créer des objets et d'invoquer des méthodes mettant à jour la base.

OQL permet aussi la navigation via les objets liés de la base, mais seulement avec des expressions de chemins monovaluées.

Il est possible de naviguer par des **expressions de chemins** un peu particulière, réduites à des chemins simples, que l'on définira comme suit :

Notion XII.8 : Expression de chemin monovalué (*Monovalued path expression*)

Séquence d'attributs ou d'associations monovaluées de la forme X1.X2…Xn telle que chaque Xi à l'exception du dernier contient une référence à un objet ou un littéral unique sur lequel le suivant s'applique.

Par exemple, voiture.appartient.personne.habite.adresse.ville est une expression de chemin valide sur la base de données exemple.

Pour parcourir les associations multivaluées, OQL utilise la notion de **collection dépendante**.

Notion XII.9 : Collection dépendante (*Depending collection*)

Collection obtenue à partir d'un objet, soit parce qu'elle est imbriquée dans l'objet, soit parce qu'elle est pointée par l'objet.

Par exemple, chaque buveur référence par l'association Boire une liste de Vins. Si B est un buveur, B.Boire est une collection dépendante, ici une collection de Vins. Par des variables ainsi liées du style B in Buveurs, V in B.Boire, OQL va permettre de parcourir les associations multivaluées. Ceci est puissant pour naviguer, mais plutôt procédural.

Pour présenter un peu plus précisément ce langage somme toute remarquable, nous procédons par des exemples un peu généralisés. Avant de les lire, il est bon de se rappeler que dans toute syntaxe une collection peut être remplacée par une requête produisant une collection. Réciproquement, il est possible de créer des collections intermédiaires et de remplacer chaque sous-requête par une collection intermédiaire. Cela peut permettre de simplifier des requêtes à nombreuses sous-requêtes imbriquées.

5.2. EXEMPLES ET SYNTAXES DE REQUÊTES

Nous présentons OQL à travers des exemples sur la base définie en ODL ci-dessus. Pour chaque exemple ou groupe d'exemples, nous abstrayons une syntaxe type dans un langage intuitif proche de la spécification de syntaxe en SQL : [a] signifie que a est

optionnel, [a]* signifie une liste de 0 à N a séparés par des virgules, [a]+ de 1 à N a séparés par des virgules, {a|b} signifie un choix entre a ou b, et... indique une syntaxe libre compatible avec celles déjà introduites. Cette syntaxe peut servir de profil (les *patterns* sont à la mode) pour formuler un type de requête. Nous appelons aussi l'expression syntaxique un profil, ou plus simplement un format. Chaque profil est précédé d'un nom de trois lettres suivi de « : » servant de référence pour un usage ultérieur éventuel. Nous donnons aussi le type du résultat inféré par le compilateur (après la requête, sous forme ===> type). Le langage étant complexe, nous ne donnons pas une définition, celle-ci pouvant être trouvée dans [ODMG97]. Il y en a d'ailleurs deux, une formelle, une autre en BNF, et elles ne semblent pas totalement cohérentes !

5.2.1. Calcul d'expressions

OQL permet de calculer des expressions arithmétiques de base, qui sont donc comme des questions.

```
(Q0)  ((STRING) 10*5/2) || "TOTO"
===>  string
```

Ceci donne en principe la chaîne « 25TOTO ». Cette requête montre à la fois le calcul d'expressions et les conversions de type. Son profil syntaxique est :

```
PEX: (<TYPE>) <EXPRESSION>
```

où l'expression peut être calculée avec tous les opérateurs classiques (+, *, –, /, mod, abs, concaténation). Plus généralement, l'expression peut aussi être une collection d'objets, et donc une requête produisant une telle collection, comme nous en verrons beaucoup ci-dessous.

5.2.2. Accès à partir d'objets nommés

OQL permet de formuler des questions sous forme d'expressions simples, en particulier construites à partir du nom d'un objet. Si MAVOITURE est le nom de l'objet désignant ma voiture, il est possible de demander :

```
(Q1)  MAVOITURE.COULEUR
===>  LITTÉRAL STRING
```

Plus généralement, OQL permet de naviguer dans la base en parcourant des chemins monovalués, comme vu ci-dessus. Par exemple, la question (Q2) retourne le nom du propriétaire de ma voiture (en principe, Gardarin !) :

```
(Q2)  MAVOITURE.APPARTIENT.NOM
===>  LITTÉRAL STRING
```

Nous appellerons de telles requêtes des **extractions d'objets**. Un profil correspondant à de telles questions est :

```
PEO: <NOM OBJET>.<CHEMIN>
```

où <CHEMIN> désigne une expression de chemins.

5.2.3. Sélection avec qualification

Nous retrouvons là les expressions de sélection du SQL de base. La notation de SQL pour la définition de variable derrière le FROM (du style BUVEURS AS B, ou BUVEURS B) peut d'ailleurs être utilisée. La requête suivante retrouve les noms et prénoms des gros buveurs :

```
(Q3)  SELECT B.NOM, B.PRENOM
      FROM B IN BUVEURS
      WHERE B.TYPE = "GROS"
==>   LITTÉRAL BAG<STRUCT<NOM:STRING,PRENOM:STRING>>
```

La syntaxe générale de sélections simples est:

```
PSE:  SELECT [<VARIABLE>.<ATTRIBUT>]+
      FROM <VARIABLE> IN <COLLECTION>
      WHERE <VARIABLE>.<ATTRIBUT> <COMPARATEUR> <CONSTANTE>
```

5.2.4. Expression de jointures

OQL permet les jointures, tout comme SQL, avec une syntaxe similaire. Par exemple, la requête suivante liste les noms et prénoms des employés gros buveurs :

```
(Q4)  SELECT B.NOM, B.PRENOM
      FROM B IN BUVEURS, E IN EMPLOYÉS
      WHERE B.NSS = E.NSS AND B.TYPE = "GROS"
==>   LITTÉRAL BAG<STRUCT<NOM:STRING,PRENOM:STRING>>
```

Le profil syntaxique d'une requête de sélection avec un ou plusieurs critères de sélection et une ou plusieurs jointures est :

```
(PSJ) SELECT [<VARIABLE>.<ATTRIBUT>]+
      FROM <VARIABLE> IN <COLLECTION>,[<VARIABLE> IN <COLLECTION>]+
      [WHERE <VARIABLE>.<ATTRIBUT> <COMPARATEUR> <VARIABLE>.<ATTRIBUT>
      [AND <VARIABLE>.<ATTRIBUT> <COMPARATEUR> <VARIABLE>.<ATTRIBUT>]*
      [{AND|OR} <VARIABLE>.<ATTRIBUT> <COMPARATEUR> <CONSTANTE>]]*
```

De manière générale, les qualifications peuvent faire intervenir, comme en SQL, des AND et des OR, et des expressions parenthésées contenant ces connecteurs logiques.

Une question équivalente peut utiliser le type EMPLOYEBUVEUR comme suit :

```
(Q5)  SELECT ((EMPLOYÉBUVEUR)B).NOM, ((EMPLOYÉBUVEUR)B).PRENOM
      FROM B IN BUVEURS
      WHERE B.TYPE = "GROS"
==>   LITTÉRAL BAG<STRUCT<NOM:STRING,PRENOM:STRING>
```

L'évaluateur doit alors vérifier à l'exécution l'appartenance des gros buveurs trouvés à la classe des EMPLOYEBUVEUR.

Le profil de telles questions avec indicateurs de classe est :

```
PIC:  SELECT [((<CLASSE>)<VARIABLE>).<ATTRIBUT>]+
      FROM...
      [WHERE...]
```

5.2.5. Parcours d'associations multivaluées par des collections dépendantes

Les collections dépendantes se parcourent par des variables imbriquées derrière le FROM. Ceci permet de traverser des chemins multivalués correspondant à des associations [1:N] ou [M:N]. Par exemple, la requête suivante retrouve le nom et le prénom des buveurs qui ont bu du Volnay :

```
(Q6)  SELECT B.NOM, B.PRENOM
      FROM B IN BUVEURS, V IN B.BOIRE
      WHERE V.CRU = "VOLNAY"
==>   LITTÉRAL BAG<STRUCT<NOM:STRING,PRENOM:STRING>
```

Le profil syntaxique général d'une requête de parcours de collections dépendantes est :

```
PCD:  SELECT [<VARIABLE>.<ATTRIBUT>]+
      FROM...,[<VARIABLE> IN <VARIABLE>.<ATTRIBUT>]+
      [WHERE... ]
```

Bien sûr, la clause WHERE peut être composée de sélections, de jointures, etc.

5.2.6. Sélection d'une structure en résultat

Il est possible de sélectionner une structure en résultat ; c'est d'ailleurs l'option par défaut que nous avions utilisée jusque là, les noms des attributs étant directement ceux des attributs sources. La requête suivante permet de retrouver des doublets (Name, City) pour chaque gros buveur :

```
(Q7)  SELECT STRUCT (NAME: B.NOM, CITY: B.HABITE.ADRESSE.VILLE)
      FROM B IN BUVEURS
      WHERE B.TYPE = 'GROS'
===>  LITTÉRAL BAG <STRUCT(NAME,CITY)>
```

Par similarité avec SQL, la collection résultat est un BAG. Il est aussi possible d'obtenir une collection de type SET en résultat ; on utilise alors le mot clé DISTINCT, comme en SQL :

```
(Q8)  SELECT DISTINCT (NAME: B.NOM, CITY: B.HABITE.ADRESSE.VILLE)
      FROM B IN BUVEURS
      WHERE B.NOM = 'DUPONT'
===>  LITTÉRAL SET <STRUCT(NAME, CITY)>
```

Le mot clé STRUCT étant implicite, le profil correspondant à ce type de requêtes est :

```
PST:  SELECT [DISTINCT][STRUCT]([<ATTRIBUT>: <VARIABLE>.<CHEMIN>]+)
      FROM...
      [WHERE...]
```

5.2.7. Calcul de collections en résultat

Plus généralement, le résultat d'une requête peut être une collection quelconque de littéraux ou d'objets. Par exemple, la requête suivante fournit une liste de structures :

```
(Q9)  LIST(SELECT STRUCT(NOM: B.NOM, VILLE: B.HABITE.ADRESSE.VILLE)
      FROM B IN BUVEURS
      WHERE B.NOM = "DUPONT")
===>  LITTÉRAL LIST <STRUCT(NOM, VILLE)>
```

Le profil syntaxique plus général est :

```
PCS:  <COLLECTION>     (SELECT [DISTINCT]
                       [STRUCT] ([<ATTRIBUT>: <VARIABLE>.<CHEMIN>]+)
                       FROM...
                       [WHERE...])
```

5.2.8. Manipulation des identifiants d'objets

Les identifiants sont accessibles à l'utilisateur. Il est par exemple possible de retrouver des collections d'identifiants d'objets par des requêtes du type :

```
(Q10) ARRAY(SELECT V
      FROM V IN VOITURES
      WHERE B.MARQUE = "RENAULT")
===>  LITTÉRAL ARRAY(<OID>)
```

dont le profil est :

```
POI:  <COLLECTION>     (SELECT <VARIABLE>
                  FROM...
                  [WHERE...])
```

5.2.9. Application de méthodes en qualification et en résultat

La question suivante sélectionne les employés ainsi que leur ville d'habitation et leur âge, dont le salaire est supérieur à 10 000 et l'âge inférieur à 30 ans (age() est une méthode) :

```
(Q11) SELECT DISTINCT E.NOM, E.HABITE.ADRESSE.VILLE, E.AGE()
      FROM E IN EMPLOYÉS
      WHERE E.SALAIRE > 10000 AND E.AGE() < 30
===>  LITTÉRAL DE TYPE SET <STRUCT>
```

Elle peut être généralisée au profil suivant :

```
PME:  SELECT   [DISTINCT]...,
               [<VARIABLE>.<CHEMIN>.<OPERATION>([<ARGUMENT>]*)]*
      FROM...
      [WHERE....
      [AND<VARIABLE>.<CHEMIN>.<OPERATION>([<ARGUMENT>]*) <COMPARATEUR>
               <CONSTANTE>]*]
```

5.2.10. Imbrication de SELECT en résultat

Afin de construire des structures imbriquées, OQL permet d'utiliser des SELECT imbriqués en résultat de SELECT. Par exemple, la question suivante calcule pour chaque employé une structure comportant son nom et la liste de ses inférieurs mieux payés :

```
(Q12) SELECT   DISTINCT STRUCT (NOM : E.NOM, INF_MIEUX_PAYES :
               LIST (SELECT I
               FROM I IN E.INFERIEUR
               WHERE I.SALAIRE > E.SALAIRE))
      FROM E IN EMPLOYÉS
===>  LITTÉRAL DE TYPE SET <STRUCT (NOM: STRING, INF_MIEUX_PAYÉS : LIST
               <EMPLOYÉS>)>
```

Voici un profil possible pour une telle question :

```
PQI:  SELECT [DISTINCT][STRUCT](...,[<ATTRIBUT>:<QUESTION>...]*)
      FROM...
      [WHERE...]
```

5.2.11. Création d'objets en résultat

Il est possible de créer des objets dans une classe par le biais de constructeurs ayant pour arguments des requêtes. Bien sûr, des constructeurs simples peuvent être appliqués pour insérer des objets dans des extensions de classes, par exemple la requête :

```
(Q13) EMPLOYÉ (NSS:15603300036029, NOM:"JEAN", SALAIRE:100000).
```

Plus généralement, il est possible d'employer tous les buveurs sans emploi par la requête suivante :

```
(Q14) EMPLOYÉ(SELECT STRUCT(NSS : B.NSS, NOM: B.NOM, SALAIRE : 4000)
      FROM B IN BUVEURS
      WHERE NOT EXIST E IN EMPLOYÉS : E.NSS=B.NSS)
==>   BAG<EMPLOYÉS> INSÉRÉ DANS EMPLOYÉS
```

Notez que cette requête utilise une quantification que nous allons expliciter plus loin. Le profil général de ces requêtes de création d'objets est donc :

```
      PCO : <CLASSE>(<QUESTION>).
```

5.2.12. Quantification de variables

OQL propose des opérateurs de quantification universelle et existentielle directement utilisables pour composer des requêtes, donc des résultats ou des qualifications.

5.2.12.1. Quantificateur universel

La question suivante retourne vrai si tous les employés ont moins de 100 ans :

```
(Q15) FOR ALL P IN EMPLOYÉS : P.AGE < 100
```

Son profil est :

```
PQU:  FOR ALL <VARIABLE> IN <QUESTION> : <QUESTION>
```

5.2.12.2. Quantificateur existentiel

La requête suivante retourne vrai s'il existe une voiture de marque Renault possédée par un buveur de plus de 60 ans :

```
(Q16) EXISTS V IN SELECT V
      FROM V IN VOITURES, B IN V.POSSEDE
      WHERE V.MARQUE = "RENAULT": B.AGE() > 60.
```

Un profil généralisé possible est :

```
PQE:  EXISTS <VARIABLE> IN <QUESTION> : <QUESTION>
```

5.2.13. Calcul d'agrégats et opérateur GROUP BY

Dans la première version, les agrégats étaient calculables par utilisation d'un opérateur fonctionnel GROUP applicable sur toute requête. La version 2 est revenue à une syntaxe proche de celle de SQL pour des raisons de compatibilité. Les attributs à grouper sont généralement définis par une liste d'attributs, alors que le critère de groupement peut être explicitement définies par des prédicats, ou implicitement par une liste d'attributs. Dans le dernier cas, le mot clé PARTITION peut être utilisé pour désigner chaque collection résultant du partitionnement. Les deux requêtes suivantes illustrent ces possibilités :

```
(Q17) SELECT E
      FROM E IN EMPLOYÉS
      GROUP BY (BAS : E.SALAIRE < 7000,
               MOYEN : E.SALAIRE £ 7000 AND E.SALAIRE < 21000,
               HAUT : E.SALAIRE ≥ 21000)
===>  STRUCT<BAS: SET(EMP.), MOYEN:SET(EMP.),HAUT:SET(EMP.)>

(Q18) SELECT STRUCT(VILLE: E.HABITE.ADRESSE.VILLE, MOY : AVG(SALAIRE))
      FROM E IN EMPLOYÉS
      GROUP BY E.HABITE.ADRESSE.VILLE
      HAVING AVG (SELECT E.SALAIRE FROM E IN PARTITION)> 5000
===>  BAG <STRUCT (VILLE: STRING, MOY: FLOAT)>
```

```
PAG:  SELECT...[<AGGREGAT>(<ATTRIBUT>)]*
      FROM...
      [WHERE...]
      GROUP BY {<ATTRIBUT>+ | <PREDICAT>+}
      HAVING <PRÉDICAT>
```

AGGREGAT désigne une fonction d'agrégats choisie parmi COUNT, SUM, MIN, MAX et AVG. PREDICAT désigne un prédicat expression logique de prédicats simples de la forme <ATTRIBUT> <COMPARATEUR> <CONSTANTE> ou <AGGREGAT>(PARTITION) <COMPARATEUR> <CONSTANTE>. On voit donc que OQL généralise les agrégats de SQL2 en permettant de grouper par prédicats explicites. C'est aussi possible en SQL3 avec d'autres constructions.

5.2.14. Expressions de collections

Il est possible de manipuler directement les collections en appliquant des opérateurs spécifiques. Les résultats de requêtes sont souvent des collections, si bien que les opérateurs peuvent être appliqués aux requêtes. Voici quelques exemples d'opérateurs.

5.2.14.1. Tris

La première syntaxe était fonctionnelle. La nouvelle est copiée sur SQL, comme suit :

```
(Q19) SELECT E.NOM, E.SALAIRE
      FROM E IN EMPLOYÉS
      WHERE E.SALAIRE > 21000
      ORDER BY DESC E.SALAIRE
```

Le profil est donc analogue à celui de SQL :

```
PTR:  <COLLECTION>
      ORDER BY {DESC|ASC} <ATTRIBUT>+
```

5.2.14.2. Union, intersection, différence

Comme en SQL, union, intersection et différence de questions sont possibles. Par exemple :

```
(Q20) EMPLOYÉS UNION BUVEURS
```

est une requête valide.

Le profil général de ce type de requête est :

```
PEN : <COLLECTION> {INTERSECT|UNION|EXCEPT} <COLLECTION>
```

Les priorités sont selon l'ordre indiqué, les parenthèses étant aussi possibles.

5.2.14.3. Accesseurs aux collections

Il est possible d'extraire un élément d'une collection en utilisant une fonction d'accès générale, comme first ou last, ou plus généralement des fonctions d'accès spécifiques

au type de collection. Les constructeurs de collections struct, set, list, bag, array sont aussi utilisables pour construire des collections comme nous l'avons déjà vu. Ainsi :

```
(Q21) SELECT B.NOM, FIRST(B.BOIRE)
      FROM B IN BUVEURS
```

sélectionne le premier vin bu par un buveur.

Plus généralement :

```
PFC : FIRST(<COLLECTION>) |
      LAST(<COLLECTION>) |
      <FONCTION>(<COLLECTION>)
```

sont des profils admissibles.

5.2.15. Conversions de types appliquées aux collections

Comme nous l'avons vu, le résultat d'une question est souvent une collection. Réciproquement, une collection correctement spécifiée est une question dont la réponse est la liste de ses éléments. Tout opérateur applicable à une ou plusieurs collections l'est aussi à une ou plusieurs questions. C'est particulièrement le cas des opérateurs ci-dessus.

Comme il existe différents types de collections, il est possible d'appliquer des opérateurs de conversion, par exemple LISTOSET qui traduit une liste en un ensemble et DISTINCT qui traduit toute collection en un ensemble. De plus, il est possible de transformer une collection ayant un seul élément en un singleton par l'opérateur ELEMENT. Par exemple, la question suivante retrouve la marque de la voiture de numéro 120 ABC 75 :

```
(Q22) ELEMENT (SELECT V.MARQUE
      FROM V IN VOITURES
      WHERE V.NUMÉRO = "120 ABC 75")
===>  STRING
```

Les collections apparaissant en résultat de requêtes sont parfois imbriquées. Les collections peuvent être aplaties à l'aide de l'opérateur FLATTEN. Par exemple :

```
(Q23) FLATTEN (SELECT B.NOM, SELECT V.MILLÉSIME
                             FROM V IN B.BOIRE
                             WHERE V.CRU = "VOLNAY"
      FROM B IN BUVEURS)
```

sélectionne simplement des doublets nom de buveurs et millésime de Volnay pour les buveurs ayant bu du Volnay.

En général, les profils permis sont donc :

```
PCO : LISTTOSET(<COLLECTION>)|
      ELEMENT(<COLLECTION>)|
      DISTINCT(<COLLECTION>)|
      FLATTEN(<COLLECTION>)
```

5.2.16. Définition d'objets via requêtes

Finalement, OQL permet aussi de nommer le résultat d'une requête. Celui-ci apparaît alors comme un littéral ou un objet nommé (selon le type du résultat). Par exemple, la requête suivante obtient un ensemble d'objets buveurs nommé `IVROGNE` :

```
(Q24) DEFINE IVROGNES AS
      SELECT DISTINCT B
      FROM B IN BUVEURS
      WHERE B.TYPE = "GROS"
```

Le profil général est simplement :

```
PDE : DEFINE <IDENTIFIANT> AS <QUESTION>.
```

5.3. BILAN SUR OQL

OQL est un langage de requête d'essence fonctionnelle très puissant. Nous n'avons donné que quelques exemples de possibilités avec une syntaxe approximative. En fait, OQL est un langage hybride issu d'un mélange d'un langage fonctionnel et d'un langage assertionnel comme SQL. D'ailleurs, SQL est lui-même un mélange d'un langage d'opérateurs (ceux de l'algèbre relationnelle) et d'un langage logique purement assertionnel (QUEL était proche d'un tel langage). Tout cela devient très complexe et la sémantique est parfois peu claire, quoi qu'en disent les auteurs. Pour être juste, disons que SQL3, décrit au chapitre suivant, souffre des mêmes défauts.

6. OML : L'INTÉGRATION À C++, SMALLTALK ET JAVA

6.1. PRINCIPES DE BASE

Le modèle de l'ODMG est abstrait, c'est-à-dire indépendant de toute implémentation. Pour l'implémenter dans un langage objet spécifique, il est nécessaire d'adapter le modèle. Par exemple, il faut préciser ce que devient un littéral dans un langage pur objet tel Smalltalk, ce que devient une interface en C++, ou une association en Java. Les types de données ODMG doivent être implémentés avec ceux du langage pour conserver un système de type unique. Il faut aussi pouvoir invoquer les requêtes OQL depuis le langage et récupérer les résultats dans des types intégrés au langage, par exemple comme des objets du langage.

L'ODMG propose trois intégrations de son modèle et de son langage de requêtes, une pour C++, une autre pour Smalltalk, une enfin pour Java. Autrement dit, la vision de l'usager d'une BD conforme aux spécifications de l'ODMG est précisée pour ces trois langages objet. Les objectifs essentiels de ces propositions sont :

1. La **fourniture d'un système de types unique à l'utilisateur**. En principe, le système de types utilisé est celui du langage de programmation parfois étendu, par exemple avec des collections. En effet, les vendeurs de SGBDO ayant longtemps décrié les SGBD relationnels pour l'hétérogénéité des types des langages de programmation et de SQL, ils se devaient d'éviter cet écueil.
2. Le **respect du langage de programmation**. Cela signifie que ni la syntaxe ni la sémantique du langage ne peuvent être modifiées pour accommoder la manipulation des données. Des classes supplémentaires seront généralement fournies, mais le langage de base ne sera pas modifié.
3. Le **respect du standard abstrait ODMG.** Ceci signifie que l'intégration proposée doit donner accès à toutes les facilités abstraites du modèle ODMG et de son langage de requêtes OQL. Ce principe n'est que partiellement suivi dans l'état actuel du standard, certaines facilités n'étant pas supportées, par exemple les associations en Java. Mais ceci devrait être le cas dans une future version.

Pour illustrer ces principes, nous traitons ci-dessous de l'intégration à Java, celles avec les autres étant similaires, mais un peu plus complexes. Auparavant, nous allons discuter des interfaces générales de gestion des bases de données et des transactions qui doivent être fournies.

6.2. CONTEXTE TRANSACTIONNEL

L'accès aux bases de données s'effectue dans un contexte transactionnel. Le langage doit donc fournir des transactions implémentées par le SGBDO. Plus précisément, une transaction est un objet créé par une `TransactionFactory` qui implémente l'interface comportant les opérations suivantes :

– `begin()` pour ouvrir une transaction ;
– `commit()` pour valider les mises à jour de la transaction, relâcher les verrous obtenus et terminer la transaction avec succès ;
– `abort()` pour défaire les mises à jour de la transaction, relâcher les verrous obtenus et terminer la transaction avec échec ;
– `checkpoint()` qui a le même effet que commit() suivi de begin(), mais ne relâche pas les verrous ;
– `join()` pour récupérer l'objet transaction récepteur sous contrôle du contexte d'exécution (*thread* courante) ;

– `leave()` pour dissocier un objet transaction du contexte d'exécution courant ;
– `isOpen()` pour tester si la transaction est ouverte.

Ce type d'opérations sur transactions est commun, à l'exception de `join()` et `leave()` prévus pour permettre à un contexte d'exécution de gérer plusieurs transactions, ou à l'inverse pour permettre à une transaction de traverser plusieurs contextes d'exécution.

Avant d'exécuter des opérations au sein de transactions, un utilisateur doit ouvrir un objet bases de données. Plus généralement, l'implémentation doit fournir des objets `Database` (créés à partir d'une `DatabaseFactory`) qui supportent les opérations suivantes :

– `open(in string nom-de-base)` pour ouvrir une base de nom donné afin d'y accéder via le langage en transactionnel ;
– `close()` pour fermer une base ;
– `bind(in any unObjet, in string nom)` pour donner un nom à l'objet passé en paramètre; c'est l'opération de base pour nommer un objet dans une base de données, le nom ne devant pas déjà exister dans la base; un même objet peut avoir plusieurs noms ;
– `Object unbind(in string nom)` pour enlever un nom à un objet précédemment nommé ;
– `Object lookup(in string nom)` pour retrouver et charger comme un objet du langage l'objet de la base de nom donné.

L'accès depuis un langage de programmation objet à une base ODMG nécessite donc l'implémentation de ces interfaces dans le langage objet. Ceci est simple puisque le SGBDO doit supporter ces opérations, les interfaces *`Transaction`* et *`Database`* étant partie intégrante du standard ODMG.

6.3. L'EXEMPLE DE JAVA

Java est un langage objet à la mode, plus simple et plus sûr que C++, idéal pour réaliser des applications distribuées robustes. Java comporte des littéraux de base (entier, réel, string, etc.) et des objets instances de classes. Il ne supporte pas les associations autrement que par des références inter-classes. Tout ceci en fait un langage de choix pour développer des applications bases de données. Nous examinons brièvement les différents aspects de Java OML ci-dessous.

6.3.1. La persistance des objets

Pour pouvoir être persistant, un objet doit être instance d'une classe connue du SGBDO. Plusieurs moyens sont possibles pour faire connaître une classe Java au

SGBDO : définir la classe en ODL et écrire un compilateur ODL générant des définitions de classes Java, avoir un préprocesseur reconnaissant les classes persistantes ou ajouter une interface spécifique de persistance à ces classes, etc. Le standard actuel ne se prononce pas sur le moyen de déclarer une classe Java persistante. De telles classes sont supposées existantes et connues du SGBDO. Elles peuvent alors contenir des objets persistants, mais aussi toujours des objets transients.

Alors, comment rendre un objet (d'une classe persistante) persistant ? Le mécanisme retenu est la **persistance par atteignabilité**. Les racines de persistance sont les objets nommés au moyen de l'opération `bind` de l'interface base de données vue ci-dessus. Tout objet référencé par un objet persistant est persistant. Tout objet non référencé par un objet persistant doit être détruit automatiquement de la base, ce qui sous-entend l'existence d'un ramasse-miettes sur disques. Au-delà, le standard propose que les objets persistants puissent conserver des attributs non persistants, ce qui n'est sans doute pas simple à déclarer sans modifier Java.

6.3.2. Les correspondances de types

Les types de l'ODMG sont traduits dans des types Java. Les littéraux `long`, `short`, `float`, `double`, `boolean`, `octet`, `char` et `string` sont traduits en type primitif Java correspondant de même nom, sauf `long` qui devient `int` et `octet` qui devient `byte`. Java propose pour chaque type primitif une représentation valeur et une représentation objet par des instances de la classe de même nom : les deux traductions sont possibles. Les types structurés `date`, `time` et `timestamp` sont traduits comme des objets définis dans le package Java.sql de JDBC, le standard d'accès aux bases relationnelles.

6.3.3. Les collections

L'interface `Collection` et ses extensions `Set`, `Bag`, `List` et `Array` sont implémentées en Java avec des opérations similaires, mais harmonisées avec Java. Java supporte le type de collection `Array` de plusieurs manières : les `Array` sont des tableaux monodimensionnels de taille fixe intégré au langage, `Vector` est une classe standard supportant des tableaux dynamiques. Il est possible d'utiliser `Array` ou `Vector` pour implémenter une classe `Varray` conforme au standard ODMG.

6.3.4. La transparence des opérations

Toute opération exécutable en Java sur des objets transients l'est également sur des objets persistants. C'est au SGBDO d'assurer la transparence à la persistance, par exemple de gérer les références entre objets persistants, de lire les objets persistants référencés non présents en mémoire, et de reporter l'état des objets modifiés dans la

base à la validation de transaction. Les techniques à mettre en œuvre sont spécifiques à chaque SGBD, mais doivent être transparentes à l'utilisateur.

6.3.5. Java OQL

L'intégration d'OQL s'effectue de deux manières. Tout d'abord l'ajout d'une opération Query(`String predicate`) permet de filtrer les objets d'une collection avec un prédicat OQL. En supposant que `Vins` désigne une collection d'objets persistants ou non, on pourra par exemple retrouver les bons vins par la construction suivante :

```
SET<OBJECT> LESBONSVINS;
LESBONSVINS = VINS.QUERY("QUALITÉ = "BONS"").
```

De manière plus complète, une classe OQLQuery est proposée afin de composer des requêtes paramétrées OQL sous forme d'objets, de demander leur exécution et de récupérer les résultats dans des objets Java, en général des collections. La classe OQLQuery comporte plus précisément les opérations :

– `OQLQuery(String question)` pour créer des requêtes avec le texte paramétré (les paramètres sont notés `$i`) en OQL passé en argument ;

– `OQLQuery()` pour créer des requêtes génériques, le texte étant passé ensuite par l'opération `create(String query)` ;

– `bind(Object parameter)` pour lier le premier paramètre non encore lié de la requête ;

– `execute()` pour exécuter une requête préalablement construite et récupérer un objet Java de type collection ou simple en résultat.

Voici une illustration simple de cette classe pour retrouver les buveurs d'un cru quelconque, par exemple de Volnay ; Java ne supportant pas les associations, on supposera que l'attribut Boire de buveurs est implémenté comme une collection de vins :

```
SET BUVEURSCRU;
QUERY = OQLQUERY  ("SELECT DISTINCT B FROM B IN BUVEURS,
                   V IN B.BOIRE WHERE V.CRU = $1");
QUERY.BIND("VOLNAY");
BUVEURSCRU = QUERY.EXECUTE();
```

6.3.6. Bilan

Java OML propose une vision simple d'une base de données ODMG en Java. La première version de cette proposition est assez brièvement décrite dans le standard. Nous en avons donné un aperçu ci-dessus. La clé du problème semble être dans le support des collections en Java, qui est loin d'être standard. Le problème de l'ODMG pourrait être à terme de faire coïncider son standard avec le standard Java, notamment pour les collections. Sinon, il y aurait vraiment opposition de phase (*impedance mismatch*), ce qui serait paradoxal ! Ceci ne serait cependant pas étonnant, car peut-on construire

deux standards (un pour les bases de données, l'autre pour le langage) indépendamment et conserver un système de type unique ? Une meilleure approche est probablement de développer un langage de programmation basé sur les types du SGBD pour éviter les problèmes de conversion de types. C'est ce que propose SQL3, comme nous le verrons dans le chapitre qui suit.

7. CONCLUSION

Depuis 1991, les vendeurs de SGBDO tentent de créer un standard afin d'assurer la portabilité des programmes utilisateurs, donc des applications. Cette démarche est importante car elle seule peut garantir la pérennité de l'approche bases de données objet pour les utilisateurs. Deux versions de ce « standard » ont été publiées, l'une en 1993, l'autre en 1997. Nous nous sommes basés sur la version 1997 (ODMG 2.0) pour écrire ce chapitre. Bien que remarquable, la version 2 est quelque peu complexe et n'est guère supportée par les systèmes actuels (ni d'ailleurs celle de 1993). Aussi, elle est parfois inégale dans le niveau de détails fourni. Elle se compose d'un modèle abstrait de bases de données objet et d'interfaces concrètes pour chacun des langages supportés.

En outre, le langage de requêtes OQL constitue une contribution importante de cette proposition. OQL offre les fonctionnalités du SQL de base avec en plus des extensions pour manipuler les expressions de chemins, les méthodes et surtout les collections. C'est un langage très puissant, mais complexe, qui a aussi quelques déficiences, comme l'absence de contrôle de droits et de gestion de vues. Il peut être intégré à C++, Smalltalk et Java de manière standard. Tout ceci est positif.

Alors, peut-on enfin parler de portabilité des applications des SGBDO, par exemple écrites en Java ? Il y a pour l'instant peu de retour d'expériences. De plus, les vendeurs de SGBDO, à l'exception sans doute de O2, n'implémentent pas OQL et ne se soucient finalement guère plus que dans le discours du standard ODMG. Pour garantir une certaine portabilité, il faut donc utiliser du pur Java, sans requête. C'est encore faible. Mais que les adeptes des langages objets, et particulièrement les défenseurs de Java se rassurent : il n'y a pas besoin de SGBDO conforme à l'ODMG pour faire persister et retrouver des objets Java et garantir la portabilité des applications. JDBC, développé par SunSoft, est un package vraiment standard pour faire persister et retrouver via SQL des objets Java dans toute base de données, y compris d'ailleurs certaines bases de données objet. Vous pouvez aussi regarder du côté de Persistant Java et d'interfaces plus spécifiques comme JSQL.

8. BIBLIOGRAPHIE

[Adiba93] Adiba M., Collet C., *Objets et Bases de Données : Le SGBD O2*, Hermès, Paris, 1993.

Cet ouvrage de 542 pages décrit la génération des SGBD objet et détaille plus particulièrement le SGBD O2. Il décrit tout le cycle d'utilisation de ce SGBD et les principaux composants du système.

[Bancilhon89] Bancilhon F., Cluet S., Delobel C., « A Query Language for an Object-Oriented Database System », *In 2nd Intl. Workshop on Database Programming Language*, DBPL, p. 301-322, 1989.

Cet article introduit la première version du langage de requêtes du système O2, précurseur du langage OQL. Cette version souligne les aspects fonctionnels du langage.

[Bancilhon92] Bancilhon F., Delobel C., Kanellakis P., *Building an Object-Oriented Database System : The Story of O2*, Morgan Kaufmann Pub., San Fransisco, CA, 1992.

Ce livre est une collection d'articles présentant l'histoire, la conception et l'implémentation du système O2, un des premiers SGBD pur objet.

[Chaudhri98] Chaudhri A.B., « Workshop Report on Experiences Using Object Data Management in the Real-World », *SIGMOD Record*, V. 27, n° 1, p. 5-10, Mars 1998.

Cet article résume les présentations effectuées à un Worshop portant sur les applications réelles des BD objet. Il discute en particulier de quelques insuffisances d'ObjectStore et des multiples différences entre O2 et le standard ODMG.

[Ferrandina95] Ferrandina F., Meyer T., Zicari R., Ferran G., Madec J., « Schema and Database Evolution in the O2 Object Database System », *Proc. of 21st Intl. Conf. On Very Large Databases*, Zurich, p. 170-181, 1995.

Cet article souligne la difficulté d'évolution des schémas ODL dans une base de données objet. Il décrit les algorithmes implémentés dans O2 pour faire évoluer la base en conformité au schéma.

[Fishman88] Fishman *et. al.*, *Overview of the IRIS DBMS*, Hewlett-Packard Internal Report, Palo Alto, 1988.

Cet article présente le système IRIS et son langage OSQL. OSQL est une version fonctionnelle d'un SQL étendu aux objets. OSQL est un langage qui permet de définir des types abstraits comme des ensembles de fonctions, de créer des classes d'objets accessibles par des fonctions, puis d'appliquer ces fonctions en les imbriquant à partir d'arguments éventuellement instanciés. Il s'agit d'un des premiers SQL étendus aux objets basé sur une approche fonctionnelle, donc un précurseur de OQL.

[ODMG93] Cattell R. (Ed.), *Object Databases : The ODMG-93 Standard*, Morgan-Kaufman, San Matéo, CA, 1993.

Ce livre présente la première version du standard ODMG pour les bases de données. Il décrit le modèle objet, le langage de définition ODL, le langage de requêtes OQL et les interfaces avec C++ et Smalltalk.

[ODMG97] Cattell R.G.G., Barry D., Bartels D., Berler M., Eastman J., Gammerman S., Jordan D., Springer A., Strickland H., Wade D., *Object Database Standard : ODMG 2.0,* Morgan Kaufmann Pub., San Fransisco, CA, 1997.

Ce livre présente la deuxième version du standard ODMG décrit dans ce chapitre. Il inclut tous les aspects étudiés et donne le détail des spécifications de ODL, OQL, OML C++, Smalltalk et Java.

[Won Kim95] Won Kim, *Modern Database Systems*, Addison-Wesley, New-York, 1995.

Cette collection d'articles sur les BD objets aborde tous les aspects d'un SGBD objet, en particulier les interfaces avec les langages objet (OQL-C++) ainsi que les promesses et déceptions des systèmes purs objet.

Chapitre XIII

L'OBJET-RELATIONNEL ET SQL3

1. INTRODUCTION

Le développement des SGBD objet s'est rapidement heurté à la nécessité de conserver la compatibilité avec l'existant. En effet, la fin des années 80 a vu le déploiement des SGBD relationnels et des applications client-serveur. Les utilisateurs n'étaient donc pas prêts à remettre en cause ces nouveaux systèmes dès le début des années 90.

Cependant, les tables relationnelles ne permettaient guère que la gestion de données alphanumériques. Avec l'avènement du Web au milieu de la décennie 90, la nécessité de supporter des données complexes au sein de la base de données s'est amplifiée. Ces données complexes peuvent être des documents textuels, des données géométriques ou géographiques, audiovisuelles ou soniques. En bref, il s'agit de supporter de manière intelligente des données multimédia (voir figure XIII.1).

Ce chapitre est consacré à l'étude du modèle objet-relationnel et du langage de requêtes associé SQL3. SQL3 est une extension de SQL2, développée par le groupe de normalisation ANSI X3 H2 et internationalisée au niveau de l'ISO par le groupe ISO/IEC JTC1/SC21/WG3. Il s'agit de la nouvelle version de SQL. SQL3 comporte beaucoup d'aspects nouveaux, l'essentiel étant sans doute son orientation vers une intégration douce de l'objet au relationnel. Ce chapitre traite en détail de cette intégration et survole les autres aspects de SQL3.

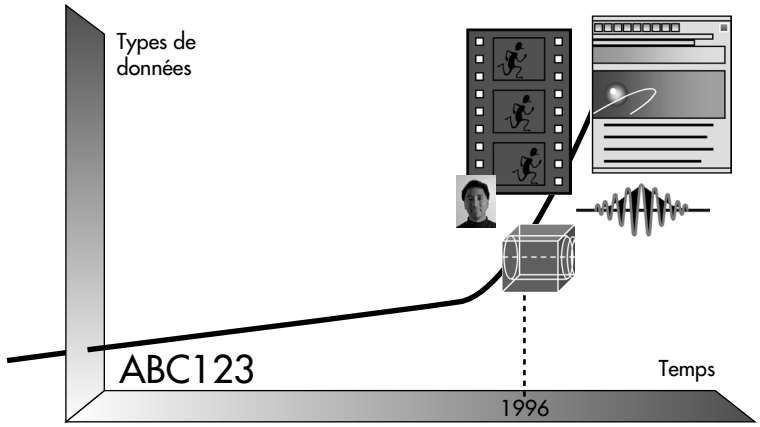


Figure XIII.1 : Le besoin de types de données multimédia

Après cette introduction, la section 2 motive l'intégration de l'objet au relationnel, en soulignant les forces et faiblesses du relationnel, qui justifient à la fois sa conservation et son extension. La section 3 définit les notions de base dérivées de l'objet et introduites pour étendre le relationnel. La section 4 propose une vue d'ensemble de SQL3 et du processus de normalisation. La section 5 détaille le support des objets en SQL3 avec de nombreux exemples. La section 6 résume les caractéristiques essentielles du langage de programmation de procédures et fonctions SQL/PSM, appoint essentiel à SQL pour assurer la complétude en tant que langage de programmation. Nous concluons ce chapitre en soulignant quelques points obscurs et l'intérêt d'une fusion entre OQL et SQL3.

2. POURQUOI INTÉGRER L'OBJET AU RELATIONNEL ?

Le modèle relationnel présente des points forts indiscutables, mais il a aussi des points faibles. L'objet répond à ces faiblesses. D'où l'intérêt d'une intégration douce.

2.1. FORCES DU MODÈLE RELATIONNEL

Le relationnel permet tout d'abord une modélisation des données adaptée à la gestion à l'aide de tables dont les colonnes prennent valeurs dans des domaines alphanumé-

riques. Le concept de table dont les lignes constituent les enregistrements successifs est simple, bien adapté pour représenter par exemple des employés, des services, ou des paiements. Avec SQL2, les domaines se sont diversifiés. Les dates, les monnaies, le temps et les intervalles de temps sont parfaitement supportés.

Dans les années 80, les SGBD relationnels se sont centrés sur le transactionnel (*On Line Transaction Processing* – OLTP) et sont devenus très performants dans ce contexte. Les années 90 ont vu le développement du décisionnel (*On Line Analysis Processing* – OLAP). Le relationnel s'est montré capable d'intégrer la présentation multidimensionnelle des données nécessaire à l'OLAP. En effet, il est simple de coder un cube 3D en table, trois colonnes mémorisant les valeurs d'une dimension et la quatrième la grandeur analysée.

En bref, grâce au langage assertionnel puissant que constitue le standard universel SQL2, à sa bonne adaptation aux architectures client-serveur de données, à sa théorie bien assise aussi bien pour la conception des bases (normalisation), l'optimisation de requêtes (algèbre, réécriture, modèle de coûts) et à la gestion de transactions (concurrence, fiabilité) intégrée, le relationnel s'est imposé dans l'industrie au cours des années 80.

2.2. FAIBLESSES DU MODÈLE RELATIONNEL

Le relationnel présente cependant des faiblesses qui justifient son extension vers l'objet. Tout d'abord, les règles de modélisation imposées pour structurer proprement les données s'avèrent trop restrictives.

L'**absence de pointeurs visibles** par l'utilisateur est très pénalisante. Dans les architectures modernes de machine où la mémoire à accès directe est importante, il devient intéressant de chaîner directement des données. Le parcours de références est rapide et permet d'éviter les jointures par valeurs du relationnel. Des enregistrements volumineux représentant de gros objets peuvent aussi être stockés une seule fois et pointés par plusieurs lignes de tables : le partage référentiel d'objets devient nécessaire. Par exemple, une image sera stockée une seule fois, mais pointée par deux tuples, un dans la table Employés, l'autre dans la table Clients. Passer par une clé d'identification (par exemple, le numéro de sécurité sociale) nécessite un accès par jointure, opération lourde et coûteuse. En clair, il est temps de réhabiliter les pointeurs sous la forme d'identifiants d'objets invariants au sein du relationnel.

Le **non-support de domaines composés** imposé par la première forme normale de Codd est inadapté aux objets complexes, par exemple aux documents structurés. L'introduction de champs binaires ou caractères longs (*Binary Large Object* – BLOB, *Character Large Object* – CLOB) et plus généralement de champs longs (*Large Object* – LOB) est insuffisante.

Notion XIII.1 : Objet long (*Large Object*)

Domaine de valeurs non structurées constitué par des chaînes de caractères (CLOB) ou de bits (BLOB) très longues (pouvant atteindre plusieurs giga-octets) permettant le stockage d'objets multimédia dans les colonnes de tables relationnelles.

Les objets longs présentent deux inconvénients majeurs : ils ne sont pas structurés et ne supportent pas les recherches associatives. La seule possibilité est de les lire séquentiellement. Ils sont donc particulièrement insuffisants pour le stockage intelligent de documents structurés, où l'on souhaite accéder par exemple à une section sans faire défiler tout le document. En clair, il faut pouvoir lever la contrainte d'atomicité des domaines, serait-ce que pour stocker des attributs composés (par exemple l'adresse composée d'une rue, d'un code postal et d'une ville) ou multivalués (par exemple les points d'une ligne).

La **non intégration des opérations** dans le modèle relationnel constitue un manque. Celui-ci résulte d'une approche traditionnelle où l'on sépare les traitements des données. Certes, les procédures stockées ont été introduites dans le relationnel pour les besoins des architectures client-serveur, mais celles-ci restent séparées des données. Elles ne sont pas intégrées dans le modèle. Il est par exemple impossible de spécifier des attributs cachés de l'utilisateur, accessibles seulement via des opérations manipulant ces attributs privés.

Tout ceci conduit à un mauvais support des applications non standard telles que la CAO, la CFAO, les BD géographiques et les BD techniques par les SGBD relationnels. En effet, ces applications gèrent des objets à structures complexes, composés d'éléments chaînés souvent représentés par des graphes. Le relationnel pur est finalement mal adapté. Il en va de même pour le support d'objets multimédia que l'on souhaite pouvoir rechercher par le contenu, par exemple des photos que l'on souhaite retrouver à partir d'un portrait robot.

2.3. L'APPORT DES MODÈLES OBJET

Les modèles objet sont issus des langages objets. Ils apportent des concepts nouveaux importants qui répondent aux manques du relationnel, comme l'identité d'objets, l'encapsulation, l'héritage et le polymorphisme.

L'**identité d'objet** permet l'introduction de pointeurs invariants pour chaîner les objets entre eux. Ces possibilité de chaînage rapide sont importantes pour les applications nécessitant le support de graphes d'objets. On pourra ainsi parcourir rapidement des suite d'associations, par navigation dans la base. Par exemple, passer d'un composé à ses composants puis à la description de ces composants nécessitera simplement deux parcours de pointeurs, et non plus des jointures longues et coûteuses.

L'**encapsulation des données** permet d'isoler certaines données dites privées par des opérations et de présenter des interfaces stables aux utilisateurs. Ceci facilite l'évolution des structures de données qui peuvent changer sans modifier les interfaces publiques seules visibles des utilisateurs. Au lieu d'offrir des données directement accessibles aux utilisateurs, le SGBD pourra offrir des services cachant ces données, beaucoup plus stables et complets. Ceux-ci pourront plus facilement rester invariants au fur et à mesure de l'évolution de la base de données.

L'**héritage d'opérations et de structures** facilite la réutilisation des types de données. Il permet plus généralement l'adaptation de composants à son application. Les composants pourront être des tables, mais aussi des types de données abstraits, c'est-à-dire un groupe de fonctions encapsulant des données cachées. Il sera d'ailleurs possible de définir des opérations abstraites, qui pourront, grâce au polymorphisme, porter le même nom mais s'appliquer sur des objets différents avec pour chaque type d'objet un code différent. Cela simplifie la vie du développeur d'applications qui n'a plus qu'à se soucier d'opérations abstraites sans regarder les détails d'implémentation sur chaque type d'objet.

2.4. LE SUPPORT D'OBJETS COMPLEXES

L'apport essentiel de l'objet-relationnel est sans doute le support d'objets complexes au sein du modèle. Ceci n'est pas inhérent à l'objet, mais plutôt hérité des premières extensions tendant à faire disparaître la première forme normale du relationnel. On parle alors de base de données en non première forme normale (NF^2).

Notion XIII.2 : Non première forme normale (*Non First Normal Form – NF^2*)

Forme normale tolérant des domaines multivalués.

Un cas particulier intéressant est le modèle relationnel imbriqué [Scholl86].

Notion XIII.3 : Modèle relationnel imbriqué (*Nested Relational Model*)

Modèle relationnel étendu par le fait qu'un domaine peut lui-même être valué par des tables de même schéma.

EXEMPLE

Des services employant des employés et enregistrant des dépenses pourront directement être représentés par une seule table externe imbriquant des tables internes :

```
SERVICES (N° INT, CHEF VARCHAR, ADRESSE VARCHAR, {EMPLOYÉS (NOM,
   AGE)}, {DÉPENSES (NDEP INT, MONTANT INT, MOTIF VARCHAR)})
```

`Employés` correspond à une table imbriquée (ensemble de tuples) de schéma (`Nom`, `Age`) pour chaque service, et de même, Dépenses correspond à une table imbriquée pour chaque service. Notons que le modèle relationnel permet une imbrication à plusieurs niveaux, par exemple, Motif pourrait correspondre à une table pour chaque dépense. ■

Services

N°	Chef	Adresse	Employés		Dépenses		
			Nom	Age	NDep	Montant	Motif
24	Paul	Versailles	Pierre	45	1	2 600	Livres
			Marie	37	2	8 700	Mission
					3	14 500	Portable
			Nom	Age	NDep	Montant	Motif
25	Patrick	Paris	Eric	42	5	3 000	Livres
			Julie	51	7	4 000	Mission

Figure XIII.2 : Exemples de relations imbriquées

Le multimédia, particulièrement la géométrie, a nécessité d'introduire des attributs multivalués plus généraux. Comme l'objet, l'objet-relationnel offre des collections génériques prédéfinies telles que des listes, ensembles, tableaux qui peuvent être imbriquées pour représenter des objets très compliqués. Par exemple, on pourra définir une ligne comme une liste de points, chaque point étant défini par une abscisse et une ordonnée. Une région pourra être définie par une liste de lignes fermée. Ces possibilités sont essentielles pour la représentation de structures complexes. Celles-ci peuvent être cachées par des fonctions offertes aux utilisateurs afin d'en simplifier la vision. Ainsi, objets complexes et encapsulation seront souvent couplés.

3. LE MODÈLE OBJET-RELATIONNEL

Le modèle objet-relationnel est fondé sur l'idée d'extension du modèle relationnel avec les concepts essentiels de l'objet (voir figure XIII.3). Le cœur du modèle reste donc conforme au relationnel, mais l'on ajoute les concepts clés de l'objet sous une forme particulière pour faciliter l'intégration des deux modèles.

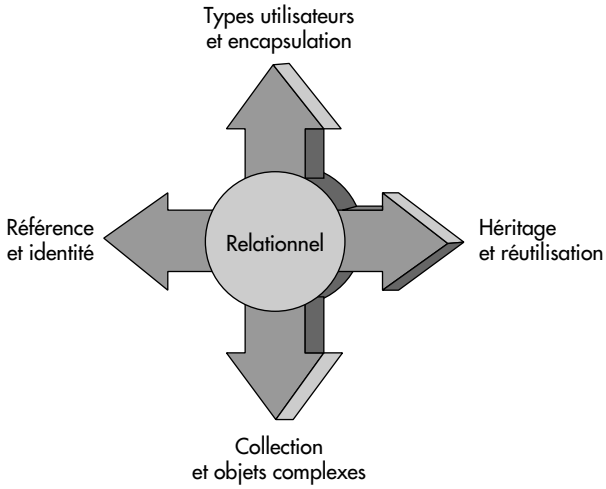


Figure XIII.3 : Les extensions apportées au relationnel

3.1. LES CONCEPTS ADDITIONNELS ESSENTIELS

Les types utilisateur permettent la définition de types extensibles utilisables comme des domaines, supportant l'encapsulation.

Notion XIII.4 : Type de données utilisateur (*User data type*)

Type de données définissable par l'utilisateur composé d'une structure de données et d'opérations encapsulant cette structure.

Le système de type du SGBD devient donc extensible, et n'est plus limité aux types alphanumériques de base, comme avec le relationnel pur et SQL2. On pourra par exemple définir des types texte, image, point, ligne, etc. Les types de données définissables par l'utilisateur sont appelés **types abstraits (ADT)** en SQL3. La notion de type abstrait fait référence au fait que l'implémentation est spécifique de l'environnement : seule l'interface d'un type utilisateur est visible.

Notion XIII.5 : Patron de collection (*Collection template*)

Type de données générique permettant de supporter des attributs multivalués et de les organiser selon un type de collection.

Les SGBD objet-relationnels offrent différents types de collections, tels que le tableau dynamique, la liste, l'ensemble, la table, etc. Le patron `LISTE(X)` pourra par

exemple être instancié pour définir des lignes sous forme de listes de points : `LIGNE LISTE(POINT)`.

Notion XIII.6 : Référence d'objet (*Object Reference*)

Type de données particulier permettant de mémoriser l'adresse invariante d'un objet ou d'un tuple.

Les références sont les identifiants d'objets (OID) dans le modèle objet-relationnel. Elles sont en théorie des adresses logiques invariantes qui permettent de chaîner directement les objets entre eux, sans passer par des valeurs nécessitant des jointures.

Notion XIII.7 : Héritage de type (*Type inheritance*)

Forme d'héritage impliquant la possibilité de définir un sous-type d'un type existant, celui-ci héritant alors de la structure et des opérations du type de base.

L'héritage de type permet donc de définir un sous-type d'un type SQL ou d'un type utilisateur. Une table correspondant à un type sans opération, l'objet-relationnel permet aussi l'**héritage de table**.

Notion XIII.8 : Héritage de table (*Table inheritance*)

Forme d'héritage impliquant la possibilité de définir une sous-table d'une table existante, celle-ci héritant alors de la structure et des éventuelles opérations de la table de base.

En résumé, le modèle objet-relationnel conserve donc les notions de base du relationnel (domaine, table, attribut, clé et contrainte référentielle) auxquelles il ajoute les concepts de type utilisateur avec structure éventuellement privée et opérations encapsulant, de collections supportant des attributs multivalués (structure, liste, tableau, ensemble, etc.), d'héritage sur relations et types, et d'identifiants d'objets permettant les références inter-tables. Les domaines peuvent dorénavant être des types utilisateurs. La figure XIII.4 illustre tous ces concepts. Deux vues sont possibles : la vue relationnelle dans laquelle les relations sont des relations entre objets (voir figure XIII.5), chaque colonne étant en fait valuée par un objet ; la vue objet (voir figure XIII.6) où une ligne d'une table peut correspondre à un objet simple (un tuple) ou complexe (un tuple avec des attributs complexes).

3.2. LES EXTENSIONS DU LANGAGE DE REQUÊTES

Au-delà du modèle, il est nécessaire d'étendre le langage de manipulation de données pour supporter les nouvelles avancées du langage de définition. SQL3 va donc comporter un langage de requêtes étendu avec notamment les appels d'opérations en résultat et en qualification, la surcharge des opérateurs de base de SQL (par exemple

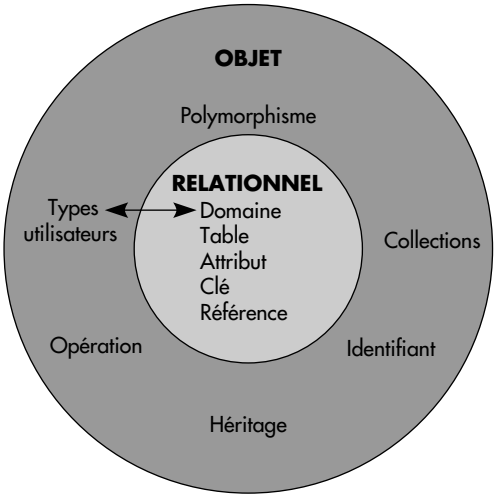


Figure XIII.4 : Les concepts de l'objet-relationnel

Accident	Rapport	Photo
134		
219		
037		

Figure XIII.5 : Une relation entre objets

Police	**Nom**	**Adresse**	**Conducteurs**		**Accidents**		
			Conducteur	Âge	Accident	Rapport	Photo
24	Paul	Paris	Paul	45	134		
			Robert	17	219		
					037		

Objet Police

Figure XIII.6 : Un objet complexe dans une table

l'addition), le support automatique de l'héritage, le parcours de références, la constitution de tables imbriquées en résultat, etc. Nous étudierons ces différentes fonctionnalités à l'aide d'exemples.

4. VUE D'ENSEMBLE DE SQL3

SQL3 est une spécification volumineuse, qui va bien au-delà de l'objet-relationnel. Nous examinons dans cette section ses principaux composants.

4.1. LE PROCESSUS DE NORMALISATION

Au-delà du groupe ANSI nord-américain très actif, la normalisation est conduite par un groupe international dénommé ISO/IEC JTC1/SC 21/WG3 DBL, DBL signifiant *Database Language*. Les pays actifs sont essentiellement l'Australie, le Brésil, le Canada, la France, l'Allemagne, le Japon, la Corée, la Hollande, le Royaume-Uni et bien sûr les États-Unis. L'essentiel du travail est bien préparé au niveau du groupe ANSI X3H2 (http://www.ansi.org) dont Jim Melton est le président. Ce groupe a déjà réalisé le standard SQL2 présenté dans la partie relationnelle. Les standards acceptés étaient contrôlés aux États-Unis par le NIST (http://ncsl.nist.gov) qui définissait des programmes de tests de conformité. Ceux-ci existent partiellement pour SQL2 mais pas pour SQL3. Depuis 1996, le congrès américain a supprimé les crédits du NIST si bien qu'il n'y a plus de programmes de validation espérés …

4.2. LES COMPOSANTS ET LE PLANNING

Alors que SQL2 était un unique document [SQL92], SQL3 a été divisé en composants pour en faciliter la lisibilité et la manipulation, l'ensemble faisant plus de 1 500 pages. Les composants considérés sont brièvement décrits dans le tableau représenté figure XIII.7.

Le planning de SQL3 a été décalé a plusieurs reprises, ces décalages témoignant de la difficulté et de l'ampleur de la tâche. Les parties 1 à 7 de SQL sont à l'état de brouillons internationaux (*Draft International Standard*) depuis fin 1997. Ils devraient devenir de véritables standards fin 1998 ou en 1999. Les deux derniers sont prévus pour l'an 2000.

Au-delà de SQL3, d'autres composants sont prévus tels SQL/MM pour la spécification de types utilisateurs multimédia et SQL/RDA pour la spécification du protocole RDA de transfert de données entre client et serveur. On parle déjà de SQL4, sans doute pour après l'an 2000.

Partie	Contenu	Titre	Description
1	le cadre	SQL/Framework	Une description non-technique de la façon dont le document est structuré et une définition des concepts communs à toutes les parties.
2	les fondements	SQL/Foundation	Le noyau de base, incluant les types de données utilisateurs et le modèle objet-relationnel.
3	l'interface client	SQL/CLI	L'interface d'appel client permet le dialogue client-serveur pour l'accès aux données via SQL2 puis SQL3.
4	les procédures stockées	SQL/PSM	Le langage de spécification de procédures stockées.
5	l'intégration aux langages classiques	SQL/Bindings	Les liens SQL dynamique et « embedded » SQL repris de SQL-92 et étendus à SQL3.
6	la gestion de transactions	SQL/Transaction	Une spécification de l'interface XA pour moniteur transactionnel distribué.
7	la gestion du temps	SQL/Temporal	Le support du temps, des intervalles temporels et des séries temporelles.
8	abandonné		
9	l'accès aux données externes	SQL/MED	L'utilisation de SQL pour accéder à des données non-SQL.
10	l'intégration aux langages objet	SQL/OBJ	L'utilisation de SQL depuis un langage objet, tel C++, Smalltalk ou Java.

Figure XIII.7 : Les composants de SQL3

4.3. LA BASE

Le composant 2 contient l'essentiel du langage SQL3, en particulier les fonctionnalités de base pour définir les autres composants (*Basic SQL/CLI capabilities*, *Basic SQL/PSM capabilities*), les déclencheurs (*Triggers*), les types utilisateurs appelés

types abstraits (*Abstract Data Types*) et les fonctionnalités orientées objets que nous allons étudier en détail ci-dessous. Un mot sur les déclencheurs : bien que partie intégrante du relationnel étudiée en tant que telle au chapitre VIII, les déclencheurs ne sont normalisés que dans SQL3. Il s'agit là de combler une lacune de la normalisation.

5. LE SUPPORT DES OBJETS

Dans cette section, nous allons présenter l'essentiel des commandes composant les fonctionnalités objet de SQL3 : définition de types abstraits, support d'objets complexes, héritage de type et de table. Pour chaque commande, nous donnons la syntaxe simplifiée et quelques exemples.

5.1. LES TYPES ABSTRAITS

5.1.1. Principes

La première nouveauté de SQL3 est l'apparition d'une commande CREATE TYPE. Au-delà des types classiques (numériques, caractères, date) de SQL, il devient possible de créer des types dépendant de l'application, multimédia par exemple (voir figure XIII.8).

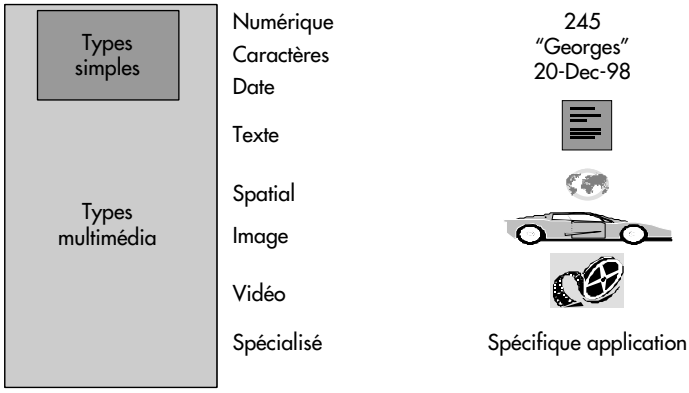


Figure XIII.8 : Exemples de types de données extensibles

Une instance d'un type peut être un objet ou une valeur. Un objet possède alors un OID et peut être référencé. Une valeur peut simplement être copiée dans une ligne d'une table. Un type possède en général des colonnes, soit directement visibles, soit

cachées et accessibles seulement par des fonctions encapsulant le type. Les opérateurs arithmétiques de SQL (+, *, –, /) peuvent être redéfinis au niveau d'un type. Par exemple, on redéfinira l'addition pour les nombres complexes. Enfin, un type peut posséder des clauses de comparaison (=, <) permettant d'ordonner les valeurs et peut être représenté sous la forme d'un autre type, une chaîne de caractères par exemple.

5.1.2. Syntaxe

La syntaxe de la commande de déclaration de type est la suivante :

```
CREATE [DISTINCT] TYPE <NOM ADT> [<OPTION OID>] [<CLAUSE SOUS-TYPE>]
[AS] (<CORPS DE L'ADT>)
```

Le mot clé DISTINCT est utilisé pour renommer un type de base existant déjà, par exemple dans la commande :

```
CREATE DISTINCT TYPE EURO AS (DECIMAL(9,2)
```

La clause :

```
<OPTION OID> ::= WITH OID VISIBLE
```

permet de préciser la visibilité de l'OID pour chaque instance (objet). Par défaut, une instance de type est une valeur sans OID. Cette clause semble plus ou moins avoir disparue de la dernière version de SQL3, le choix étant reporté sur les implémentations.

La clause :

```
<CLAUSE SOUS-TYPE> ::= UNDER <SUPERTYPE>[,<SUPERTYPE>]…
```

permet de spécifier les super-types dont le type hérite. Il y a possibilité d'héritage multiple avec résolution explicite des conflits.

La clause **<CORPS DE L'ADT>** se compose d'une ou plusieurs clauses membres :

```
<CLAUSES MEMBRES> ::=
                  < DÉFINITION DE COLONNE >
              |   < CLAUSE EGALE >
              |   < CLAUSE INFERIEURE>
              |   < CLAUSE CAST >
              |   < DÉCLARATION DE FONCTION >
              |   < DÉCLARATION D'OPÉRATEUR >
```

La définition de colonne permet de spécifier les attributs du type sous la forme <NOM> <TYPE>. Un type peut bien sûr être un type de base ou un type précédemment créé par une commande CREATE TYPE. Un attribut peut être privé ou public, comme en C++. La clause égale permet de définir l'égalité de deux instances du type, alors que la clause inférieure définit le comparateur <, important pour comparer et ordonner les valeurs du type. La clause CAST permet de convertir le type dans un autre type. Plusieurs clauses CAST sont possibles.

Une déclaration de fonction permet de définir les fonctions publiques qui encapsulent le type. Il existe différents types de fonctions : les **constructeurs** permettent de construire des instances du type ; ils portent en général le nom du type ; les **acteurs** agissent sur une instance en manipulant les colonnes du type ; parmi les acteurs, il est possible de distinguer les **observateurs** qui ne font que lire l'état et les **mutateurs** qui le modifient ; les **destructeurs** détruisent les instances et doivent être appelés explicitement pour récupérer la place disque (pas de ramasse-miettes). Plus précisément, la syntaxe d'une définition de fonction est la suivante :

```
<DÉCLARATION DE FONCTION> ::=
        [<FUNCTION TYPE>] FUNCTION <FUNCTION NAME> <PARAMETER LIST>
        RETURNS <FUNCTION RESULTS>
        { <SQL PROCEDURE> | <FILE NAME> } END FUNCTION
```

Le type est défini par :

```
<FUNCTION TYPE> ::= CONSTRUCTOR | ACTOR | DESTRUCTOR
```

Les fonctions SQL3 ne sont pas forcément associées à un type ; elles peuvent aussi être associées à une base ou à une table. Elles peuvent être programmées en SQL, en SQL/PSM (voir ci-dessous) ou dans un langage externe comme COBOL, C, C++ ou Java. Comme en C++, un opérateur est une fonction portant un nom particulier et déclarée comme opérateur.

5.1.3. Quelques exemples

Voici quelques exemples de création de types :

1. Type avec OID pouvant être utilisé comme un objet :

```
CREATE TYPE PHONE WITH OID VISIBLE (PAYS VARCHAR, ZONE VARCHAR,
NOMBRE INT, DESCRIPTION CHAR(20))
```

2. Type sans référence :

```
CREATE TYPE PERSONNE (NSS INT, NOM VARCHAR, TEL PHONE)
```

3. Sous-type :

```
CREATE TYPE ÉTUDIANT UNDER PERSONNE (CYCLE VARCHAR, ANNÉE INT)
```

4. Type énuméré :

```
CREATE TYPE JOUR-OUVRÉ (LUN, MAR, MER, JEU, VEN);
```

5. Type avec OID et fonction :

```
CREATE TYPE EMPLOYÉ WITH OID VISIBLE
(NOM CHAR(10), DATENAIS DATE, REPOS JOUR-OUVRÉ, SALAIRE FLOAT,
FUNCTION AGE (E EMPLOYÉ)RETURNS (INT)
{ CALCUL DE L'AGE DE E }
END FUNCTION;);
```

6. Autre type sans OID avec fonction :

```
CREATE TYPE ADRESSE WITHOUT OID
```

```
(RUE CHAR(20), CODEPOST INT, VILLE CHAR(10),
FUNCTION DEPT(A ADRESSE) RETURNS (VARCHAR) END FUNCTION);
```

7. Création de procédure SQL de mise à jour associée au type employé :

```
CREATE PROCEDURE AUGMENTER (E EMPLOYÉ, MONTANT MONEY)
{ UPDATE EMPLOYÉ SET SALAIRE = SALAIRE + MONTANT WHERE EMPLOYÉ.OID = E}
END PROCEDURE
```

5.2. LES CONSTRUCTEURS D'OBJETS COMPLEXES

SQL3 supporte la construction d'objets complexes par le biais de types `collection` paramétrés. Ces types génériques, appelés patrons de collections, doivent être fournis comme une bibliothèque avec le compilateur SQL3. Celui-ci doit assurer la généricité et en particulier la possibilité de déclarer des collections de types d'objets ou de valeurs quelconques. Chaque patron de collection fournit des interfaces spécifiques pour accéder aux éléments de la collection.

Tout environnement SQL3 doit offrir les patrons de base `SET(T)`, `MULTISET(T)` et `LIST(T)`. Un *multiset*, encore appelé *bag*, se comporte comme un ensemble mais permet de gérer des doubles. Par exemple, il est possible de définir un type personne avec une liste de prénoms et un ensemble de téléphones comme suit :

```
CREATE TYPE PERSONNE WITH OID VISIBLE
(NSS INT, NOM VARCHAR, PRÉNOMS LIST(VARCHAR), TEL SET(PHONE)).
```

Il est aussi possible de créer un ensemble de personnes :

```
CREATE TYPE PERSONNES (CONTENU SET (PERSONNE))
```

ou un ensemble d'identifiants de personnes :

```
CREATE TYPE RPERSONNES (CONTENU SET(REF(PERSONNE))).
```

Au-delà des types de collections `SET`, `LIST` et `MULTISET`, SQL3 propose des types additionnels multiples : piles (*stack*), files (*queue*), tableaux dynamiques (*varray*), tableaux inserables (*iarray*) pour gérer des textes par exemple. Un tableau dynamique peut grandir par la fin, alors qu'un tableau insérable peut aussi grandir par insertion à partir d'une position intermédiaire (il peut donc grandir par le milieu). Ces types de collections sont seulement optionnels : ils ne sont pas intégrés dans le langage de base mais peuvent être ajoutés.

De manière générale, il est possible de référencer des objets via des OID mémorisés comme valeurs d'attributs. De tels attributs sont déclarés références (`REF`) comme dans l'exemple ci-dessous :

```
CREATE TYPE VOITURE (NUMÉRO CHAR(9), COULEUR VARCHAR,
PROPRIÉTAIRE REF(PERSONNE))
```

5.3. LES TABLES

Les tables SQL restent inchangées, à ceci près que le domaine d'un attribut peut maintenant être un type créé par la commande CREATE TYPE. De plus, des fonctions peuvent être définies au niveau d'une table pour encapsuler les tuples vus comme des objets. Aussi, l'héritage de table qui permet de réutiliser une définition de table est possible.

Par exemple, la table représentée figure XIII.5 sera simplement créée par la commande :

```
CREATE TABLE ACCIDENTS (ACCIDENT INT, RAPPORT TEXT, PHOTO IMAGE)
```

en supposant définis les types TEXT (texte libre) et IMAGE, qui sont généralement fournis avec un SGBD objet-relationnel, comme nous le verrons plus loin.

La table de la figure XIII.6 pourra être définie à partir des mêmes types avec en plus :

```
CREATE TYPE CONDUCTEUR (CONDUCTEUR VARCHAR, AGE INT)
CREATE TYPE ACCIDENT (ACCIDENT INT, RAPPORT TEXT, PHOTO IMAGE)
```

par la commande :

```
CREATE TABLE POLICE (NPOLICE INT, NOM VARCHAR, ADRESSE ADRESSE,
   CONDUCTEURS SET(CONDUCTEUR), ACCIDENTS LIST(ACCIDENT)).
```

SQL3 donne aussi des facilités pour passer d'un type à une table de tuples de ce type et réciproquement, d'une table au type correspondant.

Par exemple, en utilisant le type EMPLOYÉ défini ci-dessus, il est possible de définir simplement une table d'employés par la commande :

```
CREATE TABLE EMPLOYÉS OF EMPLOYÉ ;
```

En définissant une table VINS, il sera possible de définir le type VIN composé des mêmes attributs comme suit :

```
CREATE TABLE VINS(NV INT, CRU VARCHAR, DEGRÉ FLOAT(5.2))
OF NEW TYPE VIN ;
```

Comme mentionné, l'héritage de table qui recopie la structure est possible par la clause :

```
CREATE TABLE <TABLE> UNDER <TABLE> [WITH (LISTE D'ATTRIBUTS TYPÉS)].
```

La liste des attributs typés permet d'ajouter les attributs spécifiques à la sous-table. Par exemple, une table de vins millésimé pourra être crée par :

```
CREATE TABLE VINSMILL UNDER VINS WITH (MILLÉSIME INT, QUALITÉ VARCHAR).
```

5.4. L'APPEL DE FONCTIONS ET D'OPÉRATEURS DANS LES REQUÊTES

Le langage de requêtes est étendu afin de supporter les constructions nouvelles du modèle. L'extension essentielle consiste à permettre l'appel de fonctions dans les

termes SQL, qui peuvent figurer dans les expressions de valeurs sélectionnées ou dans les prédicats de recherche. Une fonction s'applique à l'aide de la notation parenthésée sur des termes du type des arguments spécifiés.

Considérons par exemple la table d'employés contenant des instances du type EMPLOYÉ définit ci-dessus. La requête suivante retrouve le nom et l'âge des employés de moins de 35 ans :

```
SELECT E.NOM, AGE(E)
FROM EMPLOYÉS E
WHERE AGE(E) < 35;
```

La notation pointée appliquée au premier argument est aussi utilisable pour invoquer les fonctions :

```
SELECT E.NOM, E..AGE()
FROM EMPLOYÉS E
WHERE E..AGE() < 35;
```

Il s'agit d'un artifice syntaxique, la dernière version utilisant d'ailleurs la double notation pointée (..) pour les fonctions et attributs composés, la notation pointée simple (.) étant réservée au SQL de base (accès à une colonne usuelle de tuple).

Au-delà des fonctions, SQL3 permet aussi l'accès aux attributs composés par la notation pointée. Supposons par exemple une table d'employés localisés définies comme suit :

```
CREATE TABLE EMPLOYÉSLOC UNDER EMPLOYÉS WITH (ADRESS ADRESSE).
```

La requête suivante permet de retrouver le nom et le jour de repos des employés des Bouches-du-Rhône habitant Marseille :

```
SELECT NOM, REPOS
FROM EMPLOYÉSLOC E
WHERE DEPT(E.ADRESS) = "BOUCHES DU RHONE"
AND E.ADRESS..VILLE = "MARSEILLE";
```

Notons dans la requête ci-dessus l'usage de fonctions pour extraire le département à partir du code postal et l'usage de la notation pointée pour extraire un champ composé. Nous préférerons utiliser partout la notation pointée ; il faut cependant distinguer fonction et accès à attribut par la présence de parenthèses.

Afin d'illustrer plus complètement, supposons définis un type `point` et une fonction `distance` entre deux points comme suit :

```
TYPE POINT (ABSCISSE X, ORDONNÉE Y,
FUNCTION DISTANCE(P1 POINT, P2 POINT) RETURN (REAL)) ;
```

Considérons une implémentation des `employés` par la table :

```
EMPLOYÉS (MUNEMP INT, NOM VARCHAR, ADRESS ADRESSE, POSITION POINT) ;
```

La requête suivante recherche alors les noms et adresses de tous les employés à moins de 100 unités de distance (par exemple le mètre) de l'employé Georges :

```
SELECT E.NOM, E.ADRESS
FROM EMPLOYÉS G, EMPLOYÉS E
WHERE G.NAME = "GEORGES" AND DISTANCE(G.POSITION,E.POSITION) < 100.
```

Supposons de plus défini le type CERCLE comme suit :

```
TYPE CERCLE (CENTRE POINT, RAYON REAL,
CONSTRUCTOR FUNCTION CERCLE(C POINT, R REAL) RETURN (CERCLE))
```

Ajoutons une fonction booléenne CONTIENT au type point :

```
CREATE FUNCTION CONTIENT (P POINT, C CERCLE)
{ CODE DE LA FONCTION } RETURN (BOLEAN)
END FUNCTION.
```

La question suivante retrouve les employés dont la position est contenu dans un cercle de rayon 100 autour de l'employé Georges :

```
SELECT E.NAME, E.ADRESS
FROM EMPLOYÉS G, EMPLOYÉS E
WHERE EMPLOYÉS.NOM = "GEORGES" AND
CONTIENT(E.POSITION, CERCLE(G.POSITION,1));
```

Les deux requêtes précédentes sont de fait équivalents et génèrent les mêmes réponses.

5.5. LE PARCOURS DE RÉFÉRENCE

SQL3 permet aussi de traverser les associations représentées par des attributs de type références.

Ces attributs peuvent être considérés comme du type particulier référence, sur lequel il est possible d'appliquer les fonctions Ref pour obtenir la valeur de l'OID et DeRef pour obtenir l'objet pointé. Afin de simplifier l'écriture, DeRef peut être remplacée par la notation →. Celle-ci permet donc les parcours de chemins. On peut ainsi écrire des requêtes mélangeant la notation simple point (accès à un attribut), double point (accès à un attribut ADT composé) et flèche (parcours de chemins). Espérons que la notation pointé simple pourra être utilisée partout comme une amélioration syntaxique !

Considérons le type VOITURE déjà défini ci-dessus comme suit :

```
CREATE TYPE VOITURE (NUMÉRO CHAR(9), COULEUR VARCHAR,
PROPRIÉTAIRE REF(PERSONNE)).
```

Créons une table de voitures :

```
CREATE TABLE VOITURES OF TYPE VOITURE.
```

La requête suivante recherche les noms des propriétaires de voitures rouges habitant Paris :

```
SELECT V.PROPRIETAIRE→NOM
FROM VOITURES V
WHERE V.COULEUR = "ROUGE" AND V.PROPRIETAIRE→ADRESSE..VILLE = "PARIS".
```

SQL3 permet de multiples notations abrégées. En particulier, il est possible d'éviter de répéter des préfixes de chemins avec la notation pointée suivie de parenthèses. Par exemple, la requête suivante recherche les numéros des voitures dont le propriétaire habite les Champs-Elysés à Paris et a pour prénom Georges :

```
SELECT V.NUMÉRO
FROM VOITURES V
WHERE V.PROPRIÉTAIRE→(ADRESSE..(VILLE = "PARIS"
AND RUE = "CHAMPS ELYSÉS") AND PRÉNOM = »GEORGES").
```

5.6. LA RECHERCHE EN COLLECTIONS

Les collections de base sont donc les ensembles, listes et sacs. Ces constructeurs de collections peuvent être appliqués sur tout type déjà défini. Les collections sont rendues permanentes en qualité d'attributs de tables. La construction `TABLE` est proposée pour transformer une collection en table et l'utiliser derrière un `FROM` comme une véritable table. Toute collection peut être utilisée à la place d'une table précédée de ce mot clé `TABLE`. Par exemple, supposons l'ajout d'une colonne `PASSETEMPS` à la table des personnes valuée par un ensemble de chaînes de caractères :

```
ALTER TABLE PERSONNES ADD COLUMN PASSETEMPS SET(VARCHAR).
```

La requête suivante retrouve les références des personnes ayant pour passe-temps le vélo :

```
SELECT REF(P)
FROM PERSONNES P
WHERE "VÉLO" IN
        SELECT *
        FROM TABLE (P.PASSETEMPS).
```

Plus complexe, la requête suivante recherche les numéros des polices d'assurance dont un accident contient un rapport avec le mot clé « Pont de l'Alma » :

```
SELECT P.NPOLICE
FROM POLICE P, TABLE P.ACCIDENTS A, TABLE A.RAPPORT..KEYWORDS M
WHERE M = "PONT DE L'ALMA".
```

Cette requête suppose la disponibilité de la fonction `KEYWORDS` sur le type `TEXT` du rapport qui délivre une liste de mots clés. Nous tablons tout d'abord la liste des accidents, puis la liste des mots clés du rapport de chaque accident. Ceci donne donc une sorte d'expression de chemins multivalués. SQL3 est donc très puissant !

5.7. RECHERCHE ET HÉRITAGE

L'héritage de tables est pris en compte au niveau des requêtes. Ainsi, lorsqu'une table possède des sous-tables, la recherche dans la table retrouve toutes les lignes qui qualifient au critère de recherche, aussi bien dans la table que dans les sous-tables

Par exemple, considérons la table BUVEURS définie comme suit :

```
CREATE TABLE BUVEURS UNDER PERSONNES WITH ETAT ENUM(NORMAL,IVRE).
```

La recherche des personnes de prénom Georges par la requête :

```
SELECT NOM, PRÉNOM, ADRESSE
FROM PERSONNES
WHERE PRÉNOM = "GEORGES"
```

retournera à la fois les personnes et les buveurs de prénom Georges.

6. LE LANGAGE DE PROGRAMMATION PSM

Le composant PSM définit un L4G adapté à SQL. Il a été adopté tout d'abord dans le cadre SQL2, puis étendu à SQL3.

6.1. MODULES, FONCTIONS ET PROCÉDURES

SQL/PSM (*Persistent Store Modules*) est un langage de programmation de modules persistants et stockés dans la base de données [Melton98]. Les modules sont créés et détruits par des instructions spéciales `CREATE MODULE` et `DROP MODULE`. Un module se compose de procédures et/ou de fonctions. Il est possible de créer directement des procédures (`CREATE PROCEDURE`) ou des fonctions (`CREATE FUNCTION`) non contenues dans un module. Un module est associé à un schéma de base de données sur lequel peuvent être définies des autorisations et des tables temporaires. Il est aussi possible de définir des curseurs partagés entre les procédures et fonctions composantes.

SQL/PSM parachève SQL pour en faire un langage complet. Cependant, il est possible d'écrire des procédures stockées en pur SQL ou dans un langage pour lequel l'intégration avec SQL est définie (ADA, C, COBOL, FORTRAN, PASCAL, PL/I et MUMPS). Pour ces langages, PSM permet de définir les paramètres d'entrée et de retour des procédures (mots clés IN pour les paramètres d'entrée et OUT pour ceux de

sortie) ou des fonctions (clause RETURN <type>), Ces procédures sont en général invoquées depuis des programmes hôtes par des ordres EXEC SQL plus ou moins spécifiques du langage hôte, ou directement depuis d'autres procédures PSM par des ordres CALL <procédure> ou des appels directs de fonctions.

6.2. LES ORDRES ESSENTIELS

L'intérêt essentiel de SQL/PSM est de fournir un langage de programmation homogène avec SQL et manipulant les mêmes types de données. Ce langage comporte des déclarations de variables, des assignations, des conditionnels CASE, IF, des boucles LOOP, REPEAT, WHILE, FOR et des traitements d'erreurs et exceptions SIGNAL, RESIGNAL.

Une déclaration de variable est analogue à une déclaration de colonne en SQL :

```
<DECLARATION DE VARIABLE> ::= DECLARE <VARIABLE> <TYPE> [DEFAULT
   <VALEUR>]
```

Par exemple, une déclaration possible est :

```
DECLARE PRIX INT DEFAULT 0 ;
```

Une assignation s'effectue par une clause SET suivie d'une expression de valeur conforme à SQL, ou par l'assignation du résultat d'une requête dans une variable. Par exemple :

```
DECLARE MOYENNE DECIMAL(7.2)
SELECT AVG(SALAIRE) INTO MOYENNE FROM EMPLOYÉS
```

assigne le résultat de la requête à la variable MOYENNE.

L'ordre CASE permet de tester différents cas de valeurs d'une expression de valeur en paramètre du CASE (expression de valeur1) ou de plusieurs expressions de valeurs booléennes :

```
<ORDRE CASE> ::=
CASE [<EXPRESSION DE VALEUR1>]
WHEN <EXPRESSION DE VALEUR2 > THEN <ORDRE SQL> ;
[WHEN <EXPRESSION DE VALEUR> THEN <ORDRE SQL> ;]…
ELSE <ORDRE SQL> ;
END CASE
```

Par exemple :

```
CASE MOYENNE
WHEN <100 THEN CALL PROC1 ;
WHEN = 100 THEN CALL PROC2 ;
ELSE CALL PROC3 ;
END CASE
```

permet de traiter différents cas de la variable MOYENNE.

Les boucles LOOP, REPEAT et WHILE sont des boucles de calculs en mémoire classiques, qui ne font pas directement appel à la base. La boucle FOR permet au contraire de parcourir le résultat d'une requête par le biais d'un curseur. Sa syntaxe simplifiée est la suivante :

```
<ORDRE FOR> ::=
[<ÉTIQUETTE> :]FOR <NOM DE BOUCLE>
AS [<NOM DE CURSEUR> CURSEUR FOR] <SPECIFICATION DE CURSEUR>
DO
<ORDRE SQL>
END FOR <ÉTIQUETTE> ;
```

L'étiquette permet de référencer la boucle FOR. Le nom de boucle sert de qualifiant pour les colonnes de la table virtuelle correspondant au curseur, afin de les distinguer si nécessaire. Le nom de curseur est optionnel pour permettre la réutilisation du curseur. Une définition de curseur est classique en SQL : il s'agit en général d'une requête.

6.3. QUELQUES EXEMPLES

Voici la syntaxe des constructions essentielles du langage et un exemple de procédure et de fonction. Nous travaillons sur la base des vins composées des tables :

```
VINS (NV INT, CRU VARCHAR, MILL INT, DEGRÉ FLOAT(5.2))
COMMANDES (NCO INT, NV INT, DAT DATE, QUANTITÉ INT)
```

Une procédure pour calculer la variance des quantités commandées d'un cru donné en paramètre peut être définie comme suit :

```
CREATE PROCEDURE (IN C VARCHAR,
OUT VARIANCE DECIMAL(10.2))
BEGIN
DECLARE MOYENNE DECIMAL(10.2) ;
VARIANCE = 0 ;
SELECT AVG(QUANTITE) INTO MOYENNE
FROM COMMANDES C, VINS V
WHERE V.NV = C.NV AND V.CRU = C ;
BOUCLE1:
FOR M AS SELECT NCO, QUANTITÉ
FROM COMMANDES C, VINS V
WHERE V.NV = C.NV AND V.CRU = C
DO
SET VARIANCE = VARIANCE+(M.QUANTITE - MOYENNE)**2
END FOR BOUCLE1 ;
END
```

Tout SQL est bien sûr intégré à PSM. Il serait possible d'utiliser un curseur et une boucle WHILE avec un FETCH dans la boucle à la place de la boucle FOR.

Pour montrer la complétude du langage, nous définissons la fonction factorielle comme suit :

```
CREATE FUNCTION FACTORIELLE (N INT)
RETURNS INT
DETERMINISTIC
BEGIN
DECLARE FAC INT DEFAULT 1 ;
DECLARE I INT ;
SET I = N ;
WHILE I > 1 DO
FAC = FAC*I ;
I = I-1 ;
END WHILE ;
RETURN FAC ;
END
```

D'autres types de boucles peuvent bien sûr être utilisés pour calculer cette fonction [Melton98].

6.4. PLACE DE PSM DANS LE STANDARD SQL

SQL/PSM est un standard international dans sa version SQL2. Il a été intégré en 1996 au standard SQL2. Une version plus complète est en cours de spécification pour SQL3. Cette version présentera l'avantage de pouvoir utiliser les types abstraits et les collections de SQL3 dans le langage de programmation. SQL/PSM est assez proche des langages de quatrième génération de systèmes tels qu'ORACLE, INFORMIX ou SYBASE. Malheureusement, aucun système n'implémente actuellement exactement PSM.

7. CONCLUSION

SQL3 est un standard en évolution. Comme nous l'avons vu, les composants ne sont pas tous au même niveau d'avancement. La plupart sont au stade de brouillons internationaux, et vont donc subir encore des modifications. L'ensemble devrait être terminé vers l'an 2000.

SQL3 se situe, au moins pour la partie objet et les interfaces avec les langages objet, comme un concurrent de l'ODMG. Il est supporté par tous les grands constructeurs, comme IBM, Oracle et Sybase, et est impliqué dans un processus de standardisation international. Au contraire, l'ODMG est un accord entre quelques constructeurs de

SGBD objet autour d'un langage pur objet. L'ODMG traite bien les aspects collections et, dans une moindre mesure, traversée de chemins. Les deux propositions pourraient donc apparaître comme complémentaires, à condition de converger. Il existe d'ailleurs un accord entre les groupes ANSI X3 H2 responsable de SQL et ODMG pour explorer la convergence. Souhaitons que cet accord aboutisse.

D'autres, comme Stonebraker [Stonebraker96], pensent plutôt que les bases de données objet ont un créneau d'application différent de celui des bases de données objet-relationnel. Les premières seraient plutôt orientées vers les applications écrites en C++ ou Java, naviguant dans les bases mais ne formulant pas de questions complexes sur de gros volumes de données persistantes. Au contraire, les bases de données objet-relationnel profiteraient de l'orientation disque du relationnel et seraient capables de supporter des questions complexes sur des données complexes. La figure XIII.9 résume ce point de vue. Il est possible de mesurer la complexité des requêtes en nombre de jointures et agrégats, la complexité des données en nombre d'associations. Cependant, l'évolution des systèmes objet vers la compatibilité SQL, et donc vers l'objet-relationnel, poussée par le marché, ne plaide guère pour la validité de ce tableau.

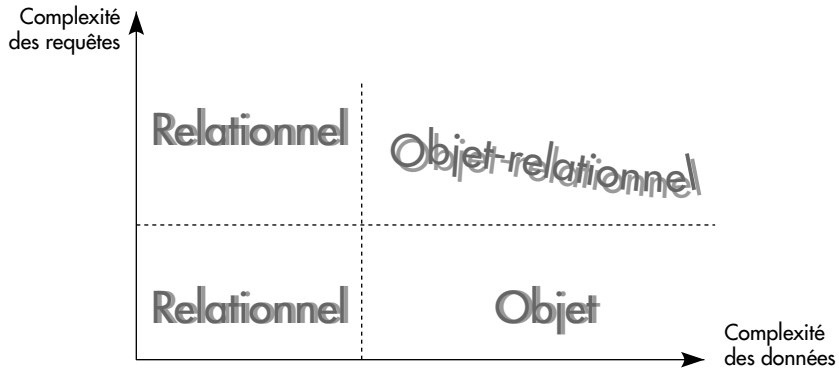


Figure XIII.9 : Relationnel, Objet ou Objet-Relationnel ?

Et après SQL3 ? On parle déjà de SQL4, notamment pour les composants orientés applications ou métiers. Il est en effet possible, voire souhaitable, de standardiser les types de données pour des créneaux d'applications, tels le texte libre, la géographie ou le multimédia. Nous étudierons plus loin ces types de données multimédia qui sont d'actualité. Au-delà, on peut aussi penser développer des types pour la CAO, la finance, l'assurance ou l'exploration pétrolière par exemple. On aboutira ainsi à des parties de schémas réutilisables, complément souhaitable d'objets métiers. Beaucoup de travail reste à faire. Reste à savoir si SQL est le cadre adéquat, et s'il n'est pas déjà trop complexe pour être étendu.

8. BIBLIOGRAPHIE

[Darwen95] Darwen H., Date, « Introducing the Third Manifesto », *Database Programming & Design Journal*, C-J.vol. 1, n° 8, p. 25-35, Jan. 1995.

Cet article plaide pour le modèle objet-relationnel vu comme une extension naturelle du relationnel, tel que proposé par Codd (et Date). Pour C. Date, l'apport essentiel utile du modèle objet est la possibilité de définir des types de données utilisateur. En clair, aucune modification n'est nécessaire au modèle relationnel qui avait déjà le concept de domaine : il suffit d'ouvrir les domaines et de les rendre extensibles.

[Gardarin89] Gardarin G., Cheiney J.P., Kiernan J., Pastre D., « Managing Complex Objects in an Extensible DBMS », *15th Very Large Data Bases International Conference*, Morgan Kaufman Pub., Amsterdam, Pays-Bas, août 1989.

Une présentation détaillée du support d'objets complexes dans le SGBD extensible Sabrina. Ce système est dérivé du SGBD relationnel SABRE et fut l'un des premiers SGBD à supportes des types abstraits comme domaines d'attributs. Il a ensuite évolué vers un SGBD géographique (GéoSabrina) qui fut commercialisé par la société INFOSYS.

[Gardarin92] Gardarin G., Valduriez P., « ESQL2: An Object-Oriented SQL with F-Logic Semantics », *8th Data Engineering International Conf.*, Phœnix, Arizona, Feb. 1992.

Cet article décrit le langage ESQL2, précurseur de SQL3 compatible avec SQL2, permettant d'interroger à la fois des bases de données à objets et relationnelles. Le langage supporte des relations référençant des objets complexes. Une notation fonctionnelle est utilisée pour les parcours de chemins et les applications de méthodes. Une version plus élaborée de ESQL2 avec classes et relations a été spécifiée dans le projet EDS. La sémantique basée sur la F-Logic (une logique pour objets) illustre les rapports entre les modèles objets et logiques.

[Godfrey98] Godfrey M., Mayr T., Seshadri P., von Eicken T., « Secure and Portable Database Extensibility », *Proc. of the 1998 ACM SIGMOD Intl. Conf. On Management of Data*, ACM Pub. SIGMOD Record vol. 27, n° 2, p. 390-401, Juin 1998.

Cet article décrit l'implémentation de types abstraits dans le SGBD PREDATOR en Java. Il montre l'intérêt de fonctions sûres et portables, mais souligne aussi les difficultés de performance par comparaison à une implémentation en C++.

[Haas90] Haas L., Chang W., Lohman G.M., McPherson J., Wilms P.F., Lapis G., Lindsay B., Pirahesh H., Carey M., Shekita E., « Starburst Mid-Flight : As the Dust Clears », *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, n° 1, Mars 1990.

Cet article décrit Starburst, un des premiers SGBD objet-relationnel. Starbust a été implémenté à IBM Almaden, et fut un des précurseurs de DB2 Universal Server. Il a conduit à de nombreux développements autour de l'objet-relationnel, notamment des techniques d'évaluation de requêtes.

[Melton98] Melton J., *Understanding SQL's Stored Procedures*, Morgan Kaufman Pub., 1998.

Ce livre présente la version 96 de PSM, intégrée à SQL2. Il est très complet, aborde les différents aspects (concepts de base, création de modules et procédures, routines externes, polymorphisme, contrôles, extensions prévues) et donne un exemple d'application.

[Scholl86] Scholl M., Abiteboul S., Bancilhon F., Bidoit N., Gamerman S., Plateau D., Richard P., Verroust A., « VERSO: A Database Machine Based On Nested Relations. Nested Relations and Complex Objects », *Intl. Workshop Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, Germany, April 1987, in Lecture Notes in Computer Science, vol. 361, p. 27-49, Springer Verlag 1989.

La machine bases de données Verso (Recto étant le calculateur frontal) fut développée à l'INRIA au début des années 1980. Basé sur le modèle relationnel imbriqué, ce projet a développé un prototype original intégrant un filtre matériel spécialisé et la théorie du modèle relationnel NF2.

[Seshadri97] Seshadri P., Livny M., Ramakrishnan R., « The Case for Enhanced Abstract Data Types »*, Proc. of 23rd Intl. Conf. On Very Large Databases*, Athens, Greece, p. 66-75, 1997.

Cet article plaide pour des types abstraits améliorés exposant la sémantique de leurs méthodes sous forme de règles. Il décrit l'implémentation et les techniques d'optimisation de requêtes dans le système PREDATOR. D'autres implémentation de types abstraits sont évoquées.

[Siberschatz91] Silberschatz A., Stonebraker M., Ullman J., « Next Generation Database Systems – Achievements and Opportunities », *Comm. of the ACM*, vol. 34, n° 10, p. 110-120, Oct. 1991.

Cet article fait le point sur l'apport des systèmes relationnels et souligne les points essentiels que la future génération de SGBD devra résoudre.

[Stonebraker86] Stonebraker M., Rowe L., Hirohama M., » The Implementation of PostGres », *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, n° 1, p. 125-142, Mars 1990.

Cet article décrit l'implémentation de PostGres, le système réalisé à Berkeley après Ingres. PostGres fut le premier système objet-relationnel. Il a été plus tard commercialisé sous le nom d'Illustra, qui fut racheté par Informix en 1996.

[Stonebraker96] Stonebraker M., Moore D., *Object-Relational DBMSs : The Next Great wave*, Morgan Kaufmann Pub., San Fransisco, CA, 1996.

Ce livre définit ce qu'est un SGBD objet-relationnel. Les fonctionnalités du SGBD Illustra sont mises en avant pour situer les objectifs essentiels des systèmes objet-relationnels, en particulier du point de vue du modèle de données, du langage de requêtes et de l'optimisation.

[Zaniolo83] Zaniolo C., « The Database Language GEM », *Proc. of 1983 ACM SIGMOD Intl. Conf. on Management of Data*, San José, Ca, p. 207-218, Mai 1983.

Cet article propose une extension du modèle relationnel et du langage de requête QUEL pour intégrer des tables fortement typées, des généralisations, des références, des expressions de chemins et des attributs multivalués. Il décrit un langage de requêtes précurseur de SQL3.

Chapitre XIV

OPTIMISATION DE REQUÊTES OBJET

1. INTRODUCTION

Une des fonctionnalités essentielles des systèmes objet ou objet-relationnel est la possibilité d'interroger la base à partir de requêtes non procédurales, exprimées en OQL ou en SQL3 (voir chapitre précédent). Ces requêtes peuvent invoquer des opérations sur des types de données utilisateur. En effet, dans les SGBD objet ou relationnel-objet, l'utilisateur, ou plutôt un implémenteur système, peut ajouter ses propres types de données, avec des comparateurs logiques et des méthodes d'accès spécifiques. L'optimiseur doit alors être extensible, c'est-à-dire capable de supporter une base de règles de connaissances pouvant être étendue par l'utilisateur afin d'explorer un espace de recherche plus riche pour les plans d'exécution.

Un premier problème est d'être capable de générer tous les plans d'exécution possibles pour une requête complexe. Ceci nécessite tout d'abord des techniques de réécriture de requêtes en requêtes équivalentes. La réécriture peut s'effectuer au niveau de la requête source ; plus souvent, elle s'applique sur des requêtes traduites en format interne, sous forme d'arbres algébriques. De nombreux travaux ont été effectués sur les techniques de réécriture de requêtes objet [Graefe93, Haas89, Cluet92, Mitchell93, Finance94, Florescu96].

Un deuxième problème est le choix des algorithmes et méthodes d'accès les plus performants pour réaliser une opération de l'algèbre d'objets. Ce choix est plus riche que dans le cas relationnel pour les jointures qui peuvent maintenant s'effectuer directement par traversées de pointeurs. Le support direct de pointeurs sous forme d'identifiants invariants est en effet une des fonctionnalités nouvelles des systèmes objet, permettant la navigation. Le choix des algorithmes d'accès va dépendre du modèle de stockage qui inclut de nouvelles techniques d'indexation de chemins et de groupage d'objets. D'autres problèmes difficiles sont la mise en place d'opérateurs spécifiques sur collections et des index sur fonctions. Nous examinerons plus particulièrement les nouveaux algorithmes de traversées de chemins.

Comme dans le cas relationnel, le choix du meilleur plan nécessite un modèle de coût permettant d'estimer le coût d'un plan à la compilation. Ce modèle doit prendre en compte les groupages d'objets, les parcours de pointeurs, les index de chemins, et aussi les méthodes utilisateur. Ceci pose des problèmes difficiles. Plusieurs modèles ont été proposés [Bertino92, Cluet92]. Nous en proposons un prenant en compte le groupage des objets.

Si l'on est capable de générer tous les plans candidats pour calculer la réponse à une requête et d'estimer le coût de chacun, il faut encore choisir le meilleur plan. Une stratégie exhaustive n'est pas possible compte tenu du grand nombre de plans en général. Des stratégies de recherche sophistiquées ont été proposées, depuis l'optimisation itérative [Ioannidis90, Lanzelotte91] jusqu'à la recherche génétique [Tang97]. Nous donnons dans ce chapitre une vue d'ensemble des méthodes aléatoire de recherche de plan optimal.

Ce chapitre présente une synthèse des techniques essentielles de l'optimisation des requêtes dans les bases de données objet, au-delà du relationnel. Il s'appuie en partie sur les travaux de recherche que nous avons menés au laboratoire PRiSM de Versailles de 1990 à 1996. Ces techniques commencent aujourd'hui à être intégrées dans les SGBD objet-relationnel et objet. Dans la section suivante, nous détaillons plus particulièrement la problématique d'optimisation spécifique à l'objet, en l'illustrant par quelques exemples. Dans la section 3, nous introduisons un modèle de stockage générique pour bases de données objet. La section 4 se consacre aux algorithmes de traversée de chemins. Nous donnons différents algorithmes de parcours d'associations et de collections imbriquées qui généralisent les algorithmes de jointures en cascade aux bases de données objet. La section 5 discute de la génération des plans équivalents et des types de règles de réécriture à considérer. Elle conduit à développer une architecture type pour un optimiseur extensible. La section 6 développe un modèle de coût pour SGBD objet. Nous examinons ensuite dans la section 7 les différentes stratégies de recherche du meilleur plan proposées par les chercheurs. Nous terminons le chapitre en proposant une stratégie génétique originale appliquée au choix d'algorithmes de traversées de chemins.

2. PROBLEMATIQUE DES REQUÊTES OBJET

Dans cette section, nous examinons plus particulièrement les problèmes soulevés par l'optimisation de requêtes OQL ou SQL3 par rapport aux requêtes purement relationnelles, c'est-à-dire exprimées en SQL2.

2.1. QU'EST-CE QU'UNE REQUÊTE OBJET ?

Les bases de données objet et plus encore relationnel-objet permettent de formuler des requêtes classiques, c'est-à-dire exprimables en SQL de base. En conséquence, l'optimiseur d'un SGBD doit toujours être capable d'effectuer les optimisations étudiées pour le relationnel. Au-delà, les requêtes objet intègrent des constructions inexprimables en relationnel. À ce titre, elles posent des problèmes d'optimisation spécifiques. Les points nouveaux sont pour l'essentiel :

1. Le **support de méthodes et opérateurs définis par l'utilisateur**. Dans les systèmes objet, l'utilisateur peut définir ses propres méthodes, ses propres comparateurs et aussi surcharger les opérateurs du langage de requêtes, par exemple l'addition. Ceci pose au moins trois problèmes d'optimisation : comment utiliser des accélérateurs, par exemple des index pour accélérer l'évaluation de requêtes ? Comment générer toutes les séquences de méthodes ou d'opérateurs possibles pour calculer la réponse à une requête ? Comment évaluer le coût d'une méthode programmée par l'utilisateur ?
2. Les **traversées de chemins.** Celles-ci sont des extensions des jointures relationnelles effectuées directement par parcours de pointeurs. Ces pointeurs implémentent les associations. Il n'est pas toujours évident de les utiliser efficacement, et parfois des jointures classiques peuvent être plus performantes.
3. Les **manipulations de collections**. Les collections sont implémentées par des structures de données spécifiques et supportent des opérateurs spécifiques, par exemple union, intersection, différence, inclusion, recherche d'appartenance, pour les ensembles. Des règles de transformation nouvelles sont applicables aux collections. Il faut pouvoir en donner connaissance à l'optimiseur, qui doit être capable de les prendre en compte. De plus, évaluer le coût d'un opérateur sur collections n'est pas chose évidente.
4. La **prise en compte de l'héritage et du polymorphisme**. L'héritage stipule que tout objet d'une sous-classe est membre implicite de la classe dont il hérite. Cette règle doit pouvoir être prise en compte lors de l'optimisation. Elle introduit des difficultés lors des surcharges ou redéfinitions de méthodes possibles grâce au polymorphisme dans les sous-classes. Les calculs de coût à la compilation des requêtes

deviennent alors très difficiles, les méthodes réellement appliquées n'étant connues qu'à la compilation.

2.2. QUELQUES EXEMPLES MOTIVANTS

Nous illustrons le concept de requête objet par quelques exemples sur la base représentée figure XIV.1. Celle-ci est une interprétation objet de la situation bien connue où des clients passent des commandes de produits à des fournisseurs. Une commande contient plusieurs lignes. Clients et fournisseurs sont des personnes. Les chemins de traversée sont implémentés par des attributs portant le nom des associations contenant des identifiants d'objets. Ils sont marqués par des flèches.

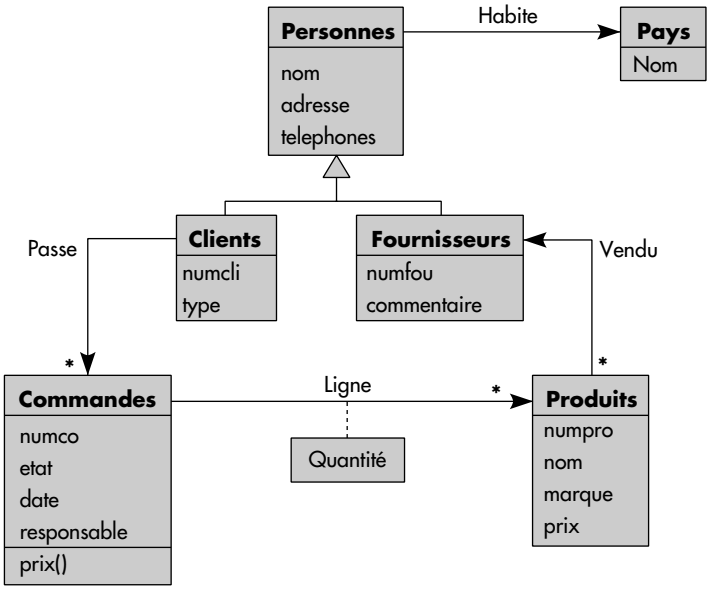


Figure XIV.1 : Schéma de base de données objet

2.2.1. Parcours de chemins

Supposons par exemple que l'on recherche le nom des produits et l'adresse des fournisseurs associés pour ceux de plus de 50 ans qui habitent en France. La question nécessite une expression de chemins en résultat mais aussi deux expressions de chemins en qualification, pour tester les prédicats `Age > 50` et `Pays = "France"`. Elle peut s'écrire comme suit en OQL :

```
(Q1)  SELECT P.NOM, P.VENDU.ADRESSE
      FROM PRODUITS P
      WHERE P.PRIX > 10.000
      AND P.VENDU.AGE > 50
      AND P.VENDU.HABITE.PAYS = "FRANCE".
```

On note que les chemins sont monovalués. Il est aussi possible de parcourir des chemins multivalués, en utilisant par exemple des collections dépendantes en OQL. La question suivante recherche par exemple les clients qui ont commandé des produits de prix supérieur à 10 000 :

```
(Q2)  SELECT C.NOM, C.ADRESSE
      FROM CLIENTS C, C.PASSE M, M.LIGNE P
      WHERE P.PRIX > 10.000
```

Le problème essentiel pour ce type de requêtes est de choisir un algorithme de parcours des chemins de traversée, de sorte à réduire au maximum le nombre d'objets traversés. Des jointures par valeur, comme en relationnel, peuvent parfois être plus avantageuses afin de trouver rapidement des résultats à partir d'un petit nombre d'objets en mémoire.

2.2.2. Méthodes et opérateurs

Une partie de la base de données décrite figure XIV.1 peut être étendue avec la localisation géométrique des clients et des fournisseurs. Nous supposons que l'unité de mesure des distances est le kilomètre. Les types de données suivant peuvent être définis :

```
TYPE POINT (  ATTRIBUTS X, Y COORDONNÉES ;
              REAL FUNCTION DISTANCE(P1 POINT, P2 POINT)) ;
TYPE ZONE (   ATTRIBUTS P1, P2 POINT ;
              BOOLEAN FUNCTION INCLUS(Z ZONE, P POINT)).
```

Considérons une implémentation des fournisseurs et clients par deux tables :

```
FOURNISSEURS (NUMFOU INT, NOM VARCHAR, ADR ADRESSE, NUMPAYS INT, TEL
   TELEPHONES, LOCALISATION POINT) ;
CLIENTS (NUMCLI INT, NOM VARCHAR, ADR ADRESSE, NUMPAYS INT, TEL
   TELEPHONE, LOCALISATION POINT).
```

L'optimiseur doit alors être capable d'évoluer en prenant en compte les nouveaux types et les règles d'optimisation des opérations sur ces types. Par exemple, on pourra maintenant rechercher tous les fournisseurs localisés dans une zone donnée (:Z) et à moins de 100 km d'un client donné (:N) par la requête :

```
(Q3)  SELECT NUMFOU, NOM, ADRESSE
      FROM CLIENTS C, FOURNISSEURS F
      WHERE C.NUMCLI = :N AND INCLUS( :Z, F.LOCALISATION)
      AND DISTANCE(C.LOCALISATION, F. LOCALISATION) < 100 ;
```

Supposons maintenant que les types figure et cercle soient connus du SGBD extensible, défini comme suit :

```
TYPE     FIGURE  (ATTRIBUT CONTOUR LIST<POINT> ;
                 BOOLEAN FUNCTION INCLUS (P POINT, F FIGURE))
TYPE     CERCLE  (ATTRIBUT CENTRE POINT, RAYON REAL ;
                 CERCLE FUNCTION CERCLE(C POINT, R REAL) ;
                 FIGURE FUNCTION INTERSECTION(C CERCLE, Z ZONE)).
```

Une requête équivalente à la requête précédente consiste à chercher tous les fournisseurs inclus dans la figure géométrique résultant de l'intersection de la zone paramètre et d'un cercle de centre le client et de rayon 100 km, ce qui donne :

```
(Q4)  SELECT NUMFOU, NOM, ADRESSE
      FROM CLIENTS C, FOURNISSEURS F
      WHERE C.NUMCLI = :N
      AND INCLUS(F.LOCALISATION, INTERSECTION(CERCLE(C.LOCALISATION,100),:Z)
```

Un bon optimiseur doit être capable de déterminer le meilleur plan d'exécution pour cette requête. Il doit donc s'apercevoir de l'équivalence des formulations. Voilà qui n'est pas chose simple et demande de bonnes notions de géométrie !

2.2.3. Manipulation de collections

Outre les parcours de chemins vus ci-dessus, les collections peuvent être manipulées par des opérations spécifiques d'extraction de sous-collections, de concaténation, d'intersection, de recherche d'éléments, etc. Ces opérations peuvent obéir à des règles de réécritures spécifiques. Par exemple, trouver si un élément appartient à l'union de deux collections revient à trouver s'il appartient à l'une ou l'autre. Supposons que `telephones` soit un attribut contenant un ensemble de numéros de téléphone. La requête suivante recherche les noms et adresses des clients de Paris dont le numéro de téléphone est donné par la variable :N :

```
(Q5)  SELECT C.NOM, C.ADRESSE
      FROM CLIENTS C
      WHERE :N IN
      UNION(   SELECT S.TELEPHONES
               FROM CLIENTS C
               WHERE ADRESSE LIKE "PARIS")
```

Grâce aux propriétés des ensembles, plus particulièrement de l'opérateur d'union par rapport à l'appartenance, elle doit pouvoir être transformée en une requête plus simple.

2.2.4. Héritage et polymorphisme

Lors d'une requête sur une super-classe, l'héritage nécessite de retrouver tous les éléments des sous-classes participant à la réponse. Couplé au polymorphisme, il peut

impliquer des difficultés pour l'optimiseur. En effet, la liaison retardée conduit à ne pas connaître lors de la compilation le code réellement appliqué suite à un appel de méthode. Supposons une fonction revenus() calculant les revenus d'une personne. Celle-ci pourra être différente pour un fournisseur et un client, donc redéfinie au niveau de la classe clients et de la classe fournisseurs. La requête suivante :

```
(Q6)  SELECT NOM, ADRESSE
      FROM PERSONNES
      WHERE REVENUS() > 50.000
```

nécessite donc a priori de calculer le revenu de chaque client et de chaque fournisseur pour vérifier s'il est supérieur à 50 000. La fonction effectivement appliquée dépend du sous-type de la personne. Le coût du plan est donc difficile à estimer lors de la compilation de la requête.

3. MODÈLE DE STOCKAGE POUR BD OBJET

Dans cette section, nous introduisons un modèle de stockage général pour bases de données objet. Ce modèle intègre différentes variantes d'identifiants d'objets, de techniques de groupages d'objets et d'indexation. Il est suffisamment général pour représenter les méthodes de stockage existant dans les SGBD. L'objectif est d'illustrer les techniques possibles, qui nous servirons plus loin de base aux calculs des coûts d'accès.

3.1. IDENTIFIANTS D'OBJETS

La plupart des SGBD stockent les objets dans des pages à fentes (*slotted pages*). Chaque page contient en fin de page un tableau de taille variable adressant les objets de la page. La figure XIV.2 illustre une page à fentes. Le déplacement conduisant du début de la page au début de chaque objet est conservé dans un petit index en fin de page.

Comme nous l'avons déjà vu, dans les SGBDO, les objets sont désignés par des identifiants d'objets. Il est possible de distinguer **identifiants physiques** et **identifiants logiques**.

Notion XIV.1 : Identifiant physique (*Physical OID*)

Identifiant d'objet composé d'un numéro de fichier, d'un numéro de page et d'un numéro de fente dans la page contenant l'objet.

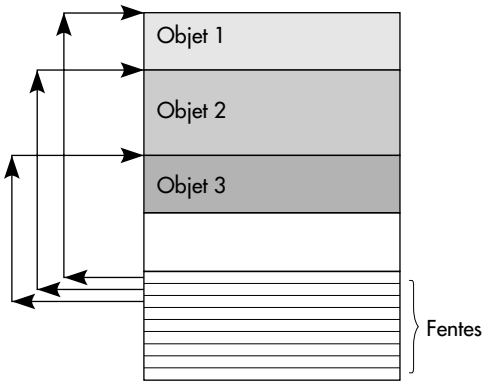


Figure XIV.2 : Page à fentes

Les identifiants physiques sont rapides à décoder pour atteindre l'objet référencé. Par contre, ils ne permettent pas le déplacement libre des objets dans la base. En cas de déplacement, une technique de chaînage doit être mise en place : l'entrée de la page initiale doit indiquer que l'objet n'est plus dans la page et pointe sur son nouvel identifiant physique. Cette technique de suite (*forwarder*) est très lourde à gérer, surtout si les objets se déplacent souvent. On lui préférera les identifiants logiques, plus coûteux lors de l'accès mais invariants au déplacement d'objet.

Notion XIV.2 : Identifiant logique (*Logical OID*)

Identifiant d'objet composé d'un numéro d'entrée dans une table permettant de retrouver l'identifiant physique de l'objet.

En cas de déplacement de l'objet, seule l'entrée dans la table est changée ; elle est positionnée à la nouvelle adresse de l'objet. En général, les SGBD gèrent une table d'identifiant logique par type d'objets. La taille de la table est ainsi limitée. Pour la limiter plus encore et éviter les entrées inutiles, la table est souvent organisée comme une table hachée : chaque identifiant logique est affecté à une entrée déterminée par une fonction de hachage ; chaque entrée contient des couples de correspondance <identifiant logique-identifiant physique>. La figure XIV.3 illustre le décodage d'un identifiant logique.

Un bon modèle de stockage permet à la fois les identifiants physiques (OIDP) et les identifiants logiques (OIDL). Le choix du type d'identifiants affecte directement le coût de traversée des pointeurs.

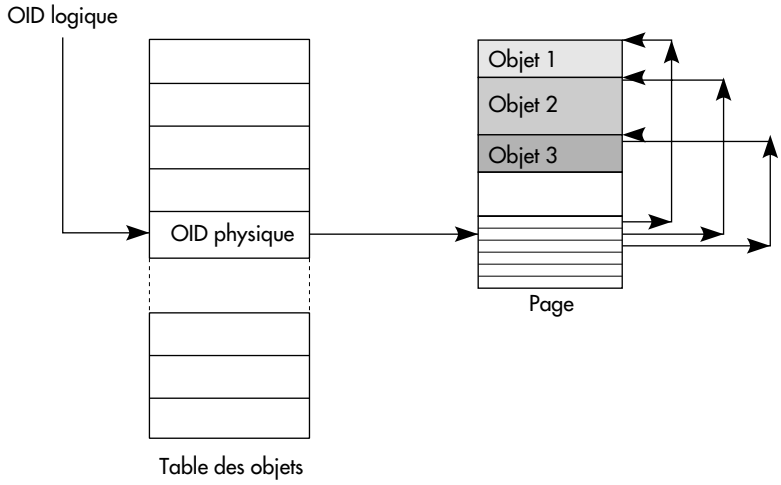


Figure XIV.3 : Décodage d'un identifiant logique

3.2. INDEXATION DES COLLECTIONS D'OBJETS

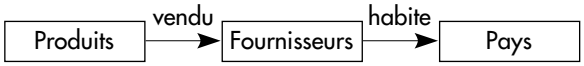
Pour accélérer l'évaluation de requêtes avec prédicats de sélection, les systèmes offrent des index, généralement organisés comme des arbres-B contenant dans les feuilles des listes d'identifiants d'objets. Un index peut être perçu comme une fonction donnant une liste d'identifiants d'objets à partir d'une valeur d'attributs. Le coût de traversée d'un index est généralement de deux E/S, voire trois pour des très grosses collections d'objets.

Au-delà des index classiques, certains systèmes à objets offrent des index de chemins.

Notion XIV.3 : Index de chemin (*Path index*)

Index donnant pour chaque valeur d'un attribut d'une collection d'objets à l'extrémité d'un chemin tous les chemins menant à des objets ayant cette valeur d'attributs.

Il existe différents types d'index de chemins [Bertino89, Kim89]. Les **index de chemin simples** [Bertino89] [Kemper90] associent pour chaque valeur figurant à la fin du chemin tous les identifiants d'objets suffixes du chemin. Ils peuvent être implémentés comme des relations : chaque colonne d'un tuple correspond à une collection du chemin, en remontant depuis la fin du chemin. Le premier attribut contient la valeur figurant à la fin du chemin, les suivants contiennent des identifiants d'objets désignant les objets successivement rencontrés en parcourant le chemin à l'envers. La figure XIV.4 illustre un index de chemins simple pour le chemin Produits → Fournisseurs → Pays de la base représentée figure XIV.1.



Pays	Fournisseurs	Produits
Allemagne	F1	P1
	F1	P2
	F2	P1
	F2	P3
France	F3	P4
	F4	P5
	F4	P1
Italie	F5	P1
USA	F6	P1
	F6	P6
	F7	P7

Figure XIV.4 : Exemple d'index de chemin simple

Une première variante consiste à imbriquer les chemins, en évitant ainsi de répéter plusieurs fois un même identifiant d'objet pour chacun de ses parents. Par exemple, pour le fournisseur F1, on indiquera directement la liste des parents, pour F2 de même, si bien que l'entrée Allemagne de l'index précédent devient (Allemagne(F1(P1, P2) F2 (P1, P3) …). Un tel index est appelé **index de chemin imbriqué** [Bertino91]. De tels index sont plus difficiles à maintenir et à utiliser que des tables lors des mises à jour.

Les **multi-index** [Kim89] sont une alternative aux index de chemin. Pour chaque couple de collections (C_i, C_{i+1}) consécutives du chemin, on gère un index de jointure [Valduriez87] contenant les couples identifiants d'objets liés par le chemin. Le dernier index est un index classique donnant pour chaque valeur terminale du chemin l'identifiant de l'objet contenant cette valeur. Le parcours du chemin s'effectue alors par des intersections de listes d'identifiants. Chaque index est finalement un index classique qui peut être implémenté comme un arbre B. Le problème des multi-index est que chaque index doit être lu ou écrit indépendamment, ce qui nécessite des entrées-sorties supplémentaires par rapport aux index de chemin. L'utilisation de tels index est par contre plus large, par exemple pour accélérer des jointures de collections intermédiaires sur le chemin ou des sélections sur la collection feuille.

3.3. LIAISON ET INCORPORATION D'OBJETS

Dans les SGBD modernes, les objets associés peuvent être stockés ensemble comme un objet composite (c'est-à-dire composés d'autres objets), ou séparément comme

plusieurs objets reliés par des identifiants logiques ou physiques. On distingue alors la **liaison** de l'**incorporation** d'objets.

Notion XIV.4 : Liaison d'objet (*Object linking*)

Représentation d'objets associés par un attribut mono– (association 1-1) ou multivalué (association 1-N) contenant un ou plusieurs identifiants d'objet pointant depuis un objet vers l'objet associé.

La liaison peut être effectuée dans les deux sens, l'objet référencé pointant lui-même son ou ses objets associés. Il existe alors pour chaque objet cible un ou plusieurs liens inverses vers l'objet source. La liaison est une technique classique pour représenter les associations nécessitant le parcours d'identifiants pour retrouver les objets associés. La liaison inverse permet le parcours de l'association dans les deux sens. La liaison a le mérite de rendre les objets liés quasiment indépendants, ce qui n'est pas le cas avec l'incorporation.

Notion XIV.5 : Incorporation d'objet (*Object embedding*)

Représentation d'objets associés par un objet composite contenant un objet englobant avec un attribut complexe mono – (association 1-1) ou multivalué (association 1-N) contenant directement la valeur des objets liés.

L'incorporation rend les objets incorporés dépendants de l'objet englobant : ils sont détruits avec lui. Le choix entre liaison ou incorporation est donc aussi un problème de sémantique : un objet incorporé a le même cycle de vie que l'objet incorporant. Du point de vue stockage, l'incorporation présente l'avantage de rendre les objets incorporant et incorporé accessibles simultanément. En outre, elle permet le placement des objets dans une même page, ce qui est aussi possible avec la liaison comme nous le verrons ci-dessous. Elle présente par contre l'inconvénient de ne pas supporter le partage référentiel des objets dépendants, qui sont de fait copiés s'ils sont référencés ailleurs. La figure XIV.5 illustre la liaison et l'incorporation dans le cas de commandes et de lignes de commandes.

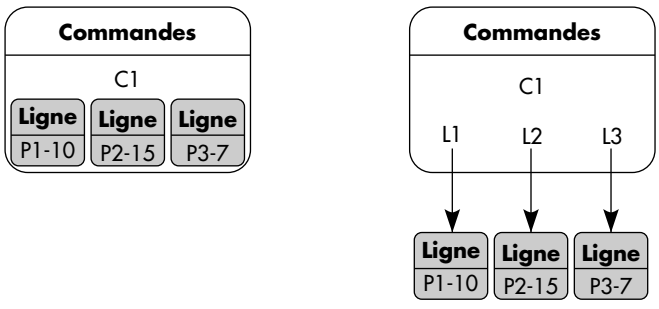


Figure XIV.5 : Incorporation et liaison d'objets

3.4. GROUPAGE D'OBJETS

Pour permettre la navigation rapide d'un objet maître vers des objets liés (par exemple, des commandes aux lignes de commandes) ou pour accélérer les jointures dans le cas d'une implémentation relationnelle, il est possible de grouper différents objets liés dans une même page.

Notion XIV.6 : Groupage (*Clustering*)

Technique consistant à stocker dans une même page des objets liés par une association afin d'accélérer les accès à ces objets par navigation ou jointure.

Le groupage est plus souple que l'incorporation car il permet une vie autonome des objets liés, qui peuvent exister sans objets associés. En cas d'objets sans maître ou d'objets partagés par plusieurs maîtres, le choix de la page de stockage n'est pas évident. Pour capturer une large classe de stratégies de groupage, il est possible de définir les groupes par des prédicats de sélection ou d'association. Un prédicat de sélection permet par exemple de définir le groupe des commandes de prix supérieur à 10 000 (`Commandes.prix>10.000`). Un prédicat d'association permet par exemple de définir un groupe pour chaque produit (`Produits vendu Fournisseurs`). Ces prédicats sont utilisés pour définir les ensembles d'objets à stocker ensemble dans une même page ou au moins dans des pages à proximité.

Les prédicats n'étant pas forcément disjoints, un objet peut appartenir logiquement à plusieurs groupes. Si l'on exclut les duplications d'objets, il faut lors du chargement choisir un groupe. Une technique possible consiste à associer des priorités à chaque prédicat de liens. Si un objet appartient à deux groupes ou plus, celui de priorité supérieure est retenu. Pour pouvoir discriminer les groupes, les priorités doivent être différentes, définies par exemple par un nombre de 0 à 10.

Afin de visualiser les informations de spécifications des groupes, il a été proposé [Amsaleg93] d'utiliser un **graphe de groupage** défini comme suit :

Notion XIV.7 : Graphe de groupage (*Clustering graph*)

Graphe dont les nœuds représentent les extensions de classes et les arcs les prédicats de liens servant au groupage, chaque arc ayant une priorité pouvant varier de 0 à 10.

Un graphe de groupage est correct si tous les arcs pointant vers un même nœud ont une priorité différente. Les objets pointés par plusieurs liens de groupage sont donc assignés au groupe de plus forte priorité. La figure XIV.6 présente différents graphes de groupage possibles pour les fournisseurs, les produits et les commandes. Elle suppose que l'association directe des fournisseurs aux produits est gérée par le SGBD. À droite du graphe de groupage, les groupes de différents types possibles sont représentés, par exemple les groupes de fournisseurs, de produits, ou les groupes mixtes.

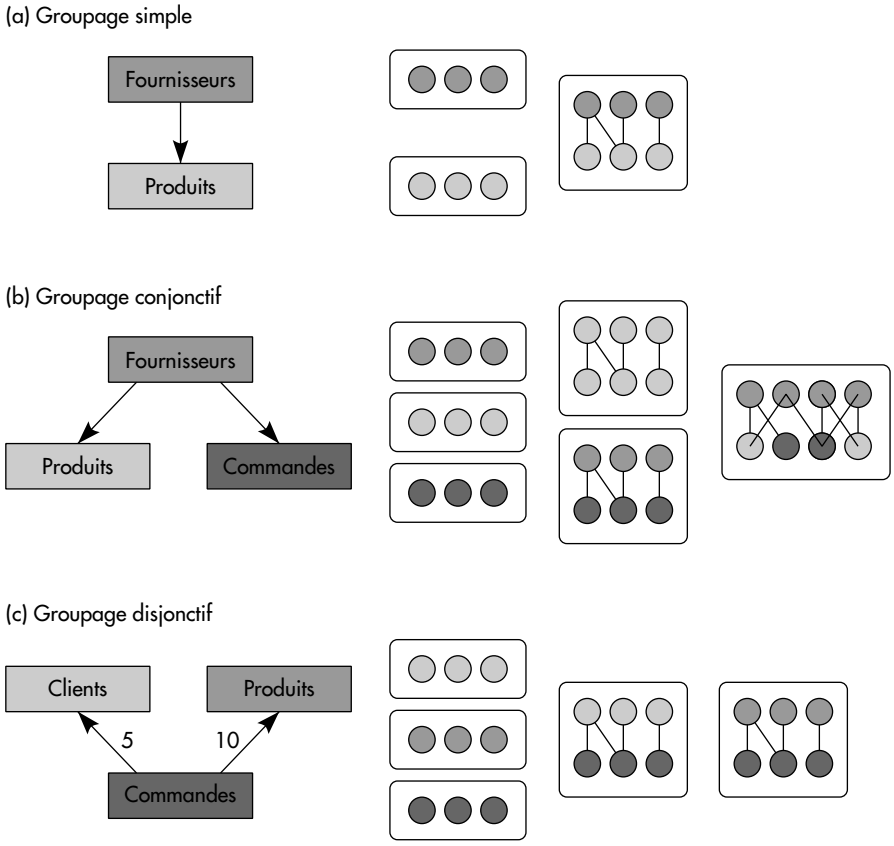


Figure XIV.6 : Exemple de graphe de groupage

La figure XIV.6 montre quatre possibilités de groupage des objets de collections :

1. Le **groupage simple**. C'est le groupage classique de deux collections selon un prédicat de lien. Ce type de groupage existe dans les bases de données en réseaux (maître et membres via un lien) et est possible dans les bases de données relationnelles selon un prédicat de jointure. Figure XIV.6a, les objets de la collection `produits` sont groupés simplement avec les objets de la collection `fournisseurs`, avec une priorité maximale de 10. Tous les produits sans fournisseurs seront groupés ensemble.
2. Le **groupage conjonctif**. Il permet de grouper plusieurs objets de collections différentes avec ceux d'une collection maître. Figure XIV.6b, tous les objets `produits` et `commandes` sont groupés avec leurs `fournisseurs`.
3. Le **groupage disjonctif**. Dans le cas où plusieurs maîtres se partagent un objet qui devrait être groupé avec eux, un choix doit être fait afin d'éviter les duplications d'objets. Selon le mécanisme de priorité, l'objet est alors placé dans le groupe déter-

miné par l'objet maître selon l'arc de poids maximal. Par exemple, figure XIV.6c, chaque objet commandes sera stocké près de son `fournisseur` s'il existe.

3.5. SCHÉMA DE STOCKAGE

Afin de regrouper et visualiser toutes les informations permettant de stocker les objets dans la base et d'y accéder, nous avons proposé [Gardarin95] d'introduire un **graphe de stockage** plus complet que le graphe de groupage vu ci-dessus. Ce graphe permet de représenter un schéma objet de données avec des notations proches d'UML (voir chapitre sur la conception), avec en plus les informations suivantes :

1. les objets incorporés liés aux objets incorporants par des flèches doubles en pointillés ;
2. les objets groupés liés aux objets groupants par des flèches simples en pointillés étiquetées par la priorité du groupage et l'éventuel prédicat, s'il ne s'agit de l'association existant entre les objets ;
3. les index par des réseaux de lignes pointillées issus de l'attribut indexé et pointant vers la ou les collections indexées.

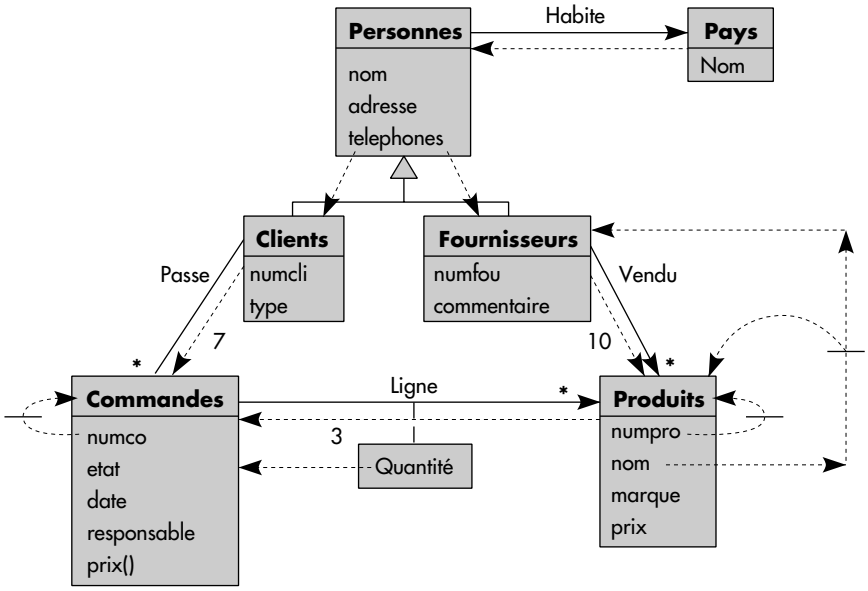


Figure XIV.7 : Exemple de graphe de stockage

La figure XIV.7 représente un exemple de graphe de stockage pour la base dont le schéma a été défini figure XIV.1. `Pays` est incorporé dans `Personnes`, dont les

attributs sont eux-mêmes incorporés dans `Clients` et `Fournisseurs`. Les `lignes` de commandes sont incorporées dans les commandes. Elles référencent les produits. Les `commandes` sont placées en priorité à proximité des `clients`, sinon à proximité des produits. Les `produits` sont placés à proximité des `fournisseurs`. Il existe un index de chemin depuis `nom` de `produit` vers `produits` et `fournisseurs`. Deux index primaires classiques sont gérés, respectivement sur numéro de commande (`numco`) et numéro de produit (`numpro`).

Un graphe de stockage induit une organisation physique des données. Il représente en fait le modèle interne de la base et est donc un élément essentiel à prendre en compte pour l'optimiseur. Le problème de l'optimiseur est de trouver le meilleur plan d'accès pour exécuter une requête compte tenu du modèle interne représenté par le graphe de stockage. Pour ce faire, l'optimiseur doit aussi prendre en compte la taille des collections et groupes. Il doit bien sûr optimiser la navigation via des pointeurs, comme nous allons le voir ci-dessous. Notez que la plupart des systèmes objet ne distinguent pas clairement le graphe de stockage et n'ont guère d'optimiseur élaboré capable de bien prendre en compte le modèle interne.

4. PARCOURS DE CHEMINS

Optimiser la navigation dans une base de données objet est l'un des soucis essentiels d'un optimiseur. Ce problème prolonge l'optimisation des séquences de jointures dans les bases de données relationnelles. Dans cette partie, nous présentons plusieurs algorithmes pour évaluer une expression de chemins avec prédicats. De tels algorithmes ont été étudiés dans [Bertino91, Shekita90, Gardarin96]. Ils correspondent à des types variés de traversée d'un graphe d'objets. Un tel graphe est représenté figure XIV.8.

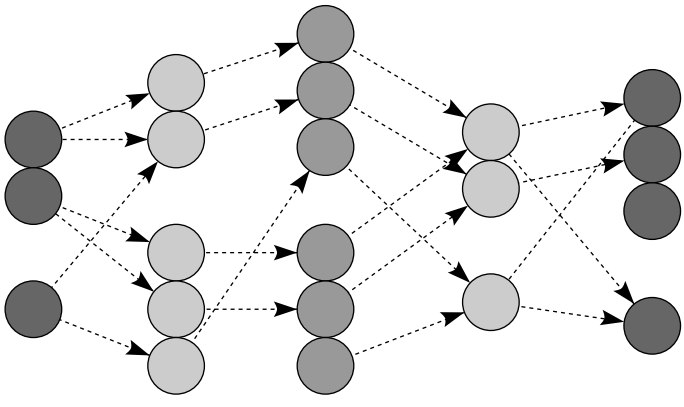


Figure XIV.8 : Exemple de graphe d'objets

Chaque famille verticale d'objets correspond à une collection C_i (par exemple, une extension de classes). Chaque objet de la collection C_i pointe sur 0 à N objets de la collection C_{i+1}, selon les cardinalités des associations traversées. Nous écrivons l'expression de chemin $C_1(P_1).C_2(P_2)\ldots C_n(P_n)$ en désignant par P_i le prédicat de sélection de la collection C_i. Il s'agit donc d'assembler les séquences $c_1.c_2\ldots c_n$ d'objets liés par des pointeurs et vérifiant respectivement ($c_1 \in C_1$ et P_1), ($c_2 \in C_2$ et P_2)… ($c_n \in C_n$ et P_n).

4.1. TRAVERSÉE EN PROFONDEUR D'ABORD

La **traversée en profondeur d'abord** (*Depth-First-Fetch, DFF*) est la méthode la plus naturelle pour évaluer une expression de chemins avec prédicats. L'algorithme suit le chemin depuis la collection racine vers la collection terminale. Chaque objet est traversé en allant toujours vers le premier objet du niveau supérieur non encore traversé, vers l'objet de même niveau s'il n'en n'existe pas, et en remontant lorsqu'il n'y a plus d'objet voisin. L'opérateur correspondant est un opérateur n-aire, qui s'applique sur N collections avec un prédicat possiblement vrai (tous les objets qualifient alors) pour chaque collection. DFF peut être vu comme un opérateur de jointures en cascade utilisant une méthode pipeline. La figure XIV.9 donne un sketch de l'algorithme.

```
// Recherche des objets d'un chemin C1.C2…Cn vérifiant les prédicats P1,P2…Pn
DFF(C1(P1).C2(P2)....Cn(Pn)) {
   i = 1 ;
   while (i≤n) {
      for each x in (Ci) {
         if Pi(x) = VRAI {
         if i < n return (x + DFF(FETCH(x.Ci+1)(Pi+1)….Cn(Pn)))
         else return(x) }
         }
      i = i+1 ;
      }
   }
```

Figure XIV.9 : Traversée en profondeur d'abord

L'avantage de DFF est qu'il s'agit d'un opérateur n-aire qui ne génère pas de résultats intermédiaires. L'algorithme assemble les objets en accédant via les OID, ce qui est efficace dans la plupart des SGBD. Les résultats sont obtenus l'un après l'autre en pipeline. L'algorithme est d'ailleurs très analogue à celui de jointures n-aire en pipeline vu dans le cadre relationnel, à ceci près qu'il navigue en suivant les pointeurs. Ainsi le SGBD peut-il retourner une réponse avant d'avoir traversé tous les objets. Intuitivement, cet opérateur est très efficace lorsque la taille mémoire est suffisante

pour contenir tous les objets sur un chemin de la racine aux feuilles, c'est-à-dire au moins une page par collection. Cependant, si les objets ne sont pas groupés selon le chemin et si la taille mémoire est insuffisante, le système peut être conduit à relire de nombreuses fois une même page.

4.2. TRAVERSÉE EN LARGEUR D'ABORD

La **traversée en largeur d'abord** (*Breadth-First-Fetch, BFF*) parcourt l'arbre d'objets par des jointures binaires en avant (*Forward Join, FJ*) successives. Les jointures sont effectuées par parcours des pointeurs de la collection C_i vers la collection C_{i+1}. Ainsi, soit une expression de chemin qualifiée $C_1(P_1).C_2(P_2)\ldots C_n(P_n)$. Pour trouver les objets qualifiants, (n-1) jointures par références successives du type $S_{i+1} = FJ(S_i \rightarrow C_{i+1}(P_{i+1}))$ sont accomplies, où Si désigne une table mémorisant les objets satisfaisant le sous-chemin $C_1(P_1).C_2(P_2)\ldots C_i(P_i)$. Le critère de jointure est une simple traversée de pointeurs. Le prédicat P_{i+1} doit être vérifié sur les objets cibles de C_{i+1}. L'algorithme est proche de celui réalisant la traversée en profondeur vue ci-dessus (DFF), mais réduit à deux collections, c'est-à-dire à un chemin de longueur 1. Une table intermédiaire répertoriant les identifiants d'objets qualifiants doit être générée en résultat de chaque jointure en avant. Dans la suite, nous appelons cette table **table support**. La jointure en avant suivante repart de la table support. La figure XIV.10 illustre l'algorithme de traversée en largeur d'abord pour trois collections, C_1, C_2 et C_3, dont les objets sont notés a_i, b_i et c_i respectivement.

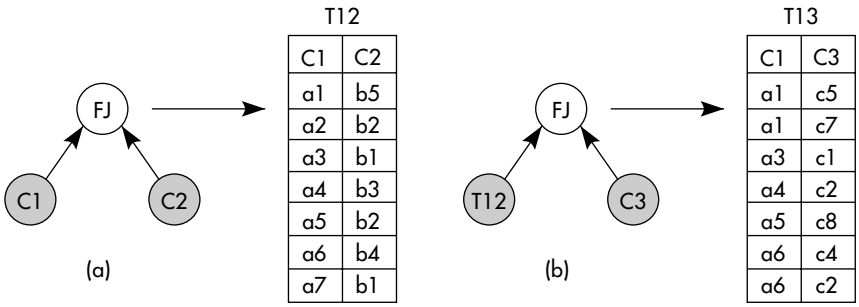


Figure XIV.10 : Traversée en largeur d'abord

L'avantage de l'algorithme en largeur d'abord est d'être basé sur des jointures binaires par référence, donc pré-calculées. Le coût de calcul est donc réduit. Cependant, pour accomplir la jointure entre les collections C_i et C_{i+1}, les états des objets de la première collection C_i doivent être chargés en mémoire pour trouver les pointeurs vers les objets de C_{i+1} (contenus dans l'attribut A_i) ; les états de la seconde C_{i+1} doivent aussi être chargés en mémoire pour tester le prédicat P_{i+1}. Si aucun groupage selon l'association

n'est réalisé, des accès multiples aléatoires aux pages peuvent être nécessaires. Un tri des identifiants des objets de la deuxième collection sera alors souvent avantageux. Contrairement à l'algorithme DFF, l'algorithme BFF nécessite de conserver si possible en mémoire des résultats intermédiaires. Si l'on veut délivrer tous les objets sur les chemins qualifiants, la table support doit être étendue avec un attribut contenant les identifiants d'objets par collection traversée. L'algorithme ne permet guère de délivrer des résultats avant la traversée totale au moins des n-1 premières collections. Soulignons que dans le cas réduit de deux collections, les algorithmes DFF et BFF sont similaires.

4.3. TRAVERSÉE À L'ENVERS

La traversée à l'envers, c'est-à-dire en partant par la dernière collection du chemin, est aussi possible en utilisant un algorithme de jointure binaire classique, de type jointure relationnelle. Comme l'algorithme BFF, la traversée en largeur d'abord à l'envers (*Reverse-Breadth-First-Fetch, RBFF*) accomplit une séquence de jointures binaires entre collections voisines sur le chemin, mais procède en ordre inverse, donc à reculons. Chaque jointure est appelée une jointure inverse (*Reverse Join, RJ*). S'il n'y a pas de lien inverse, il s'agit d'une jointure par valeur, c'est-à-dire que le critère est l'appartenance de l'identifiant de l'objet de la seconde collection aux valeurs de l'attribut de liens de la première collection. Pour traiter une expression de chemin qualifiée $C_1(P_1).C_2(P_2)\ldots C_n(P_n)$ de la collection C1 à collection Cn, (n-1) jointures successives Si = RJ(Si←Ci(Pi)) sont accomplies, où Si designe la table support des objets satisfaisant le sous-chemin $C_i(P_i)\ldots C_n(P_n)$; RJ est l'algorithme de jointure inverse testant l'appartenance de l'identifiant aux pointeurs. Une technique de jointure par hachage ou par tri-fusion peut être appliqué. La traversée à l'envers est illustrée figure XIV.11.

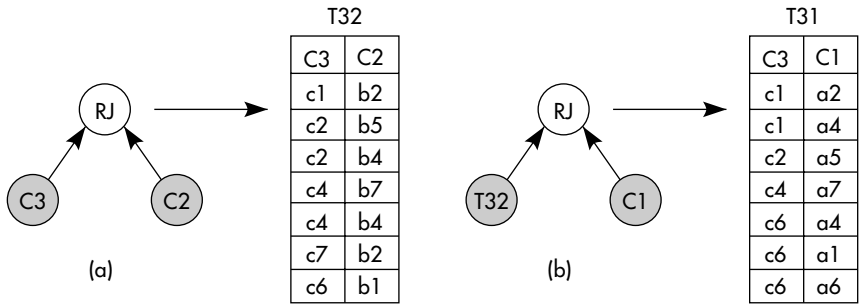


Figure XIV.11 : Traversée à l'envers

La traversée à l'envers est avantageuse lorsqu'une sélection directe est possible sur la dernière collection (par exemple par un index) et que cette sélection retrouve un nombre faible d'objets (forte sélectivité). Pour mémoriser le parcours, une table sup-

port est générée (voir figure XIV.11). Un avantage de la méthode est qu'à chaque pas la jointure est faite entre cette table support et la collection précédente sur le chemin. L'algorithme est bon si la table support est petite (expression de chemin sélective).

Soulignons qu'une expression de chemins peut toujours être divisée en plusieurs sous-expressions, chacune pouvant être traitée avec un algorithme différent, DFF, BFF ou RBFF. Si un prédicat très sélectif est rapidement évaluable sur une collection du chemin, on aura avantage à considérer RBFF pour la partie préfixant cette collection, puis BFF ou DFF pour la partie suffixe. Le choix d'un algorithme ou d'un autre n'est pas évident et nécessite un modèle de coût, comme nous le verrons plus loin.

4.4. TRAVERSÉE PAR INDEX DE CHEMIN

La traversée par index de chemin (*Path Index Traversal, PIT*) bénéficie de l'existence d'un index suivant le chemin. Comme vu ci-dessus, pour une valeur d'attribut de la collection cible, l'index donne tous les sous-chemins qualifiant. Il suffit donc d'accéder à l'index pour trouver les associations d'objets satisfaisant le prédicat. Cependant, si d'autres prédicats figurent dans les collections précédant la collection indexée, il faut accéder aux objets associés dont les identifiants sont trouvés dans l'index. Cela peut être très pénalisant si les accès sont éparpillés sur les disques. Des tris sur les identifiants physiques d'objets seront souvent avantageux, car ils permettront de grouper les accès aux objets d'une même page. Une autre faiblesse des index de chemins est bien sûr les coûts de maintenance lors des mises à jour, souvent importants.

5. GÉNÉRATION DES PLANS ÉQUIVALENTS

5.1. NOTION D'OPTIMISEUR EXTENSIBLE

Plus qu'en relationnel, la génération de plans équivalents doit se faire par application de règles de transformation. En effet, à la différence des systèmes relationnels, les systèmes objet ou objet-relationnel ne sont pas fermés : l'implémenteur de bases de données peut ajouter ses propres types de données obéissant à des règles de transformation spécifiques, comme les types géométriques (point, cercle, etc.) vus ci-dessus. Un optimiseur d'un SGBD objet ou objet-relationnel doit donc être **extensible**.

Notion XIV.8 : Optimiseur extensible (*Extensible optimiser*)

Optimiseur capable d'enregistrer de nouveaux opérateurs, de nouvelles règles de transformation de plans, de nouvelles fonctions de coût et de nouvelles méthodes d'accès lors de la définition de types de données utilisateurs, ceci afin de les utiliser pour optimiser les requêtes utilisant ces types.

Cette notion d'optimiseur extensible a été bien formalisée pour la première fois dans [Grafe93]. Un optimiseur extensible est construit autour d'une base de connaissance mémorisant les définitions de types et opérateurs associés, de méthodes d'accès, de fonctions de coût et de règles de transformation de plans d'exécution. Toutes les transformations de plans peuvent être exprimées sous la forme de règles. Comme en relationnel, les règles permettent de transformer les arbres d'opérateurs de l'algèbre représentant les requêtes en arbres équivalents. Au-delà du relationnel, pour supporter les types de données utilisateurs, l'algèbre relationnelle doit être étendue en une algèbre d'objets complexes, avec en particulier les fonctions dans les expressions de qualification et de projection, et le support d'opérations sur collections imbriquées telles que Nest, Unnest et Flatten. Une telle algèbre a été présentée au chapitre XI.

Nous représentons figures XIV.12 et XIV.13 les questions Q1 et Q3 du paragraphe 2.2 sous la forme d'arbres algébriques. L'arbre de la figure XIV.12 contient trois restrictions sur attributs ou méthodes, suivies de deux jointures par références qui traduisent le parcours de chemin. Ces deux jointures peuvent par exemple être remplacées par un opérateur de parcours du graphe en profondeur d'abord (DFF) vu ci-dessus, ou par des jointures en arrière (RBFF). Une projection finale gagnerait à être descendue par les règles classiques vues en relationnel.

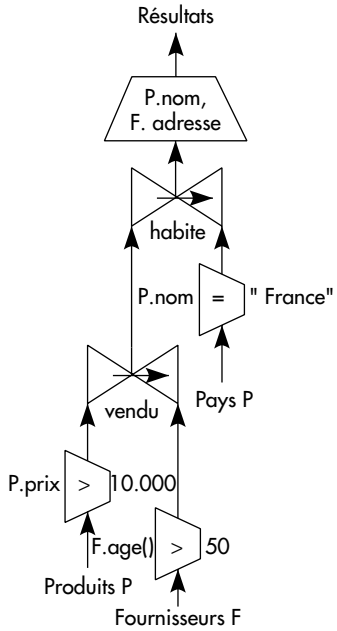


Figure XIV.12 : Arbre algébrique pour la question Q1

L'arbre de la figure XIV.13 présente une restriction sur prédicats utilisateurs (`Inclus`) et une jointure sur fonction utilisateur (`Distance < 100`). L'optimisa-

tion de telles requêtes est difficile et nécessite des connaissances spécifiques sur les fonctions utilisateurs [Hellerstein93], ici des fonctions géométriques.

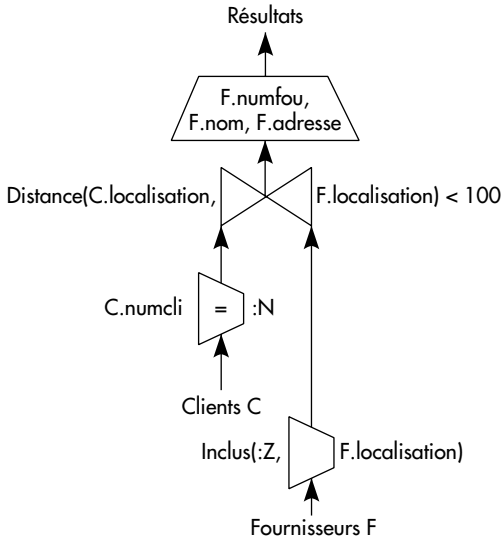


Figure XIV.13 : Arbre algébrique pour la question Q2

5.2. LES TYPES DE RÈGLES DE TRANSFORMATION DE PLANS

Comme pour le relationnel, un plan d'exécution est un arbre annoté, l'annotation associée à chaque opérateur spécifiant l'algorithme choisi. Une règle exprime que lorsqu'un sous-arbre annoté est rencontré dans un plan d'exécution, il peut être remplacé par un autre sous-arbre annoté sous certaines conditions. Le choix d'un langage de règles pour écrire les règles de transformation est un problème difficile [Finance94]. Certaines sont simplement des règles d'équivalence, conduisant à des transformations applicables dans les deux sens. Nous les noterons simplement :

`<SOUS-ARBRE>` ⇔ `<SOUS-ARBRE>`

D'autres règles ne sont applicables que dans un sens et peuvent nécessiter des conditions. À titre d'exemple, nous les écrirons sous la forme suivante :

`<SOUS-ARBRE> [<CONDITION>]` ⇒ `<SOUS-ARBRE>.`

Une telle règle signifie que lorsqu'un sous arbre est rencontré, si la condition optionnelle est satisfaite, il peut être réécrit dans le sous-arbre figurant après l'implication. En pratique, il faut en plus appliquer quelques procédures par exemple pour changer

des variables ou des annotations ; celles-ci sont supposées attachées à la règle ; elles ne sont pas mentionnées pour des raisons de simplicité.

5.2.1. Règles syntaxiques

Les premières règles que l'on peut exprimer simplement sont celles caractérisant les propriétés de l'algèbre, déjà vues dans le cadre relationnel pour la plupart. Par exemple, la règle d'associativité des jointures s'écrira simplement

```
JOIN(JOIN(X,Y),Z) ⇔ JOIN(X,JOIN(Y,Z)) ;
```

Plus complexe, la règle de permutation de la projection sur un schéma Z de la jointure naturelle de deux relations de schémas X et Y nécessite une condition :

```
PROJECT(JOIN(X,Y), Z) [X∩Y ∈ Z] ⇒
JOIN(PROJECT(X,Z∩X),PROJECT(Y,Z∩Y) ;
```

Toutes les règles classiques de l'algèbre relationnelle peuvent ainsi être codées, avec plus ou moins de difficultés. Il est possible aussi d'ajouter des règles prenant en compte les opérations sur collections, NEST,UNNEST et FLATTEN. Par exemple, la règle permettant de pousser une restriction avant une opération d'imbrication NEST peut s'écrire :

```
RESTRICT(NEST(X, G, N), Q) [ ∃ ATTRIBUT(Q) ∈ G ] ⇒
RESTRICT(NEST(RESTRICT(X, Q1), G, N), Q2) ;
```

Cette règle exprime le fait que si une restriction est appliquée sur certains attributs de groupement, alors la partie de la restriction concernant ces attributs peut être appliquée avant le groupement.

Au-delà, des règles de simplification peuvent aussi être introduites pour isoler des sous-arbres identiques et les remplacer par une relation intermédiaire utilisable en plusieurs points. Ce genre de règles est important car il permet d'éviter de calculer deux fois la même sous-question. Son expression générale nécessite un langage de règles un peu plus puissant, permettant de tester si deux sous-arbre sont équivalents.

5.2.2. Règles sémantiques

Nous groupons dans cette catégorie toutes les règles générales permettant de générer des questions équivalentes soit en prenant en compte des règles d'intégrité des données, soit en exprimant des axiomes sur les types de données abstraits.

5.2.2.1. Contraintes d'intégrité

La prise en compte des contraintes sur les données dans les bases de données objet ou objet-relationnel est difficile. Un optimiseur capable de prendre en compte de telles contraintes pour un SGBD objet a été développé à l'INRIA [Florescu96]. Une contrainte est exprimée sous la forme suivante :

```
[FOR ALL [<VARIABLE> OF TYPE <TYPE>]*] <EXPRESSION> ⇔ <EXPRESSION>.
```

Les expressions sont des termes fonctionnels du langage OQL, c'est-à-dire des expressions de chemins, des prédicats avec expressions de chemins ou des requêtes. De telles règles permettent d'exprimer divers types de contraintes, comme l'existence de liens inverses et la redondance de données. Considérons par exemple la définition en ODL de la base décrivant pays et clients, en supposant ces classes liées par une association 1→N :

```
INTERFACE CLIENTS {
   ATTRIBUT        INT NUMCLI KEY, STRING NOM,
                   ADRESSE CADRESSE, REF(PAYS) CPAYS, INT TELEPHONE,
                   INT SEGMENT, STRING COMMENTAIRE }
INTERFACE PAYS {
   ATTRIBUT        INT NUMPAYS KEY, STRING NOM, INT NUMCONT,
                   STRING COMMENTAIRE, RESIDENTS SET <CLIENTS>}
```

La règle de lien inverse spécifiant qu'un client référence un pays s'il est résident de ce pays s'écrit :

```
FOR ALL C OF TYPE CLIENTS, P OF TYPE PAYS
C.CPAYS = P ⇔ C IN P.RESIDENTS.
```

Supposons en plus que les adresses contiennent le pays :

```
INTERFACE ADRESSE {
   ATTRIBUT        INT NUM, STRING RUE,
                   STRING VILLE, INT ZIP, STRING PAYS }
```

Une règle exprimant une redondance de valeurs est :

```
FOR ALL C OF TYPE CLIENTS
C.CADRESSE.PAYS ⇔ C.CPAYS.NOM
```

De telles équivalences sont très puissantes et permettent d'exprimer toutes sortes de redondances, par exemple l'existence de vues matérialisées ; l'équivalence peut être remplacée par une implication. Les équivalences ou implications peuvent être facilement étendues pour exprimer des impossibilités, telles que la valeur nulle interdite ou l'unicité de clé (absence de doubles). Il est aussi possible de prendre en compte des assertions d'inclusion et des assertions d'appartenance [Florescu96].

Par exemple, une règle exprimant la non-nullité de l'attribut pays peut s'écrire :

```
FOR ALL C OF TYPE CLIENTS C
C.CPAYS = NIL ⇔ □
```

Une règle exprimant l'unicité de la clé de client s'écrit :

```
FOR ALL C1,C2 OF TYPE CLIENTS
C1.NUMCLI = C2.NUMCLI ⇔ C1 = C2
```

5.2.2.2. Types abstraits de données

L'équivalence de termes constitue un langage de règles puissant qui peut aussi être adapté aux types abstraits de données. Ajoutons à la base précédente des images décrivant nos clients définies comme suit :

```
INTERFACE IMAGE {
   ATTRIBUT NUMCLI INT, CONTENT ARRAY[1024,1024] INT,
   OPERATION
         ARRAY[10] SUMMARY(IMAGE),
         IMAGE ROTATE (IMAGE, ANGLE),
         IMAGE CLIP (REGION) }
```

Considérons la requête suivante qui recherche les images des clients du segment 5, les intersecte avec une fenêtre prédéfinie `$zone`, et les fait tourner de 90° avant de les afficher :

```
SELECT CLIP(ROTATE(I,90),$ZONE)
FROM CLIENTS C, IMAGES I
WHERE C.NUMCLI = I.NUMCLI
AND C.SEGMENT = 5 ;
```

Il apparaît que plutôt de faire tourner les images, on aurait intérêt à faire l'intersection (le clip) avec la zone tournée de – 90°, et à faire tourner seulement la partie intéressante. L'équivalence de termes est aussi très appropriée à la définition d'axiomes sur des types abstraits. Par exemple, la transformation nécessaire pour réécrire la question précédente de manière plus optimisée est :

```
FOR ALL I OF IMAGE, F OF FENETRE
CLIP(ROTATE(I,$A), F) ⇔ ROTATE(CLIP(I,ROTATE(F,-$A),$A).
```

On découvre alors la question équivalente en principe plus rapide à exécuter :

```
SELECT ROTATE (CLIP(I, ROTATE($ZONE,-90)),90)
FROM CLIENTS C, IMAGES I
WHERE C.NUMCLI = I.NUMCLI
AND C.SEGMENT = 5 ;
```

Plus généralement, de telles règles permettent de prendre en compte les spécificités des types abstraits afin de réécrire les expressions de fonctions dans les requêtes. Elles permettraient par exemple de spécifier des connaissances géométriques suffisantes pour montrer que les questions Q3 et Q4 vues dans la section sur les motivations sont équivalentes.

5.2.3. Règles de planning

Pour générer l'espace de recherche des plans, il faut bien sûr ajouter les règles permettant de choisir les algorithmes de sélection, jointures, et plus généralement les traversées de chemins qualifiés. Pour les systèmes relationnels, les règles de planning sont bien connues [Ioannidis90, Ioannidis91]. Pour les jointures, elles permettent par exemple de choisir le meilleur algorithme parmi le tri-fusion (*SORT-MERGE*), les boucles imbriquées (*NESTED-LOOP*) et la construction d'une table de hachage (*HASH-TABLE*). Ces règles peuvent s'exprimer comme suit :

$$JOIN_{SORT\text{-}MERGE}\ (X,Y) \Leftrightarrow JOIN_{NESTED\text{-}LOOP}\ (X,Y)$$
$$JOIN_{SORT\text{-}MERGE}\ (X,Y) \Leftrightarrow JOIN_{HASH\text{-}TABLE}(X,Y)$$

Pour les systèmes objet, il est possible de les étendre [Gardarin96] par quelques règles permettant de prendre en compte les traversées de chemin, avec les algorithmes présentés dans la section 4. Ces nouvelles règles concernent l'utilisation de l'opérateur DFF de parcours en profondeur d'abord, des jointures par références et des index de chemins. $JOIN_{FJ}$ désigne la jointure par référence en avant non considérée ci-dessus et $JOIN_{RJ}$ la jointure par référence en arrière. $JOIN_{PATH\text{-}INDEX}$ désigne l'algorithme de jointure exploitant l'existence d'un index de chemin. On obtient les nouvelles règles :

1. Jointure en avant ou en arrière :

```
A JOINFJ B ⇔ A JOINRJ B
```

2. Découpage et groupage de traversées en profondeur d'abord :

```
DFF(A,B,C...N) ⇔ JOIN(DFF(A,B..x-1),DFF(x...N))
```

3. Utilisation d'index de chemin :

```
JOINFJ(X,Y) [PATH-INDEX(X,Y)] ⇒ JOINPATH-INDEX(X,Y)
DFF(A,B,C...N)[PATH-INDEX(A,B,C...N)]⇒ JOINPATH-INDEX(X,Y)
```

5.3. TAILLE DE L'ESPACE DE RECHERCHE

Quelle est la taille de l'espace de recherche des plans pour une question donnée ? Tout dépend bien sûr de la forme de la question et des règles qui s'y appliquent. En général, l'optimisation de requêtes objet conduit à des espaces de recherche plus grands que pour les requêtes relationnelles. Cela provient surtout du fait que les requêtes sont souvent plus complexes et les règles de transformation plus nombreuses.

Prenons par exemple le cas d'une expression de chemin linéaire qualifiée. De telles requêtes s'appellent des **requêtes chaînes** (voir figure XIV.14). Le profil de requêtes OQL correspondant est :

```
SELECT (X1, X2….Xn)
FROM X1 IN C1, X2 IN C1.AC2….Xn IN Cn-1.ACn
WHERE ….
```

Pour simplifier, supposons l'absence d'index de chemins. Chaque collection est reliée à chacune de ses voisines par une association multivaluée (le cas monovalué est un cas dégénéré).

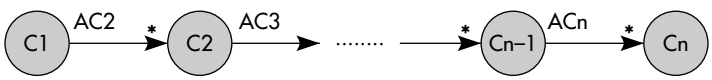


Figure XIV.14 : Une requête chaîne

FJ et RJ sont les deux algorithmes de jointures, l'un par parcours de référence en avant (FJ), l'autre par jointure sur valeur de référence en arrière (RJ). DFF est l'opéra-

teur n-aire de jointure par référence parcourant le graphe en profondeur d'abord. Dans le cas où DFF n'est pas utilisé, le problème revient à calculer le nombre d'ordres de jointures possibles avec deux opérateurs de jointures en évitant les produits cartésiens [Tan91], [Lanzelotte93]. On obtient pour l'espace de recherche de jointures binaires de n collections :

$$BJ\_SS(n) = \frac{(2n-2)!}{n!(n-1)!} * 2^{n-1}$$

Si l'on introduit l'opérateur DFF, la taille de l'espace de recherche devient bien sûr plus grande. Soit *SS(n-1)* l'espace de recherche pour traverser (n-1) collections. On a :

$$SS(n) = BJ\_SS(n) + PC\_SS(n)$$

BJ_SS(n) est l'espace de recherche pour les jointures binaires et *PC_SS(n)* est le nombre d'arbres annotés avec au moins un nœud DFF. Pour déterminer la valeur de *PC_SS(n),* supposons tout d'abord un seul DFF avec trois collections : celles-ci sont traitées ensemble par cet opérateur DFF et peuvent être vues comme une unique collection ; pour le reste, la longueur du chemin est réduite de 2, ce qui signifie que *SS(n – 2)* plans peuvent être générés. Il existe n – 2 positions parmi les n collections où il est possible d'appliquer l'opérateur DFF sur trois collections. Pour la même raison, il existe n – 3 positions où appliquer DFF sur quatre collections ; et ainsi de suite ; finalement, il existe 2 positions où appliquer DFF sur n – 1 collections et 1 position pour appliquer DFF sur les n collections. En sommant tous les cas analysés, on obtient une borne supérieure de *PC_SS(n).* Ainsi, une bonne approximation du nombre de plans est :

$$BJ\_SS(n) = \frac{(2n-2)!}{n!(n-1)!} * 2^{n-1}$$

avec SS(1) = 1, SS(2) = 2, SS(3) = 9.

Ainsi, la table présentée figure XIV.15 donne une approximation du nombre de plans pour des chemins de longueur 1 à 8. On constate qu'au-delà de 8, l'espace devient rapidement important. Il est donc très coûteux pour un optimiseur de considérer toutes les solutions, surtout dans les cas de requêtes plus complexes que les requêtes chaînes.

Nombre de collections	Espace de recherche	Nombre de collections	Espace de recherche
1	1	5	256
2	2	6	1544
3	9	7	9910
4	45	8	65462

Figure XIV.15 : Espace de recherche pour des requêtes chaînes

5.4. ARCHITECTURE D'UN OPTIMISEUR EXTENSIBLE

Un optimiseur extensible doit donc permettre la prise en compte complète de nouveaux types de données, avec opérateurs, règles, etc. Il s'agit donc d'un véritable système expert spécialisé. La figure XIV.16 représente une architecture possible pour un optimiseur extensible [Lanzelotte91]. Celui-ci explore l'espace des plans d'exécution. Un tel optimiseur accepte de l'administrateur la définition de règles de transformation d'arbres algébriques annotés. Il est extensible, en ce sens qu'il est possible d'ajouter de nouveaux types et de nouvelles règles. Il transforme un arbre algébrique d'entrée en un plan d'exécution quasiment optimal. Pour cela, il utilise le schéma de stockage de la base. Le générateur de plan applique les règles. La stratégie d'évaluation est basée sur un modèle de coût et tend à réduire le nombre de plans explorés afin d'atteindre le but qui est un plan de coût proche du minimal possible, dit quasioptimal.

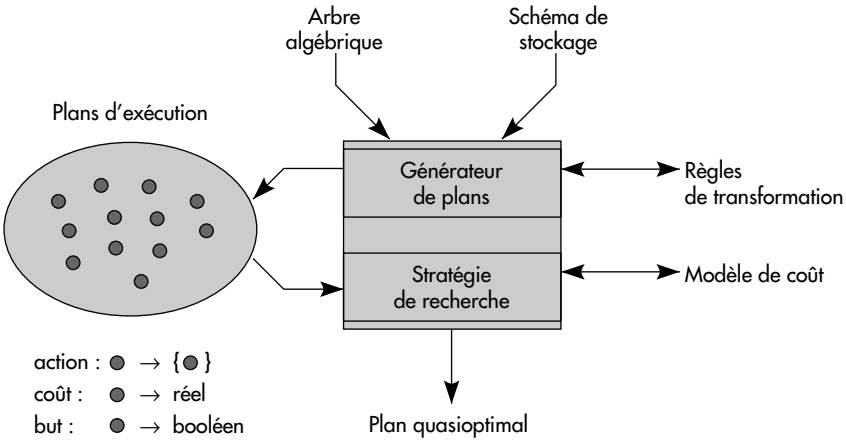


Figure XIV.16 : Architecture d'un optimiseur extensible

6. MODÈLE DE COÛT POUR BD OBJET

Un modèle de coût est un ensemble de formules permettant d'estimer le coût d'un plan d'exécution. Un optimiseur basé sur un modèle de coût cherche à sélectionner le plan de coût minimal parmi les plans équivalents. Le coût d'un plan se compose de plusieurs composants : le coût de calcul (CPU_Cost), le coût d'entrée-sorties (IO_Cost), éventuellement le coût de communication (COM_Cost). Le premier cor-

respond au temps passé à exécuter des instructions de l'unité centrale, par exemple pour évaluer un prédicat ou exécuter une boucle. Le deuxième correspond au temps de lecture et écriture sur disques. Le troisième intervient surtout dans les bases réparties où des échanges de messages sont nécessaires sur le réseau.

Dans cette section, nous présentons un modèle de coût pour bases de données objet, publié dans [Gardarin95]. Considérant une base centralisée, nous évaluons seulement les coûts d'entrées-sorties et de calcul. Pour simplifier, nous supposons que les objets ont une taille inférieure à une page et que les valeurs d'attributs sont uniformément distribuées à l'intérieur de leurs domaines de variation.

6.1. PRINCIPAUX PARAMÈTRES D'UN MODÈLE DE COÛT

Le modèle de coût utilise des statistiques sur les données de la base pour estimer le coût d'une opération. Ces statistiques se composent d'informations sur les collections de la base comme la cardinalité (nombre d'objets) de ces collections, la taille moyenne des objets, le nombre de valeurs distinctes d'un attribut, la présence d'index et de groupes d'objets. Un bon modèle de coût doit prendre en compte le placement des objets dans des groupes (*clustering*). En effet, une collection C peut être divisée en n groupes notés C_1, $C_2 \ldots C_n$. Les techniques de groupage permettent de placer dans un même groupe des objets de collections différentes.

Plus précisément, pour chaque collection d'objets C nous utilisons les statistiques suivantes :

- $||C||$ la cardinalité de la collection C,
- $|C|$ le nombre de pages de la collection C,
- $||C_i||$ la cardinalité du groupe i de la collection C,
- $|C_i|$ le nombre de pages du groupe i de la collection C,
- S_C la taille moyenne des objets dans la collection C.

Afin d'évaluer le coût de la navigation entre objets, il est nécessaire d'estimer finement les cardinalités des associations entre collections. Soient deux collections C1 et C2 reliées par un chemin de traversée implémenté par des pointeurs de C1 vers C2. Un tel chemin C1 $\rightarrow$ C2 sera caractérisé par les statistiques suivantes :

- $Fan_{C1,C2}$ le nombre moyen de références d'un objet de C1 vers des objets de C2,
- $D_{C1,C2}$ le nombre moyen de références distinctes d'un objet de C1 vers des objets de C2,
- $X_{C1,C2}$ le nombre d'objets de C1 n'ayant pas de références (ou des références nulles) vers des objets de C2.

À partir de ces paramètres, nous pouvons calculer le nombre de références distinctes des objets de C1 ayant au moins une référence non nulle vers des objets de C2 par la formule :

$$Z_{C1,C2} = \frac{D_{C1,C2} * ||C1||}{||C1|| - X_{C1,C2}}$$

De plus, nous supposons connus les paramètres suivants, dépendant essentiellement de choix d'implémentation du système :

– b le degré moyen des arbres-B utilisés (par exemple 256 clés par page),
– BLevel(I) le nombre de niveaux d'un index I (souvent 3),
– S_p la taille d'une page (par exemple 4K),
– Proj_Cost le coût moyen d'extraction d'un attribut d'un objet,
– m la taille mémoire disponible en page,
– Cost_load_page le coût de chargement d'une page en mémoire (une E/S).

Tous ces paramètres sont donc utilisés par le modèle de coût détaillé ci-dessous.

6.2. ESTIMATION DU NOMBRE DE PAGES D'UNE COLLECTION GROUPÉE

Lorsque plusieurs collections sont groupées ensemble par des associations, il n'est pas évident de déterminer le nombre de pages à balayer pour trouver les objets dans l'une d'elles. Si une collection A n'est pas groupée avec d'autres collection, il y a $\left\lfloor \frac{S_p}{S_A} \right\rfloor$ objets au plus dans une page et le nombre total de pages après chargement séquentiel peut être calculé comme suit :

$$|A| = \frac{||A||}{\left\lfloor \frac{S_p}{S_A} \right\rfloor}$$

Dans le cas de plusieurs collections groupées, certaines pages peuvent être homogènes, d'autres contenir des objets de différentes collections. Une méthode pour estimer le nombre total de pages d'une collection groupée avec d'autres a été proposée dans [Tang96]. Supposons que la collection B soit groupée avec la collection A (voir figure XIV.17). Rappelons que nous notons respectivement S_A et S_B les tailles moyennes des objets des collections A et B, et que S_A et S_B sont supposées inférieures à la taille de la page. Estimons tout d'abord la taille de la collection A qui est la racine de l'arbre de groupage. Il existe deux partitions physiques pour A après le

placement des deux collections, l'une contenant des objets groupés avec des objets de B notée $Cl_{A\rightarrow B}$, l'autre ne contenant que des objets de A notée Cl_A.

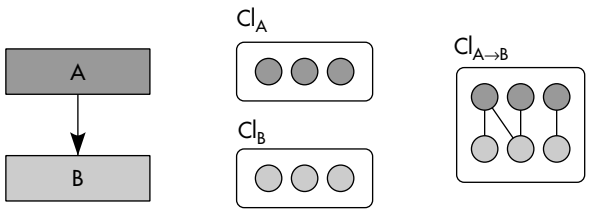


Figure XIV.17 : Différentes partitions dans le cas d'un groupage de B avec A

La partition Cl_A contient les objets A ne référençant pas d'objets B, donc $||Cl_A||=X_{A,B}$, d'où l'on déduit la taille de cette partition :

$$|A|_{Cl_A} = \frac{X_{A,B}}{\left\lfloor \frac{S_p}{S_A} \right\rfloor}$$

Pour $Cl_{A\rightarrow B}$, le nombre d'objets de A dans cette partition est $||A||-X_{A,B}$. La taille d'un groupe autour d'un objet A est $S_{ClA\rightarrow B} = S_A+(Z_{A,B}*S_B)$, d'où l'on déduit :

$$|A|_{Cl_{A\rightarrow B}} = \begin{cases} \dfrac{A - XA,B}{\left\lfloor \dfrac{S_p}{S_{Cl_{A\rightarrow B}}} \right\rfloor} & \text{si } S_{Cl_{A\rightarrow B}} < S_p \\ ||A|| - X_{A,B} \text{ si } S_{Cl_{A\rightarrow B}} \geq S_p \end{cases}$$

Le nombre total de pages de la collection A est donc donné par :

$$|A| = |A|_{ClA} + |A|_{ClA\rightarrow B}$$

D'une manière similaire, nous pouvons estimer le nombre de pages de la collection B, une collection non racine dans le graphe de placement. Les différentes partitions composant la collection B sont notées $Cl_{A\rightarrow B}$ et Cl_B.

– Pour Cl_B, le nombre d'objets de B non référencés par un quelconque objet de A est $||B||-(||A||*D_{A,B})$, d'où nous déduisons :

$$|B|_{Cl_B} = \frac{||B|| - (||A|| * D_{A,B})}{\left\lfloor \frac{S_p}{S_A} \right\rfloor}$$

– Pour $Cl_{A\rightarrow B}$, la taille et le nombre d'objets par groupe reste le même que celui calculé ci-dessus, mais le nombre de groupes auxquels accéder est maintenant $X'_{ClA\rightarrow B}= Min(Z_{A,B}*X_{A,B}, X_{A,B})$. Nous obtenons donc :

$$|B|_{Cl_{A\rightarrow B}} = \begin{cases} |A|_{Cl_{A\rightarrow B}} & \text{si } S_{Cl_{A\rightarrow B}} < S_p \\ & \text{sinon} \\ \|A\| - X_{A,B} & \text{si } Z_{A,B} * S_B \leq S_p \\ (\|A\| - X_{A,B}) * \dfrac{Z_{A,B} * S_B}{S_p} & \text{si } Z_{A,B} * S_B > S_p \end{cases}$$

Le nombre total de page de la collection B est obtenu par :

$|B| = |B|_{ClA} + |B|_{ClA\rightarrow B}$.

Ainsi, en étudiant la taille des différentes partitions, il est possible de calculer la taille globale en nombre de pages d'une collection groupée.

6.3. FORMULE DE YAO ET EXTENSION AUX COLLECTIONS GROUPÉES

Le modèle de coût doit permettre de calculer le nombre de pages auxquelles accéder lors d'une sélection d'une collection par un prédicat. Dans le cas de bases de données relationnelles, la **formule de Yao** [Yao77] permet de calculer le nombre de blocs contenant au moins un enregistrement lors d'une sélection de k enregistrements parmi n.

Notion XIV.9 : Formule de Yao (*Yao Formula*)

Formule permettant de calculer le nombre de blocs auxquels accéder lors d'une sélection de k enregistrements parmi n (k≤n) uniformément distribués dans m blocs (1<m≤n) comme suit :

$$yao(n,m,k) = m * \left[1 - \prod_{i=1}^{k} \frac{nd - i + 1}{n - i + 1} \right] \quad avec\ d = 1 - 1/m$$

Dans le cas d'une collection C groupée avec d'autres, il n'est pas possible d'appliquer simplement la formule de Yao pour estimer le nombre de pages auxquelles accédées lors d'une sélection par un prédicat P par Yao(||C||,|C|,Sel(P)*||C||). Le problème est qu'une collection groupée est composée de plusieurs partitions. Certains objets sont groupés avec des objets de collections différentes. Ainsi, la distribution des objets dans les pages n'est pas uniforme, bien que les collections soient supposées indépendantes d'un point de vue probabiliste. Il est possible d'appliquer la formule de Yao à

chaque partition, après avoir déterminé la taille de chaque partition comme vu ci-dessus, soit $||C||_i$ objets et $|C|_i$ pages pour la partition i. Nous obtenons alors la **formule de Tang** pour le calcul des entrée-sorties [Tang96].

Notion XIV.10 : Formule de Tang (*Tang Formula*)

Étant donné une collection C divisée en p partitions, chacune ayant $||C||_i$ objets et $||C||_i$ pages, la sélection aléatoire de k objets de C nécessite l'accès à :

$$Tan\ g(C,k) = \sum_{i=1}^{p} yao(||C_i||, |C_i|, k_i)$$

où k_i est le nombre d'objets sélectionnés dans la partition C_i.

Si les objets à sélectionner sont placés uniformément dans les partitions, on sélectionne $k_i = (||C_i||/||C||)*k$ dans la partition i. Quand les objets qui satisfont le prédicat de recherche ne sont pas placés uniformément, il faut calculer la sélectivité du prédicat dans chaque partition pour calculer k_i. Bien que dérivée de la formule de Yao, la formule de Tang est plus générale que celle de Yao, qui en est un cas particulier pour une collection avec une seule partition.

6.4. COÛT D'INVOCATION DE MÉTHODES

Avec l'approche objet, les attributs peuvent être publics ou privés. Les attributs publics sont directement accessibles et le coût d'extraction peut être assimilé à une constante. Il peut aussi être plus précisément déterminé en fonction du type de l'attribut et de sa longueur en octets.

Les attributs privés sont manipulés par des méthodes. Déterminer le coût d'une méthode n'est pas chose simple. Deux approches ont été proposées pour cela.

Notion XIV.11 : Révélation de coût (*Cost révélation*)

Technique consistant à fournir une méthode de calcul de coût associée à chaque opération d'une classe, avec des paramètres identiques ou simplifiés.

Par exemple, si une classe Géométrie publie une fonction distance(géométrie), elle pourra aussi publier une fonction coût_distance(géométrie). Le coût pourra être très différent selon les géométries sur lesquelles s'applique la fonction : il est par exemple beaucoup plus simple de calculer la distance de deux points que celle de deux polygones.

Notion XIV.12 : Moyennage de coût (*Cost averaging*)

Technique consistant à mémoriser le coût moyen d'application d'une méthode au fur et à mesure de son utilisation.

Au départ, le coût de la méthode est inconnu, et estimé par exemple comme une projection sur attribut. Puis, suite aux requêtes ou à des demandes d'estimation de l'administrateur (par une requête spécifique `ANALYZE <méthode>`), la méthode est exécutée sur un échantillon de N objets. Une table est gérée dans la méta-base pour maintenir le coût moyen observé, de schéma simplifié `COSTMETHOD (coût real, nombre int)`, `coût` étant le coût moyen et `nombre` le nombre d'exécutions observées. La table peut aussi mémoriser le type des paramètres, les heures d'appels, etc., afin d'obtenir des statistiques plus fines.

6.5. COÛT DE NAVIGATION VIA EXPRESSION DE CHEMIN

Ce coût dépend évidemment de l'algorithme de traversé de chemin utilisé. Nous analysons ici le cas où une expression de chemin avec prédicats est évaluée en utilisant l'algorithme en profondeur d'abord (DFF) vu ci-dessus. Soit une expression de chemin entre des collections C_1, C_2…C_n qualifiée par des prédicats P_1, P_2…P_n à chaque niveau. Une telle expression est notée C_1 (P_1).C_2 (P_2)… C_n(P_n). Désignons par A_i l'attribut de C_{i-1} référençant un ou plusieurs objets de la collection C_i. Soit S_i la sélectivité du prédicat P_i (S_i = Sel(P_i)). L'algorithme DFF accède successivement, pour chaque objet de la collection de niveau i satisfaisant le prédicat P_i, à l'attribut pointant sur les objets de la collection (i+1). Suivant ces liens, il lit les pages successives des collections en profondeur d'abord. Pour simplifier, nous supposons qu'il existe au moins une page disponible en mémoire par collection.

Calculons tout d'abord le nombre de références Ref_i distinctes moyennes pointant d'un objet de la collection C_{i-1} vers un objet de C_i dans le chemin, soit $Ref_i = (1-Prob_i)*||C_i||$. $Prob_i$ est la probabilité qu'un objet de la collection Ci ne soit pas pris en compte dans le chemin, probabilité calculée par la formule :

$$Prob_i = \left(1 - \frac{1}{||C_i||}\right)^{(\mathrm{Ref}_{i-1} * S_{i-1} * \mathrm{fan}_{c_{i-1}, c_i})}$$

Selon la taille mémoire disponible, trois cas sont possibles :

– Si la mémoire disponible p est assez grande pour contenir toutes les pages du chemin, chaque page est chargée une fois et une seule. Selon que le groupage est effectué selon la référence ou non, on effectue l'accès à Ref_i objet de la collection en 0 ou en Tang(C_i,Ref_i) entrée-sorties, d'où l'on calcule le nombre de pages lues :

$$Page_{\min} = \sum_{i=2}^{n} \begin{cases} 0 \textit{ si } C_{i-1} \textit{ est groupée avec } C_i \\ Tang(\mathrm{Ci}, \mathrm{Ref}_i) \textit{ sinon} \end{cases}$$

– Si le nombre de pages disponibles en mémoire p est exactement la taille du chemin (p = n), nous obtenons le cas limite très défavorable :

$$Page_{max} = \sum_{i=2}^{n} \begin{cases} 0 \text{ si } C_{i-1} \text{ est groupée avec } C_i \\ \|C_1\| * \prod_{j=2} fan_{C_{j-1}, C_j} * S_{j-1} \text{ sinon} \end{cases}$$

– Si le nombre de pages disponibles en mémoire p est compris entre *n* et $Page_{min}$, la mémoire ne peut contenir toutes les pages nécessaires à l'évaluation de l'expression de chemin. Le coût d'entrées-sorties dépend de la politique de remplacement des pages du SGBD. Certaines pages devront être lues plusieurs fois. Il est possible d'approcher le nombre d'entrées-sorties par une fonction linéaire entre $Page_{min}$ et $Page_{max}$, comme suit :

$$NbPage = \frac{Page_{max} - Page_{min}}{Page_{min} - n} * (n - p) + Page_{max}$$

Finalement, le coût d'évaluation d'un prédicat P constitué d'une expression de chemin qualifiée avec p pages disponibles en mémoire centrale est :

$$IO\_Eval\_Cost(P) = \begin{cases} NbPage \text{ if } n \le p < Page_{min} \\ Page_{min} \text{ if } p \ge Page_{min} \end{cases}$$

Le coût de calcul associé à l'évaluation d'un tel prédicat est le coût de projection plus le coût d'évaluation des prédicats élémentaires pour chaque objet traversé, soit :

$$CPU\_Eval\_Cost(P) = \|C_1\| * (CPU\_\text{Pr}oj\_Cost + CPU\_Comp\_Cost) * \left(\sum_{i=2}^{n} \prod_{i=2}^{i} fan_{C_{j-1}, C_J} * S_{j-1)} \right)$$

6.6. COÛT DE JOINTURE

En dehors des parcours de références évalués ci-dessus pour l'algorithme DFF, les algorithmes de jointures sont analogues à ceux des bases relationnelles. Vous pouvez donc vous reporter au chapitre concernant l'optimisation de requêtes dans les BD relationnelles pour trouver les formules de coût applicables. Des formules plus détaillées pourront être trouvées dans [Harris96]. Plus généralement, en dehors du groupage des objets, des parcours de références et des index de chemins, les BD objet n'ont guère d'autres spécificités d'implémentation.

7. STRATEGIES DE RECHERCHE DU MEILLEUR PLAN

La recherche du meilleur plan d'exécution dans une base de données objet ou objet-relationnel est encore plus complexe que dans une base relationnelle. En effet, l'espace des plans est plus large du fait de la présence de références, d'index de chemins, de collections imbriquées, etc. Au-delà de la programmation dynamique classique limitée à des espaces de recherche de faible taille [Swami88, Swami89, Vance96] vue dans le cadre relationnel, des méthodes d'optimisation pour la recherche de plans optimaux basée sur des algorithmes aléatoires ont été proposées. Après les avoir présentés, nous proposons une méthode plus avancée basée sur des algorithmes génétiques.

7.1. LES ALGORITHMES COMBINATOIRES

Les algorithmes combinatoires accomplissent une marche au hasard dans l'espace de recherche sous la forme d'une suite de déplacement. Un déplacement dans cet espace est généré par application d'une règle de transformation sur le plan d'exécution. Un déplacement est appelé descendant s'il diminue le coût du plan, montant s'il l'augmente. Un état (donc un plan) est un minimum local si tout déplacement unitaire augmente son coût. Le minimum global est celui parmi les minimums locaux qui a le plus faible coût. Cette classe d'algorithmes inclut l'**amélioration itérative** (*Iterative Improvement* [Nahar86])**,** le **recuit simulé** (*Simulated Annealing* [Ioannidis87]), l'**optimisation deux phases** (*Two Phase Optimization* [Ioannidis90]) et la **recherche taboue** (*Tabu Search* [Glover89, Glover90]). Nous présentons en détail ces méthodes ci-dessous.

7.2. L'AMÉLIORATION ITÉRATIVE (II)

L'amélioration itérative commence par un plan d'exécution initial choisi plus ou moins aléatoirement, par exemple celui résultant de la compilation directe de la requête utilisateur. Ensuite, seuls les déplacements descendants sont acceptés. C'est l'optimisation locale. Quand le minimum local est atteint, l'algorithme génère au hasard un nouveau plan, et recommence l'optimisation locale à partir de ce nouveau plan. Le processus est répété jusqu'à atteindre une condition d'arrêt, par exemple l'épuisement du temps d'optimisation. Le minimum global est le meilleur minimum local trouvé lors de l'arrêt. L'algorithme est donné en pseudo-code figure XIV.18.

Notion XIV.13 : Amélioration itérative (*Itérative improvment*)

Technique d'optimisation consistant à choisir aléatoirement des plans dans l'espace de recherche, à les optimiser localement au maximum, et enfin à retenir le meilleur des minima locaux comme plan optimal.

```
Procedure II(Question) {
   p = Initial(Question); // générer un plan intial
   PlanOptimal = p; // Initialiser le plan optimal
   while not(condition_arrêt) do { // Boucle d'optimisation globale
      while not (condition_locale) do { // Boucle d'optimisation locale
         p' = déplacer(p); // Appliquer une transformation valide à p
         if (Coût(p')<Coût(p)) then p = p'; // Garder si moins coûteux
         }
      // Sélectionner le plan optimal
      if coût(p)<coût(PlanOptimal) then PlanOptimal = p;
      p = Random(p) ; // Se déplacer au hasard vers un autre plan
      }
   Return(PlanOptimal);
   }
```

Figure XIV.18 : Algorithme d'amélioration itérative

7.3. LE RECUIT SIMULÉ (SA)

Le recuit simulé part également d'un plan initial choisi plus ou moins aléatoirement et génère des plans successifs par application de règles de transformation valide de ce plan. Á la différence de l'amélioration itérative, le recuit simuler accepte à la fois des déplacements descendants et montants. L'idée originelle est de simuler le refroidissement d'un métal recuit : tant que le métal est chaud, les mouvements de molécules sont nombreux, puis ils décroissent comme la température. Donc, quand le système est chaud, les déplacements montants qui détériorent le coût du plan sont admis afin de permettre une exploration plus large de l'espace autour du plan en cours d'analyse, pour découvrir d'autres minima locaux. Ces déplacements sont permis avec une probabilité qui décroît avec la température : $P = e^{-\text{DeltaCoût/Température}}$. Le paramètre `Température` représente la température du système. Il décroît au fur et à mesure de l'avancement de l'optimisation, si bien que les mouvements ascendants sont acceptés avec une probabilité de plus en plus faible. `DeltaCoût` est la différence de coût entre les deux plans. Quand la condition d'arrêt est atteinte (temps épuisé), le meilleur plan traversé est sélectionné et retenu comme pseudo-optimal. La figure XIV.19 donne la procédure correspondante en pseudo-code.

Notion XIV.14 : Recuit simulé (*Simulated Annealing*)

Technique d'optimisation consistant à choisir aléatoirement un plan dans l'espace de recherche et à explorer alentour par des déplacements descendants ou ascendants, ces derniers étant acceptés avec une probabilité qui tend vers 0 quand la température tend vers 0.

```
Procédure SA(Question) {
   p = Initial(Question); // générer un plan intial
   PlanOptimal = p; // Initialiser le plan optimal
   T = T0; // Initialiser la temperature
   while not(condition_arrêt) do { // Boucle d'optimisation globale
      while not(equilibrium) do { // Boucle d'optimisation locale
         p' = déplacer(p); // Appliquer une transformation valide à p
         delta = cost(p')-cost(p); // Calculer la différence de coût
         if (delta<0) then p = p'; // Si coût réduit prendre le nouveau plan
         // Si coût accru, accepté si chaud
         if (delta>0) then p = p' with probability e^- delta/T;
         // Maintenir le plan optimal
         if cost(p)<cost(PlanOptimal) then PlanOptimal = p;
         }
      T = reduce(T); // Reduire la température
      }
   Return(PlanOptimal); }
```

Figure XIV.19 : Algorithme de recuit simulé

7.4. L'OPTIMISATION DEUX PHASES (TP)

L'amélioration itérative travaille efficacement pour couvrir de grands espaces et trouver rapidement des minima locaux. Elle ne garantit guère une couverture fine autour de ces minima. Le recuit simulé est bien adapté pour explorer autour d'un minimum local. D'où l'idée de combiner les deux [Ioannidis90], ce qui conduit à l'optimisation deux phases. Tout d'abord, une phase amélioration itérative (II) est exécutée pour trouver les états initiaux de la phase recuit simulé (SA) suivante. Ainsi, l'optimisation deux phases travaille comme suit [Steinbrunn97] :

1. Pour un petit nombre de plans aléatoirement sélectionnés, des minima locaux sont cherchés en appliquant II, puis
2. À partir du (ou des) moins coûteux de ces minima locaux, SA est appliqué afin d'explorer le voisinage pour trouver de meilleures solutions.

Cette méthode mixte peut donner de bons résultats à condition de bien régler les différents paramètres (temps de chaque phase, température).

7.5. LA RECHERCHE TABOUE (TS)

La recherche taboue (TS) est une procédure du type méta-heuristique très efficace pour l'optimisation globale [Glover89]. L'idée principale est de faire à chaque pas le meilleur déplacement possible, tout en évitant une liste de plans tabous. L'utilisation d'une liste de tabous permet d'éviter des recherches inutiles et de converger plus vite vers des solutions proches de l'optimal. TS part d'un plan initial généré aléatoirement, et accomplit les meilleurs mouvements non interdits successivement. À chaque itération, la procédure génère un sous-ensemble V* de l'ensemble des voisins non interdits du plan courant. Le sous-ensemble ne contient donc pas de plans de la liste des tabous. Les plans déjà explorés sont généralement retenus, au moins pour un temps, dans la liste des tabous, ce qui évite en général de boucler. La liste des tabous est mise à jour chaque fois qu'un nouveau plan est exploré pour mémoriser les plans déjà vus. La figure XIV.20 donne le pseudo-code correspondant à cet algorithme.

Notion XIV.15 : Recherche taboue (*Tabou search*)

Technique d'optimisation consistant à se déplacer alentour vers le plan de coût minimal tout en évitant une liste de plans interdits dynamiquement mise à jour.

```
Procedure TS(Question) {
   p = Initial(Question); // générer un plan intial
   PlanOptimal = p; // Initialiser le plan optimal
   T = ϕ; // initialiser la liste taboue
   while not(condition_arrêt) do { // boucle globale
      // accepter tous les mouvements non tabous
      générer V*⊆N(p)-T en appliquant déplacer(p);
      choisir le meilleur plan p ∈ V*; // Prendre le meilleur plan
      T= (T-(plus vieux))∪{p}; // Mettre à jour la taboue liste
      // maintenir le plan optimal
      if coût(p)<coût(PlanOptimal) then PlanOptimal = p;
      }
   return(PlanOptimal);
   }
```

Figure XIV.20 : Algorithme de recherche taboue

7.6. ANALYSE INFORMELLE DE CES ALGORITHMES

La performance de tous les algorithmes précédents dépend fortement de la distribution de la fonction de coût sur l'espace de recherche. Alors que le recuit simulé (SA) et la recherche taboue (TS) dépendent fortement du plan initial, l'amélioration itéra-

tive (II) et l'optimisation deux phases (TP) utilisent une transformation aléatoire pour explorer cet espace. Pour améliorer les performances de ces algorithmes, des méthodes de transformation d'arbres introduisant une meilleure couverture de l'espace de recherche ont été mises au point. Ce sont par exemple l'échange de jointures et le *swap* [Swami89, Lanzelotte93]. L'échange de jointure permute aléatoirement des relations alors que le swap échange des parties d'arbres. De telles méthodes ont été introduites seulement pour les jointures relationnelles. Il est possible de les étendre au cas objet.

Suite aux mouvements ascendants, SA est généralement plus lent que les autres méthodes mais trouve souvent de meilleurs plans que II quand le temps de recherche est suffisant. TP combine les avantages de SA et II. Il a été montré [Ioannidis90] que TP trouve en général de meilleurs plans que SA et II. TS est rapide mais accomplit une recherche agressive. En conséquence, l'algorithme peut rester bloqué sur un minimum local. Ceci peut être évité avec une longue liste taboue, mais la recherche ralentit alors l'algorithme.

II, SA, TP et TS utilisent tous une fonction appelée `déplacer`. Celle-ci applique l'une des règles de transformation pour trouver un plan équivalent. Comme vu ci-dessus, dans une base de données objet ou objet-relationnel avec beaucoup de types, le nombre de règles peut être important. Certaines ne provoquent que de petits déplacements dans l'espace des plans, d'autres de beaucoup plus larges. Il faudrait donc pouvoir introduire des transformations permettant d'appliquer des séquences de règles, et donc de faire de grands sauts dans l'espace des plans. Nous allons étudier une méthode permettant de telles transformations dans la section suivante.

8. UN ALGORITHME D'OPTIMISATION GÉNÉTIQUE

Dans cette section, nous étudions l'approche génétique et son application possible à l'optimisation de requêtes [Bennett91]. Pour l'illustrer concrètement, nous étudions le cas de longues expressions de chemins avec prédicats. Les expressions de chemins sont importantes dans les BD objets ; certains SGBD se concentrent d'ailleurs sur leur optimisation et ne proposent guère d'autres optimisations.

8.1. PRINCIPE D'UN ALGORITHME GÉNÉTIQUE

L'**algorithme génétique** (GA) est un algorithme d'optimisation basé sur les principes d'évolution des organismes dans la nature : l'algorithme cherche à générer les

meilleurs éléments d'une population en combinant les meilleurs gènes ensemble. C'est une famille d'algorithmes utilisés dans la recherche d'extrema (minimum ou maximum) de fonctions à plusieurs variables [Holland75, Goldberg89]. Ces fonctions sont généralement définies sur des domaines discrets vastes et complexes. Dans notre cas, il s'agit de retrouver idéalement le minimum global de la fonction de coût dans l'espace des plans. Au lieu de travailler sur une solution particulière à chaque étape, l'algorithme considère une population de solutions. Diverses sortes d'algorithmes génétiques ont été proposées pour différentes tâches d'optimisation. La figure XIV.21 représente un algorithme type utilisable pour la recherche du meilleur plan.

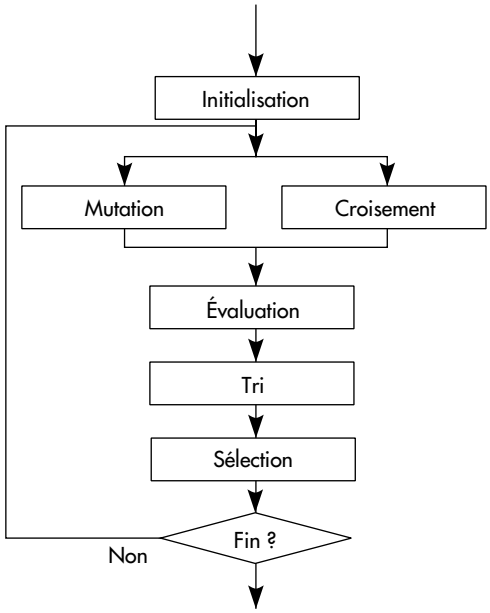


Figure XIV.21 : Algorithme génétique type

Voici les fonctions des principaux blocs d'un tel algorithme :

– **Initialisation** génère une petite population initiale de solutions (dans notre cas, des plans d'exécution) couvant l'espace de recherche.
– **Mutation** choisit une solution (c'est-à-dire un plan) de la population, et lui applique des règles de transformation.
– **Croisement** choisit deux solutions dans la population et échange leurs gènes communs (dans notre cas, des sous-arbres équivalents) pour générer deux nouvelles solutions.
– **Évaluation** évalue, pour chaque solution, la fonction à optimiser, dans notre cas la fonction de coût.

– **Tri** trie la population en fonction du coût.
– **Sélection** choisit un certain nombre des meilleurs éléments comme parents pour la prochaine génération.
– **Fin** test le critère d'arrêt afin de stopper l'optimisation.

Le cœur de l'algorithme est la génération d'une nouvelle population par les opérateurs de mutation et de croisement. À chaque génération, une partie de la population subit la mutation et l'autre le croisement. La mutation est identique au déplacement des algorithmes vus ci-dessus. Ce qui est nouveau, c'est le croisement qui permet de construire une solution à partir de deux solutions. Avec la sélection naturelle, il est permis de penser que le croisement ne laissera se propager que les bons gènes, pour converger vers les bons plans. En résumé, la notion d'algorithme génétique peut être définie comme suit :

Notion XIV.16 : Algorithme génétique (*Genetic algorithm*)

Algorithme de recherche d'optimum consistant à partir d'une population et à la faire se reproduire par mutation et croisement, en sélectionnant les meilleurs éléments à chaque génération, de sorte à converger vers une population contenant la solution optimale ou presque.

8.2. BASES DE GÈNES ET POPULATION INITIALE

Nous appliquons maintenant l'algorithme génétique à l'optimisation des expressions de chemins. Comme vu ci-dessus, différentes méthodes de jointures peuvent être appliquées : la jointure n-aire DFF en profondeur d'abord, la jointure binaire en largeur d'abord notée FJ (plusieurs jointures peuvent être exécutées successivement, ce qui donne l'algorithme BFF), la jointure en arrière notée RJ, et la traversée d'un index de chemins lorsqu'il en existe un, notée PI. Touts ces méthodes apparaissent comme des gènes permettant de générer la population des plans. La figure XIV.22 représente la base de gènes avec cinq collections liées par une chaîne d'association. S'il n'y a ni lien inverse ni index de chemin, le nombre total de gènes est le nombre de cellules dans le triangle supérieur de la figure XIV.22 généralisée à n, soit :

$$Genes = \sum 1 + 2 + ... + (n-1) + 2 * (n-1) = (n^2 + 3 * n - 2)/2$$

Dans tous les cas, le nombre de gènes est de l'ordre de $O(n^2)$. Un plan d'exécution peut être vu comme une combinaison de gènes dont le résultat permet le parcours du chemin, dans un sens ou dans l'autre, ceci afin de trouver le résultat de la requête. L'objectif de l'optimiseur est de trouver la meilleure combinaison de gènes pour obtenir le plan de coût minimal.

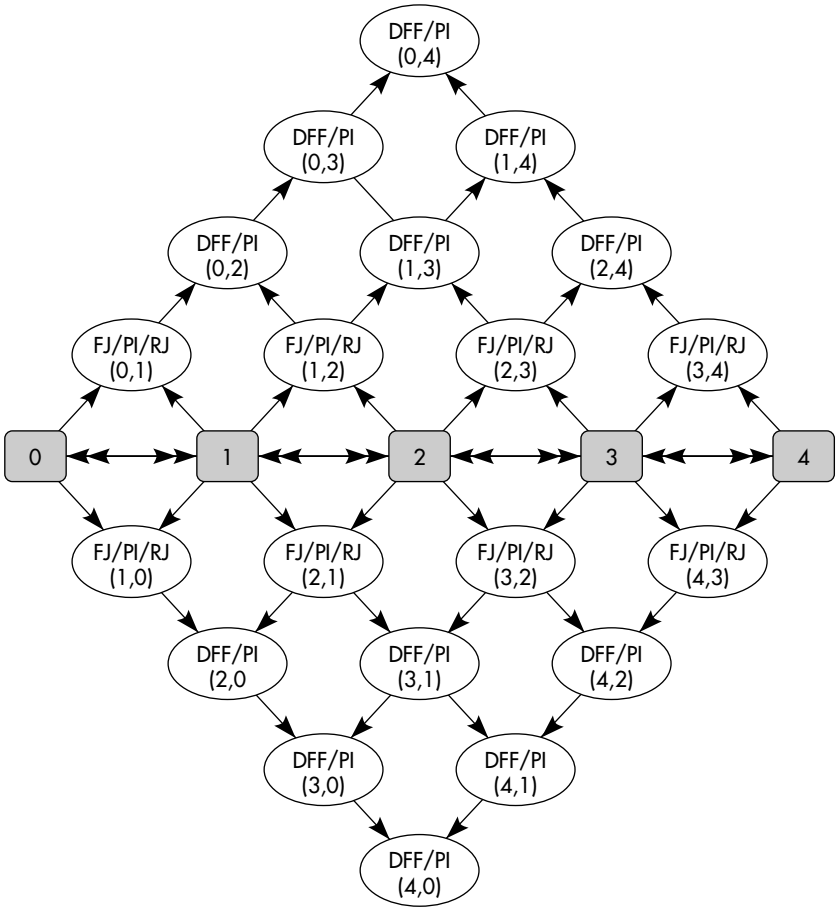


Figure XIV.22 : Base de gènes pour cinq collections

L'étape d'initialisation consiste à générer aléatoirement une population initiale de plans. Notons qu'une telle population est générée par des parcours aléatoires dans la base de gènes de la figure XIV.22, depuis la collection source jusqu'à la collection cible. De tels parcours évitent les produits cartésiens, ce qui est généralement sage.

8.3. OPÉRATEUR DE MUTATION

La mutation est un des opérateurs importants d'une méthode génétique. À chaque étape, un certain pourcentage des plans doit subir une mutation. Cette procédure consiste simplement à sélectionner un sous-arbre et à lui appliquer une règle de transformation valide. Quelques exemples de mutation possibles sont représentés figure XIV.23. La mutation apporte de nouveaux gènes qui peuvent ne pas exister dans la population courante.

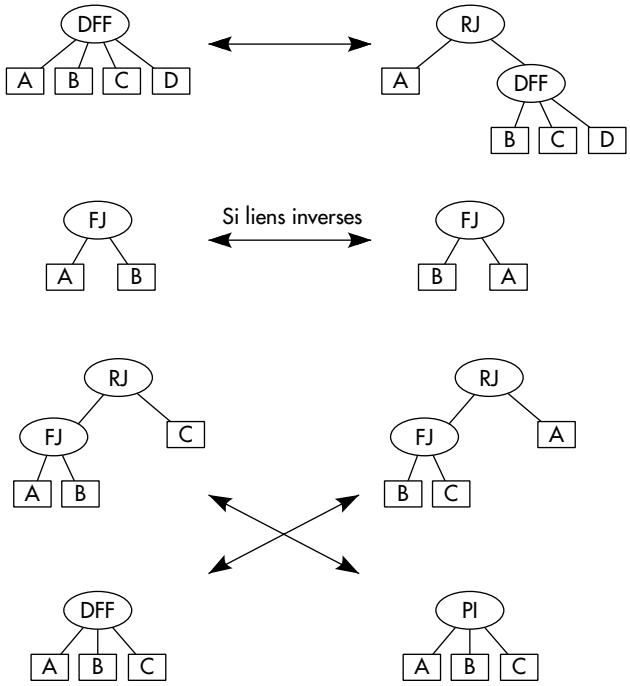


Figure XIV.23 : Exemples de mutations possibles

8.4. OPÉRATEUR DE CROISEMENT

Cet opérateur consiste à mixer deux éléments de la population pris plus ou moins au hasard. Par exemple, deux plans sont choisis et leurs sous-arbres communs sont échangés. Deux sous-arbres sont équivalents s'ils ont les mêmes feuilles et génèrent les mêmes résultats. La figure XIV.24 illustre un croisement. Alors que la mutation transforme un plan en l'un de ses voisins, en accomplissant de petits mouvements, le croisement peut accomplir de plus larges transformations. Il permet ainsi de sortir de zone voisine d'un minimum local, en sautant vers une autre partie de l'espace de recherche.

La figure XIV.25 donne le pseudo-code de l'algorithme génétique appliqué à l'optimisation des expressions de chemins. Nous utilisons le tableau `Popu` pour représenter la population. Chaque élément contient deux variables : un plan et son coût. `BasePopu` contient les plans subissant le croisement ou la mutation. Comme la population s'accroît après le croisement, la procédure de sélection est appliquée avant de passer à la nouvelle génération. Celle-ci conserve les plans de coûts minimaux. Pendant le croisement, le plus grand sous-arbre commun des deux parents est choisi. Si deux parents n'ont pas de sous-arbre commun, ils sont soumis à la mutation. Les plans résultants du croisement sont placés de `BasePopu + 1` à `NewPopu`. `Part` est le pourcentage de la population effectuant un croisement.

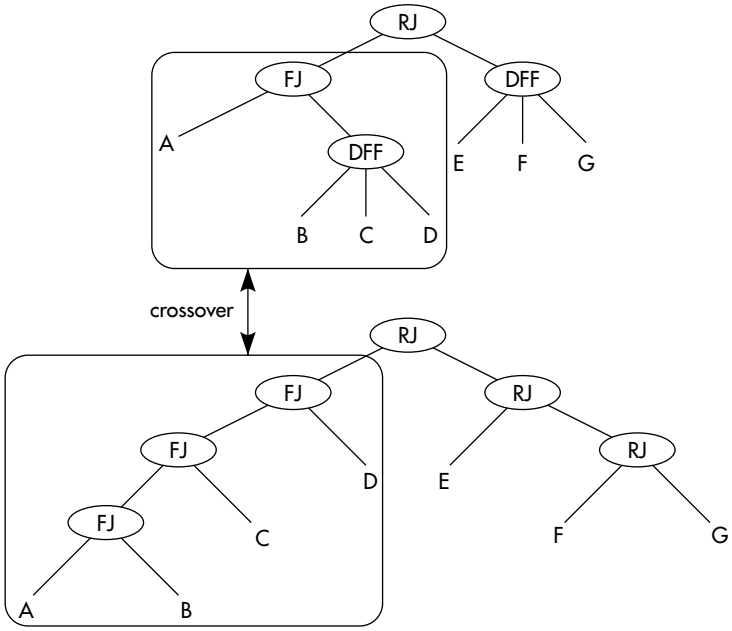


Figure XIV.24 : Opérateur de croisement

```
Procedure GA(Question) {
   Générer(Popu[BasePopu]) ; // Initialiser la population aléatoirement
   Sort(Popu) ; // Trier la population des plans par coût croissant
   PlanOptimal = Popu[0] ; // Garder le meilleur plan trouvé
   While not (condition_arrêt) do {
      Pourcent = 0;
      While Pourcent < Part * BasePopu do {
         // Appliquer croisement à Part de la population
         p1 = Popu[Random(BasePopu)] ; // Choisir p1 et p2 de Popu
         p2 = Popu[Random(BasePopu)] ; // aléatoirement
         Croisement(p1, p2) ; // Les croiser si possible
         Pourcent = Pourcent + 2 ;
         }
      For le reste de Popu do Mutation ; // Mutation pour les autres
      For (i=0 ; i < NewPopu ; i++) do evaluate(Popu[i]) ; // Calcul du coût
      Trier(Popu) ; // Trier la population par coût croissant
      PlanOptimal = Popu[0] ; // Garder le meilleur plan trouvé
      }
   return(PlanOptimal) ;
   }
```

Figure XIV.25 : Une implémentation de l'algorithme génétique

Des expériences effectuées avec cet algorithme ont montré qu'il pouvait être amélioré en ajoutant des plans aléatoirement générés à chaque génération, ceci afin de garantir la diversité des plans explorés [Tang96]. Les limites de tels algorithmes d'optimisation appliqués aux bases de données, notamment dans le cas d'un très grand nombre de règles, restent encore à découvrir.

9. CONCLUSION

Dans ce chapitre, nous avons étudié différentes techniques d'optimisation pour les SGBD objet. Tout d'abord, les techniques de groupage des objets sur disques permettent de placer à proximité les objets souvent accédés ensemble, par exemple via des parcours de chemins. Elles prolongent les techniques de groupage des relations par jointures pré-calculées de certains SGBD relationnels. Au-delà, les index de chemins constituent aussi une spécificité des BD objet ; ils peuvent être perçus comme une extension des index de jointure des BD relationnelles.

Ensuite, nous avons étudié différents algorithmes de parcours de chemins. Ces algorithmes permettent la navigation ensembliste dans les BD objet. Ils sont des extensions naturelles des algorithmes de jointures [Valduriez87], qu'ils étendent en utilisant des identifiants d'objets.

La génération de plans équivalents est plus complexe que dans le cas relationnel, surtout par la nécessité de prendre en compte les types de données utilisateurs et les réécritures d'expressions de méthodes associées. Ceci nécessite le développement d'optimiseurs extensibles, supportant l'ajout de règles de réécriture. Ce sont de véritables systèmes experts en optimisation, qui sont maintenant au cœur des SGBD objet-relationnel.

Le développement d'un modèle de coût pour bases de données objet est un problème des plus difficiles. Il faut prendre en compte de nombreuses statistiques, les groupes, les références, les index de chemins, les méthodes et les différentes formes de collections. Nous avons présenté un modèle de coût simple élaboré par extension des modèles classiques des BD relationnels. Peu d'optimiseurs prennent en compte un modèle de coût intégrant les spécificités de l'objet, par exemple les méthodes. Beaucoup reste à faire.

Nous avons enfin développé des stratégies de recherche sophistiquées pour trouver le meilleur plan d'exécution. Beaucoup ont été proposées et paraissent suffisantes pour des questions mettant en jeu une dizaine de collections. Au-delà, les stratégies génétiques semblent prometteuses. Peu de SGBD utilisent aujourd'hui ces stratégies, la plupart restant basés sur la programmation dynamique. L'intégration dans des systèmes pratiques reste donc à faire.

10. BIBLIOGRAPHIE

[Amsaleg93] Amsaleg L., Gruber O., « Object Grouping in EOS », *Distributed Object Management*, Morgan Kaufman Pub., San Mateo, CA, p. 117-131, 1993.

EOS est un gérant d'objets à un seul niveau de mémoire réalisé à l'INRIA de 1990 à 1995. Une technique originale de groupement d'objets avec priorité a été proposée, ainsi que des techniques de ramassage de miettes.

[Bertino89] Bertino E., Kim W., « Indexing Techniques for Queries on Nested Objects », *IEEE Transactions on Knowledge and Data Engineering*, vol. 1(2), June 1989, p. 196-214.

Cet article compare différentes techniques d'indexation pour les bases de données objet, en particulier plusieurs techniques d'index de chemins et d'index séparés ou groupés (multi-index) dans le cas de hiérarchie de généralisation.

[Bertino91] Bertino E., « An Indexing Technique for Object-Oriented Databases », *Proceedings of Int. Conf. Data Engineering*, Kobe, Japan, p. 160-170, April 1991.

Cet article développe le précédent et propose des techniques de recherche et mise à jour avec index de chemin.

[Bertino92] Bertino E., Foscoli P., « An Analytical Model of Object-Oriented Query Costs », *Persistent Object Systems*, Workshop in Computing Series, Springer-Verlag, 1992.

Les auteurs développent un modèle de coût analytique pour BD objet. Celui-ci prend notamment en compte les index de chemin.

[Bennett91] Bennett K., Ferris M.C., Ioannidis Y.E., « A Genetic Algorithm for Database Query Optimization », *Proc. 4th International Conference on Genetic Algorithms*, San Diego, CA, p. 400-407, June 1991.

Cet article a proposé pour la première fois l'utilisation d'un algorithme génétique dans les bases de données objet.

[Christophides96] Christophides V., Cluet S., Moerkotte G., « Evaluating Queries with Generalized Path Expressions », *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, ACM Ed., 1996.

Cet article propose d'étendre OQL avec des expressions de chemin généralisées, par exemple traversant de multiples associations. Des techniques d'évaluation de telles expressions de chemin sont discutées. Il est montré comment ces techniques peuvent être intégrées dans le processus d'optimisation.

[Cluet92] Cluet S., Delobel C., « A General framework for the Optimization of Object-Oriented Queries », *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, ACM Ed., p. 383-392, 1992.

Cet article développe un cadre général pour l'optimisation de requêtes objet basé sur la réécriture de graphes d'opérations. Il étend la réécriture d'expressions algébriques en introduisant une sorte de graphe de flux. Les nœuds représentent les collections et les arcs les opérations. Des informations de typage sont ajoutées aux nœuds. La réécriture des expressions de chemins est intégrée à la réécriture des expressions algébriques. Les index et le groupage sont aussi pris en compte.

[Cluet94] Cluet S., Moerkotte G., « Classification and Optimization of Nested Queries xin Object bases », *Journées Bases de Données Avancées*, 1994.

Cet article contient une classification des requêtes imbriquées en OQL. Il propose des techniques de désimbrication basées sur la réécriture algébrique.

[DeWitt93] DeWitt D., Lieuwen D., Mehta M., « Pointer-based join techniques for object-oriented databases », *Proceedings of Parallel and Distributed System International Conf.*, San Diego, p. 172-181, CA, 1993.

Les auteurs comparent divers algorithmes de traversées de chemin dans les bases de données objet. Ils comparent leurs performances et étudient la parallélisation.

[Finance94] Finance B., Gardarin G., « A Rule-Based Query Optimizer with Multiple Search Strategies, *Data & Knowledge Engineering Journal*, North-Holland Ed., vol. 13, n° 1, p. 1-29, 1994.

Un article proposant un optimiseur extensible pour bases de données à objets : l'optimiseur est basé sur des règles de transformations d'arbres d'opérations algébriques, exprimées sous la forme de règles de réécriture de termes. Les restructurations incluent des règles syntaxiques (descente des opérations de filtrages, permutation des jointures) et sémantiques (prises en compte de contraintes d'intégrité). La stratégie de recherche est elle-même paramétrable par des règles. Un tel optimiseur a été implémenté dans le projet Esprit EDS.

[Florescu96] Florescu D., « Espaces de Recherche pour l'Optimisation des Requêtes Orientées Objets », *Thèses de Doctorat, Université de Paris VI*, Versailles, 1996.

Cette excellente thèse de doctorat décrit la conception et la réalisation d'un optimiseur extensible pour bases de données objet. Les règles de réécriture sont introduites par des équivalences de requêtes. Ceci permet de traiter des cas variés, comme les ordonnancements de jointures, les réécritures de chemins, la prise en compte de règles sémantiques et de vues concrètes, etc. L'optimiseur est capable d'appliquer différentes stratégies de recherche à différents niveaux.

[Gardarin95] Gardarin G., Gruser J.R., Tang Z.H., « A Cost Model for Clustered Object-Oriented Databases », *Proceedings of 21st International Conference on Very Large Databases*, Zurich, Switzerland, 1995, p. 323-334.

Cet article présente les techniques de groupage avec priorité et le modèle de coût pour BD objet décrits ci-dessus.

[Gardarin96] Gardarin G., Gruser J.R., Tang Z.H., « Cost-based Selection of Path Expression Processing Algorithms in Object-Oriented Databases », *Proceedings of the 22nd International Conference on Very Large Data Bases*, Bombay, India, 1996, p. 390-401.

Cet article détaille les principaux algorithmes d'expression de chemins présentés ci-dessus, étudie leur coût et propose des heuristiques permettant de choisir le meilleur algorithme.

[Glover89] Glover F., « Tabu Search-part I », *ORSA Joinal on Computing 1*, p. 190-206, 1989.

[Glover90] Glover F., « Tabu Search-part II », *ORSA Joinal on Computing 2*, p. 4 –32, 1990.

Deux articles de référence qui développent les techniques de recherche taboues.

[Goldberg89] Goldberg D. E., *Genetic Algorithms in Search Optimization and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

Ce livre décrit des stratégies de recherche d'optimum, notamment pour l'apprentissage. Une place importante est réservée aux algorithmes génétiques.

[Graefe93] Graefe G., McKenna W., « The Volcano Optimizer Generator », Proceedings of the 9th International Conference on Data Engineering, IEEE Ed., p. 209-218, 1993.

Cet article décrit l'optimiseur extensible Volcano, basé sur des règles de réécriture dont les conditions et les actions sont écrites comme des procédures C. Volcano supporte l'optimisation de plans pour architecture parallèle.

[Haas89] Haas L. M., Cody W.F., Freytag J.C., Lapis G., Lindsay B.G., Lohman G.M., Ono K., Pirahesh H., « Extensible Query Processing in Starburst », Proceedings of the 1989 ACM SIGMOD International Conf. on Management of Data, ACM Ed., p. 377-388, 1989.

Starburst fut le projet de recherche qui a permis le développement des techniques objet-relationnel chez IBM. L'optimiseur extensible fut un des premiers réalisés. Il distinguait la phase de réécriture de celle de planning. La première était paramétrée par des règles, la seconde par des tables d'opérateurs.

[Harris96] E.P. Harris, K. Ramamohanarao, « Join Algorithm Costs Revisited », *The VLDB Journal*, vol. 5(1), p. 64-84,1996.

Cet article propose différents algorithmes de jointures et compare très précisément les coûts en entrées-sorties et en temps unité centrale.

[Hellerstein93] Hellerstein J. M., Michael Stonebraker, » Predicate Migration: Optimizing Queries with Expensive Predicates », *ACM SIGMOD Intl. Conf. On Management of Data*, p. 267-276, 1993.

Cet article discute de l'optimisation de questions avec des prédicats coûteux, incluant des fonctions utilisateurs. Il propose des méthodes de transformation.

[Holland75] Holland J.H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, 1975.

Ce livre traite de l'adaptation et introduit en particulier les algorithmes génétiques pour des fonctions multivariables.

[Ioannidis87] Ioannidis Y.E.,Wong E., « Query Optimization by Simulated Annealing », *Proceedings of the 1987 ACM SIGMOD Conference on Management of Data*, San Francisco, California, p. 9-22,1987.

Cet article présente l'adaptation de la stratégie de recherche « recuit simulé » à l'optimisation de requêtes Select-Project-Join. La méthode est évaluée en comparaison à la programmation dynamique classique.

[Ioannidis90] Ioannidis Y., Kang Y.C., « Randomized Algorithms for Optimizing Large Join Queries », *Proceedings of the 1990 ACM SIGMOD Intl. Conference on Management of Data*, Atlantic City, NJ, p. 312-321, 1990.

Cet article présente l'adaptation des stratégies de recherche amélioration itérative et recuit simulé à l'optimisation de requêtes Select-Project-Join. Il teste ces algorithmes sur de larges requêtes et montre que le recuit simulé trouve en général un meilleur plan. La méthode deux phases est alors proposée pour corriger l'amélioration itérative. Il est finalement montré que cette méthode est la meilleure des trois.

[Ioannidis91] Ioannidis Y., Kang Y.C., « Left-Deep vs. Bushy Trees : An Analysis of Strategy Spaces and its Implications for Query Optimization », *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, 1991, p. 168-177.

Cet article étudie les stratégies d'optimisation en fonction de l'espace de recherche considéré : espace limité aux arbres linéaires gauches ou prenant en compte tous les arbres, y compris les arbres branchus.

[Kim89] Kim K.C., Kim W., Dale A., « Indexing Techniques for Object-Oriented Databases », *Object-oriented concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, editors, AddisonWesley, 1989, p. 371-392.

Cet article introduit et évalue les index de chemins.

[Kemper90] Kemper A., Moerkotte G., « Advanced Query Processing in Object Bases Using Access Support Relations », *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Queensland, Australia, p. 290-301, 1990.

Cet article décrit l'optimiseur de GOM (Generic Object Model), un prototype réalisé à Karlsruhe. Cet optimiseur utilise des index de chemins sous la forme de relations support et un optimiseur basé sur des règles de réécriture. Ces dernières permettent l'intégration des index dans le processus de réécriture.

[Lanzelotte91] Lanzelotte R., Valduriez P., « Extending the Search Strategy in a Query Optimizer », *17th International Conference on Very Large Data Bases*, Barcelona, Catalonia, Spain, Morgan Kaufman Pub, p. 363-373,1991.

Cet article montre comment une conception orientée objet permet de rendre extensible un optimiseur pour bases de données. Plus particulièrement, la stratégie de recherche peut être paramétrée et changée en surchargeant quelques méthodes de base. Les stratégies aléatoires (amélioration itérative, recuit simulé, algorithmes génétiques) sont particulièrement intégrables à l'optimiseur.

[Lanzelotte93] Lanzelotte R., Valduriez P., Zait, M. « On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces », *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, p. 493-504, 1993.

Cet article analyse différentes stratégies d'optimisation pour l'optimisation de requêtes parallèles.

[Mitchell93] Mitchell G., Dayal U., Zdonick S., « Control of an Extensible Query Optimizer : A Planning-Based Approach », *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, p. 517-528,1993.

Les auteurs proposent d'organiser un optimiseur en modules de connaissance appelés régions, chaque région ayant sa propre stratégie de recherche. Les régions sont organisées hiérarchiquement, chaque région parente contrôlant ses subordonnées. Chaque région décrit ses capacités via son interface. Un niveau de contrôle global utilise les capacités des régions afin de planifier une séquence d'exécutions des régions pour traiter une requête. Le contrôle global est un planning dirigé par les buts.

[Nahar86] Nahar S., Sahni S., Shragowitz E., « Simulated Annealing and Combinatorial Optimization », *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, Las Vegas, NV, p. 293-299, 1986.

Cet article décrit la technique de recuit simulé pour l'optimisation combinatoire.

[Orenstein92] Orenstein J., Haradhvala S., Margulies B., Sakahara D., « Query Processing in the ObjectStore Database System », *ACM SIGMOD Int. Conf. on Management of Data*, San Diego, Ca., 1992.

ObjectStore est un SGBD objet parmi les plus vendus. Il supporte la persistance des objets C++ de manière orthogonale aux types de données, la gestion de transactions et les questions associatives. Il est basé sur une technique efficace de gestion de mémoire virtuelle intégrée au système. Les collections sont gérées comme des objets. Les questions sont intégrées à C++ sous forme d'opérateurs dont les opérandes sont une collection et un prédicat. Le prédicat

peut lui même contenir une question imbriquée. Cet article décrit la stratégie d'optimisation de questions implémentée.

[Ozsu95] Ozsu T., Blakeley J.A., « Query Processing in Object-oriented Database Systems », *Modern database systems*, edited by W. Kim, p. 146-174, 1995.

Cet article présente une vue d'ensemble des techniques d'optimisation dans les BD objet : architecture, techniques de réécriture algébriques, expressions de chemins, exécution des requêtes. C'est un bon tutorial sur le sujet.

[Shekita89] Shekita E. J., Carey M. J., « A Performance Evaluation of Pointer-Based Joins », *Proceddings of the 1990 ACM SIGMOD Intl. Conference on Management of Data*, New Jersey, p. 300-311, May 1990.

Cet article décrit trois algorithmes de jointures basés sur des pointeurs. Ce sont des variantes des boucles imbriquées, du tri-fusion et du hachage hybride. Une analyse permet de comparer ces algorithmes aux algorithmes correspondants qui ne sont pas basés sur des pointeurs. Il est montré que l'algorithme naturel de traversée en profondeur est peu efficace.

[Steinbrunn97] Steinbrunn M., Moerkotte G., Kemper A., « Heuristic and Randomized Optimization for the Join Ordering Problem », *VLDB Journal*, p. 191-208, 1997.

Les auteurs présentent une description assez complète et une évaluation comparée de tous les algorithmes d'optimisation combinatoire proposés pour les bases de données.

[Swami88] Swami A. N., Gupta A., « Optimization of Large Join Queries », *Proceedings of the 1988 ACM SIGMOD Intl. Conference on Management of Data*, Chicago, Illinois, p. 8-17,1988.

Il s'agit du premier article ayant introduit les techniques d'optimisation aléatoire pour chercher le meilleur plan d'exécution. Les auteurs montrent que le problème de recherche du meilleur plan est NP complet. Les méthodes combinatoires d'optimisation telles que l'itération itérative et le recuit simulé sont alors proposées. Des comparaisons montrent l'intérêt de l'amélioration itérative.

[Swami89] Swami A. N., « Optimization of Large Join Queries : Combining Heuristics and Combinatorial Techniques », *Proceedings of the 1989 ACM SIGMOD Conference*, Portland, Oregon, 1989, p. 367-376.

Les auteurs discutent la combinaison d'algorithmes combinatoires tels que l'amélioration itérative et le recuit simulé avec des heuristiques d'optimisation comme l'augmentation, l'amélioration locale et KBZ. L'augmentation consiste à choisir les relations dans un certain ordre croissant selon une mesure simple – la taille, la taille du résultat attendu, la sélectivité, le nombre de relations joignant, etc. L'heuristique KBZ consiste à étudier tous les arbres dérivant d'un

arbre choisi en permutant la racine. Ils définissent ainsi différents algorithmes combinés qui sont comparés. Les résultats sont encourageants pour la combinaison de l'amélioration itérative avec l'augmentation.

[Tan91] Tan K-L., Lu H., « A Note on the Strategy Space of Multiway Join Query Optimization Problem in Parallel Systems », *ACM SIGMOD Record*, vol. 20, 1991, p. 81-82.

Les auteurs étudient l'espace de recherche pour un arbre de jointure quelconque dans le cas d'un système parallèle. Ils estiment en particulier sa taille.

[Tang96] Tang Z., « Optimisation de requêtes avec expressions de chemin pour BD objet », *Thèse de doctorat, Université de Versailles*, 197 pages, Versailles, France, sept. 1996.

Cette thèse développe et compare trois algorithmes de traversée de chemins. L'auteur discute leur intégration dans le processus d'optimisation. Une stratégie génétique est proposée pour choisir la meilleure traversée pour de longs chemins. Diverses comparaisons démontrent l'intérêt de cette méthode. Un modèle de coût pour BD objet voisin du modèle décrit ci-dessus est développé, avec la formule de Tang.

[Valduriez87] Valduriez P., « Optimization of Complex Database Queries Using Join Indices », *Database Engineering Bulletin*, vol. 9, 1986, p. 10-16.

Cet article introduit les index de jointures, précurseur des index de chemins.

[Vance96] Vance B., Maier D., « Rapid Bushy Join-order Optimization with Cartesian Products », *Proceedings of the 1996 ACM SIGMOD Conference on Management of Data*, Montreal, Canada, p. 35-46, 1996.

Cet article montre qu'il est possible d'implémenter avec les techniques actuelles un optimiseur capable de considérer tous les arbres de jointures, y compris ceux avec des produits cartésiens.

[Yao77] Yao S.B., « Approximating the Number of Accesses in Database Organizations », *Comm. of the ACM*, vol. 20, n° 4, p. 260-270, April 1977.

Cet article introduit la fameuse formule de Yao.

Partie 4

AU-DELÀ DU SGBD

XV – Bases de données déductives (*Deductive databases*)

XVI – Gestion de transactions (*Transaction management*)

XVII – Conception des bases de données (*Database Design*)

XVIII – Nouvelles perspectives (*New developments*)

Chapitre XV

BASES DE DONNÉES DÉDUCTIVES

1. INTRODUCTION

Depuis que la notion de base de données déductive est bien comprise [Gallaire84], sous la pression des applications potentielles, les concepteurs de systèmes s'efforcent de proposer des algorithmes et méthodes efficaces pour réaliser des SGBD déductifs traitant de grands volumes de faits (les bases de données classiques) et de règles. L'objectif est de fournir un outil performant pour aider à la résolution de problèmes, exprimés sous forme de requêtes, dont la solution nécessite des volumes importants de données et de règles. Les applications potentielles sont nombreuses. Outre la gestion classique ou prévisionnelle, nous citerons par exemple l'aide à la décision, la médecine, la robotique, la productique et plus généralement toutes les applications de type systèmes experts nécessitant de grands volumes de données. Il a même été possible de penser que l'écriture de règles référençant de grandes bases de données remplacerait la programmation classique, au moins pour les applications de gestion. Ces espoirs ont été quelque peu déçus, au moins jusqu'à aujourd'hui.

Ce chapitre introduit la problématique des SGBD déductifs, puis présente le langage standard d'expression de règles portant sur de grandes bases de données, appelé DATALOG. Celui-ci permet les requêtes récursives. Nous étudions les extensions de

ce langage, telles que le support des fonctions, de la négation, des ensembles. Ensuite, nous abordons les problèmes de l'évaluation de questions sur des relations déduites. Après une brève vue d'ensemble, nous introduisons quelques techniques de représentation des règles par les graphes, puis nous nous concentrons sur la récursion. Les méthodes générales QoSaQ et Magic sont présentées. Quelques méthodes plus spécifiques sont résumées. Avant un bilan en conclusion, nous abordons, à travers des exemples, les langages de règles pour BD objet.

2. PROBLÉMATIQUE DES SGBD DÉDUCTIFS

Un SGBD déductif est tout d'abord un SGBD. En ce sens, il doit posséder un langage de description de données permettant de définir les structures des prédicats de la base B1, B2... Bn, par exemple sous forme de relations, et les contraintes d'intégrité associées. Il offre aussi un langage de requête permettant de poser des questions et d'effectuer des mises à jour. Ces deux langages peuvent être intégrés et posséder une syntaxe propre, ou plusieurs, offertes aux usagers. Parmi ces langages, il est permis de penser que SQL restera une des interfaces offertes par un SGBD déductif, surtout devant la poussée de sa normalisation.

2.1. LANGAGE DE RÈGLES

L'interface nouvelle offerte par un SGBD déductif est avant tout le **langage de règles**.

Notion XV.1 : Langage de règles (*Rule Language*)

Langage utilisé pour définir les relations déduites composant la base intentionnelle permettant d'écrire des programmes de règles du style <condition> ➔ <action>.

Le langage de règle est donc utilisé afin de spécifier les parties conditions et actions des règles de déduction. Plus précisément, à partir des prédicats B1, B2... Bn définis dans la base implantée (extensionnelle), le langage de règles permet de spécifier comment construire des prédicats dérivés R1,R2... interrogeables par les utilisateurs. Un langage de règles peut donc être perçu comme une extension des langages de définition de vues et de *triggers* des SGBD relationnels classiques.

L'extension est de taille car le langage de définition et de manipulation de connaissances va intégrer les fonctionnalités suivantes :

1. la possibilité d'effectuer les opérations classiques du calcul relationnel (union, restriction, projection, jointure, différence) ;

2. le support des ensembles incluant les fonctions d'agrégats traditionnelles des langages relationnels classiques ainsi que les attributs multivalués ;
3. la récursivité, qui permet de définir une relation déduite en fonction d'elle-même ;
4. la négation, qui permet de référencer des faits non existants dans la base ;
5. les fonctions arithmétiques et plus généralement celles définies par les utilisateurs ;
6. les mises à jour des faits au travers des règles ;
7. la modularité avec la gestion de niveaux d'abstraction successifs et de méta-règles.

En bref, toutes les facilités qui existent dans les langages de développement de bases de données vont chercher à figurer dans les langages de règles. L'objectif visé est d'ailleurs de remplacer ces langages.

2.2. COUPLAGE OU INTÉGRATION ?

La réalisation d'un SGBD déductif nécessite donc l'intégration d'un moteur de règles au sein d'un SGBD. Celui-ci doit être capable de réaliser l'inférence nécessaire lors de l'interrogation, voire la mise à jour, des prédicats dérivés. Une fonctionnalité analogue à celle d'un SGBD déductif peut être obtenue en couplant un moteur de règles à un SGBD. On distingue le couplage faible où les deux composants restent visibles à l'utilisateur du couplage fort où seul le langage de règles est visible. La figure XV.1 illustre les techniques de couplage et d'intégration. Un SGBD déductif essaie donc de réaliser l'intégration forte, en offrant un langage de définition et de manipulation de connaissances intégré.

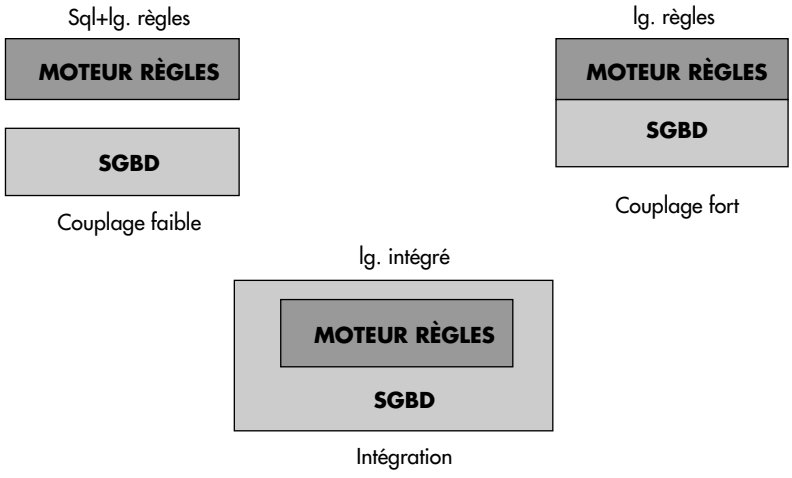


Figure XV.1 : Couplage versus intégration

La réalisation d'un SGBD déductif intégré pose de nombreux problèmes. Tout d'abord, il faut définir le langage d'expression de connaissances. Nous étudions ci-dessous l'approche DATALOG, inspirée au départ du langage de programmation logique PROLOG, qui est devenue le standard de la recherche. Ensuite, il faut choisir le modèle interne de données pour stocker les faits, mais aussi pour stocker les règles. Plusieurs approches partent d'un SGBD relationnel, soit étendu, soit modifié. D'autres ont cherché à réaliser des SGBD déductifs à partir de SGBD objet [Abiteboul90, Nicolas97] intégrant les deux paradigmes. Ensuite, il faut garantir la cohérence des données et des règles : ce problème d'intégrité étendue est très important, car il est possible en théorie de déduire n'importe quoi d'une base de connaissances (faits + règles) incohérente. Enfin, il faut répondre aux questions de manière efficace, en réalisant l'inférence à partir des faits et des règles, sans générer de faits inutiles ni redondants, mais aussi sans oublier de réponses. Le problème de l'efficacité du mécanisme d'inférence en présence d'un volume important de faits et de règles, notamment récursives, est sans doute l'un des plus difficiles.

2.3. PRÉDICATS EXTENSIONNELS ET INTENTIONNELS

Dans le contexte logique, une base de données est perçue comme un ensemble de prédicats. Les extensions des **prédicats extensionnels** sont matérialisées dans la base de données. Les prédicats extensionnels correspondent aux relations du modèle relationnel.

Notion XV.2 : Prédicat extensionnel (*Extensional predicate*)

Prédicat dont les instances sont stockées dans la base de données sous forme de tuples.

Une base de données est manipulée par des programmes logiques constitués d'une suite de clauses de Horn qui définissent des **prédicats intentionnels.** Un prédicat intentionnel est donc défini par un programme de règles logiques ; il correspond à une vue du modèle relationnel.

Notion XV.3 : Prédicat intentionnel (*Intensional predicate*)

Prédicat calculé par un programme constitué de règles logiques dont les instances ne sont pas stockées dans la base de données.

Une base de données logique est constituée d'un ensemble de prédicats extensionnels constituant la base de données extensionnelle et d'un ensemble de prédicats intentionnels constituant la base de données intentionnelle. Les règles permettant de calculer les instances des prédicats intentionnels sont donc partie intégrante de la base de données logique. Elles sont écrites dans le langage DATALOG basé sur les clauses de

Horn. La figure XV.2 illustre les notions de bases de données extensionnelle et intentionnelle, la seconde étant dérivée de la première par des règles stockées dans la méta-base du SGBD.

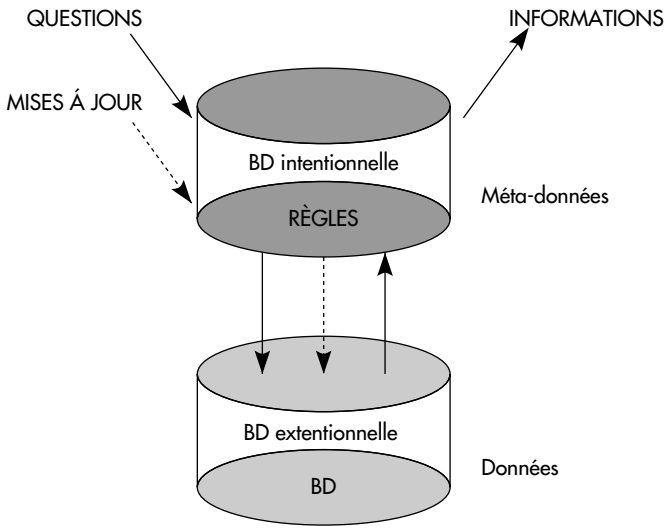


Figure XV.2 : Base de données extentionnelle et intentionnelle

2.4. ARCHITECTURE TYPE D'UN SGBD DÉDUCTIF INTÉGRÉ

Le SGBD est dit **déductif** car il permet de déduire des informations à partir de données stockées par utilisation d'un mécanisme d'inférence logique. Les informations sont les tuples des prédicats intentionnels ; elles peuvent être déduites lors de l'interrogation des prédicats intentionnels ou lors des mises à jour des prédicats extensionnels. La mise à jour des prédicats intentionnelles est difficile : il faut théoriquement répercuter sur les prédicats extensionnels, ce qui nécessite une extension des mécanismes de mise à jour au travers de vues.

Notion XV.4 : SGBD déductif (*Deductive DBMS*)

SGBD permettant de dériver les tuples de prédicats intentionnels par utilisation de règles.

En résumé, un SGBD déductif va donc comporter un noyau de SGBD permettant de stocker faits et règles dans la base, et d'exécuter des opérateurs de base comme un SGBD classique. Au-delà, il va intégrer des mécanismes d'inférence pour calculer

efficacement les faits déduits. Un langage intégré de définition et manipulation de connaissances permettra la définition des tables, des règles, l'interrogation et la mise à jour des informations (voir figure XV.3).

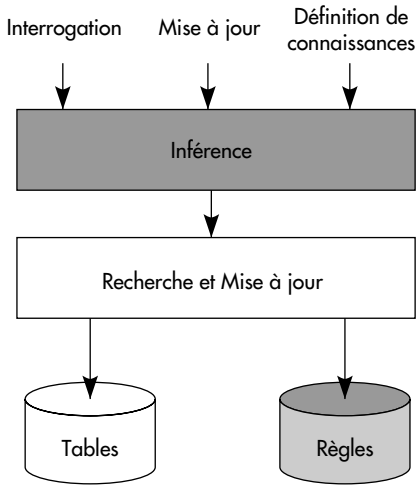


Figure XV.3 : Illustration de l'architecture d'un SGBD déductif

3. LE LANGAGE DATALOG

Le langage DATALOG est dérivé de la logique du premier ordre. C'est à la fois un langage de description et de manipulation de données. Le modèle de description de données supporté par DATALOG est essentiellement relationnel, une relation étant vue comme un prédicat de la logique. Le langage de manipulation est un langage de règles bâti à partir des clauses de Horn. Le nom DATALOG signifie « logique pour les données ». Il a été inventé pour suggérer une version de PROLOG (le langage de programmation logique) utilisable pour les données. Dans cette section, nous étudions tout d'abord la syntaxe de DATALOG, puis sa sémantique.

3.1. SYNTAXE DE DATALOG

Outre la définition des prédicats extensionnels, DATALOG permet d'écrire les clauses de Horn spécifiant les prédicats intentionnels. L'alphabet utilisé est directement dérivé de celui de la logique du premier ordre. Il est composé des symboles suivants :

1. des **variables** dénotées x, y, z … ;
2. des **constantes** choisies parmi les instances des types de base entier, numérique et chaînes de caractères ;
3. des **prédicats relationnels** dénotés par une chaîne de caractères, chaque prédicat pouvant recevoir un nombre fixe d'arguments (n pour un prédicat n-aire) ;
4. les **prédicats de comparaison** =, <, >, ≤, ≥, ≠ ;
5. les **connecteurs logiques** « et » dénoté par une virgule (,) et « implique » que l'on interprète de la droite vers la gauche, dénoté par le signe d'implication inversé (←).

À partir de cet alphabet (celui de la logique des prédicats du premier ordre particularisé et réduit), on construit des formules particulières qui sont des clauses de Horn ou règles DATALOG. Un **terme** est ici soit une constante, soit une variable. Un **atome** (aussi appelé **formule atomique** ou **littéral positif**) est une expression de la forme P(t1,t2…tn), où P est un prédicat n-aire. Un **atome instancié** est un atome sans variable (les variables ont été remplacées par des constantes). À partir de ces concepts, une **règle** est définie comme suit :

Notion XV.5 : Règle DATALOG (*DATALOG Rule*)

Expression de la forme Q ← P1, P2… Pn avec n ≥ 0 et où Q est un atome construit à partir d'un prédicat relationnel, alors que les Pi sont des atomes construits avec des prédicats relationnels ou de comparaison.

Q est appelé **tête de règle** ou **conclusion** ; P1, P2… Pn est appelé **corps de règle** ou **prémisse** ou encore **condition**. Chaque Pi est appelé **sous-but.** En appliquant l'équivalence Q ← P ⇔ ¬ P ∨ Q, une règle s'écrit aussi ¬ (P1, P2…Pn) ∨ Q ; puis en appliquant ¬ (P1, P2) ⇔ ¬ P1 ∨ ¬ P2, on obtient ¬ P1 ∨ ¬ P2 ∨ …¬ Pn ∨ Q. Donc, une règle est une clause de Horn avec au plus un littéral positif (la tête de règle Q).

La figure XV.4 donne un exemple de programme DATALOG définissant une base de données extensionnelle décrivant les employés (prédicat extensionnel `EMPLOYE`) et les services (prédicat extensionnel `SERVICE`) d'une grande entreprise. La base de données intentionnelle spécifie le chef immédiat de chaque employé (prédicat `DIRIGE1`), puis le chef du chef immédiat (prédicat `DIRIGE2`).

```
{
/* Déclaration des prédicats extensionnels */
EMPLOYE(NomService : String, NomEmploye : String) ;
SERVICE(NomService : String, NomChef : String) ;

/* Définition des prédicats intensionnels */
DIRIGE1(x,y) ← SERVICE(z,x), EMPLOYE(z,y)
DIRIGE2(x,y) ← DIRIGE1(x,z), DIRIGE1(z,y)
}
```

Figure XV.4 : Exemple de programme DATALOG

Un autre exemple de programme DATALOG est présenté figure XV.5. Il définit une base de données extensionnelle composée de pays et de vols aériens reliant les capitales des pays. La base de données intentionnelle permet de calculer les capitales proches (prédicat `CPROCHE`) comme étant les capitales atteignables l'une depuis l'autre en moins de cinq heures dans les deux sens. Les pays proches (prédicat `PPROCHE`) sont ceux ayant des capitales proches.

```
{
/* Déclaration des prédicats extensionnels */
PAYS(Nom : String, Capitale : String, Pop : Int) ;
VOLS(Num : Int, Depart : String, Arrivee : String, Duree : Int) ;

/* Définition des prédicats intensionnels */
CPROCHE(x,y) ← VOLS(z,x,y,t), t≤5, VOLS(w,y,x,u), u≤5 ;
PPROCHE(x,y) ← PAYS(x,u,p), PAYS(y,v,q), CPROCHE(u,v) ;
}
```

Figure XV.5 : Autre exemple de programme DATALOG

Parmi les règles, il en existe une classe particulièrement importante par la puissance qu'elle apporte au langage : il s'agit des **règles récursives** qui permettent de définir un prédicat intentionnel en fonction de lui-même.

Notion XV.6 : Règle récursive (*Recursive rule*)

Règle dont le prédicat de tête apparaît aussi dans le corps.

Une règle récursive dont le prédicat de tête apparaît une seule fois dans le corps est dite **linéaire**. Une règle non linéaire est **quadratique** si le prédicat de tête apparaît deux fois dans le corps. Au-delà, une règle récursive dont le prédicat de tête apparaît n ($n \geq 3$) fois dans le corps devient difficilement compréhensible.

La figure XV.6 illustre quelques exemples de règles récursives.

```
{
/* Dirige à tout niveau spécifié par une règle récursive linéaire */
DIRIGE(x,y) ← DIRIGE1(x,y) ;
DIRIGE(x,y) ← DIRIGE1(x,z),DIRIGE(z,y) ;

/* Dirige à tout niveau spécifié par une règle récursive quadratique */
DIRIGE(x,y) ← DIRIGE1(x,y) ;
DIRIGE(x,y) ← DIRIGE(x,z),DIRIGE(z,y) ;

/* Liaisons effectuables par étapes de moins de 5 heures */
LIAISON(x,y) ← VOLS(z,x,y,t), t ≤ 5 ;
LIAISON(x,y) ← LIAISON(x,z), LIAISON(z,y) ;
}
```

Figure XV.6 : Exemples de règles récursives

Chaque relation récursive nécessite une règle d'initialisation non récursive, puis une règle de calcul récursive. Le premier couple de règles définit qui dirige qui et ce, à tout niveau. Le deuxième définit la même relation, mais en utilisant une règle non linéaire. Le dernier couple spécifie les liaisons aériennes possibles entre capitales par des suites de liaisons simples effectuées en moins de cinq heures.

La figure XV.7 spécifie en DATALOG la célèbre base de données des familles à partir des prédicats extensionnels `PERE` et `MERE` indiquant qui est père ou mère de qui. La relation récursive `ANCETRE` a souvent été utilisée pour étudier les problèmes de la récursion. Nous avons ajouté la définition des grand-parents comme étant les parents des parents et celle des cousins comme étant deux personnes ayant un ancêtre commun. Les amis de la famille (`AMIF`) sont les amis (prédicat extensionnel `AMI`) ou les amis de la famille des parents. Les cousins de même génération (prédicat `MG`) se déduisent à partir des frères ou sœurs. Notez que cette définition est large : elle donne non seulement les cousins, mais aussi soi-même avec soi-même (vous êtes votre propre cousin de niveau 0 !), les frères et sœurs, puis vraiment les cousins de même génération.

```
{
/* Prédicats extensionnels Père, Mère et Ami */
   PERE(Père String, Enfant String)
   MERE(Mère String, Enfant String)
   AMI(Personne String, Personne String)

/* Parent comme union de père et mère */
   PARENT(x,y) ← PERE(x,y) ;
   PARENT(x,y) ← MERE(x,y) ;

/* Grand-parent par auto-jointure de parents */
   GRAND-PARENT(x,z) ← PARENT(x,y), PARENT(y,z) ;

/* Ancêtre défini par une règle linéaire */
   ANCETRE(x,y) ← PARENT(x,y) ;
   ANCETRE(x,z) ← ANCETRE(x,y), PARENT(y,z) ;

/* Cousin à partir d'ancêtres */
   COUSIN(x,y) ← ANCETRE(z,x), ANCETRE(z,y) ;

/* Ami de la familles comme ami des ancêtres */
   AMIF(x,y) ← AMI(x,y) ;
   AMIF(x,y) ← PARENT(x,z), AMIF(z,y) ;

/* Cousins de même génération à partir des parents */
   MG(x,y) ← PARENT(z,x), PARENT(z,y) ;
   MG(x,y) ← PARENT(z,x),MG(z,u),PARENT(u,y) ;
}
```

Figure XV.7 : La base de données des familles

Plus généralement, une **relation récursive** est une relation définie en fonction d'elle-même. Une relation récursive n'est pas forcément définie par une règle récursive. En effet, des règles mutuellement récursives permettent de définir une relation récursive. Plus précisément, il est possible de représenter la manière dont les prédicats dépendent l'un de l'autre par un graphe de dépendance. Les sommets du graphe sont les prédicats et un arc relie un prédicat P à un prédicat Q s'il existe une règle de tête Q dans laquelle P apparaît comme un sous-but. Un prédicat intentionnel est récursif s'il apparaît dans un circuit. La figure XV.8 illustre un programme DATALOG avec récursion mutuelle. Le prédicat R est récursif.

```
{
R(x,y) ← B(x,y) ;
R(x,y) ← P(x,z), B(z,y) ;
P(x,y) ← R(x,z), C(z,y) ;
}
```

Figure XV.8 : Récursion mutuelle et graphe de dépendance

En résumé, un programme DATALOG est un ensemble de règles. On note les règles constituant un programme entre crochets { }, chaque règle étant séparée de la suivante par un point virgule. L'ordre des règles est sans importance. Les prédicats relationnels sont classés en prédicats extensionnels dont les instances sont stockées dans la base sur disques, et en prédicats intentionnels qui sont les relations déduites (ou dérivées). DATALOG permet aux utilisateurs de définir à la fois des règles et des faits, un fait étant simplement spécifié comme une règle à variable instanciée sans corps (par exemple, `PERE(Pierre,Paul)`). Bien que la base extensionnelle puisse être définie en DATALOG comme une suite de prédicats, nous admettrons en général qu'elle est créée comme une base relationnelle classique, par exemple en SQL. Ainsi, DATALOG sera plutôt utilisé pour spécifier des règles, avec des corps et des têtes. Le couple base de données extensionnelle-base de données intentionnelle écrit en DATALOG constitue une base de données déductive définie en logique.

3.2. SÉMANTIQUE DE DATALOG

La sémantique d'un programme DATALOG (c'est-à-dire ce que calcule ce programme) peut être définie de plusieurs manières. Nous examinons ci-dessous les trois techniques les plus courantes.

3.2.1. Théorie de la preuve

DATALOG dérivant de la logique, il apparaît naturel d'utiliser une méthode de preuve pour calculer les instances des prédicats intentionnels. On aboutit alors à l'approche

sémantique de la preuve. Dans cette approche, un programme DATALOG calcule tout ce qui peut être prouvé en utilisant la méthode de preuve par résolution [Lloyd87]. Un fait non prouvable est considéré comme faux.

Notion XV.7 : Sémantique de la preuve (*Proof theoretic semantics*)

Sémantique selon laquelle un fait est vrai s'il peut être prouvé en appliquant la méthode de résolution à partir des axiomes logiques dérivés d'un programme DATALOG.

Afin d'illustrer la méthode, considérons la base de données définie figure XV.4, contenant :

1. le prédicat extensionnel `EMPLOYE` avec deux tuples `<informatique-Julie>` et `<informatique-Pierre>` indiquant que Pierre et Julie sont deux employés du département informatique ;
2. le prédicat extensionnel `SERVICE` avec un tuple `<informatique-Pierre>` indiquant que Pierre est le chef du service informatique.

Pour savoir si le tuple `<Pierre-Julie>` appartient logiquement au prédicat intentionnel `DIRIGE1`, il suffit d'appliquer la méthode de résolution pour prouver le théorème `DIRIGE1 (Pierre,Julie)` à partir des axiomes dérivés des relations de base `EMPLOYE (informatique,Julie)`, `EMPLOYE (informatique,Pierre)` et de celui dérivé de la règle `DIRIGE1(x,y)` ← `SERVICE(z,x), EMPLOYE(z,y)`. Ce dernier axiome se réécrit, en éliminant l'implication, `DIRIGE1(x,y)` ∨ ¬ `SERVICE (z,x)` ∨ ¬ `EMPLOYE (z,y)`. L'arbre de preuve permettant de conclure que `<Pierre-Julie>` est un fait vrai (c'est-à-dire un tuple de `DIRIGE1`) est représenté figure XV.9.

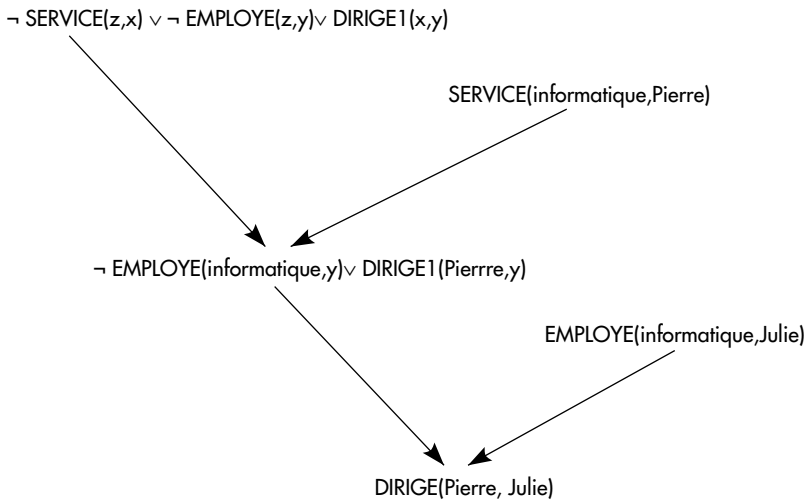


Figure XV.9 : Exemple d'arbre de preuve

En résumé, la méthode de résolution permet de donner une sémantique à un programme DATALOG : un fait appartient à un prédicat intentionnel s'il peut être prouvé comme un théorème par résolution. En cas d'échec de la méthode, le fait est supposé faux : il n'appartient pas au prédicat intentionnel. Cette interprétation de l'absence de preuve est appelée **négation par échec**. La méthode ainsi complétée est appelée méthode de résolution avec négation par échec (ou encore méthode SLDNF). Il s'agit d'une méthode qui permet de déterminer si un tuple appartient ou non à un prédicat intentionnel. C'est donc une méthode essentiellement procédurale qui fonctionne tuple à tuple. Elle est proche des méthodes de calcul appliquées par les interpréteurs PROLOG.

3.2.2. Théorie du modèle

Une seconde approche consiste à voir les règles comme définissant des modèles possibles construits à partir des instances des prédicats extensionnels. Rappelons qu'un modèle d'un ensemble de formules logiques est une interprétation dans laquelle tous les théorèmes sont vrais. Un modèle d'un programme DATALOG est donc une interprétation vérifiant les propriétés suivantes :

1. Pour chaque tuple <a1,a2…an> d'un prédicat extensionnel B, B(a1,a2…an) est vrai dans l'interprétation.
2. Pour chaque règle Q(t1,t2…tn) ← P1,P2,Pn et pour toute affectation de variable θ dans l'interprétation, si θ(P1∧ P2∧…Pn) est vrai dans l'interprétation, alors θ(Q(t1,t2…tn)) est aussi vrai.

En clair, un modèle d'un programme DATALOG est un ensemble d'instances de prédicats qui contiennent tous les faits de la base extensionnelle et tous les faits qui peuvent être inférés de ceux-ci en appliquant les règles. À partir d'un modèle, il est possible d'en générer d'autres par exemple par ajout de littéraux n'influant pas sur les conditions.

Une propriété intéressante des programmes DATALOG est que l'intersection de deux modèles reste un modèle. En conséquence, il existe un plus petit modèle qui est l'intersection de tous les modèles. Ce modèle correspond à la sémantique d'un programme DATALOG, appelée **plus petit modèle**. On peut alors définir l'approche sémantique du modèle comme suit :

Notion XV.8 : Sémantique du modèle (*Model theoretic semantics*)

Sémantique selon laquelle un fait est vrai s'il appartient au plus petit modèle des formules logiques composant un programme DATALOG.

Afin d'illustrer la méthode, le programme constitué des faits de base :

EMPLOYE (informatique,Pierre),
EMPLOYE (informatique,Julie),
SERVICE (informatique, Pierre)

et de la règle

DIRIGE1 (x,y) ← SERVICE (z,x), EMPLOYE (z,y)

a en particulier pour modèles :

1. { EMPLOYE (informatique,Pierre), EMPLOYE (informatique,Julie), SERVICE (informatique,Pierre), DIRIGE1 (Pierre,Julie), DIRIGE1 (Pierre,Pierre) } ;
2. { EMPLOYE (informatique,Pierre), EMPLOYE (informatique,Julie), SERVICE (informatique,Pierre), DIRIGE1 (Pierre,Julie), DIRIGE1 (Pierre,Pierre), DIRIGE1 (Julie,Julie)} ;

Le plus petit modèle est :

{ EMPLOYE (informatique,Pierre), EMPLOYE (informatique,Julie), SERVICE (informatique,Pierre), DIRIGE1 (Pierre,Julie), DIRIGE1 (Pierre,Pierre) }.

Il définit donc la sémantique du programme DATALOG.

3.2.3. Théorie du point fixe

Une autre approche pour calculer la sémantique d'un programme DATALOG, donc les prédicats intentionnels, consiste à interpréter le programme comme un ensemble de règles de production et à exécuter les règles jusqu'à ne plus pouvoir générer aucun fait nouveau. Cette procédure correspond à l'application récursive de l'opérateur **conséquence immédiate** [VanEmden76] qui ajoute à la base de faits successivement chaque fait généré par une règle dont la condition est satisfaite.

Dans le contexte des bases de données, on préfère en général calculer des ensembles de faits en appliquant les opérateurs de l'algèbre relationnelle que sont la restriction, la jointure, l'union et la différence. À la place de la jointure et afin de simplifier l'algorithme de traduction des règles en expressions relationnelles, nous utiliserons le produit cartésien suivi d'une restriction. Afin de traduire simplement une règle DATALOG en expression de l'algèbre relationnelle, on renommera tout d'abord les variables de même nom en introduisant des prédicats additionnels d'égalité entre variables. Par exemple, la règle R(x,y) ← B(x,z), C(z,y) sera réécrite R(x,y) ← B(x,z1), C(z2,y), z1=z2. Une telle règle est appelée **règle rectifiée**.

Chaque condition d'une règle rectifiée du type Q ← R1,R2, … Rn, P1,P2, … Pm (Ri sont des prédicats relationnels et Pj des prédicats de contraintes sur les variables) peut être traduite en une expression d'algèbre relationnelle $\sigma_{P1,P2,\ldots Pm}$(R1xR2x…Rn), où σ désigne la restriction par le critère en indice et x l'opération de produit cartésien. Cette expression calcule un prédicat dont les colonnes correspondent à toutes les variables apparaissant dans la règle rectifiée. Une projection finale doit être ajoutée afin de conserver les seules variables référencées dans la tête de la règle.

Ainsi, soit E_r l'expression de l'algèbre relationnelle résultant de la transformation d'une condition d'une règle r de la forme Q ← R1,R2, … Rn, P1,P2, … Pm. La règle est remplacée par l'équation de l'algèbre relationnelle $Q = Q \cup E_r$. Chaque programme DATALOG P est ainsi remplacé par un programme d'algèbre relationnelle noté T_P. La figure XV.10 présente des exemples de transformation de programmes DATALOG en programmes de l'algèbre relationnelle.

EMPINF(x,y) ← EMPLOYE(x,y), x = informatique;
EMPINF = EMPINF ∪ $\sigma_{A1=informatique}$(EMPLOYE)

DIRIGE1(x,y) ← SERVICE(z,x), EMPLOYE(z,y) ;
DIRIGE1 = DIRIGE1 ∪ $\Pi_{A2,A4}$ ($\sigma_{A1=A3}$(SERVICE x EMPLOYE))

DIRIGE2(x,y) ← DIRIGE1(x,z), DIRIGE1(z,y) ;
DIRIGE2 = DIRIGE2 ∪ $\Pi_{A1,A4}$ ($\sigma_{A2=A3}$(DIRIGE1 x DIRIGE1))

DIRIGE(x,y) ← DIRIGE1(x,y) ;
DIRIGE = DIRIGE ∪ DIRIGE1
DIRIGE(x,y) ← DIRIGE1(x,z), DIRIGE(z,y) ;
DIRIGE = DIRIGE ∪ $\Pi_{A1,A4}$ ($\sigma_{A2=A3}$(DIRIGE1 x DIRIGE))

Figure XV.10 : Exemple de transformation de règles en algèbre

À partir du programme d'algèbre Tp obtenu par traduction ligne à ligne en algèbre relationnelle d'un programme P, il est possible de définir la sémantique du programme DATALOG. Soit ⊥ les faits initiaux contenus dans la base de données (les tuples de la base extensionnelle). La sémantique du programme peut être définie par application successive de T_P à ⊥, puis au résultat $T_P(\perp)$, puis au résultat T_P ($T_P(\perp)$), etc., jusqu'à obtenir un point fixe $T_P^n(\perp)$. On dit qu'on a obtenu un point fixe lorsqu'une nouvelle application de T_P ne change pas le résultat. L'existence d'un point fixe est garantie car T_P est un opérateur monotone qui ne fait qu'ajouter des faits aux prédicats intentionnels ; ceux-ci étant bornés par le produit cartésien de leurs domaines, le processus converge [Tarski55].

Notion XV.9 : Sémantique du point fixe (*Fixpoint semantics*)

Sémantique selon laquelle un fait est vrai s'il appartient au point fixe d'un opérateur qui peut être défini comme étant le programme d'algèbre relationnelle obtenu par traduction une à une des règles du programme DATALOG.

Par exemple, avec le programme DATALOG :

```
P = {   PARENT(x,y) ← PERE(x,y);
        PARENT(x,y) ← MERE(x,y) ;
        ANCETRE(x,y) ← PARENT(x,y);
        ANCETRE(x,y) ← PARENT(x,z), ANCETRE(z,y); }
```

on calcule :

T_P = { PARENT = PARENT $\cup$ PERE;
PARENT = PARENT $\cup$ MERE;
ANCETRE = PARENT;
ANCETRE = ANCETRE $\cup$ $\Pi_{1,4}$ (PARENT $\bowtie$ ANCETRE); }

Soit $\perp$ = { PERE (Jean,Pierre) ; MERE (Pierre,Julie) ; } les faits initiaux de la base de données. On calcule :

$T_P(\perp)$ = { PARENT (Jean,Pierre); PARENT (Pierre,Julie);
ANCETRE (Jean,Pierre); ANCETRE (Pierre,Julie);
ANCETRE (Jean, Julie)};

$T_P(T_P(\perp)) = T_P(\perp)$ qui est donc le point fixe.

3.2.4. Coïncidence des sémantiques

En résumé, DATALOG permet de définir des prédicats dérivés dont les instances sont calculables par diverses sémantiques. Heureusement, ces dernières coïncident pour les programmes DATALOG purs, c'est-à-dire sans les extensions que nous verrons dans la suite. Cela provient du fait que toute règle DATALOG ne fait qu'ajouter des faits à un prédicat intentionnel. En général, la sémantique d'un programme DATALOG n'est pas calculée complètement, c'est-à-dire que les prédicats dérivés ne sont pas totalement calculés. Les seuls **faits relevants** sont calculés pour répondre aux questions ou pour répercuter les mises à jour. Une question peut être exprimée en SQL sur un prédicat déduit. Cependant, elle peut aussi être exprimée comme une règle sans tête : la qualification de la question est spécifiée par le corps de la règle. Afin de marquer les règles questions (sans tête), nous remplacerons l'implication par un point d'interrogation. Ainsi, la recherche des ancêtres de Julie s'effectuera par la règle ? ANCETRE(x, Julie). En pratique, une question est une règle avec une tête implicite à calculer en résultat, contenant les variables libres dans le corps de la règle.

En conclusion, si l'on compare DATALOG avec l'algèbre relationnelle, il apparaît que DATALOG permet de définir des règles et de poser des questions complexes incluant les opérateurs de l'algèbre relationnelle que sont l'union (plusieurs règles de mêmes têtes), la projection (variables du corps de règle omises en tête de règles), la restriction (prédicat non relationnel dans le corps d'une règle) et la jointure (plusieurs prédicats relationnels dans le corps d'une règle avec variables communes). Aussi, DATALOG permet la récursion que ne permet pas l'algèbre relationnelle. Cela se traduit par le fait qu'il faut effectuer une boucle sur l'opérateur T_P jusqu'au point fixe pour calculer la sémantique d'un programme DATALOG. Cette boucle est inutile si le programme ne comporte pas de relation récursive. DATALOG inclut la puissance de la récursion mais ne supporte pas la négation, au moins jusque-là. Dans la suite, nous allons étendre DATALOG avec la négation. Auparavant, nous allons examiner comment représenter les informations négatives.

4. LES EXTENSIONS DE DATALOG

Nous étudions maintenant l'extension de DATALOG avec la négation, puis avec les fonctions et les ensembles. Jusque-là, DATALOG avec récursion ne permet que d'ajouter des informations à la base intentionnelle. Le type de raisonnement supporté est **monotone**. Il devient non monotone avec la négation. Fonctions et ensembles permettent d'étendre DATALOG pour s'approcher des objets que nous intégrerons plus loin.

4.1. HYPOTHÈSE DU MONDE FERMÉ

Jusque-là, les seuls axiomes dérivés d'une base de données extensionnelle sont des faits positifs (sans négation). Afin de permettre de répondre à des questions négatives (par exemple, qui n'est pas dirigé par Pierre?), il apparaît nécessaire de compléter les axiomes avec des axiomes négatifs pour chaque fait qui n'apparaît pas dans la base. Cette dérivation d'axiomes négatifs, qui consiste à considérer que tout fait absent de la base est faux, est connue sous le nom d'**hypothèse du monde fermé**. Nous l'avons déjà introduite dans le chapitre V sur la logique. Nous la définissons plus précisément ici.

Notion XV.10 : Hypothèse du monde fermé (*Closed World Assumption*)

Hypothèse consistant à considérer que tout fait non enregistré dans la base extensionnelle et non déductible par les règles est faux.

Ainsi, si la relation PERE (Père, Enfant) contient les faits PERE (Julie,Pierre) et PERE (Jean,Paul), on en déduit :

¬ PERE (Julie,Paul), ¬ PERE (Julie,Jean), ¬ PERE (Julie,Julie), ¬ PERE (Pierre,Julie), ¬ PERE (Pierre,Jean), ¬ PERE (Pierre,Paul), etc.

L'hypothèse du monde fermé est une règle puissante pour inférer des faits négatifs. Elle suppose qu'un domaine peut prendre toutes les valeurs qui apparaissent dans la base (domaine actif) et que tous les faits correspondant à ces valeurs non connus sont faux [Reiter78]. Pour être valide, cette hypothèse nécessite des axiomes additionnels tels que l'unicité des noms et la fermeture des domaines [Reiter84].

Par exemple, en présence de valeurs nulles dans les bases de données, l'hypothèse du monde fermé est trop forte car elle conduit à affirmer comme faux des faits inconnus. Les théoriciens se sont penchés sur des hypothèses plus fines, tolérant les valeurs nulles [Reiter84]. Une variante de l'hypothèse du monde fermée consiste à modifier la méthode de résolution permettant de répondre à une question en supposant faux tout fait qui ne peut être prouvé comme vrai. Cette approche est connue comme la **négation par échec**.

4.2. NÉGATION EN CORPS DE RÈGLES

Il existe plusieurs raisons pour ajouter la négation à DATALOG ; en particulier, la négation est souhaitable pour pouvoir référencer l'inexistence d'un fait dans un prédicat. Elle permet de représenter la différence relationnelle ; elle est aussi utile pour exprimer des exceptions. Par exemple, lors d'un parcours de graphe, si l'on désire éviter le parcours des arcs enregistrés dans une relation INTERDIT (Origine, Extrémité), il est possible de compléter le programme de parcours comme suit :

{ CHEMIN(x,y) ← ARC(x,y), ¬INTERDIT(x,y) ;
CHEMIN(x,y) ← CHEMIN(x,z), ARC(z,y), ¬INTERDIT(z,y) }

Plus généralement, l'introduction de la négation permet d'écrire des règles de la forme :

Q ← L1, L2… Ln

où Q est un littéral positif et L1, L2… Ln sont des littéraux positifs ou négatifs. Rappelons qu'un littéral négatif est une négation d'une formule atomique de la forme ¬ P(t1,t2…tn), où P est un prédicat et t1, t2, … tn sont des termes. Le langage DATALOG ainsi étendu avec des prédicats négatifs dans le corps de règle est appelé DATALOG^{neg}, encore noté DATALOG$^{\neg}$.

Notion XV.11 : DATALOG avec négation (*DATALOG^{neg}*)

Version étendue de DATALOG permettant d'utiliser des littéraux négatifs dans le corps des règles.

La sémantique d'un programme DATALOG^{neg} n'est pas facile à définir. En effet, l'intersection de modèles d'un tel programme n'est en général pas un modèle. Par exemple, le programme :

{ OISEAU (Pégase) ;
PINGOIN(x) ← OISEAU(x), ¬VOLE(x) ;
VOLE(x) ← OISEAU(x), ¬ PINGOIN(x) }

a pour modèles :

– { OISEAU(Pégase); PINGOIN(Pégase) } et

– { OISEAU(Pégase); VOLE(Pégase) }

dont l'intersection { OISEAU(pégase) } n'est pas un modèle. Un tel programme n'a donc pas de plus petit modèle. En fait, en utilisant les équivalences logiques, la deuxième et la troisième règles peuvent être réécrites :

VOLE(x) ∨ PINGOIN(x) ← OISEAU(x).

Nous avons là une règle disjonctive qui introduit des informations ambiguës. D'où l'existence de plusieurs modèles possibles.

Sous certaines conditions syntaxiques, le problème du choix d'un modèle peut être résolu en divisant le programme en strates successives, chacune ayant un plus petit

modèle [Apt86, Pryzymusinski88]. Un module d'un programme DATALOG étant un ensemble de règles, un **programme stratifié** peut être défini comme suit :

Notion XV.12 : Programme DATALOG stratifié (*Stratified Datalog Program*)

Programme DATALOG divisé en un ensemble ordonné de modules {S1, S2…Sn} appelés strates, de telle manière que la strate Si n'utilise que des négations de prédicats complètement calculés par les strates S1, S2, Si-1 (ou éventuellement aucune négation de prédicats).

En fait, le plus petit modèle d'une strate est calculé à partir du plus petit modèle de la strate précédente, la négation étant remplacée par un test de non-appartenance. Si M est le plus petit modèle de la strate Si-1, ¬ P(x) est interprété comme « P(x) n'appartient pas à M » dans la strate Si.

Tout programme DATALOG n'est pas stratifiable ; il doit être possible de calculer complètement toute l'extension d'un prédicat avant d'utiliser sa négation pour pouvoir stratifier un programme. Cela n'est pas vrai si la récursion traverse la négation. Les programmes stratifiables ont un plus petit modèle unique qui est caractérisé en définissant un ordre partiel entre les prédicats. L'ordre correspond à un calcul de plus petit modèle strate par strate en utilisant l'hypothèse du monde fermé.

Par exemple, les programmes :

– {P(x) ← ¬ P(x)} et

– { PAIRE(0); IMPAIRE(x) ← ¬ PAIRE(x) ; PAIRE(x) ← ¬IMPAIRE(x) }

ne sont pas stratifiables car la négation se rencontre sur un cycle de récursion (c'est-à-dire traverse la récursion). Le programme

– { R(1) ← ; P(1) ← ; P(2) ← ; Q(x) ← R(x); T(x) ← P(x),¬Q(x)}

est stratifiable; deux strates possibles sont :

– S1 = {R(1) ← ; Q(x) ← R(x)} puis

– S2 = { P(1) ← ; P(2) ← ; T(x) ← P(x),¬ Q(x)};

S1 calcule Q sans utiliser ¬ Q, puis S2 calcule T en utilisant ¬ Q et P. Le plus petit modèle de S1 est {R(1); Q(1)}. Celui de S2 est {R(1); Q(1); T(2)}. Notez que l'ordre de calcul des prédicats est fondamental : si on commençait à calculer T avant Q, on obtiendrait T(1) et un résultat qui ne serait pas un modèle.

La négation en corps de règle est importante car elle permet de réaliser la différence relationnelle. La stratification correspond à la sémantique couramment admise en relationnelle : avant d'appliquer une différence à une relation, il faut calculer complètement cette relation (autrement dit, le « pipe-line » est impossible avec l'opérateur de différence).

4.3. NÉGATION EN TÊTE DE RÈGLES ET MISES À JOUR

Comme PROLOG, DATALOG est construit à partir des clauses de Horn. Aussi, les règles sont en principe limitées à des clauses de Horn pures, disjonctions d'un seul

prédicat positif avec plusieurs prédicats négatifs (0 à n). Il n'y a pas de difficultés à étendre les têtes de règles à plusieurs prédicats positifs reliés par conjonction (et) ; une règle à plusieurs têtes est alors interprétée comme plusieurs règles à une seule tête avec le même corps. Une autre possibilité est de tolérer un prédicat négatif en tête de règle. Une information négative étant en général une information non enregistrée dans la base, une interprétation possible pour un tel prédicat est une suppression des faits correspondant aux variables instanciées satisfaisant la condition de la règle. Le langage DATALOG avec négation possible en corps et en tête de règle est appelé **DATALOG^{neg,neg}**, encore noté **DATALOG$^{\neg\neg}$**.

Notion XV.13 : DATALOG avec double négation (*DATALOG^{neg,neg}*)

Version étendue de DATALOG permettant d'utiliser des littéraux négatifs en tête et en corps de règle.

La négation en tête de règle est donc interprétée comme une suppression. C'est elle qui confère réellement la non monotonie à des programmes DATALOG. Au-delà, il est possible de placer plusieurs prédicats en tête d'une règle DATALOG. Supporter à la fois les règles à têtes multiples et des prédicats négatifs en tête de règle conduit à permettre les mises à jour dans le langage de règles. On parle alors du langage DATALOG avec mise à jour, noté DATALOG*.

Par exemple, la règle suivante, définie sur la base de données intentionnelle { PARENT(Ascendant, Descendant), ANCETRE(Ascendant, Descendant) } est une règle DATALOG* :

ANCETRE(x,z), ¬ ANCETRE(z,x) ← PARENT(x,y), ANCETRE(y,z)

Cette règle génère de nouveaux ancêtres et supprime des cycles qui pourraient apparaître dans la relation ancêtre (si x est le parent de y et y l'ancêtre de z, alors x est l'ancêtre de z mais z n'est pas l'ancêtre de x).

Une interprétation de DATALOG* est possible à partir des règles de production, très populaires dans les systèmes experts. Une **règle de production** est une expression de la forme :

<condition> → <expression d'actions>.

Une **expression d'actions** est une séquence d'actions dont chacune peut être soit une mise à jour, soit un effet de bord (par exemple, l'appel à une fonction externe ou l'édition d'un message). Une **condition** est une formule bien formée de logique. Quand l'ordre d'évaluation des règles est important, celles-ci sont évaluées selon des priorités définies par des **méta-règles** (des règles sur les règles) ; par exemple, une méta-règle peut simplement consister à exécuter les règles dans l'ordre séquentiel. Ainsi, un langage de règles de production n'est pas complètement déclaratif mais peut contenir une certaine procéduralité. Il en va de même pour les programmes DATALOG avec mises à jour.

Un programme DATALOG* peut être compris comme un système de production [Kiernan90]. Chaque règle est exécutée jusqu'à saturation pour chaque instanciation possible des variables par des tuples satisfaisant la condition. Un prédicat positif en tête de règle correspond à une insertion d'un fait et un prédicat négatif à une suppression. Une règle peut à la fois créer des tuples et en supprimer dans une même relation. Plus précisément, soit une expression d'actions de la forme R(p1), R(p2)…, ¬ R(n1), ¬ R(n2)… avec des actions conflictuelles sur une même relation R. Notons P = {p1, p2…} l'ensemble des tuples insérés dans R et N = {n1, n2…} l'ensemble des tuples supprimés. Certains tuples pouvant être à la fois insérés et supprimés, il faut calculer l'effet net des insertions et suppressions. De sorte à éviter les sémantiques dépendant de l'ordre d'écriture des actions, l'opération effectuée peut être définie par : $R = R - (N - P) \cup (P - N)$. Ainsi, une telle expression d'actions effectue une mise à jour de la relation R ; l'ordre des actions dans l'expression est sans importance.

L'existence d'un point fixe pour un programme DATALOG* n'est pas un problème trivial. Certaines règles peuvent supprimer des tuples que d'autres règles créent. Un programme de règles avec mises à jour peut donc boucler ou avoir différents états stables selon l'ordre d'application des règles, donc avoir une sémantique ambiguë. Un programme est **confluent** s'il conduit toujours au même résultat (point fixe), quel que soit l'ordre d'application des règles. Afin d'éviter les programmes à sémantique ambiguë, il est possible d'utiliser une méta-règle implicite analogue à la stratification : une règle avec suppression sur une relation R ne peut être exécutée que quand toutes les règles insérant dans R ont été exécutées jusqu'à saturation. Une telle règle est très restrictive et impose par exemple de rejeter les règles à la fois insérant et supprimant (mises à jour).

Par exemple, pour démontrer la puissance de DATALOG ainsi étendu, nous proposons des règles de transformation de circuits électriques. Cet exemple suppose une base de données relationnelle composée d'une unique relation `CIRCUIT(Nom, Fil, Origine, Extrémité, Impédance)`. Nom est le nom du circuit. Fil est un identifiant donné à chaque fil. Les origines et extrémités de chaque fil sont données par les attributs correspondants, alors que le dernier attribut donne l'impédance du circuit. Un problème typique est de calculer l'impédance du circuit. Pour cela, il faut appliquer les transformations série et parallèle, classiques en électricité. Chaque règle remplace deux fils par un fil qu'il faut identifier. Il faut donc pouvoir créer de nouveaux identifiants de fils, ce que nous ferons par une fonction de création d'objets à partir de deux objets existants f1 et f2, dénotée new (f1,f2). En conséquence, la règle suivante effectue la transformation parallèle (1/i = 1/i1 + 1/i2) :

¬ CIRCUIT(x,f1,o,e,i1),¬ CIRCUIT(x,f2,o,e,i2),
CIRCUIT(x,new(f1,f2),o,e,(i1*i2)/(i1+i2))
← CIRCUIT(x,f1,o,e,i1), CIRCUIT(x,f2,o,e,i2)

La transformation série est similaire, mais il faut s'assurer de la non existence d'un fil partant de la jonction des deux fils en série, ce qui nécessite une négation en corps de règle. D'où la règle suivante qui cumule les impédances de deux fils en série :

¬ CIRCUIT(x,f1,o,e1,i1),¬ CIRCUIT(x,f2,e1,e,i2),
CIRCUIT(x,new(f1,f2),o,e,i1+i2)
← CIRCUIT(x,f1,o,e1,i1),CIRCUIT(x,f2,e1,e,i2),¬ CIRCUIT(x,f3,e1,e2,i3)

Ces deux règles réduisent les circuits par remplacement des fils en parallèle et en série appartenant à un même circuit par un fil résultat de la transformation effectuée. Notez qu'elles ne sont pas stratifiables. Cependant, le programme de règles est confluent, au moins pour des circuits électriques connexes.

D'autres sémantiques que la sémantique opérationnelle des règles de production introduite ci-dessus ont été proposées pour Datalog* [Abiteboul95], comme la **sémantique inflationiste** et la **sémantique bien fondée**. La sémantique inflationiste est simple : elle calcule les conditions, puis tous les faits déduits de toutes les règles à la fois. Elle applique donc un opérateur de point fixe global modifié. Malheureusement, le résultat ne correspond guère à la signification courante. La sémantique bien fondée est plus générale : elle est basée sur une révision de l'hypothèse du monde fermée, en autorisant des réponses inconnues à des questions. Pour Datalog^{neg}, elle coïncide avec la sémantique stratifiée lorsque les programmes sont stratifiables. Les problèmes de sémantique de Datalog* sont en résumé très difficiles et ont donné lieu à de nombreux travaux théoriques de faible intérêt.

4.4. SUPPORT DES FONCTIONS DE CALCUL

Pour accroître la puissance de DATALOG, il est souhaitable d'intégrer les fonctions de la logique du premier ordre au langage. Des symboles de fonctions pourront alors être utilisés dans les arguments des prédicats, en tête ou en corps de règle. Des exemples de fonctions sont les fonctions arithmétiques (+,-,/,*) ou plus généralement des fonctions mathématiques (`LOG`, `EXP`, `SIN`…), voire des fonctions programmées par un utilisateur. Les fonctions sont importantes car elles permettent en général la manipulation d'objets complexes [Zaniolo85], par exemple, des figures géométriques. En général, les fonctions permettent d'invoquer des types abstraits de données [Stonebraker86].

D'un point de vue syntaxique, l'extension consiste à introduire dans l'alphabet des symboles de fonctions, notés f, g, h … Chaque fonction a une arité n, qui signifie que la fonction accepte n paramètres. De nouveaux termes peuvent être construis de la forme f(t1, t2, … tn) où chaque ti est lui-même un terme (qui peut être construit en utilisant un symbole de fonction). Les termes fonctionnels peuvent être utilisés à l'intérieur d'un prédicat comme un argument. Le langage résultant est appelé **DATALOG^{fonc}**.

Notion XV.14 : DATALOG avec fonction (*DATALOG^{fun}*)

Version étendue de DATALOG dans laquelle un terme argument de prédicat peut être le résultat de l'application d'une fonction à un terme.

Ainsi, des prédicats tels que P(a,x,f(x),f(g(x,a))) où f est une fonction unaire et g une fonction binaire sont acceptés. La sémantique de DATALOG doit alors être complétée pour intégrer les fonctions. Cela s'effectue comme en logique, en faisant correspondre à chaque fonction n-aire une application de D^n dans D, D étant le domaine d'interprétation.

Les fonctions sont très utiles en pratique pour effectuer des calculs. Par exemple, un problème de cheminement avec calcul de distance sur un graphe pourra être exprimé comme suit :

{ CHEMIN(x,y,d) ← ARC(x,y,d) ;
  CHEMIN(x,y,d+e) ← CHEMIN(x,z,e), ARC(z,y,d) }

La recherche des longueurs de tous les chemins allant de Paris à Marseille s'effectuera par la requête ? `CHEMIN(Paris,Marseille,x)`.

Un problème qui devient important avec DATALOG^{fonc} est celui de la finitude des réponses aux questions (ou des relations déduites). Une question est **saine** (en anglais *safe*) si elle a une réponse finie indépendamment des domaines de la base (qui peuvent être finis ou infinis). Le problème de déterminer si une question est saine existe déjà en DATALOG pur. Si les domaines sont infinis, un programme DATALOG peut générer des réponses infinies. Par exemple, le programme:

{ SALAIRE(100) ;
  SUPERIEUR(x,y) ← SALAIRE(x), x < y ;
  ? SUPERIEUR(x,y) }

génère une réponse infinie. Pour éviter des programmes a modèle infini, une caractérisation syntaxique des programmes sain a été proposée [Zaniolo86]. Cette caractérisation est basée sur la notion de **règle à champ restreint**. Une règle est à champ restreint si toutes les variables figurant en tête de règle apparaissent dans un prédicat relationnel dans le corps de la règle. Par exemple, la règle `SUPERIEUR(x,y) ← SALAIRE(x), x < y` n'est pas à champ restreint car y n'apparaît pas dans un prédicat relationnel dans le corps de la règle. Si toutes les règles d'un programme DATALOG sans fonction sont à champ restreint, alors le programme est sain et ne peut générer des réponses infinies.

Avec des fonctions, le problème de savoir si un programme est sain est plus difficile. Par exemple, le programme :

{ ENTIER(0);
  ENTIER(x+1) ← ENTIER(x) }

n'est pas sain car il génère un prédicat infini (les entiers positifs sont générés dans ENTIER). Cependant, ce programme est à champ restreint. Notez cependant que la question ? ENTIER(10) a une réponse finie unique (vrai). Vous trouverez une méthode générale pour déterminer si un programme avec fonctions est sain dans [Zaniolo86].

En conclusion, il est intéressant de remarquer qu'il est possible d'étendre l'algèbre relationnelle avec des fonctions [Zaniolo85], comme vu dans le chapitre XI sur le modèle objet. En fait, il est nécessaire d'inclure des fonctions dans les qualifications de jointures et restrictions (les qualifications sont alors des expressions de logique du premier ordre avec fonctions). Il est aussi nécessaire d'inclure des fonctions dans les critères de projection. On projette alors sur des termes fonctionnels calculés à partir d'attributs. Le plus petit modèle d'un programme DATALOG^{fonc} peut alors être calculé par un programme utilisant des boucles d'expressions d'algèbre relationnelle sans différence. Ainsi, DATALOG^{fonc} a la puissance de l'algèbre relationnelle avec fonction sans différence, mais avec la récursion. Pour avoir la différence, il faut passer à DATALOG^{fonc,neg} et pour avoir les mises à jour à DATALOG^{fonc,*}.

4.5. SUPPORT DES ENSEMBLES

Une caractéristique intéressante des langages de manipulation de bases de données relationnelles comme SQL est la possibilité de manipuler des ensembles à travers des fonctions agrégats. Plusieurs auteurs ont proposé d'introduire les ensembles dans les langages de règles [Zaniolo85, Beeri86, Kupper86]. Le but est de supporter des attributs multivalués contenant des ensembles de valeurs. DATALOG étendu avec des ensembles est appelé **DATALOG^{ens}**.

Notion XV.15 : DATALOG avec ensemble (*DATALOG^{set}*)

Version étendue de DATALOG permettant de manipuler des ensembles de valeurs référencés par des variables ensembles.

Afin d'illustrer l'intérêt des ensembles, considérons la relation de schéma PIECE(COMPOSE, COMPOSANT) dont un tuple <a,b> exprime le fait que a est composé avec le composant b. Pour une pièce donnée a, il existe autant de tuples que de composants de a dans cette relation. Trouver toutes les sous-pièces de chaque pièce et les répertorier dans une relation COMPOSE(PIECE, ENSEMBLE_DE_SOUS-PIECE) est une opération intéressante, possible dès que l'on tolère des attributs multivalués. Une telle opération est appelée **groupage** (en anglais, *nest*). L'opération inverse est le **dégroupage** (en anglais, *unest*). Ces deux opérations déjà étudiées dans le chapitre XI sur le modèle objet sont illustrées figure XV.11.

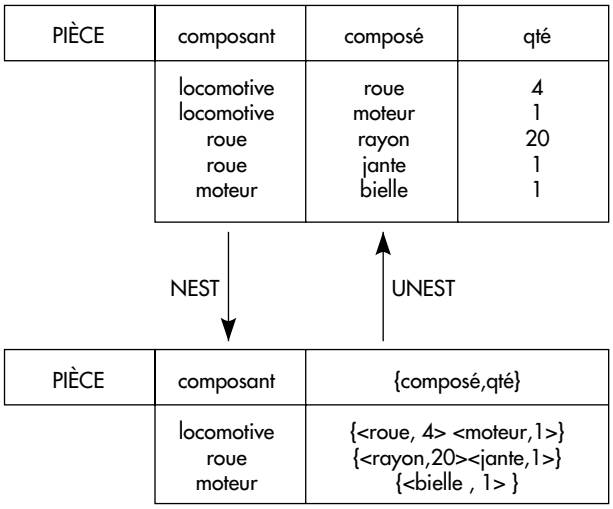


Figure XV.11 : Opérations de groupage et dégroupage

Il existe plusieurs manières d'introduire les ensembles dans un langage de règles. Il est possible d'introduire de nouveaux domaines dans la base de données intentionnelle dont les valeurs sont des ensembles de constantes. Cependant, il est sage d'interdire les ensembles d'ensembles, donc de se limiter à un niveau d'ensemble ; dans le cas contraire, il serait possible de générer des ensembles d'ensembles à des profondeurs infinies. Pour éviter de confondre les variables qui référencent des ensembles avec celles qui référencent des valeurs simples, nous utiliserons des majuscules pour les variables simples et des minuscules pour les variables simples, comme [Kuper86].

Pour accomplir l'opération de groupage, un opérateur spécial appelé groupage peut être introduit comme dans [Beeri86]. Le groupage range dans un ensemble toutes les valeurs des variables en arguments qui satisfont la condition du corps de règle. Appliqué à une variable x, le groupage est noté <x>. Ainsi, la relation `COMPOSE` résultat du groupage de la relation `PIECE` illustrée figure XV.11 peut être déclarée comme suit :

COMPOSE (x,Y) ← PIECE(x,y), Y = <y>.

Etant donné des ensembles et des éléments, un prédicat d'appartenance, noté $\in$, peut être introduit. En utilisant ce prédicat, le dégroupage illustré figure XV.11 est exécuté par la règle :

PIECE(x,y) ← COMPOSE(x,Y), y $\in$ Y.

Afin de comparer des ensembles, d'autres prédicats classiques peuvent être introduits, comme l'inclusion stricte notée $\subset$. Des fonctions sur ensembles sont aussi possibles, telles que les fonctions agrégats classiques `COUNT`, `MIN`, `MAX`, `AVG`… qui délivrent des valeurs simples, ou des fonctions binaires entre ensembles qui délivrent des ensembles ($\cup$, $\cap$, –).

Remarquons que le groupage ne donne pas plus de puissance au langage que les fonctions interprétées et la négation. En fait, si l'on définit la fonction unaire interprétée qui à un élément fait correspondre un ensemble composé de cet élément { }: $x \rightarrow \{x\}$ et la fonction binaire interprétée sur ensemble $\cup$: $X,Y \rightarrow X \cup Y$ (c'est-à-dire l'union classique), il est possible d'effectuer le groupage par les règles suivantes :

SOUS-PIECE (x,Y) ← PIECE (x,y), Y = {y}
SOUS-PIECE (x,Y) ← SOUS-PIECE (x,Z), SOUS-PIECE (x,T), Y = Z ∪ T
COMPOSE (x,Y) ← SOUS-PIECE (x,Y), ¬SOUS-PIECE (x,Z), Z ⊂ Y

En clair, la deuxième règle fait l'union de tous les éléments composant chaque pièce x et la dernière garde le plus grand ensemble de sous-pièces Y ainsi généré. Il va de soi qu'utiliser l'opérateur de groupage est plus simple ; aussi, si le système réalise efficacement cet opérateur, il sera probablement plus performant pour effectuer le groupage qu'en exécutant les trois règles ci-dessus.

Pour conclure sur les ensembles, notons que l'introduction d'ensembles en DATALOG nécessite la stratification pour définir la sémantique d'un programme sans ambiguïté. Cela provient du fait que les ensembles intègrent un cas particulier de négation. La sémantique de la stratification nécessaire peut être exprimée comme suit : « calculer tout ensemble avant d'utiliser son contenu ». Ainsi, les règles doivent être partiellement ordonnées en strates dans lesquelles les opérations de groupage sont effectuées dès que possible. Si des groupages sont effectués dans des cycles de calculs de prédicats récursifs, le programme n'est pas stratifiable et a une sémantique ambiguë : il doit être rejeté.

5. ÉVALUATION DE REQUÊTES DATALOG

Cette section présente les techniques de base pour évaluer des questions sur des prédicats dérivés définis en Datalog.

5.1. ÉVALUATION BOTTOM-UP

La solution la plus simple pour répondre à une question portant sur un prédicat déduit par un programme DATALOG est de calculer ce prédicat, puis de le filtrer avec la question. Le calcul du prédicat peut se faire par calcul de point fixe comme vu formellement ci-dessus, lors de l'étude de la sémantique du point fixe. Ce calcul commence à partir des prédicats de base contenant les faits et génère des faits dans les prédicats dérivés par application successive des règles. Pour appliquer une règle, les variables

sont instanciées avec des faits connus. Cela nécessite d'évaluer la condition composant le corps de règle sur les faits connus ; pour chaque instance des variables satisfaisant la condition, l'action (ou les actions) définie par le prédicat de tête est exécutée. La procédure de génération est appliquée jusqu'à saturation, c'est-à-dire jusqu'au point où aucune règle ne peut produire de nouveau fait. Une telle procédure est connue en intelligence artificielle comme la génération en **chaînage avant** [Nilsson80]. Elle est résumée figure XV.12.

```
{ Tant que « Il existe une relation dérivée R non saturée » faire{
   « sélectionner une règle r, dont l'action s'applique sur R » ;
   pour chaque « tuple de la base satisfaisant la condition de r » faire
      « exécuter les actions de r » ;
   }
} ;
```

Figure XV.12 : Génération en chaînage avant

L'approche proposée part des données pour élaborer la réponse à l'usager. Pour cette raison, elle est souvent appelée **méthode d'évaluation *bottom-up***.

Notion XV.16 : Evaluation Bottom-up (Bottom-up evaluation)

Technique d'évaluation partant des tuples de la base de données, consistant à appliquer les règles en avant jusqu'à saturation pour générer la réponse à la question finalement obtenue par filtrage des données générées.

Une illustration de la génération *bottom-up* apparaît figure XV.13.

Une technique d'évaluation *bottom-up* calcule le plus petit modèle d'un programme logique ; donc, elle génère la base intentionnelle. La question est appliquée à la base intentionnelle. L'ordre et la manière selon lesquels les règles sont appliquées sont importants pour au moins deux raisons :

1. Ils peuvent changer les performances du processus de génération. Puisque les règles peuvent interagir de différentes manières (le résultat d'une règle peut changer la valeur de la condition d'une autre), il est souhaitable de choisir un ordre. Dans le cas de DATALOG pur, le problème du choix de l'ordre des règles est une extension de l'optimisation de question des SGBD relationnels.
2. Dans le cas de règles avec négations ou ensembles et de programmes stratifiés, il est nécessaire de générer chaque strate dans l'ordre de stratification pour obtenir un modèle correct. Plus complexe, avec des langages de règles de production (règles avec mises à jour), le résultat peut dépendre de l'ordre d'application des règles.

PROGRAMME DATALOG AVEC QUESTION

(r1) PARENT(x,z) ← MERE(x,z)
(r2) PARENT(x,z) ← PERE(x,z)
(r3) GRANPARENT(x,z) ← PARENT(x,y), PARENT(y,z)
(q) ? GRANDPARENT(x, Jean)

BASE EXTENSIONNELLE

MÈRE	ASC	DESC
	Marie	Jean
	Julie	Marie
	Julie	Jack

PÈRE	ASC	DESC
	Pierre	Chris
	Ted	Marie
	Ted	Jack
	Jef	Pierre

APPLICATION DE r1

PARENT	ASC	DESC
	Marie	Jean
	Julie	Marie
	Julie	Jack

APPLICATION DE r2

PARENT	ASC	DESC
	Marie	Jean
	Julie	Marie
	Julie	Jack
	Pierre	Chris
	Ted	Marie
	Ted	Jack
	Jef	Pierre

APPLICATION DE r3

GRANDPARENT	ASC	DESC
	Julie	Jean
	Ted	Jean
	Jef	Chris

APPLICATION DE q

GRANDPARENT	ASC	DESC
	Julie	Jean
	Ted	Jean

Figure XV.13 : Exemple de génération bottom-up

5.2. ÉVALUATION TOP-DOWN

Au lieu de partir de la base extensionnelle pour générer les réponses aux questions, il est possible de partir de la question. Le principe est d'utiliser le profil de la question (nom de prédicat et valeurs connues) et de le remonter via les règles en **chaînage arrière** jusqu'à la base extensionnelle pour déterminer les faits capables de générer des réponses. Un tel procédé est connu sous le nom chaînage arrière en intelligence artificielle [Nilsson80]. Si des faits de la base sont retrouvés en utilisant le chaînage arrière, alors la question est satisfaite ; donc, une réponse oui/non est facile à élaborer

en chaînage arrière. Plus généralement, la remontée des constantes de la question vers les tuples de la base permet de générer des **faits relevants**, seuls capables de produire des réponses aux questions [Lozinskii86].

Notion XV.17 : Faits relevants (*Relevant facts*)

Tuples de la base extensionnelle qui participent à la génération d'au moins un tuple de la réponse à la question.

Les faits relevants peuvent être utilisés afin de générer toutes les réponses à la question si nécessaire, en appliquant une procédure de chaînage avant à partir de ces faits. Cependant, la technique est très différente de l'évaluation *bottom-up* puisqu'elle profite des constantes de la question pour réduire les espaces de recherche. La méthode part de la question de l'usager pour remonter aux faits de la base. En conséquence, elle est appelée **méthode *top-down***.

Notion XV.18 : Évaluation *top-down* (*Top-Down Evaluation*)

Technique d'évaluation partant de la question et appliquant les règles en arrière pour dériver la réponse à la question à partir des faits relevants.

Une évaluation *top-down* de la question `PARENT(x,Jean)` est représentée figure XV.14. La question `GRANDPARENT(x,Jean)` de la figure XV.13 est plus difficile à évaluer. En première approche, tous les faits seront considérés comme relevants du fait de la règle (r3) (aucune variable ne peut être instanciée en chaînage arrière dans la première occurrence du prédicat `PARENT`). Le problème de déterminer précisément les faits relevants est difficile et nous l'étudierons plus généralement pour les règles récursives dans la section suivante.

La méthode d'évaluation *top-down* est formellement basée sur la méthode de résolution. Soit ? R(a,y) une question portant sur un prédicat dérivé R. Toute règle du type suivant :

$$R(..) \leftarrow B1,B2...Q1,Q2...$$

dont la conclusion peut être unifiée par une substitution μ avec la question permet de calculer une résolvante de la question. Pour chaque règle de ce type, une sous-question $\{B1,B2...Q1,Q2...\}_{[\mu]}$ est générée. Le processus doit être répété pour les sous-questions $Q1_{[\mu]}$, $Q2_{[\mu]}$... Dans le cas où aucune relation n'est récursive, on finit toujours par aboutir à des sous-questions portant sur les prédicats de base B1,B2... qui peuvent être évaluées par le SGBD ; la collecte des résultats permet d'élaborer les réponses à la question initiale.

En résumé, la dérivation *top-down* transmet les constantes depuis la question vers la base afin de filtrer les faits relevants. Malheureusement, la remontée des constantes qui réduit grandement les espaces de recherche n'est en général pas simple. Elle

demande de comprendre les connexions entre les règles dans un programme de règles, c'est-à-dire les unifications possibles entre prédicats. Dans ce but, plusieurs représentations par des graphes d'un programme DATALOG ont été proposées.

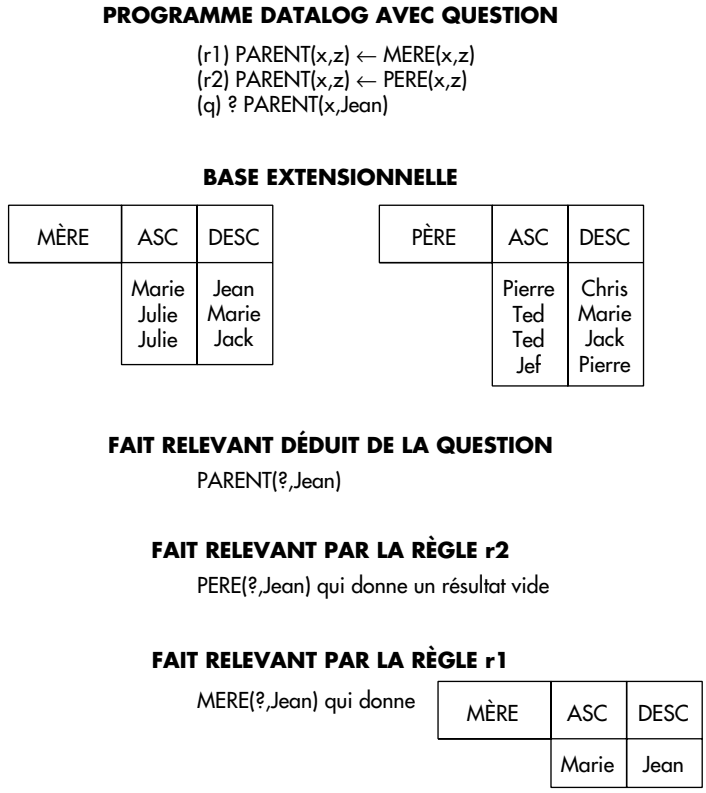


Figure XV.14 : Exemple d'évaluation top-down

6. LA MODÉLISATION DE RÈGLES PAR DES GRAPHES

Il est important de comprendre les connections entre les règles d'un programme. Un modèle basé sur un graphe est généralement utilisé pour visualiser les liens entre règles. Un tel modèle capture les unifications possibles entre les prédicats en tête de

règles et ceux figurant dans les conditions. Il permet souvent d'illustrer le mécanisme d'optimisation-exécution ou de vérifier la cohérence des règles. Par exemple, des règles peuvent être contradictoires ou organisées de telle manière que certaines relations restent vides. Dans cette section, nous allons présenter les modèles graphiques les plus connus.

6.1. ARBRES ET GRAPHES RELATIONNELS

Toute règle peut être interprétée comme une production relationnelle. Les conditions dans le corps de règle représentent des restrictions (conditions de la forme x Θ v, où Θ est un opérateur de comparaison et v une valeur), des jointures (présence d'une même variable dans deux prédicats relationnels), des différences (négation de prédicats relationnels). Les implications correspondent à des projections. Plusieurs règles générant le même prédicat correspondent à une union. Il est donc possible de représenter un programme de règle DATALOG^{neg} par un graphe d'opérations relationnelles. Le graphe est un arbre dans le cas où aucun prédicat n'est récursif.

Le type de nœud correspondant à chaque opération relationnelle est représenté figure XV.15. Une duplication est simplement une recopie en un ou plusieurs exemplaires d'un résultat intermédiaire. Une addition est une union d'une relation avec la relation cible (cas particulier d'union dans laquelle le résultat est cumulé dans une des deux relations). Un arbre est généré par combinaison de ces nœuds. La figure XV.16 illustre la méthode de construction pour les règles suivantes :

(r1) PARENT(x,z) ← MERE(x,t,z), t >16
(r2) PARENT(x,z) ← PERE(x,t,z), t > 16
(r3) ANCETRE(x,z) ← PARENT(x,z)
(r4) ANCETRE(x,z) ← ANCETRE(x,y), PARENT(y,z)
(r5) REPONSE(x) ← ANCETRE(x,toto)

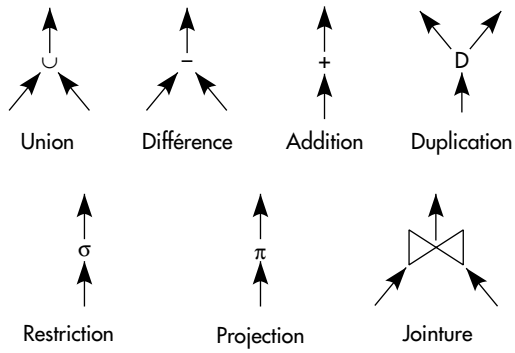


Figure XV.15 : Opérateurs de l'algèbre relationnelle utilisés

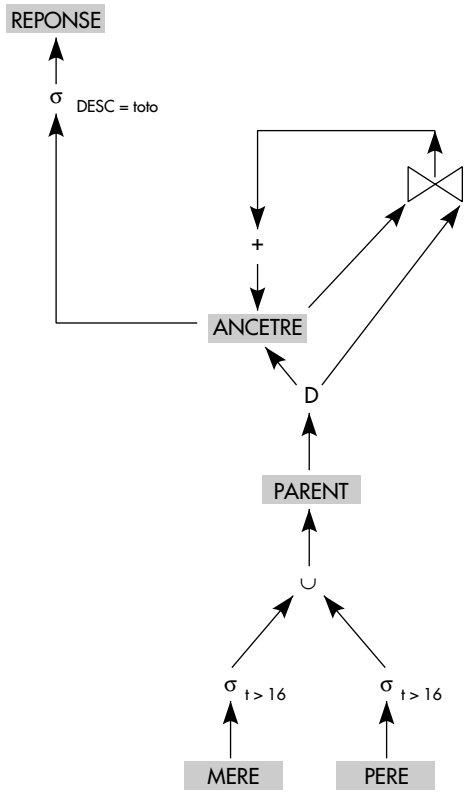


Figure XV.16 : Calcul des ancêtres par un graphe d'opérateurs relationnels

Notez qu'en général, la possibilité de compléter le graphe par un opérateur de duplication permet d'éviter de dupliquer des branches similaires de l'arbre et d'optimiser l'exécution correspondante. Par exemple, si l'on ajoute la règle :

(r0) AIME(x,z) ← MERE(x,t,z), t >16

il n'est pas nécessaire de dupliquer toute la branche de restriction sur mère, mais seulement d'introduire un opérateur de duplication au niveau du résultat de cette branche. Celui-ci génère les tuples de AIME. Soulignons aussi que le nom d'un prédicat intermédiaire peut être omis lorsque ce dernier ne doit pas être gardé. Finalement, un graphe d'opérateurs relationnels donne un moyen de générer par simple traduction un programme *bottom-up* calculant la réponse à une question déductive. Tout le problème d'optimisation de questions déductives peut être vu comme un problème d'optimisation de graphes relationnels. Pour résumer, nous définirons informellement la notion de **graphe relationnel de règles** comme suit :

Notion XV.19 : Graphe relationnel de règles (*Relational Rule Graph*)

Graphe d'opérateurs relationnels représentant l'exécution bottom-up d'un programme de règles pour répondre à une question.

6.2. ARBRES ET/OU ET GRAPHES RÈGLE/BUT

Le modèle le plus connu vient de l'intelligence artificielle, où il est utilisé pour représenter l'exécution de règles en chaînage arrière. Il s'agit des **arbres ET/OU**.

Notion XV.20 : Arbre ET/OU (*AND/OR tree*)

Arbre composé de deux sortes de nœuds, les nœuds OU représentant les prédicats et les nœuds ET représentant les règles, dans lesquels la racine représente une question et les arcs l'évaluation top-down de la question.

Étant donné un programme DATALOG sans négation et une question, un arbre ET/OU est construit comme suit. Chaque occurrence de prédicat est représentée par un ou plusieurs nœuds OU. Une règle correspond à un ou plusieurs nœuds ET. La racine de l'arbre est le prédicat de la question, donc un nœud OU. Les enfants d'un nœud OU sont toutes les règles dont la tête s'unifie avec le prédicat représenté par le nœud OU ; ce sont donc des nœuds ET. Les enfants d'un nœud ET sont toutes les occurrences des prédicats relationnels qui apparaissent dans le corps de la règle représentée. En principe, l'arbre est développé jusqu'à ce que les prédicats de la base extensionnelle apparaissent comme des feuilles. Pour spécifier les unifications (passages de paramètres) effectuées lorsque l'on passe d'une règle à une autre, la substitution qui unifie la tête de règle avec le prédicat parent (apparaissant dans le corps de la règle précédente) peut être mémorisée comme une étiquette sur l'arc allant du nœud représentant la règle au prédicat s'unifiant avec sa tête.

Par exemple, considérons le programme DATALOG suivant :

(r1) PARENT(x,z) ← MERE(x,t,z)
(r2) PARENT(x,z) ← PERE(x,t,z)
(r3) GRANDPARENT(x,z) ← PARENT(x,y), PARENT(y,z)
(q) ? GRANDPARENT(x,Jean)

L'arbre ET/OU associé à la résolution de la question (q) en dérivation top-down est représenté figure XV.17. Un nœud ET est représenté par un rectangle et un nœud OU par une ellipse.

Un arbre ET/OU montre la propagation des constantes depuis la question vers les relations de la base extensionnelle en chaînage arrière. Malheureusement, dans le cas de règles récursives, un arbre ET/OU peut devenir infini car de nouvelles occurrences

de règles sont ajoutées pour développer les nœuds qui correspondent à des relations récursives. Nous développerons la récursion dans la section suivante. Cependant, il apparaît déjà que des graphes plus sophistiqués sont nécessaires pour représenter les règles récursives.

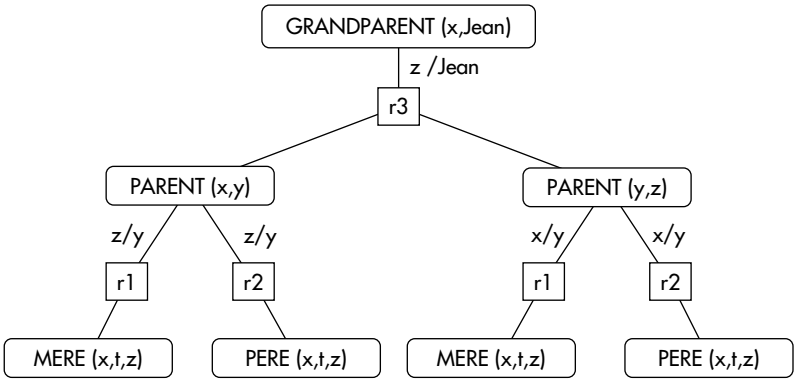


Figure XV.17 : Exemple d'arbre ET/OU

Une extension de l'arbre ET/OU pour éviter les branches infinies en cas de prédicats récursifs est le graphe règle/but. Un graphe règle/but est aussi associé à une question qui spécifie un prédicat à évaluer. Il contient en outre des nœuds circulaires représentant les prédicats et des nœuds rectangulaires représentant les règles. Les seuls arcs d'un graphe règle/but sont définis par la règle suivante : s'il existe une règle r de la forme P ← P1, P2… Pn dans le programme de règles, alors il existe un nœud allant du nœud r au nœud P et, pour chaque Pi, il existe un arc allant du nœud Pi au nœud r. Dans sa forme la plus simple, un **graphe règle/but** peut être défini comme une variation d'un graphe ET/OU :

Notion XV.21 : Graphe règle/but (*Rule/goal graph*)

Graphe représentant un ensemble de règles dérivé d'un arbre ET/OU en remplaçant l'expansion de tout prédicat dérivé déjà développé dans l'arbre par un arc cyclique vers la règle correspondant à ce développement.

Ainsi, un graphe règle/but ne peut être infini. En présence de règles récursives, il contient simplement un cycle. Il correspond donc à un graphe ET/OU dans lequel une expansion déjà faite est remplacée par une référence à cette expansion. La figure XV.18 représente le graphe règle/but correspondant au programme suivant :

{ (r1) PARENT(x,z) ← MERE(x,t,z), t >16
(r2) PARENT(x,z) ← PERE(x,t,z), t > 16
(r3) ANCETRE(x,z) ← PARENT(x,z)
(r4) ANCETRE(x,z) ← ANCETRE(x,y), PARENT(y,z) }.

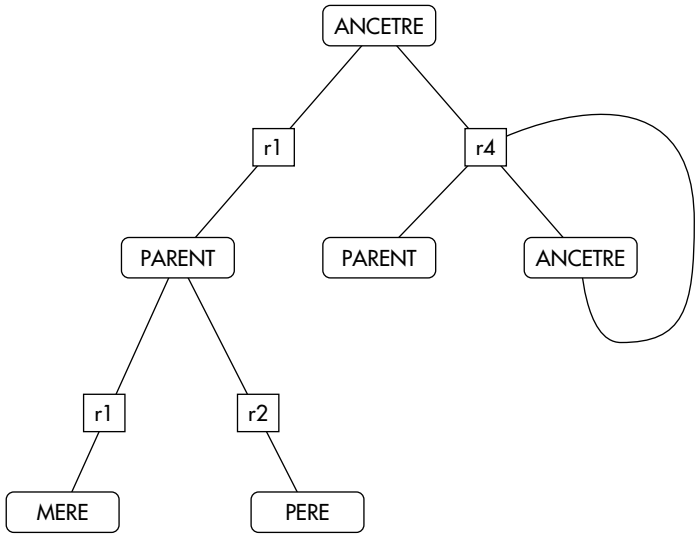


Figure XV.18 : Exemple de graphe règle/but

6.3. AUTRES REPRÉSENTATIONS

Plusieurs autres représentations graphiques d'un ensemble de règles ont été proposées, parmi lesquelles les **graphes de connexion de prédicats** (PCG) [McKay81] et les **réseaux de Petri à prédicats** (PrTN) [Gardarin85, Maindreville87].

Un **graphe de connexion de prédicats** (PCG) modélise toutes les unifications possibles entre les prédicats d'un programme de règles. Plus précisément, une règle est modélisée comme un nœud multiple du PCG ; un tel nœud est représenté par un rectangle contenant le prédicat de tête puis les prédicats du corps. Un arc représente une unification entre un prédicat apparaissant en tête de règle et un prédicat apparaissant dans le corps d'une règle ; un arc est orienté et étiqueté par la substitution de variables réalisant l'unification. La figure XV.19 représente le PCG associé au programme :

(r1) PARENT(x,z) ← MERE(x,t,z)
(r2) PARENT(x,z) ← PERE(x,t,z)
(r3) ANCETRE(x,z) ← PARENT(x,z)
(r4) ANCETRE(x,z) ← ANCETRE(x,y), PARENT(y,z)

Un PCG actif correspond à un parcours simulant le chaînage arrière du PCG à partir d'une question représentée comme un nœud singulier sans tête [McKay81].

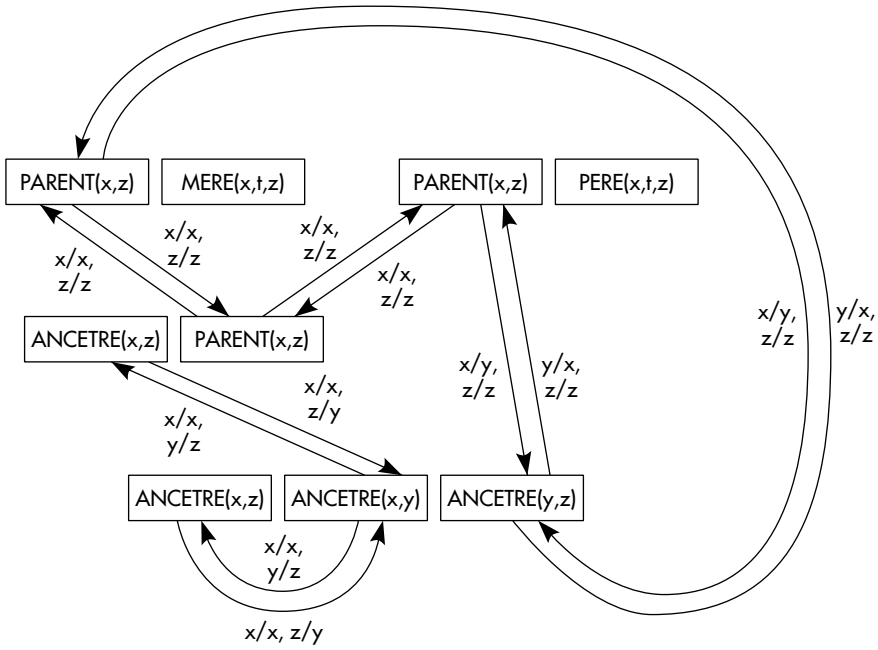


Figure XV.19 : Un exemple de PCG

Un **réseau de Petri à prédicats** (*PrTN*) modélise directement une évaluation *bottom-up* (en chaînage avant) d'un programme de règles. Chaque relation (prédicat intentionnel ou extentionnel) est modélisée par une place ; les jetons de marquage correspondent aux tuples des relations. Chaque règle est modélisée par une transition ; le prédicat correspond à la condition de la règle. Les règles génèrent en sortie de nouveaux jetons (tuples) ; pour repérer quels jetons sont générés, on étiquette en général les arcs sortant des transitions avec des variables qui représentent les attributs des tuples générés. Les arcs entrants sont aussi étiquetés afin de faciliter l'écriture des conditions. Une exécution du PrTN modélise un calcul de point fixe en chaînage avant. Afin d'éviter la destruction des tuples dans les places lors d'un déclenchement d'une transition, il faut ajouter un arc qui remet le jeton après exécution de la transition. Pour alléger, nous notons par un point le fait que le jeton doit être conservé dans sa place lors de l'exécution d'une transition. L'absence de point permet de modéliser les mises à jour par les règles. La figure XV.20 illustre le PrTN correspondant au programme de règles déjà modélisé figure XV.18 et XV.19.

La figure XV.21 présente une synthèse des principales caractéristiques des différents modèles graphiques introduits ci-dessus. Notez que l'élément déterminant apparaît finalement être l'orientation top-down (dirigée par les questions) ou bottom-up (dirigée par les données) du modèle. Différentes variantes de ces modèles ont aussi été

proposées ; elles permettent en général de mieux prendre en compte une fonctionnalité du langage de règles modélisé.

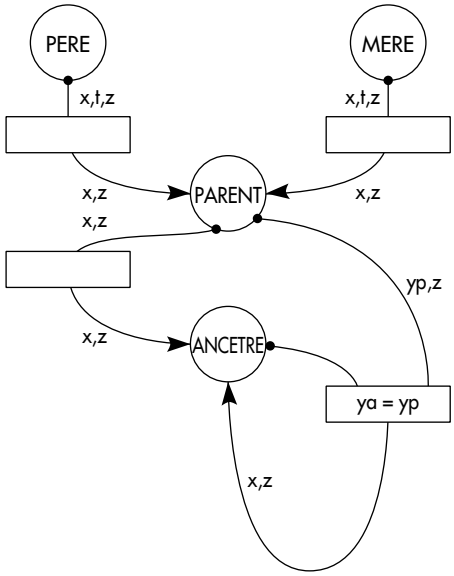


Figure XV.20 : Un exemple de PrTN

Modèle / Caract.	Graphe Relationnel	Graphe Règle/but	PCG Actif	PrTN Etendu
Fonction	oui	non	non	oui
Negation	oui	non	non	oui
Mise à jour	oui	non	non	oui
Condition Non Horn	oui	non	non	oui
Bottom up	oui	non	non	oui
Top Down	non	oui	oui	non
Recursion	oui	oui	oui	oui

Figure XV.21 : Comparaison des modèles de graphes

7. EVALUATION DES REGLES RECURSIVES

Cette section présente les techniques essentielles permettant d'optimiser des requêtes dans le cas de règles récursives.

7.1. LE PROBLÈME DES RÈGLES RÉCURSIVES

Le problème de l'optimisation et de l'évaluation de requêtes portant sur des prédicats dérivés récursifs est étudié depuis longtemps [Gallaire78, Minker82]. Il s'agit d'un problème difficile et important compte tenu de l'influence de la récursion, notamment dans les applications d'intelligence artificielle. [Bancilhon86b] expose les premières solutions connues.

Un exemple typique, déjà vu ci-dessus, de prédicat dérivé récursif est la définition des ancêtres `ANC` à partir d'une relation de base parent notée `PAR(PARENT,ENFANT)` comme suit :

(r1) ANC(x,y) ← PAR(x,y)
(r2) ANC(x,y) ←PAR(x,z), ANC(z,y).

Un tel exemple est **linéaire** car la relation `ANC` est définie en fonction d'une seule occurrence de la relation ANC dans la seule règle récursive (r2). Il est possible de donner une définition non linéaire des ancêtres en remplaçant la règle r2 par :

ANC(x,y) ← ANC(x,z),ANC(z,y)

Cette règle est quadratique car `ANC` apparaît deux fois dans le corps.

Un exemple typique est celui du calcul des cousins de même génération. Pour initialiser la relation `MG(PERSONNE,PERSONNE)` définissant les cousins de même génération, il est commode de constater que chacun est son propre cousin de même génération. Toutes les personnes connues étant stockées dans une relation unaire HUM (humains), on obtient la règle d'initialisation :

(r1) MG(x,x) ← HUM(x)

Notez que la relation humain (`HUM`) peut être obtenue par les deux règles :

HUM(x) ← PAR(x,y)
HUM(y) ← PAR(x,y)

Pour compléter la relation `MG`, on dira qu'une personne x est le cousin de même génération qu'une personne y si deux de leurs parents sont aussi cousins de même génération. On aboutit alors à la règle :

(r'2) MG(x,y) ← PAR(xp,x),MG(xp,yp),PAR(yp,y)

Il existe plusieurs autres manières de définir les cousins de même génération. En particulier, comme MG est une relation symétrique (`MG(x,y)` ⇔ `MG(y,x)`), il est

possible d'intervertir les variables xp et yp à intérieur de MG dans r2. Cela conduit à la définition suivante des cousins de même génération :

(r1) MG(x,x) ← HUM(x)
(r2) MG(x,y) ← PAR(xp,x),MG(yp,xp),PAR(yp,y)

Un exemple plus complexe car **quadratique** est la définition des personnes au même niveau hiérarchique par un prédicat déduit MC(EMPLOYE,EMPLOYE), spécifiée à partir d'une relation SER(SERVICE,EMPLOYE) et d'une relation CHEF(EMPLOYE,EMPLOYE) :

(r1) MSER(x,y) ← SER(s,x),SER(s,y)
(r2) MC(x,y) ← MSER(x,y)
(r3) MC(x,y) ← CHEF(x,z1),MC(z1,z2),MSER(z2,z3),MC(z3,z4),CHEF(y,z4)

Cet exemple est quadratique car la règle récursive r3 définit le prédicat dérivé MC en fonction de deux occurrences du même prédicat. Elle est illustrée figure XV.22. Il est évidemment possible de définir la relation MC de manière plus simple, la complexité ayant ici pour but de mieux illustrer dans des cas non triviaux les différents algorithmes que nous verrons ci-dessous.

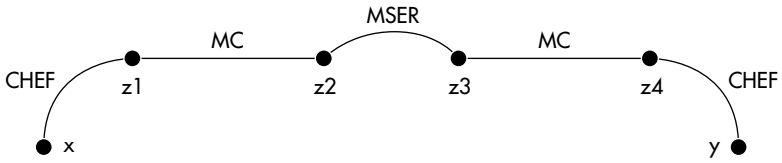


Figure XV.22 : Définition de la relation récursive Mêmes Chefs

Il est aussi possible d'utiliser DATALOG^{fonc} afin de définir des prédicats récursifs avec calculs de fonctions. Des exemples typiques sont dérivés des problèmes de parcours de graphes. Un graphe valué peut être représenté par une relation :

ARC (SOURCE, CIBLE, VALUATION).

Chaque tuple de la relation représente un arc. Les règles suivantes déduisent une relation CHEMIN représentant tout les chemins du graphe, avec une valuation composée calculée par une fonction f :

CHEMIN(x,y,z) ← ARC(x,y,z)
CHEMIN(x,y,f(z1,z2)) ← ARC(x,z,z1), CHEMIN(z,y,z2)

Par exemple, si f(z1,z2) = z1 + z2 et si les valuations représentent des distances, les règles précédentes définissent tous les chemins du graphe avec leur longueur associée.

Une autre application possible des fonctions dans les règles récursives est la manipulation d'objets structurés. Par exemple, des listes peuvent être manipulées avec la fonction concaténation de deux listes X et Y, notée X|Y. La base de données peut être composée d'un prédicat définissant l'ensemble des listes constructibles à partir de n

éléments. On spécifie l'ajout (par la fin) d'une liste X à une liste Y comme étant une liste Z définie par le prédicat `AJOUT(X,Y,Z)`, puis l'inversion des éléments d'une liste X comme étant une liste Y définie par le prédicat `INVERSE(X,Y)`. Les règles suivantes définissent ces prédicats dérivés ([] désigne la liste vide) :

(r1) AJOUT(X,[],X|[]) ← LISTE(X)

(r2) AJOUT(X,W|Y,W|Z) ← AJOUT(X,Y,Z),LISTE(W)

(r3) INVERSE([],[]) ←

(r4) INVERSE(W|X,Y) ← INVERSE(X,Z),AJOUT(W,Z,Y)

`AJOUT` et `INVERSE` sont ainsi deux prédicats récursifs linéaires.

Le problème qui se pose est bien sûr d'évaluer des questions portant sur des prédicats récursifs. Ces questions spécifient en général des constantes pour certaines variables, ou des ensembles de constantes (question avec ≤, <, ≥,>). Voici quelques questions typiques sur les prédicats définis précédemment :

(1) Qui sont les ancêtres de Jean ?
? ANC(x,Jean)

(2) Pierre est-il un ancêtre de Jean ?
? ANC(Pierre,Jean)

(3) Qui sont les cousins de même génération de Jean ?
? MG(Jean,x)

(4) Qui sont les chefs au même niveau que Jean ?
? MC(Jean,x)

(5) Quel est l'inverse de la liste [1,2,3,4,5] :
? INVERSE([1,2,3,4,5], x)

Il est bien sûr possible de remplacer par des constantes tout sous-ensemble de paramètres afin de retrouver les autres (cependant, selon les instanciations, DATALOG^{fonc} peut conduire à des réponses infinies comme vu ci-dessus), ou encore d'instancier tous les paramètres afin d'obtenir une réponse booléenne du type VRAI/FAUX.

Dans la suite, nous allons étudier les principales solutions proposées, en commençant par les solutions simples, en général peu performantes ou incomplètes. Nous étudierons ensuite les solutions interprétées qui transforment progressivement la requête en sous-requêtes adressées au SGBD, puis les solutions compilées qui, dans un premier temps, génèrent un programme d'algèbre relationnelle et dans un deuxième temps demandent l'exécution de ce programme. Bien que décrivant les principales approches, notre étude est loin d'être complète. En particulier, nous ignorons des approches importantes basées sur des règles de réécriture [Chang81], des automates [Marque-Pucheu84] ou une méthode de résolution adaptée [Henschen84].

7.2. LES APPROCHES *BOTTOM-UP*

7.2.1. La génération naïve

La solution la plus naïve pour évaluer la réponse à une requête consiste simplement dans un premier temps à appliquer un chaînage avant direct à partir des prédicats de base jusqu'à saturation des prédicats dérivés. Dans un deuxième temps, une sélection sur les prédicats dérivés instanciés permet de retenir les tuples répondant à la question. Cette solution est connue sous le nom de **génération naïve**.

Notion XV.22 : Génération naïve (*Naïve Generation*)

Technique d'évaluation bottom-up calculant une relation déduite par application de toutes les règles à tous les tuples produits à chaque étape du processus de génération, jusqu'à saturation de la relation déduite.

Cette méthode de calcul de point fixe part donc des faits de la base et calcule les instances des prédicats dérivés par application des règles afin de répondre à la question. En général, une valeur R_0 est calculée pour le prédicat récursif à partir des règles d'initialisation. Puis, par un programme de l'algèbre relationnelle qui résulte d'une compilation élémentaire de la règle récursive, on effectue un calcul itératif $R = R \cup E(R)$, où E est l'expression de l'algèbre relationnelle traduisant la règle récursive. Le processus s'arrête quand l'application de E à R ne permet pas de générer de nouveau tuple : on a alors le point fixe du prédicat récursif défini par R = E(R). Ce point fixe existe en DATALOG [Aho-Ullman79]. Un programme de calcul naïf d'une relation récursive R s'écrit donc comme représenté figure XV.23, E étant une expression de l'algèbre relationnelle dérivée de la règle récursive.

```
Procedure Naïve (R) {
   R:= R0 ;
   Tant que « R change » Faire
      R := R ∪ E(R) ;
   }
```

Figure XV.23 : Programme de génération naïve

Par exemple, dans le cas des ancêtres, on initialisera ANC par la règle r1 avec les parents, puis on appliquera la règle r2 en effectuant une boucle de jointures de la relation ANC avec la relation PAR, et ce jusqu'à ce que l'on ne génère plus de nouvel ancêtre. Le calcul est illustré figure XV.24. Le processus sera similaire pour générer la relation MC : les règles r1 et r2 permettent de lui attribuer une valeur initiale MC_0, puis les jointures des relations CHEF,MC_0,MSER,MC_0,CHEF permettront de calculer MC_1. De manière itérative, les jointures de CHEF,MC_i,MSER,MC_i,CHEF permettront de calculer MC_{i+1}. Le calcul s'arrêtera quand aucun nouveau tuple sera généré : on aura

obtenu le point fixe de la relation MC qui décrit toutes les personnes de la base à un même niveau hiérarchique.

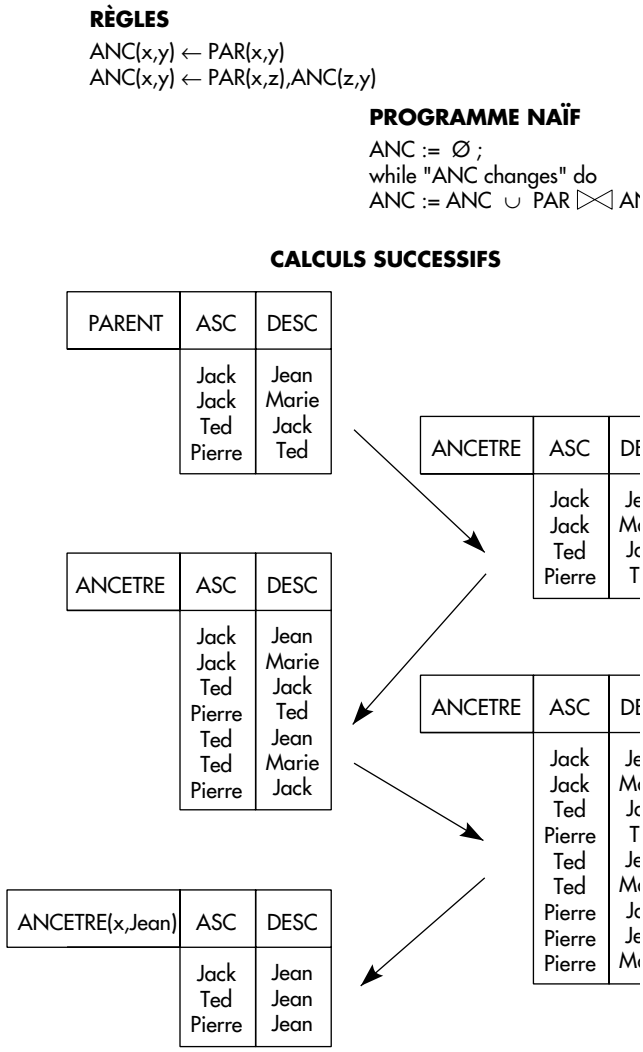


Figure XV.24 : Évaluation naïve des ancêtres de Jean

Si les programmes de règles sont en DATALOG pur, comme pour ANC et MC, il n'existe pas de fonction pour générer des valeurs qui ne sont pas dans la base : ceci garantit la finitude du processus de génération [Aho-Ullman79]. Dans le cas des listes, la stratégie *bottom-up* ne peut être appliquée telle quelle ; en effet, par ajout de listes, on génère des listes sans fin. Le processus de génération ne se termine pas.

Une étude plus détaillée du processus de génération naïf démontre plusieurs faiblesses de la méthode, outre le fait qu'une terminaison n'est garantie qu'en cas d'absence de fonction :

1. À chaque itération, un travail redondant est effectué, qui consiste à refaire les inférences de l'itération précédente ; cela provient du fait que l'on cumule les résultats avec les anciens et que l'on applique à nouveau la jointure à tous les résultats sans distinction.
2. Dans le cas où des constantes figurent dans la question, un travail inutile est effectué car la génération produit des résultats qui ne figurent pas dans la réponse ; ils sont éliminés par la sélection finale. Par exemple, pour calculer les seuls ancêtres de Jean, le processus conduit à générer les ancêtres de toutes les personnes de la base.

Voici maintenant les principales méthodes qui répondent à ces critiques.

7.2.2. La génération semi-naïve

Afin d'éviter des calculs redondants, il est possible de considérer à chaque itération les seules inférences nouvelles, c'est-à-dire celles incluant au moins un tuple généré à l'étape précédente qui n'existait pas déjà dans la relation dérivée. Ainsi, en utilisant les tuples nouveaux à l'itération i-1, il est possible de calculer les tuples nouveaux à l'itération i sans refaire des inférences déjà faites à l'étape i-1. Cette méthode est appelée **génération semi-naïve**.

Notion XV.23 : Génération semi-naïve (*Seminaïve generation*)

Technique d'évaluation bottom-up calculant une relation déduite par application itérative des règles seulement aux nouveaux tuples produits à chaque étape du processus d'inférence, jusqu'à saturation de la relation déduite.

Dans le cas de règles linéaires, l'approche consiste à remplacer à l'itération i la relation récursive R_i produite jusque-là par $\Delta R_i = R_i - R_{i-1}$. Si E(R) est l'expression de l'algèbre relationnelle représentant le corps de la règle récursive, on calcule alors $\Delta R_{i+1} = E(\Delta R_i) - R_i$, puis $R_{i+1} = R_i \cup \Delta R_{i+1}$. Dans le cas de règles non linéaires (par exemples quadratiques), le problème est plus complexe car il faut aussi considérer les inférences possibles entre les anciens tuples de R_i et les nouveaux de ΔR_i. Étant donné l'expression $R_{i+1} = E(R_i)$ où E est une expression de l'algèbre relationnelle, il est toujours possible de calculer $\Delta R_{i+1} = \delta E(R_i)$ par dérivation de E. On peut ainsi transformer un programme d'algèbre relationnelle effectuant un chaînage avant naïf en un programme effectuant un chaînage avant semi-naïf, ce qui permet d'éviter le travail redondant.

Par exemple, dans le cas des ancêtres, il suffira d'effectuer à chaque itération la jointure de la relation parent avec les nouveaux tuples ancêtres générés à l'étape précédente. Dans le cas de la relation MC, il faudra effectuer à l'itération i les jointures :

$$\text{CHEF} \bowtie \Delta\text{MC}_i \bowtie \text{MSER} \bowtie \text{MC}_i \bowtie \text{CHEF}$$
$$\text{CHEF} \bowtie \text{MC}_i \bowtie \text{MSER} \bowtie \Delta\text{MC}_i \bowtie \text{CHEF}$$
$$\text{CHEF} \bowtie \Delta\text{MC}_i \bowtie \text{MSER} \bowtie \Delta\text{MC}_i \bowtie \text{CHEF}$$

afin de considérer toutes les inférences nouvelles possibles, puis l'union des résultats et la différence avec les résultats déjà connus pour obtenir ΔR_{i+1}. Le processus différentiel apparaît donc ici comme assez lourd, mais il évite la jointure $\texttt{CHEF} \bowtie \texttt{MC}_i \bowtie \texttt{MSER} \bowtie \texttt{MC}_i \bowtie \texttt{CHEF}$.

Fondé sur les principes précédents, l'algorithme semi-naïf [Bancilhon86b] effectue donc un calcul différentiel de la relation récursive R comme représenté figure XV.25. Quand les mêmes tuples ne peuvent être produits deux fois par deux itérations différentes, la différence $\Delta R = \Delta R - R$ n'est pas nécessaire. Ce cas nécessite l'acyclicité des règles et des données [Gardarin87], comme dans le cas du calcul des ancêtres de Jean illustré figure XV.26 (page suivante).

```
Procedure SemiNaïve (R) {
   ΔR := R0
   R := ΔR ;
   Tant que ΔR ≠ ∅ faire {
      ΔR = ΔE(R,ΔR) ;
      ΔR = ΔR − R ;
      R = R ∪ ΔR ;
      };
   } ;
```

Figure XV.25 : Programme de génération semi-naïf

L'algorithme semi-naïf peut être implanté directement au-dessus d'un SGBD. Cependant, une telle implantation ne sera pas très efficace car l'algorithme accomplit des boucles de jointures, unions et différences afin d'atteindre la saturation. Pour améliorer les performances, une gestion de mémoire judicieuse tirera profit du fait que les relations dans la boucle restent les mêmes (R, ΔR, relations de E). Néammoins, pas plus que le chaînage avant naïf, le chaînage avant semi-naïf ne s'applique aux listes, ceci étant dû au problème de génération infinie.

7.3. DIFFICULTÉS ET OUTILS POUR LES APPROCHES TOP-DOWN

7.3.1. Approches et difficultés

Au lieu d'appliquer le chaînage avant pour générer la réponse à une question, il est possible de travailler en chaînage arrière « à la PROLOG ». Une telle approche part

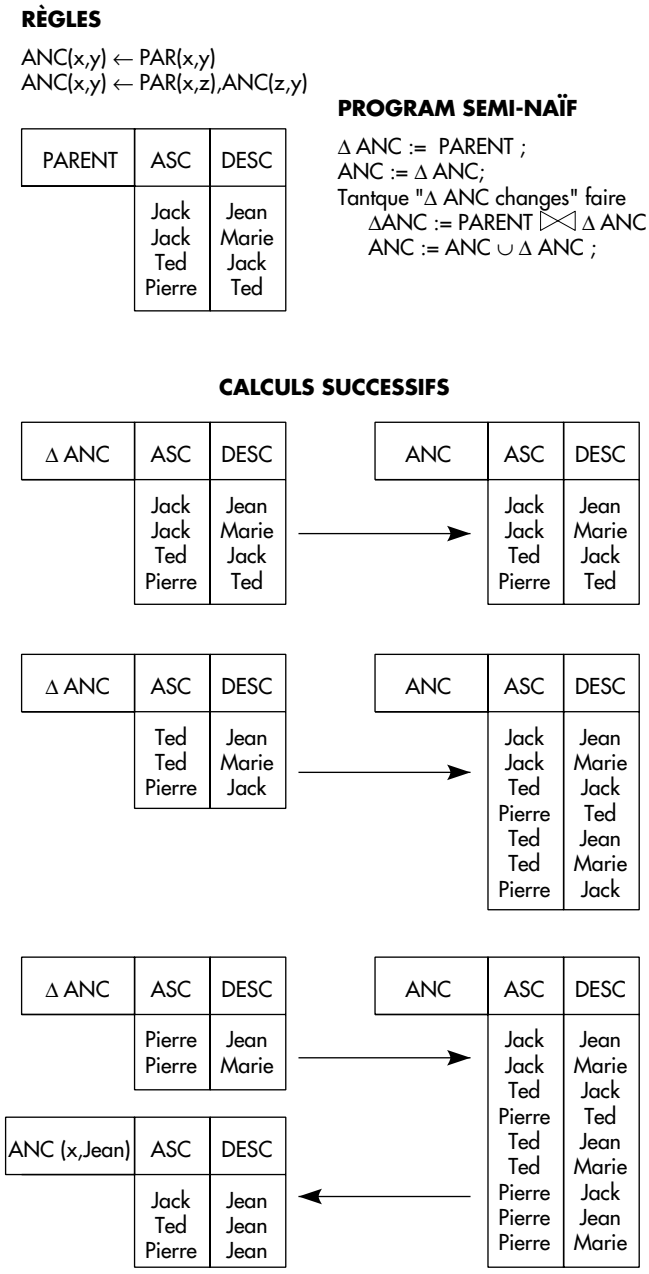


Figure XV.26 : Évaluation semi-naïve des ancêtres de Jean

de la question pour remonter aux faits via les règles. En conséquence, elle est qualifiée de stratégie *top-down* comme vu ci-dessus. Malheureusement, dans le cas de règles récursives la méthode *top-down* peut conduire à boucler.

Nous allons tout d'abord montrer une application de la méthode dans le cas favorable. Par exemple, avec la règle :

ANC(x,y) ←PAR(x,z), ANC(z,y)

la question ANC(Jean,y) génère la sous-question PAR(Jean,z),ANC(z,y). Tenter de résoudre directement ANC(z,y) en chaînage arrière conduit à boucler. Par contre, si l'on évalue PAR(Jean,z) sur la relation de base PARENT, on obtient des constantes c0, c1... pour z. Il devient alors possible de résoudre en chaînage arrière `ANC(c0,y)`,`ANC(c1,y)...` qui génèrent les sous-questions `PAR(c0,z)`,`ANC(z,y)`,`PAR(c1,z)`,`ANC(z,y)`, etc. Le processus s'arrête quand il n'existe plus de sous-question générable du type `PAR(ci,z)` ayant une réponse non vide.

La méthode n'est cependant applicable que dans le cas particulier où la constante de la question se propage pour regénérer la même sous-question. Par exemple, dans le cas de la règle :

ANC(x,y) ←ANC(x,z), PAR(z,y)

la question `ANC(Jean,y)` conduit à générer la sous-question `ANC(Jean,z)`. Changer y en z ne fait pas progresser le problème et le processus boucle totalement. Nous verrons dans la suite des extensions de cette approche « à la Prolog » qui permettent d'éviter le bouclage en utilisant les autres règles et en mêlant chaînage arrière-chaînage avant. Auparavant, nous allons étudier plus en détail la propagation des constantes dans les règles.

7.3.2. La remontée d'informations via les règles

Quand elles s'appliquent, les méthodes par chaînage arrière remontent les constantes depuis la question vers les relations de base et permettent donc d'effectuer les restrictions avant la récursion. Afin de mieux comprendre la remontée des constantes dans le corps des règles, il est possible de l'illustrer par un graphe appelé **graphe de propagation**. Il s'agit là d'une représentation des **propagations d'informations** entre prédicats au sein d'une règle [Beeri87], appelées en anglais *Sideways Information Passing* [Ullman86].

Notion XV.24 : Propagation d'informations (*Information Passing*)

Passage de constantes instanciant une variable x d'un prédicat P(…,x…y…) à un autre prédicat Q(…,y…) en effectuant une sélection sur P avec les valeurs de x, suivie d'une jointure avec Q sur y.

Les propagations d'informations dans les règles se représentent donc par des graphes. Un graphe de propagation pour une règle est un graphe étiqueté qui décrit comment les constantes obtenues par la tête de la règle se propagent aux autres prédicats de la règle. Il existe plusieurs graphes de propagation possibles pour une règle, suivant les choix de propagation effectués. Pour faciliter la compréhension et éviter de répéter des morceaux de règles, une représentation simplifiée avec des bulles représentant les sommets du graphe est proposée. Les prédicats de la règle sont écrits de gauche à droite, depuis la tête jusqu'au dernier prédicat de la condition. Un sommet du graphe (donc représenté par une bulle) est, soit un groupe de prédicats, soit un prédicat récursif. Un arc N $\rightarrow$ p relie un groupe de prédicats N à une occurrence d'un prédicat récursif p. L'arc N $\rightarrow$ p est étiqueté par des variables dont chacune apparaît dans un prédicat de N et dans p. Un graphe de propagation est valide s'il existe un ordre des sommets (appelé ici ordre de validité) tel que, pour chaque arc, tout membre de sa source apparaisse avant sa cible. Pour faciliter la visualisation d'un graphe de propagation valide, il est souhaitable d'écrire les occurrences de prédicats selon l'ordre de validité. Les figures XV.27 et XV.28 présentent deux graphes de propagation valides pour les règles récursives des exemples Ancêtres et Chefs.

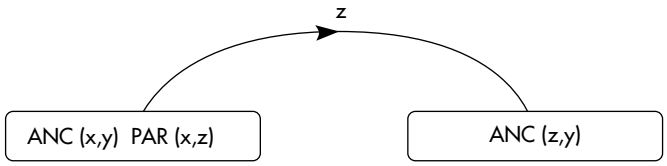


Figure XV.27 : Un graphe de propagation valide pour la règle ANC(x,y) ← PAR(x,z), ANC(z,y)

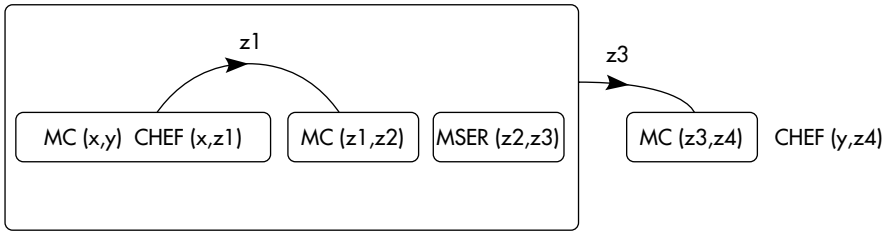


Figure XV.28 : Un graphe de propagation valide pour la règle MC(x,y) ← CHEF(x,z1), MC(z1,z2),MSER(z2,z3),MC(z3,z4),CHEF(y,z4)

La signification intuitive d'un graphe de propagation valide est la suivante : en supposant instanciée la jointure des prédicats entourés par une bulle source d'un arc, il serait possible de propager les valeurs des variables étiquettes vers le prédicat cible de l'arc. Si la bulle source contient le prédicat de tête de la règle, un arc modélise la pro-

pagation de constantes vers un prédicat récursif en chaînage arrière suite à une instantiation d'une variable de la tête. Une telle propagation est donc une propagation par effet de bord.

Un graphe de propagation valide est **linéaire** si pour tout arc N → p, N contient tous les nœuds qui précédent p selon l'ordre de validité. Suivant [Beeri87], nous dirons qu'un graphe de propagation valide linéaire est **total** si toute variable d'un prédicat récursif r cible d'un arc apparaissant dans un prédicat qui précède r étiquette un arc entrant dans r. Nous utiliserons les graphes de propagation de constantes dans la suite, notamment pour la méthode des ensembles magiques généralisés [Beeri87].

7.3.3. Les règles signées

Une **signature** (en anglais *adornment* [Ullman86]) consiste à associer à un prédicat un vecteur de bits qui indique par des 1 les variables liées par des constantes et par des 0 les variables libres (nous suivons ici la méthode de [Rohmer86], [Ullman86] ayant pour sa part utilisé les lettres b (*bounded*) et f (*free*) pour 1 et 0). Par exemple, le prédicat ANC(Jean,x) peut être écrit sous forme d'un prédicat signé par un index binaire ANC^{10}(Jean,x). Les règles sont souvent réécrites avec des prédicats signés ; elles sont alors appelées **règles signées** [Beeri87]. Ainsi, si P^{110}(x,y,z) apparaît en tête d'une règle, cela signifie que la règle sera invoquée avec x et y instanciés par des constantes.

Notion XV.25 : Règle signée (*Adorned Rule*)

Règle réécrite avec des prédicats signés (indexés par des 1 signifiant position instanciée et des 0 signifiant position libre) selon une question et un graphe de propagation.

La tête de la règle est tout d'abord signée selon l'unification réalisée avec la question. La signature est ensuite propagée aux autres prédicats récursifs en suivant les arcs du graphe de propagation et en marquant à 1 les variables correspondant aux étiquettes dans les prédicats cibles [Beeri87]. L'utilisation de règles signées permet de distinguer des prédicats de mêmes noms ayant des variables instanciées différentes. Cela est très utile dans le cas de règles non stables, où les constantes changent de position dans les occurrences d'un même prédicat [Henschen-Naqvi84]. Voici les règles récursives signées pour les `Ancêtres` et les `Chefs`, selon les graphes de propagation représentés respectivement figure XV.27 et XV.28, pour les questions ? `ANC(Jean,x)` et ? `MC(Jean,y)` :

ANC^{10}(x,y) ←PAR(x,z), ANC^{10}(z,y)
MC^{10}(x,y) ← CHEF(x,z1),MC^{10}(z1,z2),MSER(z2,z3),
MC^{10}(z3,z4),CHEF(y,z4)

Dans la suite, nous utiliserons les règles signées pour tous les algorithmes basés sur la réécriture de règles.

7.4. LA MÉTHODE QoSaQ

Les **méthodes interprétées** sont des extensions des méthodes de chaînage arrière « à la Prolog » qui traitent correctement les règles récursives, d'une part en utilisant bien les règles d'initialisation et d'autre part en incluant un tantinet de chaînage avant pour générer des sous-questions. Elles sont caractérisées par le fait qu'elles sont appliquées lors de l'exécution d'une question et non pas lors de l'entrée des règles, à la compilation.

L'approche QSQ (*Query Sub-Query*) décrite dans [Vieille86] a été implantée à l'ECRC (Munich). Elle est devenue QoSaQ lorsqu'elle a été généralisée au support complet de DATALOG. Elle peut être vue comme une amélioration de la méthode de chaînage arrière consistant à mémoriser les sous-questions générées lors de l'unification et à appliquer globalement les sous-questions de même forme, afin de limiter le nombre de sous-questions à traiter. Un ensemble de sous-questions {? R(ai,y) | i ∈ [1,n] } est ainsi remplacé par une seule sous-question avec critère disjonctif ? R({a1∨ a2 ∨...∨ an},y). La remontée des constantes pour déterminer les faits relevants s'effectue par chaînage arrière comme en PROLOG. De plus, une sous-question n'est exécutée que si elle ne l'a pas déjà été. Le critère d'arrêt est donc double : pas de nouvelle sous-question générée ou pas de nouveau fait relevant généré.

Ces techniques de mémorisation conduisent à une méthode rapide en temps d'exécution, mais relativement consommatrice en mémoire puisque faits relevants et sous-questions doivent être mémorisés. Deux versions de la méthode ont été implantées, l'une itérative qui parcourt la liste des sous-questions à traiter, l'autre récursive qui se rappelle à chaque sous-question avec critère disjonctif généré. Notez que la méthode QoSaQ résout les problèmes de cycles dans les données (par exemple, une personne qui serait son propre ancêtre) puisque elle n'exécute que des sous-questions nouvelles.

7.5. LES ENSEMBLES MAGIQUES

La méthode des ensembles magiques est une méthode compilée permettant de générer un programme d'algèbre relationnel étendu avec des boucles. Elle se compose d'une phase de réécriture des règles puis d'une seconde phase appliquant la génération semi-naïve vue ci-dessus. Elle est basée sur l'idée d'un petit génie [Bancilhon86a] qui, avant application du chaînage avant semi-naïf, marque les tuples utiles du prédicat récursif R à l'aide d'un prédicat associé, appelé magique et noté MAGIQUE_R. Ainsi, les calculs en chaînage avant ne sont effectuées que pour les tuples susceptibles de générer des réponses, qui sont marqués par le prédicat magique. La première version des ensembles magiques [Bancilhon86a] ne s'appliquait guère qu'aux règles récursives linéaires. Une version généralisée a été publiée sous le nom « d'ensembles magiques généralisés » [Beeri87]. Nous allons présenter maintenant cette version.

Le point de départ est un programme de règles signées comme vu ci-dessus et un graphe SIP de propagation de constantes par effets de bord. L'intention est de générer un programme de règles qui, à l'aide du ou des prédicats magiques, modélise le passage des informations entre prédicats selon le graphe SIP choisi. Pour chaque prédicat récursif signé R^{xx}, un prédicat magique est créé, noté MAGIQUE_R^{xx} dont les variables sont celles liées dans R^{xx} ; l'arité du prédicat MAGIQUE_R^{xx} est donc le nombre de bits à 0 dans la signature xx. Ainsi, un prédicat magique contiendra des constantes qui permettront de marquer les tuples de R^{xx} utiles. Chaque règle est modifiée par addition des prédicats magiques à sa partie condition, pour restreindre l'inférence aux tuples marqués par les prédicats magiques. Par exemple, une règle :

$$B1^{xx},B2^{xx}...R^{10},C1^{xx},C2^{xx}... \rightarrow R^{10}(x,y)$$

sera transformée en :

$$MAGIC\_R^{10}(x),\ B1^{xx},B2^{xx}...R^{10},C1^{xx},C2^{xx}... \rightarrow R^{10}(x,y)$$

qui signifie que la condition ne doit être exécutée que pour les tuples de R^{10} marqués par le prédicat MAGIC_R^{10}(x).

Le problème un peu plus compliqué à résoudre est de générer les constantes du (ou des) prédicat(s) magique(s) pour marquer tous les tuples utiles et, si possible, seulement ceux-là. Une première constante est connue pour un prédicat magique : celle dérivée de la question ; par exemple, avec la question ? R(a,x), on initialisera le processus de génération des prédicats magiques par MAGIC_R^{10}(a). La génération des autres tuples du (ou des) prédicat(s) magique(s) s'effectue en modélisant les transmissions de constantes à tous les prédicats dérivés par le graphe SIP choisi. Plus précisément, pour chaque occurrence de prédicat dérivé R^{xx} apparaissant dans le corps d'une règle, on génère une règle magique définissant les variables liées de R^{xx} comme suit : la conclusion de la règle est MAGIQUE_R^{xx}(...) et la condition est composée de tous les prédicats qui précédent R^{xx} dans le graphe SIP, les prédicats dérivés dont certaines variables sont liées étant remplacés par les prédicats magiques correspondants. Les variables sont déterminées par le graphe SIP et par les bits à 1 des signatures pour les prédicats magiques. Vous trouverez une explication plus détaillée et une preuve partielle de la méthode dans [Beeri87].

Nous traitons les exemples des ancêtres et des mêmes chefs avec la méthode des ensembles magiques généralisés figure XV.29. Les résultats montrent que les règles de production générées répètent pour partie la condition de la règle précédente. Ce n'est pas optimal, car on est conduit à réévaluer des conditions complexes. Une solution consistant à mémoriser des prédicats intermédiaires de continuation a été proposée sous le nom d'« ensembles magiques généralisés supplémentaires » [Beeri87]. Cette méthode est voisine de celle d'Alexandre proposée par ailleurs [Rohmer86]. Pour le cas des règles linéaires, une version améliorée des ensembles magiques a aussi été proposée [Naughton89]. Le mérite de la méthode reste cependant la généralité.

(1) Les ancêtres définis linéairement (question ?ANC(Jean,y)):
(r1) $MAGIC\_ANC^{10}(x), PAR(x,y) \rightarrow ANC^{10}(x,y)$
(r2) $MAGIC\_ANC^{10}(x), PAR(x,z) \rightarrow MAGIC\_ANC^{10}(z)$
$MAGIC\_ANC^{10}(x), PAR(x,z), ANC^{10}(z,y) \rightarrow ANC^{10}(x,y)$

La question permet d'initialiser le chaînage avant avec:
$MAGIC\_ANC^{10}(Jean)$.

(2) Les mêmes chefs définis quadratiquement (question ?MC(Jean,y)) :
(r1) $SER(s,x), SER(s,y) \rightarrow MSER(x,y)$
(r2) $MAGIC\_MC^{10}(x), MSER(x,y) \rightarrow MC^{10}(x,y)$
(r3) $MAGIC\_MC^{10}(x), CHEF(x,z1) \rightarrow MAGIC\_MC^{10}(z1)$
$MAGIC\_MC^{10}(x), CHEF(x,z1), MC^{10}(z1,z2), MSER(z2,z3) \rightarrow MAGIC\_MC^{10}(z3)$
$MAGIC\_MC^{10}(x), CHEF(x,z1), MC^{10}(z1,z2), MSER(z2,z3), MC^{10}(z3,z4), CHEF(y,z4) \rightarrow MC^{10}(x,y)$

Le prédicat $MAGIC\_MC^{10}$ sera initialisé comme suit par la question :
$MAGIC\_MC^{10}(Jean)$

Figure XV.29 : Application des ensembles magiques

7.6. QUELQUES MÉTHODES D'OPTIMISATION MOINS GÉNÉRALES

7.6.1. La méthode fonctionnelle

Les deux méthodes présentées précédemment conduisent à réécrire un programme de règles avec des prédicats problèmes ou magiques, pour pouvoir l'exécuter de manière optimisée en chaînage avant semi-naïf. Malheureusement, le programme de règles obtenu est en général complexe (plus de règles que le programme initial, récursion mutuelle) et l'exécution semi-naïve peut être lourde. Par exemple, dans le cas des ancêtres, la question ?ANC(Jean,y) conduit à trois règles vues ci-dessus, alors que la solution consiste simplement à calculer la fermeture transitive de la relation parent à partir de Jean. Ces remarques, avec en plus le fait que la démonstration de la complétude des méthodes précédentes est complexe, nous ont conduit à développer une méthode plus formelle basée sur des équations au point fixe qui s'applique dans un contexte général avec fonctions [Gardarin87].

Une méthode d'évaluation simple qui découle de la sémantique de DATALOG consiste à écrire des équations au point fixe en algèbre relationnelle. Malheureusement, la propagation directe des constantes n'est pas possible dans de telles équations dès que celles-ci se reportent sur une condition de jointure. Afin de modéliser dans les équations au point fixe la propagation des constantes, nous utilisons **des fonctions à arguments ensemblistes**. L'idée intuitive de la méthode est

qu'une instance de relation définit en fait des fonctions ; chaque fonction transforme un ensemble de valeurs d'une colonne dans l'ensemble de valeurs correspondant dans une autre colonne. Ainsi toute question ? R(a,y) est interprétée comme une fonction r appliquant l'ensemble des parties du domaine de a dans l'ensemble des parties du domaine de y, c'est-à-dire avec des notations classiques, $r : 2^{dom(a)} \rightarrow 2^{dom(y)}$. Évaluer la question ? R(a,y) revient alors à évaluer la fonction r({a}).

Etant donnés une question et un ensemble de règles, un algorithme permet de générer une **équation fonctionnelle au point fixe** du type r = F(r), où F est une expression fonctionnelle. Pour ce faire, des opérations monotones entre fonctions ont été introduites :

1. la composition de deux fonctions f o g (X) = f(g(X)) modélise en fait une jointure suivie d'une projection ;
2. l'addition (f+g) (X) = f(X) ∪ g(X) modélise l'union ;
3. l'intersection (f ∩ g) (X) = f(X) ∩ g(X) permet de modéliser des jointures cycliques.

Dans le cas de règles à prédicats binaires sans fonction, l'algorithme est simple et repose sur un graphe de connexion des variables dans la règle. Ce graphe représente simplement par un nœud chaque variable et par un arc les prédicats (ou un hyper arc dans le cas de prédicats non binaires). Il est orienté pour visualiser les passages d'information en chaînage arrière. Ce graphe peut être perçu comme une simplification du graphe de propagation d'information vu ci-dessus.

La génération d'équation fonctionnelle s'effectue par un algorithme de cheminement dans le graphe, chaque arc étant interprété comme une fonction de calcul du nœud cible en fonction du nœud source. Cet algorithme décrit dans [Gardarin86] est appliqué ci-dessous aux cas des ancêtres et des chefs. Pour simplifier les notations, pour toute relation binaire R(x,y), R(X) dénote la fonction qui applique {x} sur {y} et R'(Y) celle qui applique {y} sur {x}. Après obtention, l'équation fonctionnelle au point fixe est simplement résolue par application du théorème de Tarski [Tarski55].

Nous illustrons tout d'abord la méthode avec le problème des ancêtres. La figure XV.30 représente les graphes de connexion des variables des règles :

ANC(x,y) ← PAR(x,y)
ANC(x,y) ← PAR(x,z),ANC(z,y).

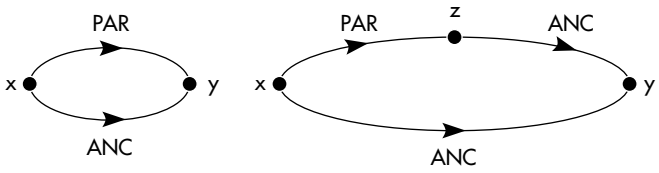


Figure XV.30 : Graphes de connexion des variables pour les ancêtres

L'équation fonctionnelle au point fixe résultante est :

$$ANC(X) = PAR(X) + ANC(PAR(X)).$$

En appliquant le théorème de Tarski à cette équation, la solution obtenue est :

$$ANC(X) = PAR(X) + PAR(PAR(X)) +... + PAR(PAR(PAR...(PAR(X))))$$

qui peut être traduite directement en algèbre relationnelle par un retour aux notations classiques (nous supposons ici la relation ANC acyclique ; une hypothèse contraire nécessite d'élaborer le test d'arrêt). Le programme résultant (figure XV.31) calcule ainsi les ancêtres d'un ensemble de personnes X dans ANC.

```
PROCEDURE CALCUL(ANC,X); {
   ANC := ∅;
   DELTA := X;
   Tant que DELTA ≠ ∅ faire {
      DELTA := π_PAR.2(σ_PAR.1∈DELTA(PAR));
      ANC := ANC ∪ DELTA;
      } ;
   };
```

Figure XV.31 : Calcul des ancêtres de X

De manière plus générale, la méthode s'applique très simplement aux règles **fortement linéaires** de la forme :

$$R(X,Y) \leftarrow B1(X,Y)$$
$$R(X,Y) \leftarrow B2(X,X1),R(X1,Y1),B3(Y1,Y)$$

où B1, B2 et B3 sont des relations de base ou simplement des jointures/restrictions de relations de base et où X, Y, X1, Y1 représentent des variables ou des n-uplets de variables. Le graphe de connexion des variables de ce type de règles est représenté figure XV.32.

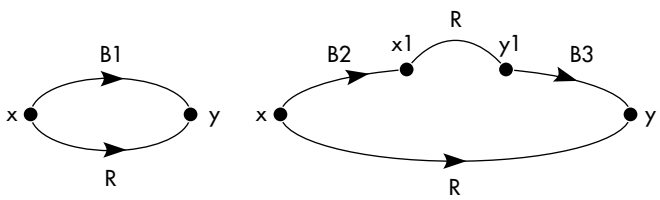


Figure XV.32 : Graphe de connexion des variables de règles fortement linéaires

L'équation fonctionnelle au point fixe est, pour des questions du type ? R(a,Y) :

$$r(X) = b1(X) + b3(r(b2(X))).$$

Le théorème de Tarski permet d'obtenir la forme générale de la solution :

$$r(X) = b1(X) + b3(b1(b2(X))) +... + b3^n(b1(b2^n(X))).$$

Le calcul de ce polynôme est effectué par le programme de l'algèbre relationnelle optimisé suivant dont la terminaison suppose l'acyclicité des données de la relation B2 (sinon, il faut ajouter une différence). Pour obtenir la réponse à une question du type ? R(a,Y), il suffit de remplacer X par a dans le programme de la figure XV.33. Celui-ci est utilisable quand X est instancié par un ensemble de valeurs.

```
PROCEDURE CALCUL(R,X); {
   i,n : integer;
   n:= 0 ;
   R := π_B1.2(σ_B1.1 ∈ X(B1));
   B2N := X;
   Tant que B2N ≠ ∅ faire {
      n := n + 1;
      B2N := π_B2.2(σ_B2.1 ∈ B2N(B2));
      DELTA := π_B1.2(σ_B1.1 ∈ B2N(B1));
      Pour i : = 1 à n faire {
         DELTA := π_B3.2(σ_B3.1 ∈ B2N(B3));
         R := R ∪ DELTA;
         } ;
      } ;
   } ;
```

Figure XV.33 : Calcul des réponses à des requêtes sur règles fortement linéaires

7.6.2. Les méthodes par comptages

La méthode par comptages a été proposée dans [Sacca86]. Elle s'applique aux **règles fortement linéaires** de la forme :

$R(X,Y) \leftarrow B1(X,Y)$

$R(X,Y) \leftarrow B2(X,X1),R(X1,Y1),B3(Y1,Y)$

où B1, B2 et B3 sont des relations de base ou simplement des jointures et restrictions de relations de base et où X, Y, X1, Y1 représentent des variables ou des n-uplets de variables. Ce type de règles assez triviales (mais n'est-ce pas le seul type qui corresponde à des exemples pratiques ?) a déjà été étudié ci-dessus avec la méthode fonctionnelle. À partir d'une question ? R(a,Y), l'idée est de réécrire les règles en ajoutant des compteurs qui mémorisent la distance parcourue depuis a en nombre d'inférences « vers le haut » via B2 et en nombre d'inférence « vers le bas » via B3. On ne retiendra alors que les tuples obtenus par le même nombre d'inférences dans les deux sens, puisque seulement ceux-ci peuvent participer au résultat (rappelons que le résultat s'écrit en forme fonctionnelle $b3^n(b1(b2^n(a)))$: il faut donc effectuer n inférences via b2, passer par b1 puis effectuer n autres inférences via b3). Le système de règles modifié pour générer la réponse devient :

COMPTE(0,a)

COMPTE_B2(J,X), B2(X,X1) → COMPTE_B2(J+1,X1)
COMPTE_B2(J, X), B1(X,Y) → DECOMPTE_B3(J,Y)
DECOMPTE_B3(J,Y1),B3(Y1,Y) → DECOMPTE_B3(J-1,Y)
DECOMPTE_B3(0,Y) → R(Y)

Ce programme de règles compte à partir de « a » en effectuant la fermeture de B2 : il conserve dans COMPTE_B2 les éléments de la fermeture avec leur distance à « a ». Puis il effectue la jointure avec B1 et commence un décompte en effectuant la fermeture descendante via B3 à partir des éléments trouvés. Lorsque le compteur atteint 0, c'est qu'un tuple résultat est trouvé. La méthode ne fonctionne bien évidemment qu'en présence de données acycliques. Une extension pour résoudre le problème des données cycliques consiste à borner le compteur, mais ceci s'effectue au détriment des performances. La méthode est donc assez peu générale : il s'agit d'une écriture par règles d'un algorithme de double fermetures transitives, montante puis descendante, pour des graphes acycliques. La méthode est certes efficace; de plus, elle peut être combinée avec les ensembles magiques pour donner des méthodes de type **comptage magique** [Sacca86].

7.6.3. Les opérateurs graphes

Une méthode basée sur des **parcours de graphes** est applicable au moins dans le cas de règles fortement linéaires. L'idée est de représenter l'auto-jointure d'une relation par un graphe qui peut être construit en mémoire à partir de l'index de jointure de la relation avec elle-même. Les nœuds correspondent aux tuples et les arcs aux inférences (donc aux jointures) possibles entres ces tuples. La résolution d'un système de règles fortement linéaires du type vu ci-dessus se réduit alors à des parcours de même longueur des deux graphes d'auto-jointures des relations B2 et B3 en mémoire, en partant des nœuds représentant des tuples du type B2(a,y) et en transitant par les tuples de B1 pour passer de B3 à B2. Il est possible de détecter les cycles par marquage et de faire fonctionner les algorithmes avec des données cycliques. En addition, des calculs de fonctions peuvent être effectués à chaque nœud, les graphes étant valués par les valeurs des tuples. Ceci conduit à des méthodes très efficaces pour traiter des problèmes du type plus court chemin ou calcul de prix dans des relations du type composant-composé.

8. RÈGLES ET OBJETS

De nombreuses tentatives pour combiner les bases de données objet et les règles ont été effectuées. Citons notamment IQL [Abiteboul90], Logres [Ceri90], Coral

[Srivastava93]et Rock&Roll [Barja94]. Le sigle DOOD (*Deductive Object-Oriented Databases*) a même été inventé et une conférence portant ce sigle est née. Aucun système n'a réellement débouché sur un produit, à l'exception sans doute de Validity [Nicolas97]. L'inférence sur les objets est cependant un problème ancien, déjà traité par les systèmes experts au début des années 80, par exemple OPS 5 et Nexpert-Object.

8.1. LE LANGAGE DE RÈGLES POUR OBJETS ROL

Afin d'illustrer la problématique, nous introduisons ici le langage **ROL** (*Rule Object Language*) prototypé au laboratoire PriSM au début des années 1990 [Gardarin94]. ROL s'appuie sur le modèle objet de C++. Un schéma de bases de données est défini par une description dans un langage analogue à ODL de l'ODMG. La figure XV.34 illustre un schéma ROL. Celui-ci est directement implémenté en C++. Pour les calculs intermédiaires, il est possible de définir des classes temporaires par le mot clé TRANSIENT.

```
CLASS Point { Float Abs ; Float Ord ;} ;
CLASS Personne {String Nom; String Prénom;
   LIST<Point> Caricature; };
CLASS Acteur : Personne { Real Salaire;
   Real Augmenter (Real Montant) ; };
CLASS Film {Int Numf; String Titre; SET<String> Categories;
   Relationship Set<Acteur> Joue ;};
CLASS Domine { Int Numf; Acteur Gagnant; Acteur Perdant; };
TRANSIENT CLASS Calcul { } ;
```

Figure XV.34 : Exemple de schéma ROL

Les règles en ROL sont organisées en modules. Chaque module commence par des définitions de variables valables pour tout le module, de la forme :

```
VAR <nom> IN <collection>.
```

Le nom d'une variable est généralement une lettre, alors qu'une collection est une extension de classe ou une collection dépendantes, comme en OQL. Il est aussi possible d'utiliser des variables symboliques (chaînes de caractères), afin d'améliorer la clarté des programmes. Nous définissons simplement les variables :

```
VAR X IN Film, VAR Y IN X.Categories ;

VAR Z IN Calcul ;
```

Les règles sont perçues comme des règles de production de la forme :

```
IF <condition > THEN <action>.
```

La condition est analogue à une qualification de requêtes OQL. C'est une combinaison logique de conditions élémentaires. Une condition élémentaire permet de comparer deux termes. Un terme est une constante, un attribut, une expression de chemin ou de méthode. Les actions sont :

– **Des insertion de valeurs.** Une valeur éventuellement complexe (tuple par exemple) est insérée dans une collection temporaire ou persistante par la construction +<collection>(<valeurs>).
– **Des créations d'objets.** Un objet est créé dans une collection temporaire ou persistante par la construction +<classe>(<paramètres>). La classe doit être définie auparavant. L'action provoque simplement l'appel au constructeur d'objet avec les paramètres indiqués.
– **Des destruction de valeurs.** Une valeur d'une collection est détruite par la construction –<collection>(<valeurs>).
– **Des destructions d'objets.** Un objet est détruit dans une collection temporaire ou persistante par la construction –<classe>(). La classe doit être définie auparavant. L'action provoque simplement l'appel au destructeur de l'objet avec les paramètres indiqués sur les objets de la classe sélectionnés par la condition.
– **Des appels de méthodes.** Toute méthode définie au niveau du schéma est valide en partie action précédée de + pour faciliter la lecture des expressions.
– **Un bloc d'actions élémentaires** (création et destruction d'objets et/ou valeurs, appels de méthodes). Une règle peut avoir en conséquence une suite d'actions séparées par des + et des –. Elle peut donc accomplir des mises à jour, des impressions, etc.

Une sémantique ensembliste type point fixe est donnée au langage par traduction en algèbre d'objet complexe [Gardarin94]. Les difficultés des langages objets est qu'ils intègrent à la fois la création de nouveaux objets (l'invention d'objets d'après [Abiteboul90]) et les fonctions. De ce fait, il est facile d'écrire des programmes infinis dès qu'on utilise des règles récursives. À notre connaissance, aucun compilateur n'est capable de détecter les programmes non sains.

À titre d'illustration, nous définissons figure XV.35 deux règles en ROL. La première réalise une suite d'actions pour tous les films d'aventure dans lesquels joue Quinn avec un salaire supérieur à 1 000 $ de l'heure. La seconde calcule la fermeture transitive de la relation Domine, puis imprime qui fut (transitivement) meilleur que Marilyn. Il n'y a pas de boucle infinie car MEILLEUR est défini comme un ensemble. Il n'en serait rien avec une liste. Notons aussi que la structure de blocs imbriqués permet de définir des ordres de calcul, donc une certaine stratification. De tels langages sont complexes, mais puissants. Les perspectives d'applications sont importantes d'après [Nicolas97].

```
{  VAR F IN Film, A IN F.Joue, C IN F.Categorie ;
   IF A.Name='Quinn' AND A.salary≥1000 AND C='Aventure')
   THEN {      +Calcul(F.Titre, F.Categories, A.Salaire)
         +F.Categories('cher')
         -F.categories('gratuit')
         +Print(F.titre, 'Mon cher Quinn') };

{  TRANSIENT SET Meilleur (Gagnant String, Perdant string);
   VAR Domine D, Meilleur B, C ;
   { IF EXISTS(D) THEN +Meilleur(D.Gagnant, D.Perdant);
    IF B.Perdant = C.Gagnant THEN
      +Meilleur(B.Gagnant, C.Perdant);}

   IF B.Perdant.Nom ="Marilyn" THEN
         +Print(B.Gagnant, "fut meilleur que", B.Perdant); } ;
```

Figure XV.35 : Exemples de règles ROL

8.2. LE LANGAGE DE RÈGLES POUR OBJETS DEL

DEL est le langage supporté par le système Validity, commercialisé par la société Next Century Media [Next98]. DEL signifie *Datalog Extended Language*. DEL étend en effet Datalog avec des objets typés, des prédicats prédéfinis (*built-in*), des quantificateurs et des ordres impératifs. L'ensemble est un langage déductif très puissant, permettant d'inférer sur de grandes bases d'objets.

Le modèle de données est un modèle objet. Les types DEL sont organisés selon une hiérarchie d'héritage. Ils sont atomiques ou composites. Les types composites comportent les collections classiques de l'objet (*set*, *bag* et *list*) et les tuples. Les types atomiques sont les booléens, les littéraux numériques (entiers et flottants), les chaînes de caractères et les références aux objets. Il est possible de définir des types utilisateurs en étendant la hiérarchie.

Les types peuvent être encapsulés dans des méthodes et fonctions écrites dans le langage impératif de DEL, en C ou en C++. Le langage impératif est un langage structuré par blocs, analogue aux langages classiques. Il est possible de calculer des expressions et d'effectuer des contrôles. Un ordre *foreach* est disponible pour parcourir les collections. Une collection peut être l'extension d'un prédicat dérivé par des règles.

Le langage de règles permet d'écrire des règles en DATALOG étendu avec des fonctions et des références. Les règles ne sont pas limitées aux clauses de Horn : il est possible d'utiliser des connecteurs logiques `AND`, `OR` et `NOT`, et des quantificateurs `EXISTS` et `FORALL`. En outre, le calcul de prédicats est étendu pour manipuler des identifiants d'objets, des opérations d'agrégats et des expressions de chemins limitées. Une tête de règle est un prédicat dérivé. DEL supporte les règles récursives.

Au-delà des corps de règles, les conditions permettent d'exprimer des contraintes d'intégrité et peuvent être utilisées dans les ordres impératifs conditionnels (*if* et *while*). DEL supporte aussi de nombreux prédicats prédéfinis appelables depuis les conditions ou les ordres impératifs. Le temps est également supporté sous différentes formes. DEL apparaît donc comme un langage très puissant, capable d'améliorer la productivité des développeurs d'applications intelligentes. L'implémentation du système déductif a été réalisée à Bull, à partir des travaux effectués à la fin des années 80 à l'ECRC à Munich. Un gérant d'objets spécifique avec des méthodes de reprise et de gestion des accès concurrents efficaces sert de support au moteur d'inférence qui applique un dérivé de la méthode QoSaQ vue ci-dessus.

9. CONCLUSION

Ce chapitre a proposé une synthèse des recherches et développements en matière de bases de données déductives. Parmi les multiples approches, nous avons surtout insisté sur l'intégration d'un langage de règles au sein d'un SGBD relationnel étendu. Cette approche est probablement valable à long terme, bien que quelques produits commencent à apparaître. Une approche plus immédiate consiste à coupler un interpréteur de règles à un SGBD relationnel. De nombreux produits offrent de tels couplages. Ils constituent une approche très primitive aux bases de données déductives.

Plusieurs caractéristiques souhaitables d'un langage de règles pour bases de données ont été étudiées. La plupart ont été intégrées à un langage de programmation logique appelé DATALOG, qui a une sémantique de type point fixe. Les extensions essentielles de DATALOG nécessaires à la programmation sont finalement les fonctions et les attributs multivalués (ensembles). L'approche DATALOG est utile pour la compréhension des problèmes, mais sans doute difficile à utiliser vu sa syntaxe plutôt formelle. Un langage plus pratique que DATALOG peut être dérivé de SQL, à partir des qualifications de questions et des actions d'insertion et de mise à jour. Il s'agit essentiellement d'un choix de syntaxe. De même, avec les objets il est possible de dériver un langage de règles du langage OQL, comme le montre le langage ROL ébauché ci-dessus. DEL est une autre variante plus proche de DATALOG. La puissance de tous ces langages est résumée figure XV.36.

Le problème de l'optimisation des requêtes dans un contexte de règles récursives est complexe. Vous pouvez consulter [Ceri91] pour un approfondissement. L'optimisation en présence de fonctions et d'objets complexes est encore mal connue. De plus, les techniques de maintenance de la cohérence des règles et des données sont à peine étudiées. Finalement, il n'existe encore guère de SGBD déductif fonctionnant avec des applications en vraie grandeur, sans doute à l'exception de Validity. Cependant, les

techniques essentielles pour étendre un SGBD relationnel ou objet avec des règles sont connues. Elles ont été présentées dans ce chapitre.

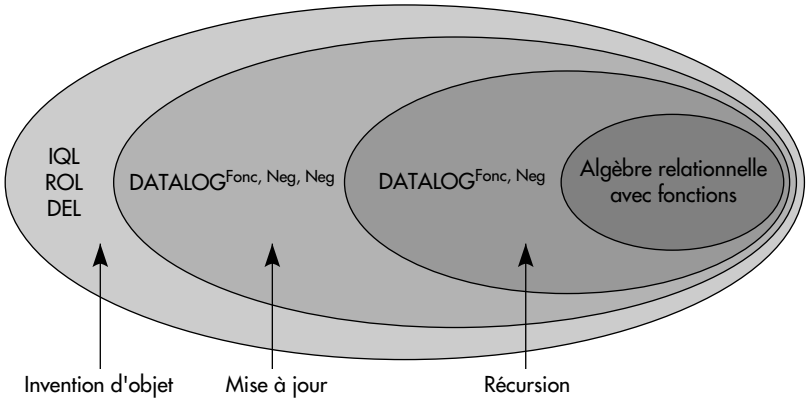


Figure XV.36 : Puissance des langages de règles

10. BIBLIOGRAPHIE

[Abiteboul89] Abiteboul S., Kanellakis P., « Object Identity as a Query Language Primitive », *ACM SIGMOD Intl. Conf. on Management of Data*, p. 159-173, 1989.

Une présentation des fondements du langage IQL, un langage de règles capable de créer de nouveaux objets. IQL utilise des identifiants d'objets pour représenter des structures de données partagées avec de possibles cycles, pour manipuler les ensembles et pour exprimer toute question calculable. IQL est un langage typé capable d'être étendu pour supporter des hiérarchies de types.

[Abiteboul90] Abiteboul S., « Towards a Deductive Object-Oriented Database Language », *Data and Knowledge Engineering,* vol. 5, n° 2, p. 263-287, 1990.

Une extension d'IQL vers un langage plus praticable. Un langage de règles pour objets complexes est présenté, incluant toutes les extensions de Datalog, l'invention d'objets, l'appel de méthodes et la construction de fonctions dérivées.

[Abiteboul91] Abiteboul S., Simon E., « Fundamental Properties of Deterministic and Nondeterministic Extensions of Datalog », *Theoretical Computer Science*, n° 78, p. 137-158, 1991.

Une étude des différentes extensions de Datalog avec négations et mises à jour. La puissance des différents langages obtenus est caractérisée en termes de complexité.

[Abiteboul95] Abiteboul S., Hull R., Vianu V., *Foundations of Databases*, Addison-Wesley Pub. Comp., Reading, Mass, USA, 1995.

Ce livre présente les fondements théoriques des bases de données. Une large place est donnée à Datalog et ses extensions, ainsi qu'à la sémantique de tels langages.

[Aho79] Aho A.V., Ullman J.D., « Universality of data retrieval languages », Conf. POPL, San-Antonio, Texas, 1979.

Cet article est l'un des premiers à souligner l'incapacité des langages d'interrogation de bases de données à supporter la récursion. La notion de plus petit point fixe d'une équation de l'algèbre est notamment introduite.

[Aiken92] Aiken A., Widom J., Hellerstein J., « Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism », *ACM SIGMOD Int. Conf. on Management of Data*, San Diego, Calif., 1992.

Des méthodes d'analyse statiques d'un programme de règles sont présentées pour déterminer si un ensemble de règles de production référençant une base de données : (1) se terminent ; (2) produisent un état base de données unique ; (3) produisent une chaîne d'actions observables unique.

[Apt86] Apt C., Blair H., Walker A., « Towards a theory of declarative knowledge », *Foundations of Deductive Databases and Logic Programming*, Minker J. Ed., Morgan Kaufmann, Los Altos, 1987, aussi IBM Research Report RC 11681, avril 1986.

Une étude sur les points fixes minimaux et la stratification. Il est montré qu'un programme avec négation à plusieurs points fixes minimaux et que la stratification consiste à choisir le plus naturel.

[Bancilhon86a] Bancilhon F., Maier D., Sagiv Y., Ullman J.D., « Magic sets and other strange ways to implement logic programs », *5th ACM Symposium on Principles of Database Systems*, Cambridge, USA, 1986.

Cet article expose la méthode des ensembles magiques pour effectuer les restrictions avant la récursion lors de l'évaluation de questions déductives. Comme la méthode d'Alexandre, les ensembles magiques réécrivent les règles en simulant un chaînage arrière à partir de la question. Une évaluation en chaînage avant permet alors de ne s'intéresser qu'aux seuls faits capables de générer des réponses (les faits relevants).

[Bancilhon86b] Bancilhon F., Ramakrishnan R., « An Amateur's Introduction to Recursive Query Processing Strategies », *ACM SIGMOD Intl. Conf. on Management of Data*, Washington D.C., mai 1986.

Une vue d'ensemble des méthodes d'évaluation de règles récursives proposées en 1986. Les méthodes Query-Subquery et Magic Set sont particulièrement bien décrites.

[Barja94] Barja L. M., Paton N. W., Fernandes A.A., Williams H.M., Dinn A., « An Effective Deductive Object-Oriented Database Through Language Integration », *20th Very Large Data Bases International Conference*, Morgan Kaufman Pub., Santiago, Chile, p. 463-474, août 1994.

Cet article décrit une approche au développement d'un SGBD objet déductif (DOOD) réalisée à l'Université d'Edinburgh. Le prototype comporte notamment un composant ROCK & ROLL constitué du modèle d'objets, d'un langage impératif et d'un langage de type DATALOG associés.

[Beeri86] Beeri C. *et. al.* « Sets and negation in a logical database language (LDL1) », *Proc. of ACM PODS*, 1987; aussi MCC Technical Report, novembre 1986.

L'auteur discute de l'introduction des ensembles et de la négation dans le langage LDL1. Il montre que les deux fonctionnalités nécessitent la stratification. De nombreux exemples sont proposés.

[Beeri87] Beeri C., Ramakrishnan R., « On the Power of Magic », *ACM PODS 1987*, aussi Technical Report DB-061-86, MCC.

Cet article développe différentes variantes de la méthode des ensembles magiques pour optimiser les règles récursives. Il montre qu'une extension plus performante de cette méthode est identique à la méthode d'Alexandre préalablement proposée par J.M. Kerisit [Rohmer86].

[Bidoit91] Bidoit N., « Negation in Rule-Based Database Languages : A Survey », *Theoretical Computer Science*, n° 78, p. 1-37, 1991.

Une vue d'ensemble de la négation en Datalog et des diverses sémantiques possibles.

[Ceri90] Ceri S., Widom J., « Deriving Production Rules for Constraint Maintenance », *16th Very Large Data Bases International Conference*, Morgan Kaufman Pub., Brisbane, Australie, août 1990.

Un ensemble de méthodes différentielles pour générer des « triggers » afin de maintenir des contraintes d'intégrité dans une base de données relationnelle. Un article similaire a été publié par les mêmes auteurs au VLDB 91 afin de maintenir cette fois des vues concrètes.

[Ceri91] Ceri S., Gottlob G., Tanca L., *Logic Programming and Databases*, Surveys in Computer Sciences, Springer Verlag, 1990.

Un livre fondamental sur le modèle logique. Il introduit Prolog comme un langage d'interrogation de données, les bases de données relationnelles vues d'un point de vue logique, et enfin les couplages de Prolog et des bases de données. Dans la deuxième partie, Datalog et ses fondements sont présentés. La troi-

sième partie est consacrée aux techniques d'optimisation de Datalog et à un survol des prototypes implémentant ces techniques.

[Chang86] Chang C. L., Walker A., « PROSQL: A Prolog Programming Interface with SQL/DS », *1st Int. Workshop on Expert Database Systems*, Benjamin/ Cummings Pub., L. Kerschberg Ed., 1986.

La description d'une intégration de SQL à Prolog. Un prototype a été réalisé à IBM selon ces principes permettant d'accéder à DB2 depuis Prolog.

[Clark78] Clark C. « Negation as failure » *Logic and databases*, édité par Gallaire et Minker, Plenum Press, New York, 1978.

Un article de base sur la négation en programmation logique. Il est proposé d'affirmer qu'un fait est faux s'il ne peut être démontré vrai (négation par échec). Cela conduit à interpréter les règles comme des équivalences : « si » peut être lu comme « si et seulement si » à condition de collecter toutes les règles menant à un même but comme une seule.

[Clocksin81] Clocksin W.F., Mellish C.S., *Programming in Prolog*, Springer Verlag, Berlin-Heidelberg-New York, 1981.

Un livre de base sur le langage Prolog.

[Gallaire78] Gallaire H., Minker J., *Logic and Databases,* Plenum Press, 1978.

Le premier livre de base sur la logique et les bases de données. Il s'agit d'une collection d'articles présentés à Toulouse lors d'un premier workshop tenu en 1977 sur le sujet.

[Gallaire81] Gallaire H., Minker J., Nicolas J.M., *Advances in database theory*, vol. 1, Plenum Press, 1981.

Le second livre de base sur la logique et les bases de données. Il s'agit d'une collection d'articles présentés à Toulouse lors d'un second workshop tenu en 1979 sur le sujet.

[Gallaire84] Gallaire H., Minker J., Nicolas J.M., « Logic and databases: a deductive approach », ACM Computing Surveys, vol. 16, n° 2, juin 1984.

Un état de l'art sur les bases de données et la logique. Différents types de clauses (fait positif ou négatif, contrainte d'intégrité, loi déductive) sont isolés. L'interprétation des bases de données comme un modèle ou comme une théorie de la logique est discutée. Les différentes variantes de l'hypothèse du monde fermé sont résumées.

[Gardarin85] Gardarin G., De Maindreville C., Simon E., « Extending a Relational DBMS towards a Rule Base Systems : A PrTN based Approach », CRETE WORKSHOP on AI and DB, June 1985, THANOS and SCHMIDT Ed., aussi dans *Database and Artificial Intelligence*, Springer Verlag Ed, 1989.

Cet article présente les premières idées qui ont permis de développer le langage RDL1 à l'INRIA. Citons comme apports importants l'introduction des

mises à jour en têtes de règles avec l'exemple du circuit électrique et la modélisation du processus d'évaluation de questions par les réseaux de Petri à prédicats. Les principales techniques d'optimisation de réseaux par remontée des sélections sont décrites.

[Gardarin86] Gardarin G., de Maindreville C., « Evaluation of Database Recursive Logic Programs as Recurrent Function Series », *ACM SIGMOD Intl. Conf. on Management of Data*, Washington D.C., May 1986.

Cet article introduit l'approche fonctionnelle décrite ci-dessus. La réécriture est basée sur des graphes de connexion de variables.

[Gardarin87] Gardarin G., « Magic functions : A technique for Optimization of Extended Datalog Recursive Programs », *Intl. Conf. on Very Large Data Bases*, Brighton, England, septembre 1987.

Cet article propose une deuxième version de l'approche fonctionnelle plus générale. Une question et les règles associées sont réécrites en systèmes d'équations fonctionnelles. Le système est alors optimisé puis évalué sur la base de données. La technique est particulièrement adaptée aux règles récursives.

[Gardarin94] Gardarin G., « Object-Oriented Rule Language and Optimization Techniques », *Advances in Object-Oriented Systems*, Springer Verlag, Computer and Systems Sciences Series, vol. 130, p. 225-249, 1994.

Cet article compare les différents langage de règles et introduit le langage ROL pour BD objet de type C++ persistant.

[Hanson92] Hanson E., « Rule Condition Testing and Action Execution in Ariel », *ACM SIGMOD Int. Conf. on Management of Data*, San Diego, Calif., 1992.

Cet article décrit les tests des conditions des règles et l'exécution des actions dans le SGBD actif Ariel. Pour tester les conditions, Ariel utilise un réseau de discrimination maintenu en mémoire proche du réseau Rete, bien connu dans les systèmes de production en intelligence artificielle. Ce réseau est une version modifiée du réseau Treat qui évite de maintenir les données correspondant à certaines jointures.

[Hayes-Roth85] Hayes-Roth F., « Rule Based systems », *Communications of the ACM*, vol. 28, n° 9, septembre 1985.

Une introduction aux systèmes de règles de production et leurs applications.

[Kiernan90] Kiernan J., de Maindreville C., Simon E., « Making Deductive Database a Practical Technology: A Step Forward », *ACM SIGMOD Intl. Conf. on Management of Data*, Atlantic City, juin 1990.

Une présentation du langage RDL1 et de son implémentation. Le langage RDL1 supporte toutes les extensions de Datalog vues dans ce chapitre avec une syntaxe plus proche du calcul relationnel de tuples. De plus, il possède une par-

tie contrôle permettant de spécifier les priorités d'exécution des règles. RDL1 a été implanté à l'INRIA à l'aide de réseaux de Petri à prédicats étendus résultant de la compilation des règles. Ces réseaux sont interprétés sur la couche basse du système SABRINA qui offre une interface proche d'une algèbre relationnelle étendue avec des types abstraits.

[Kifer87] Kifer M., Lozinskii E., « Implementing Logic Programs as a Database System », *IEEE Intl. Conf. on Data Engineering,* Los Angeles, 1987.

Une présentation d'un prototype supportant Datalog. L'idée essentielle de cet article est de compiler un programme logique en un réseau de transitions (le « system graph ») représentant les règles. Le réseau peut alors être optimisé par des techniques de génération de sous-questions.

[Kifer89] Kifer M., Lausen G., « F-Logic : A Higher-Order Language for Reasoning about Objects, Inheritance and Schema », *ACM SIGMOD Intl. Conf. on Management of Data*, Portland, 1989.

Une logique pour objets complexes. Cet article définit un langage de règles supportant héritage, méthodes et objets complexes. Une méthode de résolution étendue est proposée pour démontrer un théorème dans un tel contexte.

[Kuper86] Kuper G., « Logic programming with sets », *Proc. ACM PODS*, 1987.

Cet article discute du support des ensembles en programmation logique.

[Lloyd87] Lloyd J., *Foundations of logic programming*, 2e édition, Springer Verlag, 1987.

Le livre de base sur les fondements de la programmation logique. Les différentes sémantiques d'un programme logique sont introduites. Les techniques de preuve de type résolution avec négation par échec sont développées.

[McKay81] MacKay D., Shapiro S., « Using active connection graphs for reasoning with recursive rules », *Proc. Intl. Conf. on Artificial Intelligence IJCAI,* 1981.

Cet article introduit les PCG comme support d'analyses de règles et méthodes de preuves.

[Minker82] Minker J., Nicolas J.M., « On Recursive Axioms in Deductive Databases », *Information Systems Journal* Vol.8, n° 1, p. 1-13, Jan. 1982.

Un des premiers articles sur le problème des règles récursives.

[Naqvi89] Naqvi S., Tsur S., « A Logical Language for Data and Knowledge Bases », *Principles of Computer Science*, Computer Science Press, New York, 1989.

Ce livre décrit le langage LDL. LDL est un langage dérivé de Prolog pour manipuler les bases de données relationnelles supportant des objets complexes (ensembles). Il est assez proche de Datalog étendu avec la négation, des mises à jour et des ensembles. Une version opérationnelle a été développée à MCC et distribuée dans les universités américaines. Le livre est illustré de nombreux exemples.

[Next98] Next Century Media, *Validity Database System, DEL Language Reference Manual*, Next Century Media Inc., Versailles, France

Ce document présente le langage DEL du SGBD déductif (DOOD) Validity, commercialisé par Next Century Media, une start-up issue du groupe Bull.

[Nicolas97] Nicolas J-M., « Deductive Object Oriented Databases – Technology, Products and Applications : Where are we ? », *Intl. Symp. On Digital Media Information Base* DBIM'97, Nara, Japan, Nov. 1997.

Cet article donne une vue quelque peu optimiste du passé, du présent et du futur des bases de données déductives. Il est écrit par le président de Next century Media.

[Nilsson80] Nilsson N., *Principles of Artificial Intelligence*, Tioga Publication, 1980.

Un des livres de base sur l'intelligence artificielle. Les systèmes de règles de production pertinents aux bases de données déductives sont particulièrement développés.

[Przymusinski88] Przymusinski T., « On the declarative semantics of stratified deductive databases and logic programs », in *Foundations of Deductive Databases and Logic Programming*, Morgan & Kaufmann, Los Altos, 1988.

Une discussion de la sémantique des programmes de règles stratifiés. La sémantique inflationniste où toutes les règles sont déclenchées en parallèle est notamment introduite.

[Ramakrishnan95] Ramakrishnan R., Ullman J.D., « A Survey of Deductive Database Systems », *Journal of Logic Programming*, vol. 23, n° 2, Elsevier Science Pub. Comp., New York, 1995.

Une vue d'ensemble récente des bases de données déductives et de leurs apports et perspectives théoriques.

[Reiter78] Reiter R., « On closed world data bases », in *Logic and databases*, Édité par Gallaire et Minker, Plenum Press, New York, 1978.

L'article de base sur l'hypothèse du monde fermé.

[Reiter84] Reiter R., « Towards a Logical Reconstruction of Relational Database Theory », in *On Conceptual Modelling*, p. 191-234, Springer Verlag, 1984.

Une redéfinition des bases de données relationnelles en logique. Les différents axiomes nécessaires à l'interprétation d'une base de données comme une théorie en logique sont présentés : fermeture des domaines, unicité des noms, axiomes d'égalité, etc. Les calculs relationnels sont unifiés et généralisés.

[Rohmer86] Rohmer J., Lescœur R., Kerisit J.M., « The Alexander Method – A Technique for the Processing of Recursive Axioms in Deductive Databases », *New Generation Computing*, n° 4, p. 273-285, Springer-Verlag, 1986.

Les auteurs introduisent la méthode d'Alexandre, première méthode proposée dès 1985 lors du stage de J.M. Kerisit pour transformer les règles récursives en

règles de production appliquées aux seuls faits relevants. Cette méthode est très proche de la méthode des ensembles magiques proposée un peu plus tard par Bancilhon et Ullman.

[Sacca86] Sacca D., Zaniolo C., « Magic Counting Methods », *MCC Technical Report* DB-401-86, Dec. 1986.

Cet article décrit la méthode de comptage étudiée ci-dessus pour optimiser les règles récursives.

[Simon92] Simon E., Kiernan J., de Maindreville C., « Implementing High Level Active Rules on top of Relational DBMS », *Proc. of the 18th Very Large Data Bases Int. Conf.*, Morgan Kaufman, Vancouver, Canada, 1992.

Cet article présente le langage de définition de déclancheurs dérivé du langage de règle RDL1. Les règles deviennent actives du fait qu'elles sont déclenchées suite à des événements (par exemple des mises à jour). Elles sont définies dans un langage de haut niveau avec une sémantique de point fixe. Un gérant d'événements implémenté au-dessus du SGBD SABRINA déclenche l'évaluation des règles.

[Srivastava93] Srivastava D., Ramakrishnan, Seshadri P., Sudarshan S., « Coral++ : Adding Object-Orientation to a Logic Database Language », *Proc. of the 19th Very Large Data Bases Int. Conf.*, Morgan Kaufman, p. 158-170, Dublin, Ireland, 1993.

Cet article présente le langage de règles CORAL, extension de Datalog avec des concepts objet. Le modèle objet de CORAL est inspiré de C++. Les méthodes sont écrites en C++. CORAL est compilé en C++. Ce langage a été réalisé à l'université du Maddisson aux USA.

[Stonebraker86] Stonebraker M., Rowe A.L., « The Postgres Papers », *UC Berkeley, ERL M86/85*, BERKELEY, novembre 1986.

Une collection d'articles sur POSTGRES. Ce système supporte la définition de règles exploitées lors des mises à jour. Les règles sont donc utilisées pour définir des déclencheurs. Elles permettent de référencer des objets complexes définis sous forme de types abstraits de données. Elles se comportent comme des règles de production de syntaxe if <opération> on <relation> then <opérations>. Les opérations incluent bien sûr des mises à jour. Un système de priorité est proposé pour régler les conflits en cas de règles simultanément déclenchables. Ce mécanisme de règles est aujourd'hui commercialisé au sein du SGBD INGRES.

[Tarski55] Tarski A., « A lattice theoretical fixpoint theorem and its applications », *Pacific journal of mathematics*, n° 5, p. 285-309, 1955.

L'article de base sur la théorie du point fixe. Il est notamment démontré que toute équation au point fixe construite avec des opérations monotones sur un

treillis a une plus petite solution unique. Ce théorème peut être appliqué pour démontrer que l'équation au point fixe R = F(R) sur le treillis des relations, où F est une expression de l'algèbre relationnelle sans différence, a un plus petit point fixe.

[Tsur86] Tsur D., Zaniolo C., « LDL: a Logic-Based Data Language », Very Large Databases *Int. Conf.,* Kyoto, Japon, septembre 1986.

Une présentation synthétique du langage LDL1 implémenté à MCC. Cet article insiste notamment sur le support de la négation, des ensembles et des fonctions externes.

[Ullman85] Ullman J.D., « Implementation of logical query languages for Databases », *ACM SIGMOD Intl. Conf. on Management of Data*, aussi dans ACM TODS, vol. 10, N. 3, p. 289-321, 1986.

Une description des premières techniques d'optimisation de programmes Datalog. Le passage d'informations de côté est notamment introduit pour remonter les constantes dans les programmes Datalog, y compris les programmes récursifs.

[VanEmden76] Van Emden M., Kowalski R., « The semantics of predicate logic as a programming language », *Journal of the ACM* Vol.23, n° 4, octobre 1976.

Une première étude sur la sémantique du point fixe, développée dans le contexte de la programmation logique. Plus tard, cette approche a été retenue pour définir la sémantique de Datalog.

[Vieille86] Vieille L., « Recursive axioms in deductive databases : the query subquery approach », *Proc. First Intl. Conference on Expert Database Systems*, Charleston, 1986.

Cet article décrit la méthode QSQ, dont l'extension QoSaQ a été implémenté dans le prototype EKS à l'ECRC entre 1986 et 1990. Ce prototype a été ensuite repris pour donner naissance au produit Validity, un des rares DOOD.

[Zaniolo85] Zaniolo C., « The representation and deductive retrieval of complex objects », *11th Int Conf. on Very Large Data Bases*, août 1985.

Une extension de l'algèbre relationnelle au support de fonctions. Cet article présente une extension de l'algèbre relationnelle permettant de référencer des fonctions symboliques dans les critères de projection et de sélection, afin de manipuler des objets complexes. Des opérateurs déductifs de type fermeture transitive étendue sont aussi intégrés.

[Zaniolo86] Zaniolo C., « Safety and compilation of non recursive horn clauses », *MCC Tech. Report DB-088-85*, 1986.

Une discussion des conditions assurant la finitude des prédicats générés par des clauses de Horn et des questions posées sur ces prédicats.

Chapitre XVI

GESTION DE TRANSACTIONS

1. INTRODUCTION

En bases de données, le concept de transaction a été introduit depuis près de trente ans [Bjork72]. Ce concept est aussi d'actualité dans les systèmes d'exploitation. Dans ce chapitre, nous étudions les problèmes liés à la gestion de transactions dans les systèmes de bases de données. Le concept de transaction est fondamental pour maintenir la cohérence des données. Une transaction est une unité de mise à jour composée de plusieurs opérations successives qui doivent être toutes exécutées ou pas du tout. En gestion, les transactions sont courtes (quelques secondes). En conception, elles peuvent être beaucoup plus longues (quelques minutes voire quelques heures). Le SGBD doit garantir les fameuses propriétés ACID des transactions. Outre l'atomicité mentionnée, les transactions effectuent des accès concurrents aux données qui doivent être contrôlés afin d'éviter les conflits entre lecteurs et écrivains. Cela nécessite d'isoler les mises à jour dont les effets doivent par ailleurs être durables. En conséquence, la gestion de transactions mélange intimement les problèmes de fiabilité et de reprise après panne avec les problèmes de concurrence d'accès. Les problèmes de sécurité qui recouvrent la confidentialité sont aussi connexes.

Ce chapitre essaie de faire le point sur tous les aspects de la gestion de transactions dans les SGBD centralisés. Après quelques rappels de base, nous traitons d'abord les problèmes de concurrence. Nous introduisons la théorie de la concurrence, qui s'appuie sur le concept de sérialisation, conduisant à n'accepter que les exécutions simultanées de transactions produisant les mêmes résultats qu'une exécution séquen-

tielle des transactions, c'est-à-dire l'une après l'autre. Nous exposons la technique de verrouillage qui est une méthode préventive de conflits plutôt pessimiste. Bien qu'entraînant des difficultés telles que le verrou mortel, cette méthode est de loin la plus appliquée. Cependant, nous analysons aussi les méthodes optimistes qui laissent les conflits se produire et les corrigent après, telles que l'ordonnancement par estampille, la certification et les contrôles avec versions multiples.

Nous étudions ensuite les principes de la gestion de transactions. Tout repose sur les algorithmes de validation et de reprise après panne. Nous approfondissons les outils nécessaires à ces algorithmes puis les différents algorithmes de validation et de reprise, en incluant la validation en deux étapes, généralement appliquée même dans les systèmes centralisés. Comme exemple de méthode de reprise intégrée, nous décrivons la méthode ARIES implémentée à IBM, la référence en matière de reprise. Nous terminons la partie sur les transactions proprement dites en présentant les principaux modèles de transactions étendus introduits dans la littérature. Ces modèles sont destinés à permettre un meilleur support des transactions longues, en particulier pour les applications de conception qui nécessitent de longs travaux sur des objets complexes. Ces modèles sophistiqués commencent à être introduits dans les SGBD, notamment dans les SGBD objet ou objet-relationnel.

Pour terminer ce chapitre, nous traitons du problème un peu orthogonal de confidentialité. Beaucoup peut être dit sur la confidentialité. Nous résumons l'essentiel des méthodes DAC (*Discretionary Access Control*) et MAC (*Mandatory Access Control*). La première permet d'attribuer des autorisations aux utilisateurs et de les contrôler, la seconde fonctionne avec des niveaux de confidentialité. Nous concluons par quelques mots sur le cryptage des données.

2. NOTION DE TRANSACTION

Un modèle simplifié de SGBD se compose de programmes utilisateurs, d'un système et de mémoires secondaires. Les programmes accèdent aux données au moyen d'opérations du SGBD (*Select*, *Insert*, *Update*, *Delete*), généralement en SQL. Ces opérations déclenchent des actions de lecture et écriture sur disques (*Read*, *Write*), qui sont exécutées par le système, et qui conduisent à des entrées-sorties sur les mémoires secondaires. Afin de pouvoir déterminer les parties de programmes correctement exécutées, le concept de **transaction** a été proposé depuis longtemps.

2.1. EXEMPLE DE TRANSACTION

Les opérations classiques réalisées par une transaction sont les mises à jour ponctuelles de lignes par des écrans prédéfinis. Les transactions sont souvent répétitives.

Elles s'appliquent toujours sur les données les plus récentes. Un exemple typique de transaction est fourni par le banc d'essais TPC A [Gray91]. Il s'agit de réaliser le débit/crédit sur une base de données bancaire. Le benchmark a pour objectif la mesure des performances du SGBD en nombre de transactions exécutées par seconde.

La base se compose des tables Agences, Comptes, Caissiers et Historique (voir figure XVI.1). Une agence gère 100 000 comptes, possède 100 caissiers et comporte un système avec 10 terminaux.

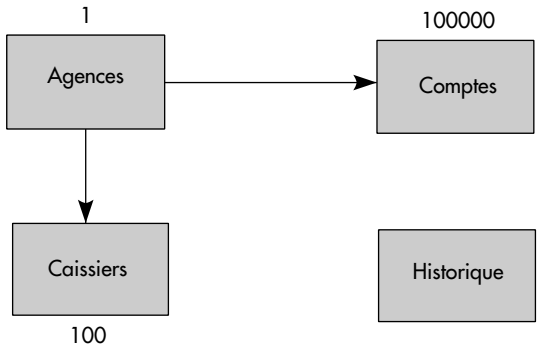


Figure XVI.1 : Schéma de la base TPC A

Le code de la transaction TPC A apparaît figure XVI.2. Il réalise le crédit ou le débit d'un compte d'un montant Delta et répercute cette mise à jour dans les autres tables. On voit qu'il s'agit bien d'un groupe de mises à jour ponctuelles mais liées. Le benchmark suppose que chaque terminal lance l'exécution d'une transaction toute les 10 secondes. 90 % des transactions doivent avoir un temps de réponse inférieur à 2 secondes.

```
Begin-Transaction
   Update Account Set Balance = Balance + Delta
      Where AccountId = Aid ;
   Insert into History (Aid, Tid, Bid, Delta, TimeStamp) ;
   Update Teller Set Balance = Balance + Delta
       Where TellerId = Tid ;
   Update Branch Set Balance = Balance + Delta
      Where TellerId = Tid ;
End-Transaction.
```

Figure XVI.2 : La transaction TPC

La performance est obtenue en divisant le nombre de transactions exécutées par le temps d'exécution moyen. On obtient ainsi un nombre de transactions par seconde.

Ces vingt dernières années, l'objectif de tous les SGBD a été d'exécuter un maximum de telles transactions par seconde.

En pratique, les transactions doivent souvent cohabiter avec des requêtes décisionnelles, traitant un grand nombre de tuples en lecture, souvent pour calculer des agrégats. Par exemple, une requête décisionnelle pourra calculer la moyenne des avoir des comptes par agence comme suit :

```
SELECT B.BRANCHID, AVG(C.BALANCE)
FROM BRANCH B, ACCOUNT C
WHERE B.BRACHID = C.BRANCHID
GROUP BY B.BRANCHID ;
```

Une telle requête parcourt tous les comptes et peut en conséquence empêcher les transactions débit/crédit de s'exécuter. Elle peut aussi être bloquée par les transactions débit/crédit. Nous allons étudier les solutions aux problèmes des transactions ci-dessous.

2.2. PROPRIÉTÉ DES TRANSACTIONS

Une **transaction** est donc composée d'une suite de requêtes dépendantes à la base qui doivent vérifier les propriétés d'**atomicité**, de **cohérence**, d'**isolation** et de **durabilité**, résumées par le vocable ACID.

Notion XVI.1 : Transaction (*Transaction*)

Séquence d'opérations liées comportant des mises à jour ponctuelles d'une base de données devant vérifier les propriétés d'atomicité, cohérence, isolation et durabilité (ACID).

Nous analysons brièvement les propriétés ACID ci-dessous.

2.2.1. Atomicité

Une transaction doit effectuer toutes ses mises à jour ou ne rien faire du tout. En cas d'échec, le système doit annuler toutes les modifications qu'elle a engagées. L'atomicité est menacée par les pannes de programme, du système ou du matériel, et plus généralement par tout événement susceptible d'interrompre une transaction en cours.

2.2.2. Cohérence

La transaction doit faire passer la base de données d'un état cohérent à un autre. En cas d'échec, l'état cohérent initial doit être restauré. La cohérence de la base peut être violée par un programme erroné ou un conflit d'accès concurrent entre transactions.

2.2.3. Isolation

Les résultats d'une transaction ne doivent être visibles aux autres transactions qu'une fois la transaction validée, afin d'éviter les interférences avec les autres transactions. Les accès concurrents peuvent mettre en question l'isolation.

2.2.4. Durabilité

Dès qu'une transaction valide ses modifications, le système doit garantir qu'elles seront conservées en cas de panne. Le problème essentiel survient en cas de panne, notamment lors des pannes disques.

Ces propriétés doivent être garanties dans le cadre d'un système SGBD centralisé, mais aussi dans les systèmes répartis. Elles nécessitent deux types de contrôle, qui sont intégrés dans un SGBD : contrôle de concurrence, résistance aux pannes avec validation et reprise. Nous allons les étudier successivement, puis nous étudierons la méthode intégrée ARIES qui est une des plus efficaces parmi celles implémentées dans les SGBD.

3. THÉORIE DE LA CONCURRENCE

Cette section aborde les problèmes de gestion des accès concurrents. Les solutions proposées permettent de garantir la cohérence et l'isolation des mises à jour des transactions (le C et le I de ACID). Elles sont basées sur la théorie de la sérialisabilité des transactions, que nous examinons maintenant.

3.1. OBJECTIFS

L'objectif général est de rendre invisible aux clients le partage simultané des données. Cette transparence nécessite des contrôles des accès concurrents au sein du SGBD. Ceux-ci s'effectuent au moyen de protocoles spéciaux permettant de synchroniser les mises à jour afin d'éviter les **pertes de mises à jour** et l'**apparitions d'incohérences.**

Une **perte de mise à jour** survient lorsqu'une transaction T1 exécute une mise à jour calculée à partir d'une valeur périmée de donnée, c'est-à-dire d'une valeur modifiée par une autre transaction T2 depuis la lecture par la transaction T1. La mise à jour de T2 est donc écrasée par celle de T1. Une perte de mise à jour est illustrée figure XVI.3. La mise à jour de la transaction T2 est perdue.

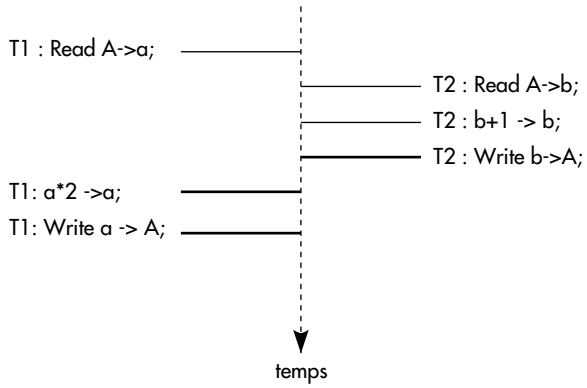


Figure XVI.3 : Exemple de perte de mise à jour

Une **incohérence** apparaît lorsque des données liées par une contrainte d'intégrité sont mises à jour par deux transactions dans des ordres différents, de sorte à enfreindre la contrainte. Par exemple, soient deux données A et B devant rester égales. L'exécution de la séquence d'opérations {T1 : A = A+1 ; T2 : B = B*2 ; T2 : A = A*2; T1: B=B+1} rend en général A différent de B, du fait de la non-commutativité de l'addition et de la multiplication. Elle provoque donc l'apparition d'une incohérence. Cette situation est illustrée figure XVI.4.

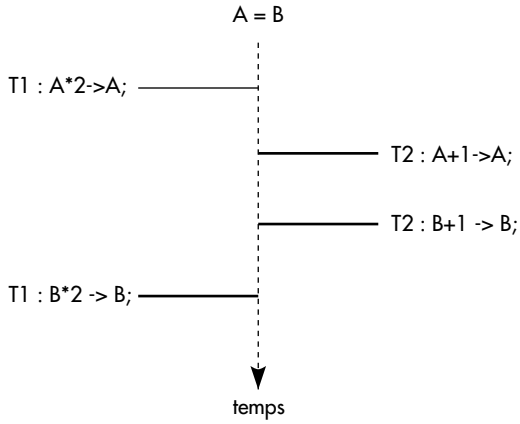


Figure XVI.4 : Exemple d'introduction d'incohérence

Un autre problème lié aux accès concurrents est la **non-reproductibilité des lectures** : deux lectures d'une même donnée dans une même transaction peuvent conduire à des valeurs différentes si la donnée est modifiée par une autre transaction entre les deux lectures (voir figure XVI.5). Le problème ne survient pas si les mises à jour sont isolées, c'est-à-dire non visibles par une autre transaction avant la fin de la transaction. Il en va de même de l'apparition d'incohérences. Pour les pertes de mise à

jour, l'isolation des mises à jour n'est pas suffisante : il faut aussi ne pas laisser deux transactions modifier simultanément une même donnée.

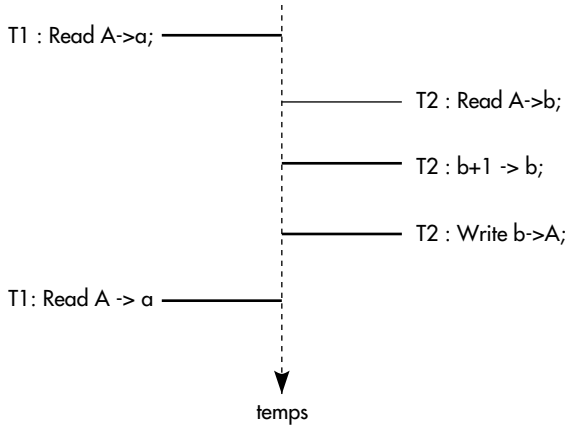


Figure XVI.5 : Exemple de non-reproductibilité des lectures

La résolution dans un système des problèmes évoqués nécessite la mise en place d'algorithmes de contrôle de concurrence spécialisés. Ces algorithmes s'appuient sur la théorie de la concurrence, que nous examinons ci-dessous.

3.2. QUELQUES DÉFINITIONS DE BASE

Pour éviter perte d'opérations, incohérences et non reproductibilité des lectures, le SGBD doit contrôler l'accès aux données. L'unité de données contrôlée dépend du SGBD. De plus en plus de SGBD permettent des contrôles variables selon le type de transactions. Nous appellerons cette unité **granule de concurrence ;** le terme objet est parfois aussi employé.

Notion XVI.2 : Granule de concurrence (*Concurrency granule*)

Unité de données dont les accès sont contrôlés individuellement par le SGBD.

Un granule au sens de la concurrence peut être une ligne, une page ou une table dans un système relationnel. Ce peut être un objet ou une page dans un SGBD objet. Nous discuterons de la taille du granule de concurrence plus loin.

Les granules de concurrence sont lus et écrits par les utilisateurs, éventuellement par parties. On appelle **action** un accès élémentaire à un granule.

Notion XVI.3 : Action (*Action*)

Unité indivisible exécutée par le SGBD sur un granule pour un utilisateur, constituée généralement par une lecture ou une écriture.

Un système de bases de données exécute donc une suite d'actions résultant de transactions concurrentes. Après complétude d'un ensemble de transactions (T1, T2…Tn), une histoire du système peut être représentée par la suite des actions exécutées. Plus généralement, toute suite d'actions pouvant représenter une histoire possible sera appelée simplement **exécution**.

Notion XVI.4 : Exécution de transactions (*Schedule*, ou *Log*, ou *History*)

Séquence d'actions obtenues en intercalant les diverses actions des transactions T1, T2 …Tn tout en respectant l'ordre interne des actions de chaque transaction.

Une exécution respecte donc l'ordre des actions de chaque transaction participante et est, par définition, séquentielle. Par exemple, considérons les transactions T1 et T2 figure XVI.6, modifiant les données A et B reliées par la contrainte d'intégrité A = B ; A et B appartiennent à deux granules distincts, maximisant ainsi les possibilités de concurrence. Une exécution correcte de ces deux transactions est représentée figure XVI.7 (a). Une autre exécution est représentée figure XVI.7 (b), mais celle-là est inacceptable car elle conduit à une perte d'opérations.

T1	**T2**
Read A → a1	Read A → a2
a1+1 → a1	a2*2 → a2
Write a1 → A	Write a2 → A
Read B → b1	Read B → b2
b1+1 → b1	b2*2 → b2
Write b1 → B	Write b2 → B

Figure XVI.6 : Deux transactions T1 et T2

T1 : Read A → a1	T2 : Read A → a2
T1 : a1+1 → a1	T2 : a2*2 → a2
T1 : Write a1 → A	T1 : Read A → a1
T2 : Read A → a2	T1 : a1+1 → a1
T2 : a2*2 → a2	T2 : Write a2 → A
T2 : Write a2 → A	T2 : Read B → b2
T1 : Read B → b1	T2 : b2*2 → b2
T1 : b1+1 → b1	T1 : Write a1 → A
T1 : Write b1 → B	T1 : Read B → b1
T2 : Read B → b2	T1 : b1+1 → b1
T2 : b2*2 → b2	T1 : Write b1 → B
T2 : Write b2 → B	T2 : Write b2 → B
(a)	(b)

Figure XVI.7 : Deux exécutions des transactions T1 et T2

3.3. PROPRIÉTÉS DES OPÉRATIONS SUR GRANULE

Un granule accédé concurremment obéit à des contraintes d'intégrité internes. Lors des modifications de la base de données, les granules sont modifiés par des suites d'actions constituant des unités fonctionnelles appelées **opérations**. Les opérations respectent la cohérence interne du granule, c'est-à-dire les contraintes d'intégrité qui relient les données appartenant au granule.

Notion XVI.5 : Opération (*Operation*)

Suite d'actions accomplissant une fonction sur un granule en respectant sa cohérence interne.

Par exemple, si le granule est la page, les opérations de base sont souvent LIRE (page) et ECRIRE (page), qui sont également dans bien des systèmes des actions indivisibles. Si le granule est l'article, des opérations plus globales nécessitant plusieurs actions indivisibles sont LIRE (article) et ECRIRE (article), mais aussi MODIFIER (article) et INSERER (article). Avec ces opérations de base, il est possible d'en construire d'autres plus globales encore. Sur un objet typé, tel un compte en banque, on peut distinguer des opérations, créer, créditer, débiter, détruire, etc.

L'application d'opérations à des granules conduit à des **résultats**. Le résultat d'une opération est constitué par l'état du granule concerné après l'application de l'opération considérée et par les effets de bords qu'elle provoque. Par exemple, le résultat d'une opération LIRE est représenté par la valeur du tampon récepteur après exécution, alors que le résultat d'une transaction modifiant une base de données est l'état des granules modifiés après exécution ainsi que la valeur des messages édités.

Les opérations sont enchevêtrées au niveau des actions lors de l'exécution simultanée de transactions. Deux opérations qui ne modifient aucun granule et qui appartiennent à deux transactions différentes peuvent être enchevêtrées de manière quelconque sans modifier les résultats de leur exécution. Autrement dit, toute intercalation d'opérations n'effectuant que des lectures conduit à des résultats identiques à une exécution successive de ces opérations. Plus généralement, il est possible de définir la notion **d'opérations compatibles**.

Notion XVI.6 : Opérations compatibles (*Compatible operations*)

Opérations 0i et 0j dont toute exécution simultanée donne le même résultat qu'une exécution séquentielle 0i suivie de 0j ou de 0j suivie de 0i (à noter que les résultats 0i puis 0j, et 0j puis 0i peuvent être différents).

Considérons par exemple les opérations représentées figure XVI.8. Les opérations O11 et O21 sont compatibles ; O11 et O12 ne le sont pas.

Il est important de remarquer que deux opérations travaillent sur deux granules différents sont toujours compatibles. En effet, dans ce cas aucune perte d'opérations ne

peut survenir si l'on intercale les opérations. Or il est simple de voir que deux opérations sont incompatibles lorsque qu'il existe une possibilité d'intercalation générant une perte d'opérations.

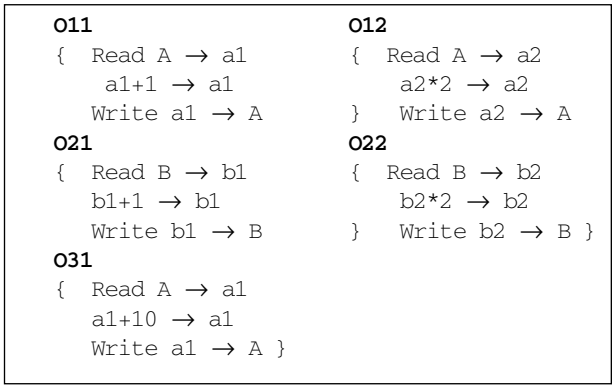


Figure XVI.8 : Exemple d'opérations

Les problèmes surviennent avec les opérations incompatibles, lorsqu'une au moins modifie un granule auquel l'autre a accédé. L'ordre d'exécution des deux opérations peut alors changer les résultats. Dans d'autres cas, il peut être indifférent. Plus généralement, nous définirons la notion d'**opérations permutables**, qu'il faut bien distinguer de celle d'opérations compatibles (la première est une notion indépendante de l'ordre d'exécution, alors que la seconde est définie à partir de la comparaison des ordres d'exécution).

Notion XVI.7 : Opérations permutables (*Permutable operations*)

Opérations 0i et 0j telles que toute exécution de 0i suivie par 0j donne le même résultat que celle de 0j suivie par 0i.

Par exemple, les opérations O11 et O31 représentées figure XVI.8 sont permutables alors que les opérations O11 et O12 ne le sont pas. Soulignons que deux opérations travaillant sur des granules différents sont toujours permutables. En effet, dans ce cas, l'exécution de la première ne peut modifier le résultat de la seconde et réciproquement. Par exemple, O11 et O12 sont permutables. Plus généralement, deux opérations compatibles sont permutables, mais la réciproque n'est pas vraie.

3.4. CARACTÉRISATION DES EXÉCUTIONS CORRECTES

Certaines exécutions introduisent des pertes d'opérations ou des inconsistances, comme nous l'avons vu ci-dessus. L'objectif du contrôle de concurrence consiste à ne

laisser s'exécuter que des exécutions sans pertes d'opérations ou inconsistances. Il est bien connu que l'exécution successive de transactions (sans simultanéité entre transactions) est un cas particulier d'exécution sans perte d'opérations ni inconsistances. Une telle exécution est appelée **succession** et peut être définie plus formellement comme suit :

Notion XVI.8 : Succession (*Serial Schedule*)

Exécution E d'un ensemble de transactions {T1, T2…Tn} telle qu'il existe une permutation π de (1,2…n) telle que :

$E = < T\pi(1) ; T\pi(2) ; \ldots T\pi(n) >$

Afin d'assurer l'absence de conflit, il est simple de ne tolérer que les exécutions qui donnent le même résultat qu'une succession pour chaque transaction. De telles exécutions sont dites **sérialisables**.

Notion XVI.9 : Exécution sérialisable (*Serialisable Schedule*)

Exécution E d'un ensemble de transactions {T1, T2…Tn} donnant globalement et pour chaque transaction participante le même résultat qu'une succession de (T1, T2…Tn).

Le problème du contrôle de concurrence est donc d'assurer qu'un système centralisé (ou réparti) ne peut générer que des exécutions sérialisables. C'est là une condition suffisante pour assurer l'absence de conflit dont la nécessité peut être discutée [Gardarin77]. En fait, la condition est nécessaire si le système n'a pas de connaissances sur la sémantique des opérations.

Afin de caractériser les exécutions sérialisables, nous introduisons deux transformations de base d'une exécution de transactions. Tout d'abord, la séparation d'opérations compatibles Oi et Oj exécutées par des transactions différentes consiste à remplacer une exécution simultanée des opérations E (Oi, Oj) par la séquence donnant le même résultat, soit < Oi ; Oj > ou < Oj ; Oi >. La séparation d'opérations permet donc de mettre en succession des opérations compatibles exécutées par des transactions différentes. Ensuite, la permutation d'opérations permutables Oi et Oj exécutées par des transactions différentes consiste à changer l'ordre d'exécution de ces opérations ; par exemple la séquence < Oi ; Oj > est remplacée par la séquence < Oj ; Oi >.

Une condition suffisante pour qu'une exécution soit sérialisable est qu'elle puisse être transformée par séparation des opérations compatibles et permutations des opérations permutables en une succession des transactions composantes. En effet, par définition, séparations et permutations conservent les résultats. Par suite, si l'exécution peut être transformée en une succession, elle donne le même résultat que cette succession pour chaque transaction et est donc sérialisable. La condition n'est pas nécessaire car, au moins pour certaines valeurs des données, des opérations incompatibles ou non permutables peuvent être exécutées simultanément sans conflits.

À titre d'exemple, considérons l'exécution représentée figure XVI.7(a). En représentant seulement globalement les opérations, cette exécution s'écrit :

T1 : A + 1→ A
T2 : A * 2 → A
T1 : B + 1 → B
T2 : B * 2 → B

Les opérations A *2 → A et B + 1 → B sont permutables car elles agissent sur des granules différents. Par suite, cette exécution peut être transformée en :

T1 : A + 1 → A
T1 : B + 1 → B
T2 : A * 2 → A
T2 : B * 2 → B

qui est une succession de T1 puis T2. Par suite, l'exécution figure XVI.7(a) est sérialisable.

3.5. GRAPHE DE PRÉCÉDENCE

Une exécution sérialisable est correcte car elle donne un résultat que l'on obtiendrait en exécutant les transactions l'une après l'autre. Lorsqu'on examine une séquence d'opérations résultant d'une exécution simultanée d'un ensemble de transactions, il apparaît que l'ordre de certaines opérations ne peut être changé sans changer le résultat, du fait de la non-commutativité des opérateurs exécutés (par exemple, addition et multiplication).

Les chercheurs ont ainsi abouti à définir la notion de **précédence** de transactions dans une exécution simultanée.

Notion XVI.10 : Précédence (*Precedence*)

Propriété stipulant qu'une transaction a accompli une opération Oi sur une donnée avant qu'une autre transaction accomplisse une opération Oj, Oi et Oj n'étant pas commutatives ({Oi; Oj} ≠ {Oj; Oi}).

La notion de précédence est générale et s'applique à tout type d'opération. En pratique, les systèmes ne considèrent d'ordinaire que les opérations de lecture et d'écriture. Les précédences sont alors créées par les séquences d'actions de base lecture et écriture. Les séquences non commutatives lecture puis écriture, écriture puis lecture, écriture puis écriture, d'une même donnée introduisent des précédences. Plus précisément, l'une des séquences :

– Ti : lire(D) … Tj : écrire(D) ;

– Ti : écrire(D) … Tj : écrire(D) ;

– Ti : écrire(D) … Tj : lire(D) ;

implique que Ti précède Tj.

Considérons une exécution simultanée de transactions. La relation de précédence entre transactions peut être représentée par un graphe :

Notion XV.11 : Graphe de précédence (*Precendency graph*)

Graphe dont les nœuds représentent les transactions et dans lequel il existe un arc de Ti vers Tj si Ti précède Tj dans l'exécution analysée.

Une exécution simultanée des transactions T1, T2 et T3 et le graphe de précédence associé sont illustrés figure XVI.9.

Il est simple de montrer qu'une condition suffisante de sérialisabilité est que le graphe de précédence soit sans circuit. En effet, dans ce cas, il est toujours possible de transformer l'exécution simultanée en une succession en séparant puis permutant les opérations. L'ordre des transactions dans la succession est induit par le graphe sans circuit. Par exemple, l'exécution simultanée représentée figure XVI.9 n'est pas sérialisable puisque le graphe de précédence possède un circuit.

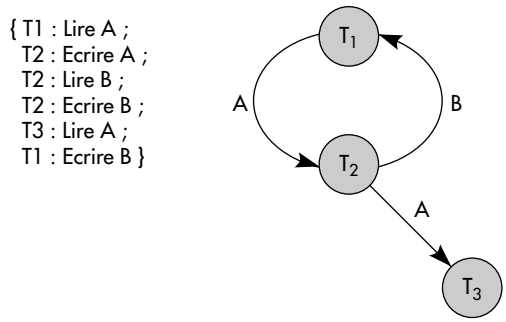


Figure XVI.9 : Exemple de graphe de précédence

4. CONTRÔLE DE CONCURRENCE PESSIMISTE

Deux types de techniques ont été développées pour garantir la sérialisabilité des transactions : les techniques de prévention des conflits qui empêchent leur apparition et

les techniques de détection qui laissent les conflits se produire mais les détectent et annulent leurs effets. Les premières sont dites pessimistes car elles préviennent des conflits qui ne surviennent en général pas. Elles sont basées sur le verrouillage. Les secondes sont dites optimistes. Dans cette partie, nous étudions le verrouillage qui est de loin la technique la plus appliquée.

4.1. LE VERROUILLAGE DEUX PHASES

Le verrouillage deux phases est une technique de prévention des conflits basée sur le blocage des objets par des verrous en lecture ou écriture avant d'effectuer une opération de sélection ou de mise à jour. En théorie, une transaction ne peut relâcher de verrous avant d'avoir obtenu tous ceux qui lui sont nécessaires, afin de garantir la correction du mécanisme [Eswaran76].

Notion XVI.12 : Verrouillage deux phases (*Two Phase Locking*)

Technique de contrôle des accès concurrents consistant à verrouiller les objets au fur et à mesure des accès par une transaction et à relâcher les verrous seulement après obtention de tous les verrous.

Une transaction comporte donc deux phases (voir figure XVI.10) : une phase d'acquisition de verrous et une phase de relâchement. Cette condition garantit un ordre identique des transactions sur les objets accédés en mode incompatible. Cet ordre est celui d'exécution des points de verrouillage maximal φ. En pratique, afin de garantir l'isolation des mises à jour, les verrous sont seulement relâchés en fin de transaction, lors de la validation.

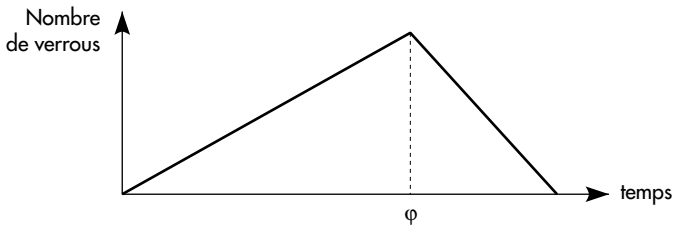


Figure XVI.10 : Comportement des transactions deux phases

Les verrous sont demandés au moyen de l'opération `Lock(G,M)` et relâchés au moyen de l'opération `Unlock(G)`, G étant le granule à verrouiller/déverrouiller et M le mode de verrouillage. Les compatibilités entre opérations découlent des précédences ; elles sont décrites par la matrice représentée figure XVI.11. Les algorithmes `Lock` et `Unlock` sont détaillés figure XVI.12. Lors d'une demande de verrouillage,

si l'objet demandé est verrouillé, la transaction demandante est mise en attente jusqu'à libération de l'objet. Ainsi, toute transaction attend la fin des transactions incompatibles, ce qui garantit un graphe de précédence sans circuit. Une analyse fine montre que les circuits sont transformés en verrous mortels.

	L	E
L	V	F
E	F	F

Figure XVI.11 : Compatibilité des opérations de lecture/écriture

```
Bool Function Lock(Transaction t, Granule G, Mode M){
Cverrou := 0 ;
Pour chaque transaction i ≠ t ayant verrouillé l'objet G faire {
   Cverrou := Cverrou ∪ t.verrou(G) }; // cumuler verrous sur G
   si Compatible (Mode, Cverrou) alors {
      t.verrou(G) = t.verrou(G) ∪ M; // marquer l'objet verrouillé
      Lock := true ; }
   sinon {
      insérer (t, Mode) dans queue de G ; // mise en attente de t
      bloquer la transaction t ;
      Lock := false ; } ;
} ;
```

```
Procédure Unlock(Transaction t, Granule G){
t.verrou(G) := 0 ; // Remise à 0 du verrou de la transaction sur G
Pour chaque transaction i dans la queue de G faire {
   si Lock(i, G,M) alors { // Tentative de verrouillage pour Ti
      enlever (i,M) de la queue de G ;
      débloquer i ; } ; };
} ;
```

Figure XVI.12 : Algorithmes de verrouillage et déverrouillage

L'application du verrouillage dans un système pose le problème du choix du granule de verrouillage. Dans une base de données relationnelle, les objets à verrouiller peuvent être des tables, des pages ou des tuples. Une granularité variable des verrous est souhaitable, les transactions manipulant beaucoup de tuples pouvant verrouiller au niveau table ou page, celles accédant ponctuellement à quelques tuples ayant la capacité de verrouiller au niveau tuple. Nous examinerons ci-dessous le problème des granules de taille variable. Le choix d'une unité de verrouillage fine (par exemple le tuple) minimise bien sûr les risques de conflits. Elle maximise cependant la complexité et le coût du verrouillage.

4.2. DEGRÉ D'ISOLATION EN SQL2

Le verrouillage, tel que présenté ci-dessus, est très limitatif du point de vue des exécutions simultanées possibles. Afin de proposer une approche plus permissive et de laisser s'exécuter simultanément des transactions présentant des dangers limités de corruption des données, le groupe de normalisation de SQL2 a défini des **degrés d'isolation** emboîtés, du moins contraignant au plus contraignant, ce dernier correspondant au verrouillage deux phases. Le groupe distingue les **verrous courts** relâchés après exécution de l'opération et les **verrous longs** relâchés en fin de transaction. Le degré de verrouillage souhaité est choisi par le développeur de la transaction parmi les suivants :

– Le degré 0 garantit les non pertes des mises à jour ; il correspond à la pose de verrous courts exclusifs lors des écritures.
– Le degré 1 garantit la cohérence des mises à jour ; il génère la pose de verrous longs exclusifs en écriture par le système.
– Le degré 2 assure la cohérence des lectures individuelles ; il ajoute la pose de verrous courts partagés en lecture à ceux du degré 1.
– Le degré 3 atteste de la reproductibilité des lectures ; il complète le niveau 2 avec la pose de verrous longs partagés en lecture.

Ainsi, le développeur peut contrôler la pose des verrous. Un choix autre que le degré 3 doit être effectué avec précaution dans les transactions de mises à jour, car il implique des risques d'incohérence. Seul en effet le degré 3 assure la sérialisabilité des transactions.

4.3. LE PROBLÈME DU VERROU MORTEL

4.3.1. Définition

Le verrouillage soulève quelques problèmes. Le problème essentiel est le risque de **verrous mortels** entre transactions.

Notion XVI.13 : Verrou mortel (*Deadlock*)

Situation dans laquelle un groupe de transactions est bloqué, chaque transaction du groupe attendant qu'une autre transaction du groupe relâche un verrou pour pouvoir continuer.

Une transaction Ti attend une transaction Tj si Ti a demandé l'obtention d'un verrou sur un objet verrouillé par Tj en mode incompatible. La figure XVI.13 donne un exemple de verrou mortel.

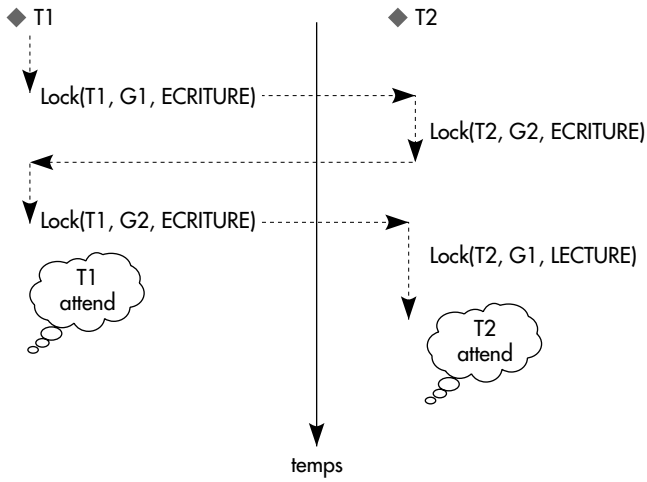


Figure XVI.13 : Exemple de verrou mortel

Deux classes de solutions sont possibles dans les SGBD afin de résoudre le problème du verrou mortel : la première, appelée **prévention**, empêche les situations de verrous mortels de survenir ; la seconde, appelée **détection**, est une solution curative qui consiste à supprimer les verrous mortels par reprise de transactions.

4.3.2. Représentation du verrou mortel

Nous présentons ci-dessous deux types de graphes représentant les verrous mortels : le graphe des attentes et le graphe des allocations.

4.3.2.1. Graphe des attentes

Le **graphe des attentes** [Murphy68] est un graphe G (T,Γ) où T est l'ensemble des transactions concurrentes $\{T_1, T_2 \ldots T_n\}$ se partageant les granules $G_1, G_2 \ldots G_m$ et Γ est la relation « attend » définie par : T_p « attend » T_q si et seulement si T_p attend le verrouillage d'un objet G_i alors que cette requête ne peut pas être acceptée parce que G_i a déjà été verrouillé par T_q.

Notion XVI.14 : Graphe des attentes (*Waiting graph*)

Graphe dont les nœuds correspondent aux transactions et les arcs représentent les attentes entre transactions.

Le théorème suivant a été introduit dès 1968 [Murphy68] : il existe une situation de verrou mortel si et seulement si le graphe des attentes possède un circuit. La figure XVI.14 illustre ce théorème sur l'exemple introduit ci-dessus. La preuve est

simple. En effet, si le graphe des attentes possède un circuit, c'est qu'il existe un groupe de transactions tel que : T_1 attend T_2, T_2 attend $T_{3...}$,T_k attend T_1. Chaque transaction du groupe est donc bloquée en attente d'un objet du fait de l'utilisation de cet objet par une autre transaction du groupe. La fin d'exécution de toutes les transactions n'appartenant pas au groupe ne permet donc pas de débloquer une transaction du groupe.

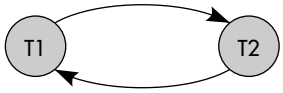


Figure XVI.14 : Exemple de graphe d'attente avec circuit

Réciproquement, l'existence d'une situation de verrou mortel implique l'existence d'au moins un circuit. S'il n'en était pas ainsi, tout groupe de transaction serait tel que le sous-graphe des attentes qu'il engendre ne possèderait pas de circuit. Après exécution de toutes les transactions n'appartenant pas au groupe, il serait donc possible de débloquer une transaction du groupe puisqu'un graphe sans circuit possède au moins un sommet pendant. Toutes les transactions appartenant à un circuit sont en situation de verrou mortel ; de plus, une transaction attendant une transaction en situation de verrou mortel est elle-même en situation de verrou mortel (voir figure XVI.15).

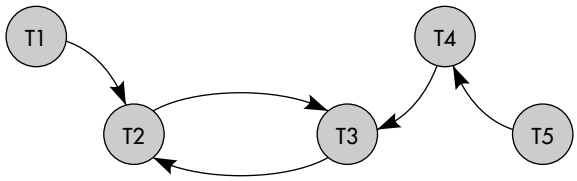


Figure XVI.15 : Transactions en situation de verrou mortel

Il est intéressant d'établir le rapport entre graphes des attentes et graphe de précédence. Par définition, si une transaction T_i `attend` une transaction T_j, alors T_j a verrouillé un objet O dont le verrouillage est demandé par T_i dans un mode incompatible. Ainsi, l'opération pour laquelle Tj a verrouillé O sera exécutée avant celle demandée par Tj car les deux opérations sont incompatibles et donc non permutables. Donc Tj `précède` Ti. Toutefois, la relation de précédence n'implique généralement pas la relation d'attente. Donc, en changeant l'orientation des arcs du graphe des attentes, on obtient un sous-graphe du graphe de précédence. Cela implique que si le graphe des attentes a un circuit, il en sera de même pour le graphe de précédence. En conséquence, une situation de verrou mortel ne peut pas donner lieu à une exécution sérialisable même s'il était possible de terminer les transactions interbloquées.

4.3.2.2. Graphe des allocations

Le **graphe des allocations** [Holt72] est composé de deux ensembles de sommets :

1. l'ensemble T des transactions
2. l'ensemble O des objets.

Un arc relie l'objet Oi à la transaction Tp si et seulement si Tp a obtenu le verrouillage de Oi dans au moins un mode d'opération ; l'arc est valué par les modes d'opérations alloués. Un arc relie la transaction Tp à l'objet Oi si et seulement si Tp a demandé et n'a pas encore obtenu l'allocation de ce granule ; l'arc est valué par les modes d'opérations demandés. La figure XVI.16 représente le graphe des allocations de l'exemple de la figure XVI.13.

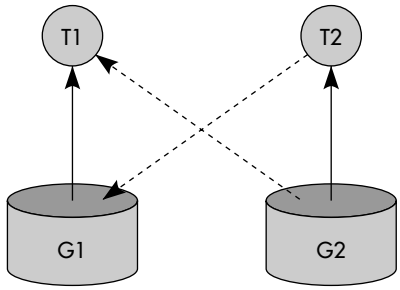


Figure XVI.16 : Exemple de graphe des allocations

Il est simple de démontrer le théorème suivant : une condition nécessaire d'existence de verrou mortel est la présence d'un circuit sur le graphe des allocations. Cette condition n'est en général pas suffisante. La preuve s'effectue par l'absurde. En effet, il est possible de prouver que s'il n'existe pas de circuits sur le graphe des allocations, il ne peut y avoir d'interblocage. En effet, soit T un groupe quelconque de transactions. Du fait que le graphe des allocations est sans circuit, le sous-graphe obtenu après exécution des transactions n'appartenant pas à T est sans circuit. Il possède donc un sommet pendant. Ce sommet ne peut être un granule car un granule non verrouillé ne peut être attendu. Dans tout groupe de transactions T, l'exécution supposée de toutes les transactions n'appartenant pas au groupe T conduit donc à débloquer une transaction du groupe. Il n'y a donc pas situation de verrou mortel.

Il faut remarquer que la condition est suffisante dans le cas où les seuls modes de lecture et d'écriture sont distingués. Ceci permet en général de détecter les situations de verrou mortel par détection de circuits dans le graphe des allocations dans la plupart des systèmes classiques. Nous pouvons noter qu'en général il n'y a pas de rapport direct entre graphe de précédence et graphe des allocations. Cependant, si les seuls modes existants sont lecture et écriture, la présence d'un circuit dans le graphe des allocations est équivalente à l'existence d'une situation de verrou mortel et donc à

celle d'un circuit dans le graphe des attentes. Sous cette condition, la présence d'un circuit dans le graphe des allocations entraîne ainsi celle d'un circuit dans le graphe de précédence.

4.3.3. Prévention du verrou mortel

La prévention consiste à appliquer une stratégie de verrouillage garantissant que le problème ne survient pas. Il existe classiquement deux approches, l'une basée sur l'ordonnancement des ressources, l'autre sur celui des transactions. L'ordonnancement des ressources (tables, pages, objets, tuples) pour les allouer dans un ordre fixé aux transactions est impraticable vu le grand nombre d'objets distribués. L'ordonnancement des transactions est possible à partir d'une **estampille**.

Notion XVI.15 : Estampille de transaction (*Transaction Timestamp*)

Numéro unique attribué à une transaction permettant de l'ordonner strictement par rapport aux autres transactions.

En général, l'estampille attribuée à une transaction est son horodate de lancement concaténée avec le numéro de processeur sur lequel elle est lancée, ceci afin d'empêcher l'égalité des estampilles pour deux transactions lancées au même instant : celles-ci diffèrent alors par le numéro de processeur en poids faibles. Le numéro de processeur n'est utile que dans les architectures parallèles.

À partir des estampilles, deux algorithmes ont été proposés [Rosenkrantz77] pour prévenir les verrous mortels. Tous deux consistent à défaire plus ou moins directement une transaction dans le cas d'attente, de sorte à ne permettre que des attentes sans risque de circuit. L'algorithme WAIT-DIE consiste à annuler les transactions qui demandent des ressources tenues par des transactions plus anciennes. La transaction la plus récente est alors reprise avec la même estampille ; elle finit ainsi par devenir ancienne et par passer. Il ne peut y avoir de verrou mortel, les seules attentes possibles étant dans l'ordre où une transaction ancienne attend une transaction récente. Le contrôle des attentes imposé par l'algorithme est précisé figure XVI.17.

```
// Algorithm WAIT-DIE
Procédure Attendre (Ti,Tj) {
   // Ti réclame un verrou tenu par Tj
   si ts(Ti) < ts(Tj) alors Ti waits sinon Ti dies ; }
```

Figure XVI.17 : Contrôle des attentes dans l'algorithme WAIT-DIE

L'algorithme WOUND-WAIT est un peu plus subtil. Tout type d'attente est permis. Mais si une transaction plus ancienne attend une plus récente, la récente est blessée (*wounded*), ce qui signifie qu'elle ne peut plus attendre : si elle réclame un verrou

tenu par une autre transaction, elle est automatiquement défaite et reprise. Le contrôle des attentes imposé par l'algorithme est représenté figure XVI.18 ; une transaction blessée ne peut donc attendre.

```
// Algorithm WOUND-WAIT
Procédure Attendre (Ti,Tj) {
   // Ti réclame un verrou tenu par Tj
   si ts(Ti) < ts(Tj) alors Tj is wounded sinon Ti waits ; }
```

Figure XVI.18 : Contrôle des attentes dans l'algorithme WOUND-WAIT

Les deux algorithmes empêchent les situations de verrous mortels en donnant priorité aux transactions les plus anciennes. L'algorithme WOUND-WAIT provoque en principe moins de reprises de transactions et sera en général préféré.

4.3.4. Détection du verrou mortel

La prévention provoque en général trop de reprises de transactions, car les méthodes défont des transactions alors que les verrous mortels ne sont pas sûrs d'apparaître. Au contraire, la détection laisse le problème se produire, détecte les circuits d'attente et annule certaines transactions afin de rompre les circuits d'attente.

Un algorithme de détection de l'interblocage peut se déduire d'un algorithme de détection de circuits appliqué au graphe des attentes ou des allocations. Nous présentons ici une mise en œuvre de l'algorithme qui consiste à tester si un graphe est sans circuit par élimination successive des sommets pendants.

Sur le graphe des attentes, un sommet est pendant si la transaction qu'il représente n'attend le verrouillage d'aucun granule. Soit N(k) le nombre de granules dont la transaction T_k attend le verrouillage. Une première réduction du graphe peut être obtenue par élimination des sommets pendants, donc tels que N(k) = 0. Le problème est alors de recalculer les nombres de granules attendus N(k) après réduction pour pouvoir effectuer la réduction suivante. Ceci peut être fait en comptant les demandes qui peuvent être satisfaites après chaque réduction, et en décrémentant N(k) chaque fois que l'on compte une demande de la transaction T_k. L'application de la méthode nécessite deux précautions :

1. marquer les demandes comptées pour ne pas les compter deux fois ;
2. disposer d'une procédure permettant de tester si une demande peut être satisfaite compte tenu de l'état des verrouillages des transactions non encore éliminées du graphe des attentes.

Soit donc `SLOCK (k, G, M)` une procédure booléenne permettant de tester si la demande du granule G en mode M de la transaction Tk peut être satisfaite compte tenu de l'état d'allocation des granules aux transactions présentes dans le graphe des attentes. Cette procédure répond `VRAI` si la demande peut être satisfaite et `FAUX`

sinon. Le code de cette procédure est analogue à celui de l'algorithme LOCK vu ci-dessus, à ceci près que seule les transactions de T sont prises en compte (les autres sont supposées exécutées et terminées) et que l'état de verrouillage n'est pas modifié. La figure XVI.19 présente une procédure DETECTER répondant VRAI s'il y a situation de verrou mortel et FAUX sinon. Cette procédure élimine donc progressivement les transactions pendantes du graphe des attentes.

```
Bool Procedure Detecter {
T = {Liste des transactions telles que N(k) ≠ 0 }
G = {liste des granules allouées aux transactions dans T}
Pour chaque entrée g de G faire
   Pour chaque demande non marquée M de Tk en attente de g faire {
         Si SLOCK(k, g, Q) = VRAI alors {
               Marquer Q ;
               N(k) = N(k) −1 ;
               Si N(k) = 0 alors {
                      Eliminer Tk de T ;
                      Ajouter les granules verrouillés par Tk à G ;
                      } ;
               } ;
         } ;
   Si T = ∅ alors DETECTER = FAUX
   Sinon DETECTER = VRAI ;
} ;
```

Figure XVI.19 : Algorithme de détection du verrou mortel

Quand une situation d'interblocage est détectée, le problème qui se pose est de choisir une transaction à recycler de façon à briser les circuits du graphe des attentes. L'algorithme de détection présenté ci-dessus fournit la liste des transactions en situation d'interblocage. Il faut donc choisir une transaction de cette liste. Cependant, tous les choix ne sont pas judicieux, comme le montre la figure XVI.20. Une solution à ce problème peut être de recycler la transaction qui bloque le plus grand nombre d'autres transactions, c'est-à-dire qui correspond au sommet de demi-degré intérieur le plus élevé sur le graphe des attentes. Le choix de la transaction à reprendre doit aussi chercher à minimiser le coût de reprise.

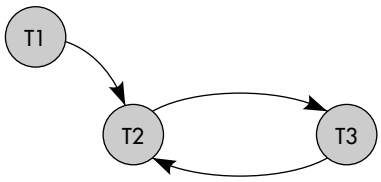


Figure XVI.20 : Exemple de choix difficile de transaction à reprendre

Le coût d'une solution de type détection avec reprise peut être réduit. En effet, il est possible de déclencher un algorithme détection seulement quand une transaction attend un verrouillage depuis un temps important (par exemple, quelques secondes), plutôt qu'à chaque début d'attente.

D'autres algorithmes de détection sont possibles. Le graphe d'allocation est souvent utilisé dans les systèmes répartis. Lors d'une attente qui dure, un algorithme envoie une enquête le long des arcs du graphe des allocations. Cette enquête est transmise au granule attendu, puis aux transactions bloquant ce granule, puis aux granules attendus s'il en existe, etc. Si l'enquête revient à la transaction initiale, c'est qu'il y a un verrou mortel. Cet algorithme est moins efficace en centralisé.

4.4. AUTRES PROBLÈMES SOULEVÉS PAR LE VERROUILLAGE

Un autre problème soulevé par le verrouillage est le problème de famine, encore appelé **blocage permanent**. Ce problème survient dès qu'un groupe de transactions se coalise, en effectuant des opérations compatibles entre elles (par exemple des lectures), contre une transaction individuelle qui désire effectuer une opération incompatible avec les précédentes (par exemple une écriture). La transaction individuelle peut alors attendre indéfiniment. Les solutions à ce problème consistent en général à mettre en file d'attente les demandes de verrouillage dans leur ordre d'arrivée et à n'accepter une requête de verrouillage que si elle est compatible avec les verrouillages en cours et ceux demandés par les requêtes les plus prioritaires en attente. Il faut noter que les algorithmes de prévention DIE-WAIT et WOUND-WAIT ne conduisent jamais une transaction à l'attente infinie. En effet, une transaction qui meurt garde son ancienne estampille lorsqu'elle est relancée. Elle devient ainsi plus vieille et finit toujours par passer, le principe étant d'avorter les transactions les plus jeunes.

Le **problème des fantômes** a également été soulevé [Eswaran76]. Il survient lorsqu'un objet est introduit dans la base de données et ne peut être pris en compte par une transaction en cours qui devrait logiquement le traiter. Par exemple, soit la base de données de réservation de places d'avions, représentée figure XVI.21, composée de deux relations : PASSAGER (nom, numéro de vol, numéro de siège) et OCCUPATION (numéro de vol, nombre de passagers). Considérons maintenant les transactions suivantes :

– T1 (1re partie) : lister la relation PASSAGER en lisant tuple à tuple ;
– T1 (2e partie) : lister la relation OCCUPATION d'un seul tuple ;
– T2 : insérer dans PASSAGER le tuple (Fantomas, 100, 13) et incrémenter le nombre de passagers du vol numéro 100.

Les transactions sont supposées verrouiller les tuples. Supposons que les transactions s'enchevêtrent dans l'ordre : T1 (1re partie), T2, T1 (2e partie). C'est une exécution

valide puisque T2 accède à un granule non verrouillé qui n'existe même pas lorsque T1 exécute sa 1re partie : le tuple « Fantomas ». Toutefois, le résultat de T1 est une liste de 4 noms alors que le nombre de passagers est 5. « Fantomas » est ici un fantôme pour T1.

Passagers	Nom	N°Vol	N°Siège
	Dubois	100	3
	Durand	100	5
	Dupont	100	10
	Martin	100	15

Occupation	N°Vol	NbrePass
	100	4

Figure XVI.21 : Illustration du problème des fantômes

Ce problème, ainsi que la difficulté de citer les granules à verrouiller, peuvent être résolus par la définition de granules logiques (dans l'exemple, les passagers du vol 100) au moyen de prédicats [Eswaran76]. Le verrouillage par prédicat permet également de définir des granules de tailles variables, ajustées aux besoins des transactions. Malheureusement, il nécessite des algorithmes pour déterminer si deux prédicats sont disjoints et ce problème de logique n'a pas de solution suffisamment efficace pour être appliqué dynamiquement lors du verrouillage des objets. De plus, les prédicats sont définis sur des domaines dont les extensions ne sont pas consultables dans la base pour des raisons évidentes de performance. Donc, il est très difficile de déterminer si deux prédicats sont disjoints ; par exemple, PROFESSION = "Ingénieur" et SALAIRE < 7000 seront déterminés logiquement non disjoints, alors qu'ils le sont dans la plupart des bases de données. Le verrouillage par prédicat est donc en pratique source d'attentes inutiles et finalement inapplicable.

4.5. LES AMÉLIORATIONS DU VERROUILLAGE

Malgré le grand nombre de solutions proposées par les chercheurs, les systèmes continuent à appliquer le verrouillage deux phases avec prévention ou détection des verrous mortels. Les degrés d'isolation choisis par les transactions permettent de maximiser le partage des données en limitant le contrôle. Le verrouillage est cependant très limitatif. Un premier problème qui se pose est le choix de la granularité des objets à verrouiller. Au-delà, la recherche sur l'amélioration du verrouillage continue et des solutions parfois applicables ont été proposées. Nous analysons les plus connues ci-dessous.

4.5.1. Verrouillage à granularité variable

Une granularité variable est possible. La technique consiste à définir un graphe acyclique d'objets emboîtés et à verrouiller à partir de la racine dans un mode d'intention jusqu'aux feuilles désirées qui sont verrouillées en mode explicite. Par exemple, une transaction désirant verrouiller un tuple en mode écriture verrouillera la table en intention d'écriture, puis la page en intention d'écriture, et enfin le tuple en mode écriture. Les modes d'intentions obéissent aux mêmes règles de compatibilités que les modes explicites, mais sont compatibles entre eux. Le verrouillage en intention permet simplement d'éviter les conflits avec les modes explicites. Sur un même objet, les modes explicites règlent les conflits. La figure XVI.22 donne la matrice de compatibilité entre les modes lecture (L), écriture (E), intention de lecture (IL) et intention d'écriture (IE).

	L	E	IL	IE
L	V	F	V	F
E	F	F	F	F
IL	V	F	V	V
IE	F	F	V	V

Figure XVI.22 : Compatibilités entre les modes normaux et d'intention

Une telle méthode peut être appliquée dans les bases relationnelles, mais aussi objet. Le graphe d'inclusion pour une base de données objet peut être base, extension de classe, page ou groupe (*cluster*) et objet. Il est représenté figure XVI.23.

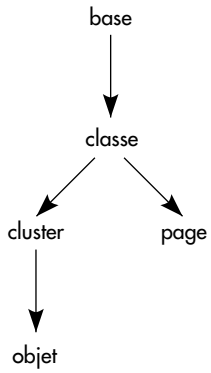


Figure XVI.23 : Granules variables pour une BD objet

4.5.2. Verrouillage multi-versions

Le **verrouillage multi-versions** suppose l'existence d'au moins une version précédente d'un objet en cours de modification. C'est généralement le cas dans les systèmes puisque, comme nous le verrons ci-dessous, un journal des images avant mise à jour est géré en mémoire. Le principe est simple : lors d'un verrouillage en lecture, si le granule est occupé par une transaction en mode incompatible (donc en écriture en pratique), la version précédente du granule est délivrée à l'utilisateur. Une telle technique est viable lorsque les granules verrouillés sont des pages ou des tuples. Au-delà, il est difficile de constituer une version cohérente du granule rapidement.

Avec le verrouillage multi-versions, tout se passe comme si la transaction qui lit avait été lancée avant la transaction qui écrit. Malheureusement, la sérialisabilité n'est pas garantie si la transaction qui accède à la version ancienne écrit par ailleurs. En effet, la mise à jour de la transaction en quelque sorte sautée n'est pas prise en compte par la transaction lisant, qui risque d'obtenir des résultats dépassés ne pouvant servir aux mises à jour. Seule les transactions n'effectuant que des lectures peuvent utiliser ce mécanisme, aussi appelé lecture dans le passé. Si l'on veut de plus garantir la reproductibilité des lectures, il faut gérer au niveau du système un cache des lectures effectuées dans le passé, afin de les retrouver lors de la deuxième lecture. Ce type de verrouillage est réservé au décisionnel qui ainsi n'est pas perturbé par les mises à jour.

4.5.3. Verrouillage altruiste

Le **verrouillage altruiste** suppose connu les patterns d'accès aux granules des transactions, c'est-à-dire au moins l'ordre d'accès aux granules et les granules non accédés. Il devient alors possible de relâcher les verrous tenus par une transaction longue lorsqu'on sait qu'elle n'utilisera plus un granule. Ce granule est alors ajouté à l'ensemble des granules utilisés par la transaction, appelée la **traînée** (*wake*) de la transaction. En cas de reprise, toutes les transactions dans la traînée d'une transaction sont aussi reprises (c'est-à-dire celles ayant accédé à des objets dans la traînée). C'est l'**effet domino**, selon lequel une transaction implique la reprise d'autres pour compenser la non isolation (voir ci-dessous). Par exemple, figure XVI.24, on voit que T2 peut s'exécuter alors que T1 n'est pas terminée car on sait que T1 ne reviendra pas sur c et que T2 n'accédera pas a. Si T3 est reprise, T4 doit l'être aussi car elle a modifié c, lui-même modifié par T3.

Le verrouillage altruiste est difficile à appliquer en pratique car on ne connaît pas les patterns d'accès des transactions. De plus, l'effet domino reste mal maîtrisé. Cette technique pourrait être intéressante pour faire cohabiter des transactions longues avec des transactions courtes [Barghouti91].

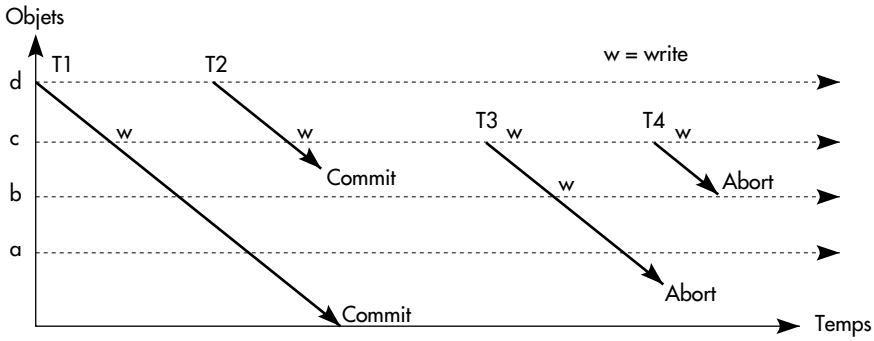


Figure XVI.24 : Exemple de verrouillage altruiste

4.5.4. Commutativité sémantique d'opérations

Il est possible d'exploiter la **sémantique des opérations**, notamment dans les systèmes objet ou objet-relationnel où il existe des types (ou classes) [Gardarin76, Weihl88, Cart90]. Chaque type est caractérisé par une liste d'opérations (méthodes). Comme nous l'avons vu ci-dessus, les opérations commutatives sont permutables et n'entraînent pas de conflits de précédence. Il est donc intéressant de distinguer des modes de verrouillage plus fins que lecture et écriture, permettant de prendre en compte les opérations effectives sur les objets typés, et non pas les actions de base lecture et écriture.

Introduire la commutativité entre opérations est utile si les opérations de mises à jour, a priori incompatibles, sont souvent commutatives. Si l'on regarde seulement le nom de l'opération, la commutativité est rare car elle dépend souvent des paramètres, notamment de la réponse. Les chercheurs ont donc introduit des modes d'opérations incluant les réponses. Par exemple, avec des ensembles, il est intéressant de distinguer insérer avec succès `[Ins,ok]`, supprimer avec succès `[Del,ok]`, tester l'appartenance avec succès `[In, true]` et avec échec `[In, False]`. La matrice de commutativité est représentée figure XVI.25.

Dans un système typé, chaque objet peut posséder en option un contrôle de concurrence défini au niveau de la classe. Les verrouillages sont alors délégués au contrôleur du type d'objet. Celui-ci laisse passer simultanément les verrouillages en modes d'opérations commutatives. Par exemple, un ensemble pourra être verrouillé simultanément par deux transactions en mode `[Ins,ok]`, ou en `[In,False]` et `[Del,ok]`. Le contrôleur bloque seulement les opérations non commutatives (ordonnancement).

Les reprises en cas de panne ou d'obtention d'un résultat invalide (non verrouillé par exemple) sont cependant difficiles. En effet, comme pour le verrouillage altruiste, le modèle est ouvert et permet à des transactions de voir des données modifiées par des

transactions non encore terminées. Il faut donc gérer la portée des transactions, par exemple sous forme de listes de transactions dépendantes. L'effet domino introduit ci-dessus survient : lors de la reprise d'une transaction, toutes les transactions dépendantes doivent être reprises.

	[Ins,ok]	**[Del,ok]**	**[In,true]**	**[In,False]**
[Ins,ok]	1	0	0	0
[Del,ok]	0	1	0	1
[In,true]	0	0	1	1
[In,False]	0	1	1	1

Figure XVI.25 : Commutativité d'opérations sur ensemble

Certains auteurs [Weihl88] ont aussi considéré la commutativité en avant et la commutativité en arrière. Par exemple `[In,true]` et `[Insert,ok]` commutent en avant mais pas en arrière : si l'on exécute ces deux opérations à partir d'un état s sur lequel elles sont définies, on obtient bien le même résultat quel que soit l'ordre. Mais si l'on a l'exécution `[Insert, ok] [In, true]`, on n'est pas sur que `[In,true]` soit définie sur l'état initial (l'objet inséré peut être celui qui a permis le succès de l'opération `In`). Donc, on ne peut pas commuter lorsqu'on défait et refait une exécution. Tout cela complique les procédures de reprises et l'exploitation de la commutativité des opérations.

5. CONTRÔLES DE CONCURRENCE OPTIMISTE

Le verrouillage est une solution pessimiste : il empêche les conflits de se produire, ou plutôt les transforme en verrou mortel. Analysons maintenant une autre gamme de

solutions qualifiées d'optimistes qui laissent se produire les conflits et les résout ensuite.

5.1. ORDONNANCEMENT PAR ESTAMPILLAGE

Bien que le verrouillage avec prévention ou détection du verrou mortel soit la technique généralement appliquée dans les SGBD, de nombreuses autres techniques ont été proposées. En particulier, l'**ordonnancement par estampille** peut être utilisé non seulement pour résoudre les verrous mortels comme vu ci-dessus, mais plus complètement pour garantir la sérialisabilité des transactions.

Une méthode simple consiste à conserver pour chaque objet accédé (tuple ou page), l'estampille du dernier écrivain W et celle du plus jeune lecteur R. Le contrôleur de concurrence vérifie alors :

1. que les accès en écriture s'effectuent dans l'ordre croissant des estampilles de transactions par rapport aux opérations créant une précédence, donc l'écrivain W et le lecteur R.
2. que les accès en lecture s'effectuent dans l'ordre croissant des estampilles de transactions par rapport aux opérations créant une précédence, donc par rapport à l'écrivain W.

On aboutit donc à un contrôle très simple d'ordonnancement des accès conformément à l'ordre de lancement des transactions [Gardarin78, Bernstein80]. En cas de désordre, il suffit de reprendre la transaction ayant créé le désordre. Les contrôles nécessaires en lecture et écriture sont résumés figure XVI.26.

```
// Contrôle d'ordonnancement des transactions
Fonction Ecrire(Ti, O) { // la transaction Ti demande l'écriture de O;
   si ts(Ti) < W(O) ou ts(Ti) < R(O) alors abort(Ti)
   sinon executer_ecrire(Ti,O) } ;

Fonction Lire(Ti,O) {// la transaction Ti demande la lecture de l'objet O;
   si ts(Ti) < W(O) alors abort(Ti)
   sinon executer_lire(Ti,O) } ;
```

Figure XVI.26 : Algorithme d'ordonnancement des accès par estampillage

L'algorithme d'ordonnancement par estampillage soulève plusieurs problèmes. De fait, les estampilles W et R associées à chaque objet remplacent les verrous. Il n'y a pas d'attente, celles-ci étant remplacées par des reprises de transaction en cas d'accès ne respectant pas l'ordre de lancement des transactions. Ceci conduit en général à

beaucoup trop de reprises. Une amélioration possible consiste à garder d'anciennes versions des objets. Si l'estampille du lecteur ne dépasse pas celle du dernier écrivain, on peut délivrer une ancienne version, plus exactement, la première inférieure à l'estampille du lecteur. Ainsi, il n'y a plus de reprise lors des lectures. La méthode est cependant difficile à mettre en œuvre et n'est guère utilisée aujourd'hui.

5.2. CERTIFICATION OPTIMISTE

La certification optimiste est une méthode de type curative, qui laisse les transactions s'exécuter et effectue un contrôle garantissant la sérialisabilité en fin de transaction. Une transaction est divisée en trois phases : phase d'accès, phase de certification et phase d'écriture. Pendant la phase d'accès, chaque contrôleur de concurrence garde les références des objets lus/écrits par la transaction. Pendant la phase de certification, le contrôleur vérifie l'absence de conflits avec les transactions certifiées pendant la phase d'accès. S'il y a conflit, la certification est refusée et la transaction défaite puis reprise. La phase d'écriture permet l'enregistrement des mises à jour dans la base pour les seules transactions certifiées.

En résumé, nous introduirons ainsi la notion de certification qui peut être effectuée de différentes manières :

Notion XVI.16 : Certification de transaction (*Transaction certification*)

Action consistant à vérifier et garantir que l'intégration dans la base de donnés des mises à jour préparées en mémoire par une transaction préservera la sérialisabilité des transactions.

Vérifier l'absence de conflits pourrait s'effectuer en testant la non-introduction de circuits dans le graphe de précédence. L'algorithme commun [Kung81] de certification est plus simple. Il consiste à mémoriser les ensembles d'objets lus (*Read Set RS*) et écrits (*Write Set WS*) par une transaction. La certification de la transaction Ti consiste à tester que RS(Ti) n'intersecte pas avec WS(Tj) et que WS(Ti) n'intersecte pas avec WS(Tj) ou RS(Tj) pour toutes les transactions Tj lancées après Ti. On vérifie donc que les transactions n'agissent pas en modes incompatibles avec les transactions concurrentes avant de les valider. L'algorithme est représenté figure XVI.27.

En résumé, cette méthode optimiste est analogue au verrouillage, mais tous les verrous sont laissés passants et les conflits ne sont détectés que lors de la validation des transactions. L'avantage est la simplicité du contrôleur de concurrence qui se résume à mémoriser les objets accédés et à un test simple d'intersection d'ensembles de références lors de la validation. L'inconvénient majeur est la tendance à reprendre beaucoup de transactions en cas de conflits fréquents. La méthode optimiste est donc seulement valable pour les cas où les conflits sont rares.

```
Bool Fonction Certifier(Ti) {
   Certifier = VRAI ;
   Pour chaque transaction t concurrente faire {
      Si RS(Ti) ∩ WS(t) ≠ ∅ ou WS(Ti) ∩ RS(t) ≠ ∅
         ou WS(Ti) ∩ WS(t) ≠ ∅
      Alors {
               Certifier = FAUX ;
               Abort(Ti) ;
         } ;
   } ;
} ;
```

Figure XVI.27 : Algorithme de certification optimiste

5.3. ESTAMPILLAGE MULTI-VERSIONS

Comme pour le verrouillage deux phases et même mieux, la stratégie d'ordonnancement par estampillage vue ci-dessus peut être améliorée en gardant plusieurs versions d'un même granule [Reed79]. Pour chaque objet O, le système peut maintenir :

1. un ensemble d'estampilles en écriture {EEi(O)} avec les valeurs associées { Oi }, chacune d'elles correspondant à une version i ;
2. un ensemble d'estampilles en lecture {ELi(O)}.

Il est alors possible d'assurer l'ordonnancement des lectures par rapport aux écritures sans jamais reprendre une transaction lisant. Pour cela, il suffit de délivrer à une transaction Ti demandant à lire l'objet O la version ayant une estampille en écriture immédiatement inférieure à i. Ainsi, Ti précédera toutes les créations d'estampilles supérieures écrivant l'objet considéré et suivra celles d'estampilles inférieures. Ti sera donc correctement séquencée. Tout se passe comme si Tj avait demandé la lecture juste après l'écriture de la version d'estampille immédiatement inférieure. L'algorithme de contrôle de l'opération LIRE avec un dispositif d'ordonnancement partiel multi-versions est représenté figure XVI.28.

```
// Lecture de la bonne version dans le passé
Fonction Lire(Ti,O) {// la transaction Ti demande la lecture de l'objet O;
   j = index de la dernière version de O ;
   Tant que ts(Ti) < W(O) faire j = j-1 ; // chercher la version avant Ti
   executer_lire(Ti,Oj) } ; // lire la bonne version
}
```

Figure XVI.28 : Algorithme de lecture avec ordonnancement multi-versions

Il est en général très difficile de refaire le passé. Cependant, il est parfois possible de forcer l'ordonnancement des écritures de Ti en insérant une nouvelle version créée par Ti juste après celle d'estampille immédiatement inférieure, soit Oj. Malheureusement, si une transaction Tk (k >i) a lu la version Oj, alors cette lecture doit aussi être reséquencée. Ce n'est possible que si la transaction Tk pouvait être reprise. Afin d'éviter la reprise de transactions terminées, on préférera reprendre l'écrivain Tj avec une nouvelle estampille i' supérieure à k.

L'algorithme de contrôle de l'opération WRITE correspondant est représenté figure XVI.29. Les notations sont identiques à celles utilisées ci-dessus, les indices désignant les numéros de versions d'objets.

```
// Réordonnancement des écritures dans le passé
Fonction Ecrire(Ti, O) { // la transaction Ti demande l'écriture de O;
   j = index de la dernière version de O ;
   Tant que ts(Ti) < W(Oj) faire j = j-1 ; // chercher la version avant Ti
   Si ts(Ti) < R(Oj) alors abort(Ti) // abort si lecture non dans l'ordre
   sinon executer_ecrire(Ti,Oj) } ; // écrire en bonne place
}
```

Figure XVI.29 : Algorithme d'écriture avec ordonnancement multi-versions

La figure XVI.30 illustre l'algorithme.

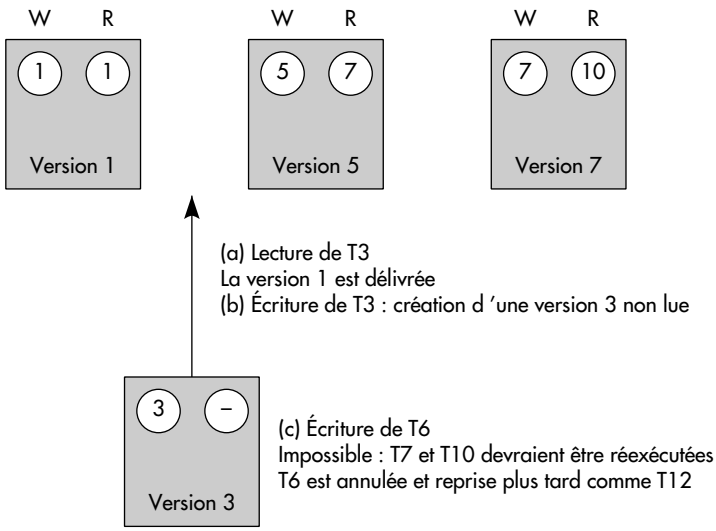


Figure XVI.30 : Exemple de reséquencement

Les transactions entrent en conflit sur un objet O unique dont les versions successives sont représentées par des rectangles. La situation originale est représentée en haut de la figure. Trois versions de l'objet existent, successivement créées par les transactions 1, 5 et 7. La version 1 a été lue par la transaction 1, la version 5 par la transaction 7 et la version 7 par la transaction 10. Nous supposons que T3 accomplit une écriture sur l'objet O après l'avoir lu. La nouvelle version 3 créée est insérée en bonne place. Nous supposons ensuite que T6 procède à une écriture sur O. L'objet ayant été lu par T7, il faudrait refaire le passé. On préférera annuler T6 et la relancer plus tard.

En résumé, beaucoup d'algorithmes basés sur des estampilles peuvent être inventés pour contrôler les accès concurrents. Il est même possible de mixer estampilles et verrouillage, comme déjà vu au niveau des algorithmes DIE-WAIT et WOUND-WAIT. Cependant, les performances de ces algorithmes restent faibles car ils provoquent tous des reprises qui deviennent de plus en plus fréquentes lorsqu'il y a un plus grand nombre de conflits, donc lorsque le système est chargé. Voilà sans doute pourquoi la plupart des SGBD utilisent le verrouillage deux phases.

6. LES PRINCIPES DE LA RÉSISTANCE AUX PANNES

Nous avons étudié ci-dessus les mécanismes de contrôle de concurrence permettant de limiter les interférences entre transactions. Abordons maintenant les techniques de résistance aux pannes, qui permettent aussi d'assurer les propriétés ACID des transactions, particulièrement l'atomicité et la durabilité.

6.1. PRINCIPAUX TYPES DE PANNES

Ils existe différentes sources de pannes dans un SGBD. Celles-ci peuvent être causées par une erreur humaine, une erreur de programmation ou le dysfonctionnement d'un composant matériel. On peut distinguer [Gray78, Fernandez80] :

1. **La panne d'une action** survient quand une commande au SGBD est mal exécutée. En général, elle est détectée par le système qui retourne un code erreur au programme d'application. Ce dernier peut alors tenter de corriger l'erreur et continuer la transaction.
2. **La panne d'une transaction** survient quand une transaction ne peut continuer par suite d'une erreur de programmation, d'un mauvais ordonnancement des accès concurrents, d'un verrou mortel ou d'une panne d'action non corrigeable. Il faut alors défaire les mises à jour effectuées par la transaction avant de la relancer.

3. **La panne du système** nécessite l'arrêt du système et son redémarrage. La mémoire secondaire n'est pas affectée par ce type de panne ; en revanche, la mémoire centrale est perdue par suite du rechargement du système.
4. **La panne de mémoire secondaire** peut survenir soit suite à une défaillance matérielle, soit suite à une défaillance logicielle impliquant de mauvaises écritures. Alors, une partie de la mémoire secondaire est perdue. Il s'agit du type de panne le plus catastrophique.

Les différents types de panne sont de fréquence très différente. Par exemple, les deux premiers peuvent survenir plusieurs fois par minute alors qu'une panne système apparaît en général plusieurs fois par mois et qu'une panne mémoire secondaire n'arrive que quelquefois par an, voire moins. Aussi, seul le dernier type de panne conduit à faire appel aux archives et peut s'avérer, dans certains cas très rares, non récupérable.

6.2. OBJECTIFS DE LA RÉSISTANCE AUX PANNES

L'objectif essentiel est de minimiser le travail perdu tout en assurant un retour à des données cohérentes après pannes. Compte tenu de l'aspect non instantané de l'apparition d'une panne et de sa détection, nous considérerons généralement que la transaction est l'unité de traitement atomique, ou si l'on préfère l'unité de reprise. Cependant, ceci n'est pas toujours vrai et une unité plus faible a été retenue dans les systèmes basés sur SQL à l'aide de la notion de point de reprise de transaction (*savepoint*). Une transaction est divisée en étapes, encore appelées **unités d'œuvre** (voir figure XVI.31). L'atomicité de chaque unité d'œuvre doit être garantie par le système transactionnel. Une panne de transaction provoque le retour au dernier point de reprise de la transaction. L'exécution d'un point de reprise permet de conserver les variables en mémoire de la transaction, bien que la reproductibilité des lectures ne soit en général pas garantie entre les unités d'œuvre. Dans la suite, pour simplifier, nous ne considérerons en général que des transactions composées d'une seule unité d'œuvre.

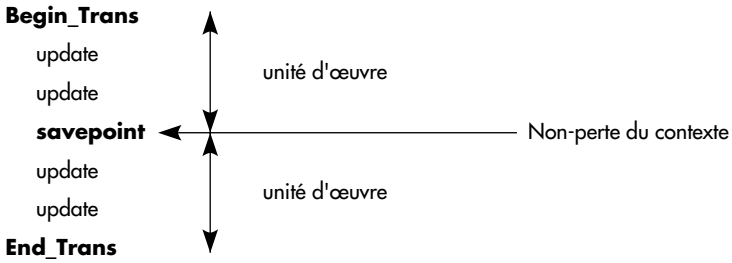


Figure XVI.31 : Transaction composée d'unités d'œuvre multiples

Les objectifs premiers de la résistance aux pannes sont de fournir un protocole aux applications permettant d'assurer l'atomicité. Pour ce faire, une application doit pouvoir commencer l'exécution d'une transaction et la terminer avec succès ou par un échec. Des actions atomiques sont ainsi fournies par le système de gestion de transactions aux applications. En plus de la création d'une transaction, ces actions correspondent aux trois notions de **validation** (encore appelée *commitment*, consolidation ou confirmation), **d'annulation** (encore appelée *abort* ou abandon ou retour arrière) et de **reprise** (encore appelée restauration ou *redo*). Nous définissons ci-dessous ces trois notions.

Notion XVI.17 : Validation de transaction (*Transaction commitment*)

Action atomique spéciale (appelée Commettre ou Commit), exécutée en fin de transaction, provoquant l'intégration définitive de toutes les mises à jour non encore commise de la transaction exécutante dans la base de données.

La validation est donc la terminaison avec succès d'une transaction. Dans le cas de transactions composées de plusieurs unités d'œuvre, une validation est effectuée à la fin de chaque unité d'œuvre. L'opposé de la validation est **l'annulation**.

Notion XVI.18 : Annulation de transaction (*Transaction abort*)

Action atomique spéciale (appelée Annuler ou Défaire ou Abort *ou* Undo*), généralement exécutée après une défaillance, provoquant l'annulation de toutes les mises à jour de la base effectuées par la transaction et non encore commises.*

Notez que seules les transactions non validées peuvent être annulées. Défaire une transaction validée est une opération impossible sauf à utiliser des versions antérieures de la base. Par contre, une transaction défaite peut être refaite (on dit aussi rejouée) : c'est l'objet de la **reprise**.

Notion XVI.19 : Reprise de transaction (*Transaction redo*)

Exécution d'une action spéciale (appelée Refaire ou Redo) qui refait les mises à jour d'une transaction précédemment annulée dans la base de données.

La reprise peut s'effectuer à partir de journaux des mises à jour, comme nous le verrons ci-dessous. Elle peut aussi nécessiter une nouvelle exécution de la transaction.

6.3. INTERFACE APPLICATIVE TRANSACTIONNELLE

La mise à disposition des fonctionnalités de validation, annulation et reprise a nécessité le développement d'une interface entre les applications et le système transactionnel. Cette interface a été standardisée par l'X/OPEN dans le cadre de l'architecture

DTP (*Distributed Transaction Processing*). Pour des transactions simples, elle se résume à trois actions de base :

– `Trid` **`Begin`** `(context)` permet de débuter une transaction en fournissant un contexte utilisateur ; elle retourne un identifiant de transaction Trid ;

– **`Commit`** `(TrId)` valide la transaction dont l'identifiant est passé en paramètre ;

– **`Abort`**`(TrId)` annule la transaction dont l'identifiant est passé en paramètre.

Des points de sauvegardes peuvent être introduits, comme vu ci-dessus avec les opérations :

– `SaveId` **`Save`**`(TrId)` déclare un point de sauvegarde pour la transaction et demande la validation de l'unité d'œuvre en cours ; elle retourne un identifiant de point de sauvegarde ;

– **`Rollback`** `(Trid, SaveId)` permet de revenir au point de sauvegarde référencé, en général le dernier.

Quelques opérations de service supplémentaires sont proposées, telles par exemple :

– **`ChainWork`** `(context)` valide la transaction en cours et ouvre une nouvelle transaction ;

– `TrId` **`MyTrId`**`()` retourne l'identifiant de la transaction qui l'exécute ;

– **`Status`**`(TrId)` permet de savoir quel est l'état de la transaction référencée en paramètre ; elle peut être active, annulée, commise, en cours d'annulation ou de validation.

En résumé, l'objectif d'un système transactionnel au sein d'un SGBD est de réaliser efficacement les opérations précédentes. En plus, celui-ci doit bien sûr intégrer un contrôle de concurrence correct et efficace.

6.4. ÉLÉMENTS UTILISÉS POUR LA RÉSISTANCE AUX PANNES

Nous décrivons maintenant les différents éléments utilisés pour la validation et la reprise de transactions.

6.4.1. Mémoire stable

Avant tout, il est nécessaire de disposer de mémoires secondaires fiables et sûres. Plus précisément, la notion de **mémoire stable** recouvre l'espace disque qui n'est ni perdu ni endommagé lors d'une panne simple, d'action, de transaction ou de système. La mémoire stable est organisée en pages. Une écriture de page dans la mémoire stable est atomique : une page est soit correctement écrite sur mémoire secondaire, soit pas du tout ; elle ne peut être douteuse ou partiellement écrite. De plus, si elle est écrite,

elle ne peut être détruite que par une panne catastrophique explicite ou par une réécriture.

Notion XVI.20 : Mémoire stable (*Stable store*)

Mémoire découpée en pages dans laquelle une écriture de page est soit correctement exécutée, soit non exécutée, garantissant la mémorisation de la page jusqu'à réécriture, donc sa non perte suite à des pannes simples.

Dans les SGBD, la mémoire stable est le disque ; il permet de mémoriser les données persistantes. La réalisation d'une mémoire sûre garantissant l'atomicité des écritures n'est pas triviale. Les techniques utilisées sont en général les codes de redondances, ainsi que les doubles écritures. Dans la suite, nous considérons les mémoires stables comme sûres.

6.4.2. Cache volatile

Les SGBD utilisent des caches des bases de données en mémoire afin d'améliorer les performances. Comme vu ci-dessus, la mémoire centrale peut être perdue en cas de panne système : le **cache** est donc une mémoire **volatile**.

Notion XVI.21 : Cache volatile (*Transient cache*)

Zone mémoire centrale contenant un cache de la base, considérée comme perdue après une panne système.

Les transactions actives exécutent des mises à jour dont l'effet apparaît dans le cache et n'est pas instantanément reporté sur disque. En théorie, l'effet d'une transaction devrait être reporté de manière atomique lors de la validation. La figure XVI.32 illustre les mouvements entre mémoire volatile (le cache) et mémoire stable souhaitables lors de la validation ou de l'annulation.

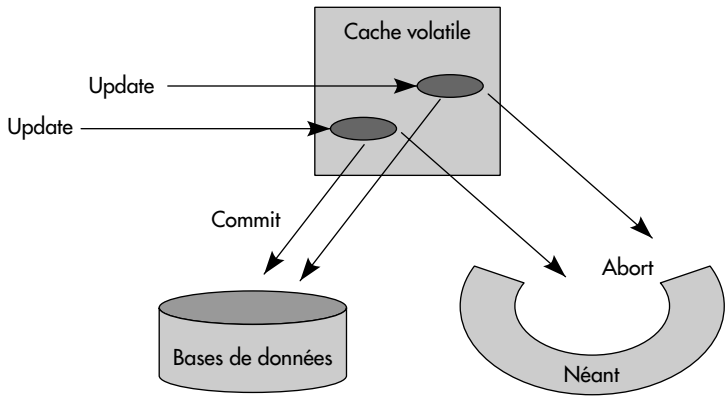


Figure XVI.32 : Mouvements de données entre cache et mémoire stable

Ceci est la vue logique donnée à l'utilisateur. En pratique, les mécanismes pour assurer un tel fonctionnement logique sont plus complexes.

6.4.3. Journal des mises à jour

À un instant donné, l'état de la base est déterminé par l'état de la mémoire stable et l'état du cache. En effet, des mises à jour ont été effectuées et ne sont pas encore reportées sur disque. Certaines effectuées par des transactions venant juste d'être validée peuvent être en cours de report. Il faut cependant garantir la non-perte de mise à jour des transactions commises en cas de panne. Si le système reporte des pages dans la mémoire stable avant validation d'une transaction, par exemple pour libérer de la place dans le cache, il faut être capable de défaire les reports de pages contenant des mises à jour de transactions annulées.

La méthode la plus classique pour permettre la validation atomique, l'annulation et la reprise de transaction consiste à utiliser des journaux [Verhofstad78]. On distingue le **journal des images avant** et le **journal des images après**, bien que les deux puissent être confondus dans un même journal.

Notion XVI.22 : Journal des images avant (*Before image log*)

Fichier système contenant d'une part les valeurs (images) avant modification des pages mises à jour, dans l'ordre des modifications avec les identifiants des transactions modifiantes, ainsi que des enregistrements indiquant les débuts, validation et annulation de transactions.

Le journal des images avant est utilisé pour **défaire** les mises à jour d'une transaction (*undo*). Pour cela, il doit être organisé pour permettre d'accéder rapidement aux enregistrements correspondant à une transaction. Un fichier haché sur l'identifiant de transaction (TrId) sera donc opportun.

Notion XVI.23 : Journal des images après (*After image log*)

Fichier système contenant d'une part les valeurs (images) après modifications des pages mises à jour, dans l'ordre des modifications avec les identifiants des transactions modifiantes, ainsi que des enregistrements indiquant les débuts, validation et annulation de transactions.

Le journal des images après est utilisé pour **refaire** les mises à jour d'une transaction (*redo*). Comme le journal des images après, il doit être organisé pour permettre d'accéder rapidement aux enregistrements correspondant à une transaction. Un fichier haché sur l'identifiant de transaction (TrId) sera donc opportun.

En guise d'illustration, la figure XVI.33 représente un enregistrement d'un journal contenant à la fois les images avant et après. Les enregistrements sont précédés d'une lettre R (Redo) pour les images après, U (Undo) pour les images avant, et T (Transaction) pour les changements d'états des transactions.

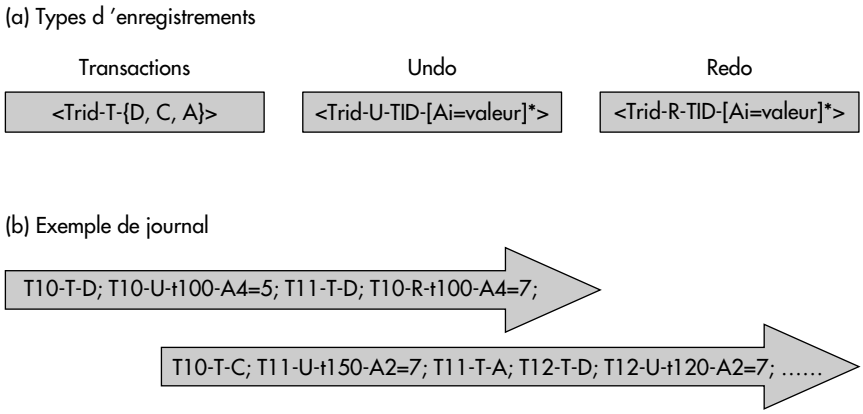


Figure XVI.33 : Enregistrements dans le journal global

Comme indiqué ci-dessus, les modifications sont tout d'abord exécutées dans des caches en mémoire. Ces caches sont **volatils**, c'est-à-dire perdus lors d'une panne. Ce n'est bien souvent que lors de la validation que les mises à jour sont enregistrées dans le journal et dans la base. Afin d'être capable d'annuler une transaction dans tous les cas, il faut écrire les enregistrements dans le journal avant de reporter le cache dans la base, comme illustré figure XVI.34.

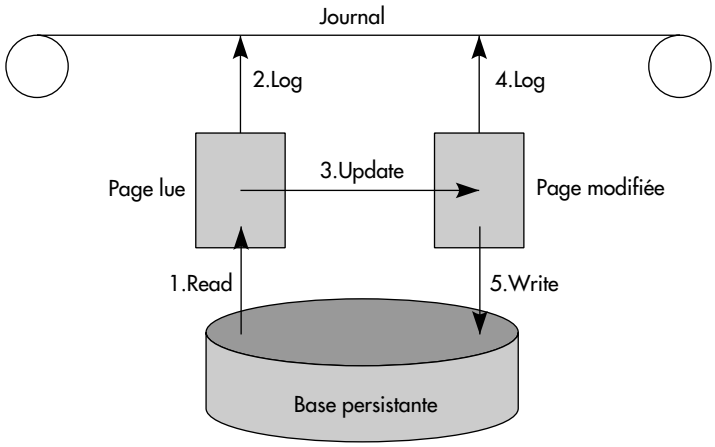


Figure XVI.34 : Ordre d'enregistrement des mises à jour

L'ordre dans lequel les opérations doivent être accomplies est indiqué sur la figure. Les règles suivantes sont souvent conseillées [Bernstein87] :

1. Avant d'écrire une page modifiée en mémoire stable, il faut enregistrer son image avant dans le journal (pour pouvoir défaire) ainsi que son image après (pour pou-

voir refaire). Cette règle est connue sous le nom de **journalisation avant écriture** (*log ahead rule*).

2. Toutes les pages modifiées en mémoire volatile par une transaction doivent être écrites en mémoire stable, donc sur disque, avant la validation de la transaction. Cette dernière règle est connue sous le nom de **validation après écriture** (*commit after rule*).

L'application de ces deux règles conduit naturellement à enregistrer journal puis page modifiée sur disques soit à chaque mise à jour, soit en fin de transaction avant le commit effectif. Elles sont donc très limitatives. En conséquence, la première est généralement suivie pour éviter de ne pouvoir défaire des transactions non validées ou refaire des validées. La seconde peut être relaxée avec quelques précautions. Dans certains cas, par exemple si les journaux sont doublés ou si la base est doublée par une base miroir, ces deux règles peuvent être remplacées par des règles plus souples.

L'utilisation d'un journal coûte très cher : chaque mise à jour nécessite a priori trois entrées-sorties. Afin d'améliorer les performances, les enregistrements du journal sont généralement gardés dans un tampon en mémoire et vidés sur disques lorsque le tampon est plein. Malheureusement, il faut écrire le journal avant d'écrire dans la base pour pouvoir défaire ou refaire les mises à jour de la mémoire stable. La technique utilisée pour résoudre ce problème consiste à bloquer ensemble plusieurs validations de transactions.

Notion XVI.24 : Validation bloquée (*Blocked commit*)

Technique consistant à valider plusieurs transactions ensemble pour pouvoir écrire ensemble les enregistrements dans le journal.

Ainsi, le système peut attendre qu'un tampon du journal soit plein avant de le vider. Lorsqu'une transaction commet, si le tampon n'est pas plein, elle attend que d'autres transactions effectuent des mises à jour pour remplir le tampon. Si le système est peu actif, un délai maximal (*time out*) permet de forcer le vidage du tampon journal. Soulignons que le journal est en général compressé, pour réduire sa taille au maximum et aussi maximiser le nombre de transactions commises simultanément.

Un dernier problème concernant le journal est celui de sa purge. En effet, on ne peut enregistrer indéfiniment les enregistrements étudiés sur un disque dans un fichier haché. Même avec du hachage extensible, le fichier risque de devenir important et difficile à gérer. En conséquence, les systèmes changent périodiquement de fichier journal. Il est possible par exemple de tourner sur N fichiers hachés, en passant au suivant chaque fois que l'un est plein ou au bout d'une période donnée. Un fichier journal qui n'est plus actif est vidé dans une archive. Il sera réutilisé plus tard pour une nouvelle période de journalisation.

6.4.4. Sauvegarde des bases

En cas de perte de la mémoire stable, donc d'une panne des disques en général, il faut pouvoir retrouver une archive de la base détruite. Pour cela, des sauvegardes doivent être effectuées périodiquement, par exemple sur un disque séparé ou sur bandes magnétiques.

Notion XVI.25 : Sauvegarde (*Backup copy*)

Copie cohérente d'une base locale effectuée périodiquement alors que cette base est dans un état cohérent.

Une sauvegarde peut par exemple être effectuée en début de chaque semaine, de chaque journée, de chaque heure. La prise de sauvegarde pendant le fonctionnement normal du système est un problème difficile. Pour cela, un mécanisme de verrouillage spécifique – ou mieux, un mécanisme de multi-versions [Reed79] – peut être utilisé.

6.4.5. Point de reprise système

Après un arrêt normal ou anormal du système, il est nécessaire de repartir à l'aide d'un état machine correct. Pour cela, on utilise en général des points de reprise système.

Notion XVI.26 : Point de reprise système (*System checkpoint*)

Etat d'avancement du système sauvegardé sur mémoires secondaires à partir duquel il est possible de repartir après un arrêt.

Les informations sauvegardées sur disques comportent en général l'image de la mémoire, l'état des travaux en cours et les journaux. Un enregistrement « point de reprise système » est écrit dans le journal. Celui-ci est recopié à partir des fichiers le contenant. Lors d'une reprise, on repart en général du dernier point de reprise système. Plus ce point de reprise est récent, moins le démarrage est coûteux, comme nous le verrons ci-dessous.

7. VALIDATION DE TRANSACTION

Comme indiqué ci-dessus, la validation de transaction doit permettre d'intégrer toutes les mises à jour d'une transaction dans une base de données de manière atomique, c'est-à-dire que toutes les mises à jour doivent être intégrées ou qu'aucune ne doit l'être. L'atomicité de la validation de transaction rend les procédures d'annulation de

transactions non validées simples. Le problème est donc de réaliser cette atomicité. Plusieurs techniques ont été introduites dans les systèmes afin de réaliser une validation atomique. Elles peuvent être combinées afin d'améliorer la fiabilité [Gray81]. La plupart des SGBD combinent d'ailleurs les écritures en place et la validation en deux étapes.

7.1. ÉCRITURES EN PLACE DANS LA BASE

Avec cette approche, les écritures sont effectuées directement dans les pages de la base contenant les enregistrements modifiés. Elles sont reportées dans la base au fur et à mesure de l'exécution des commandes Insert, Update et Delete des transactions. Ce report peut être différé mais doit être effectué avant la validation de la transaction. Comme vu ci-dessus, les mises à jour d'une transaction sont écrites dans le journal avant d'être mises en place dans la base de données. L'atomicité de la validation d'une transaction est réalisée par écriture d'un enregistrement dans le journal [Verghofstad78]. Les écritures restent invisibles aux autres transactions tant qu'elles ne sont pas validées ; pour cela, les pages ou les enregistrements sont verrouillés au fur et à mesure des écritures. La validation proprement dite consiste à écrire dans le journal un enregistrement « transaction validée ». Ensuite, les mises à jour sont rendues visibles aux autres transactions.

Dans tous les cas, la validation d'une transaction ayant mis à jour la base de données génère un nouvel état, ainsi que des enregistrements dans le journal (images des pages modifiées et enregistrement transaction validée), comme indiqué figure XVI.35.

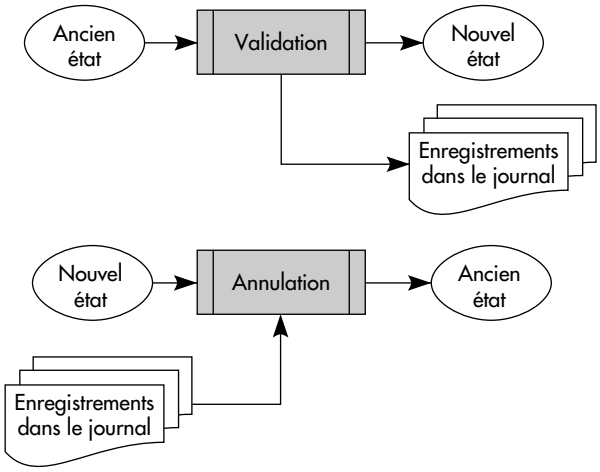


Figure XVI.35 : Principes de la validation et de l'annulation avec journal

L'annulation d'une transaction ayant mis en place des mises à jour dans la base est une procédure difficile. Elle s'effectue à partir du journal des images avant. L'annulation nécessite le parcours du journal à l'envers, c'est-à-dire en reculant dans le temps.

7.2. ÉCRITURES NON EN PLACE

Avec cette approche, les pages modifiées sont recopiées dans de nouvelles pages en mémoire et écrites dans de nouveaux emplacements dans la base. L'atomicité de la validation d'une transaction peut être réalisée par la **technique des pages ombre** [Lorie77]. Les pages nouvelles séparées et attachées à la transaction modifiante sont appelées pages différentielles. Les pages anciennes constituent les pages ombre. Avant toute lecture, il faut alors que le SGBD consulte les pages différentielles validées et non encore physiquement intégrées à la base. Cela conduit à alourdir les procédures de consultation et accroître les temps de réponse.

Une solution plus efficace a été proposées dans [Lampson76]. Elle consiste à réaliser une intégration physique atomique par **basculement des tables des pages** (voir figure XVI.34). Pour cela, chaque fichier de la base de données possède un descripteur pointant sur la table des pages du fichier. Quand une transaction désire modifier un fichier, une copie de la table des pages est effectuée et les adresses des pages modifiées sont positionnées à leurs nouvelles valeurs, de sorte que la copie de la table des pages pointe sur la nouvelle version du fichier (anciennes pages non modifiées et nouvelles pages modifiées). La validation consiste alors simplement à mettre à jour le descripteur du fichier en changeant l'adresse de la table des pages.

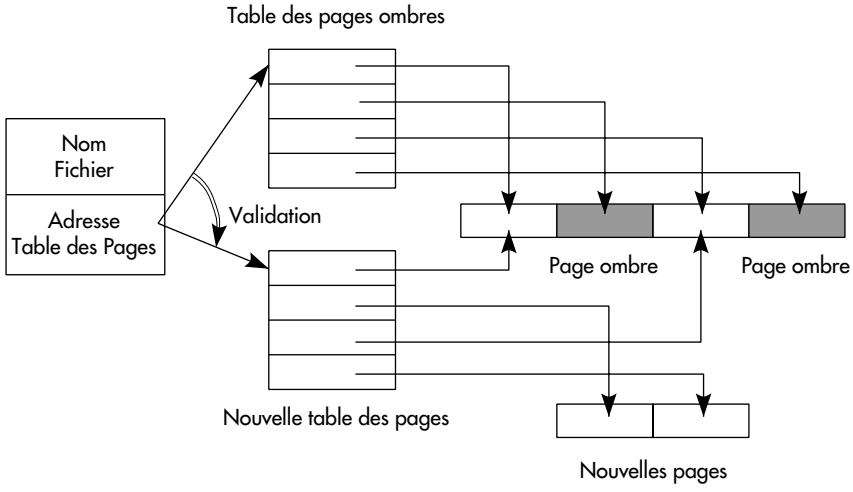


Figure XVI.36 : Validation par basculement de la table des pages

7.3. VALIDATION EN DEUX ÉTAPES

Dans la plupart des systèmes, un ensemble de processus collabore à la vie d'une transaction. Afin de coordonner la décision de valider une transaction, un **protocole de validation en deux étapes** est généralement utilisé. Ce protocole a été proposé dans un contexte de système réparti [Lampson76, Gray78] mais est aussi utile dans un contexte centralisé multi-processus. La validation en deux étapes peut être perçue comme une méthode de gestion des journaux. Lors de la première étape, les images avant et après sont enregistrées dans le journal si bien qu'il devient ensuite possible de valider ou d'annuler la transaction quoi qu'il advienne (excepté une perte du journal) ; cette étape est appelée préparation à la validation. La seconde étape est la réalisation de la validation ou de l'annulation atomique, selon que la première étape s'est bien ou mal passée. La preuve de correction d'un tel protocole déborde le cadre de cet ouvrage et pourra être trouvée dans [Baer81].

Le protocole de validation en deux phases coordonne l'exécution des commandes COMMIT par tous les processus coopérant à l'exécution d'une transaction. Le principe consiste à diviser la validation en deux phases. La phase 1 réalise la préparation de l'écriture des résultats des mises à jour dans la BD et la centralisation du contrôle. La phase 2, réalisée seulement en cas de succès de la phase 1, intègre effectivement les résultats des mises à jour dans la BD répartie. Le contrôle du système est centralisé sous la direction d'un processus appelé **coordinateur**, les autres étant des **participants**. Nous résumons ces concepts ci-dessous.

Notion XVI.27 : Protocole de validation en deux étapes (*Two Phase Commit*)

Protocole permettant de garantir l'atomicité des transactions dans un système multi-processus ou réparti, composé d'une préparation de la validation et de centralisation du contrôle, et d'une étape d'exécution.

Notion XVI.28 : Coordinateur de validation (*Commit Coordinator*)

Processus d'un système multi-processus ou réparti qui dirige le protocole en centralisant le contrôle.

Notion XVI.29 : Participant à validation (*Commit Participant*)

Processus d'un système multi-processus ou réparti qui exécute des mises à jour de la transaction et obéit aux commandes de préparation, validation ou annulation du coordinateur.

Le protocole de validation en deux étapes découle des concepts précédents. Lors de l'étape 1, le coordinateur demande aux autres sites s'ils sont prêts à commettre leurs mises à jour par l'intermédiaire du message PREPARER (en anglais *PREPARE*). Si tous les participants répondent positivement (prêt), le message VALIDER (en anglais *COMMIT*) est diffusé : tous les participants effectuent leur validation sur ordre du coordinateur. Si un participant n'est pas prêt et répond négativement (KO) ou ne répond pas (*time out*), le coordinateur demande à tous les participants de défaire la transaction (message ANNULER, en anglais *ABORT*).

Le protocole nécessite la journalisation des mises à jour préparées et des états des transactions dans un journal local à chaque participant, ceci afin d'être capable de retrouver l'état d'une transaction après une éventuelle panne et de continuer la validation éventuelle. Le protocole est illustré figure XVI.37 dans le cas favorable où un site client demande la validation à deux sites serveurs avec succès. Notez qu'après exécution de la demande de validation (VALIDER), les participants envoient un acquittement (FINI) au coordinateur afin de lui signaler que la transaction est maintenant terminée.

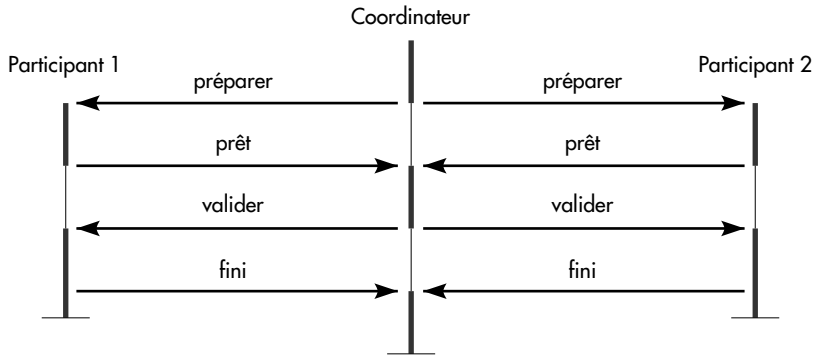


Figure XVI.37 : Validation en deux étapes avec succès

Le cas de la figure XVI.38 est plus difficile : le participant 2 est en panne et ne peut donc répondre au message de demande de préparation. Une absence de réponse est assimilée à un refus. Le coordinateur annule donc la transaction et envoie le message ANNULER. Le participant 1 annule la transaction. Le participant 2 l'annulera lorsqu'il repartira après la panne, une transaction non prête étant toujours annulée.

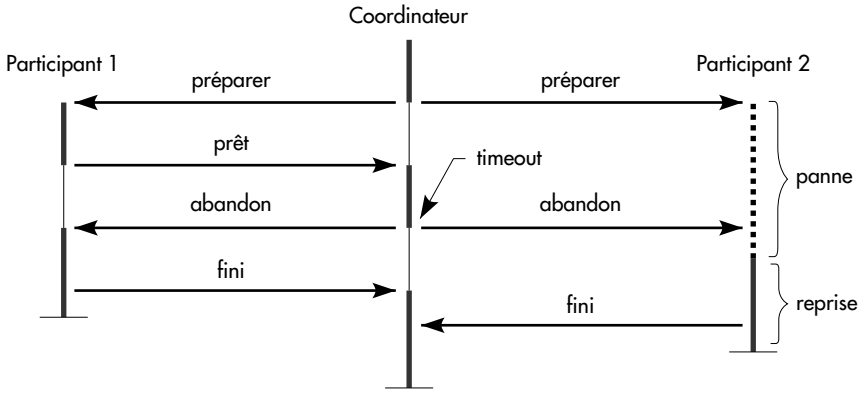


Figure XVI.38 : Validation en deux étapes avec panne totale du participant 2

Le cas de la figure XVI.39 est encore plus difficile : le participant 2 tombe en panne après avoir répondu positivement à la demande de préparation. Le coordinateur envoie donc le message COMMIT qui n'est pas reçu par le participant 2. Heureusement, celui-ci a dû sauvegarder l'état de la sous-transaction et ses mises à jour dans un journal fiable sur disque. Lors de la procédure de reprise, le journal sera examiné. La transaction étant détectée prête à commettre, son état sera demandé au coordinateur (ou a un participant quelconque) par un message appelé STATUS. En réponse à ce message, la validation sera confirmée et les mises à jour seront appliquées à partir du journal. Les deux sous-transactions seront donc finalement validées.

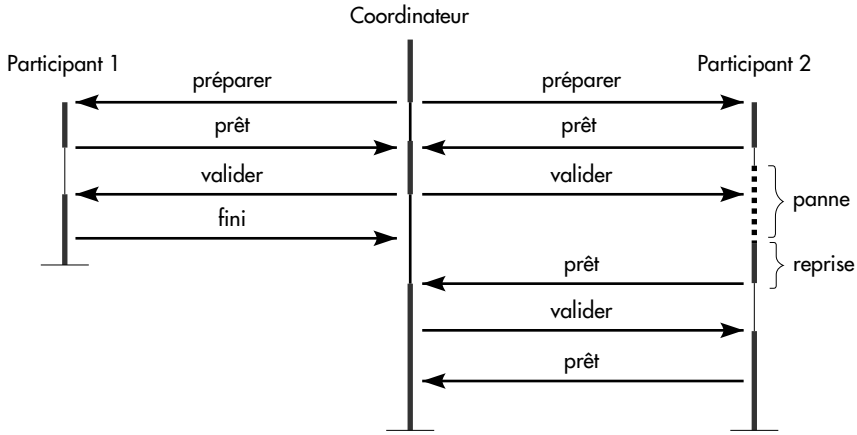


Figure XVI.39 : Validation en deux étapes avec panne partielle du participant 2

Au-delà du protocole en deux étapes, il existe des protocoles en trois étapes qui évitent les blocages du protocole précédent en cas de panne du coordinateur. Le plus courant est le protocole dit d'*abort* supposé, qui en cas de panne du coordinateur suppose l'abandon de la transaction. Ceci est même possible en deux phases [Mohan86].

Le protocole en deux étapes a été standardisé par l'ISO. TP est le protocole standard proposé par l'ISO dans le cadre de l'architecture OSI afin d'assurer une validation cohérente des transactions dans un système distribué. Le protocole est arborescent en ce sens que tout participant peut déclencher une sous-transaction distribuée. Un site responsable de la validation de la sous-transaction est choisi. Un coordinateur est responsable de ses participants pour la phase 1 et collecte les PREPARE. Il demande ensuite la validation ou l'*abort* selon la décision globale à un site appelé ***point de validation*** qui est responsable de la phase 2 : c'est le point de validation qui envoie les COMMIT aux participants. L'intérêt du protocole, outre l'aspect hiérarchique, est que le point de validation peut être différent du coordinateur : ce peut être un nœud critique dans le réseau dont la présence à la validation est nécessaire. La figure XVI.40 illustre ce type de protocole hiérarchique avec point de validation.

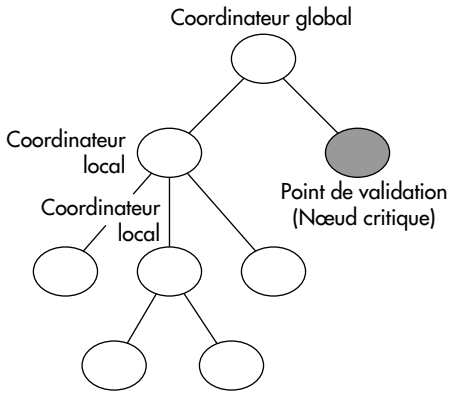


Figure XVI.40 : Le protocole hiérarchique avec point de validation TP

8. LES PROCÉDURES DE REPRISE

Dans cette section, nous examinons les procédures de reprises. Après quelques considérations générales, nous introduisons divers protocoles de reprise possibles [Bernstein87].

8.1. PROCÉDURE DE REPRISE

On appelle **procédure de reprise** la procédure exécutée lors du redémarrage du système après un arrêt ou une panne. Plusieurs types de reprises sont distingués et un schéma de principe possible pour chaque type est étudié ci-dessous. Ces procédures consistent à repartir à partir du journal et éventuellement de sauvegardes, pour reconstituer une base cohérente en perdant le minimum du travail exécuté avant l'arrêt. Afin de préciser les idées, nous définissons la notion de procédure de reprise.

Notion XVI.30 : Procédure de reprise (*Recovery procedure*)

Procédure système exécutée lors du redémarrage du système ayant pour objectif de reconstruire une base cohérente aussi proche que possible de l'état atteint lors de la panne ou de l'arrêt du système.

La reprise normale ne pose pas de problème. Elle a lieu après un arrêt normal de la machine. Lors de cet arrêt, un point de reprise système est écrit comme dernier enre-

gistrement du journal. La reprise normale consiste simplement à restaurer le contexte d'exécution sauvegardé lors de ce point de reprise. Une telle procédure est exécutée lorsque le dernier enregistrement du journal est un point de reprise système.

8.2. REPRISE À CHAUD

La reprise après une panne du système est appelée **reprise à chaud**. Rappelons qu'une panne système entraîne la perte de la mémoire centrale sans perte de données sur mémoires secondaires. Dans ce cas, le système doit rechercher dans le journal le dernier point de reprise système et restaurer l'état machine associé. Le journal est alors traité en avant à partir du point de reprise afin de déterminer les transactions validées et celles non encore validées (appelées respectivement gagnantes et perdantes dans [Gray81]). Puis le journal est traité en sens inverse afin de défaire les mises à jour des transactions non validées (les perdantes). Une telle procédure est illustrée figure XVI.41.

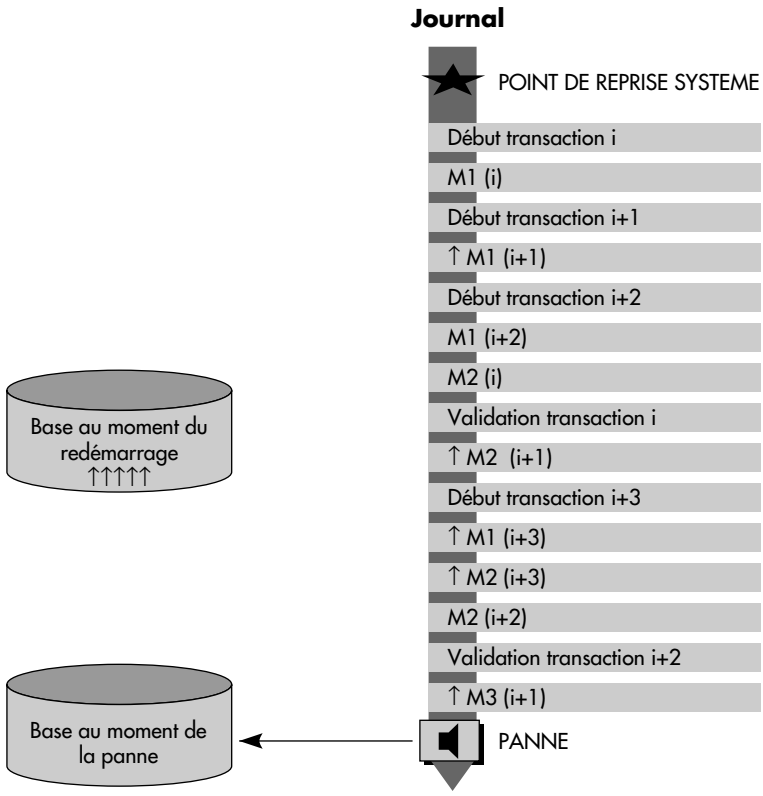


Figure XVI.41 : Exemple de reprise à chaud

Les transactions i et (i+2) sont gagnantes alors que (i+1) et (i+3) sont perdantes. Mj(i) désigne la j^e^ mise à jour de la transaction i. Les mises à jour défaites sont marquées par une flèche.

Cette procédure est exécutée lors des redémarrages du système quand le dernier enregistrement du journal n'est pas un point de reprise et que l'opérateur ne signale pas une perte de mémoire secondaire.

8.3. PROTOCOLES DE REPRISE À CHAUD

Lors d'une reprise à chaud, il est généralement nécessaire de défaire (*Undo*) des transactions non commises et refaire (*Redo*) des transactions commises. Cependant, différents protocoles sont applicables selon les cas indiqués figure XVI.42. Les conditions indiquées en colonne portent sur les mises à jour des transactions commises, celles en ligne portent sur les mises à jour des transactions non commises au moment de la panne. Les actions nécessaires sur les transactions exécutées depuis le dernier point de reprise système sont indiquées au carrefour de la ligne et de la colonne, quand les deux conditions sont vérifiées.

Non commises → **Commises ↓**	**Pas de mise à jour en base**	**Certaines mises à jour en base**
Certaines mise à jour en base	No Undo, Redo (1)	Undo, Redo (2)
Toute mise à jour en base	No Undo, No Redo (3)	Undo, No Redo (4)

Figure XVI.42 : Protocoles de reprise à chaud possibles

Les renvois numériques du tableau ci-dessus correspondent aux protocoles suivants :

1. Le **protocole Refaire sans Défaire (*No Undo, Redo*)** est applicable dans le cas où l'on est sûr que toute transaction non commise n'a aucune mise à jour installée en base permanente. L'application d'un tel protocole nécessite donc le report des mises à jour d'une transaction simultanément ou après l'écriture de l'enregistrement COMMIT pour cette transaction dans le journal. La simultanéité nécessite un matériel spécial capable d'assurer l'atomicité de plusieurs mises à jour simultanées de disques différents. Différer les mises à jour en base après la validation relâche la règle de la validation après écritures vue ci-dessus.
2. Le **protocole Défaire et Refaire (*Undo, Redo*)** est le plus classique. Il est appliqué dans le cas où certaines mises à jour de transactions non commises peuvent être

enregistrées dans la base et certaines mises à jour de transactions commises peuvent être perdues dans la base. Il est général et est le plus fréquemment utilisé. Il ne nécessite pas de différer les mises à jour en base permanente.

3. Le **protocole Défaire sans Refaire (*Undo, No Redo*)** est applicable dans le cas où l'on est sûr que toute transaction commise a toutes ses mises à jour reportées en base permanente. Ce peut être vrai si le système respecte la règle de validation après report des écritures vue ci-dessus.
4. Le **protocole Ni Refaire Ni Défaire (*No Undo, No Redo*)** est rarement applicable. Il nécessite que toute transaction commise ait toutes ses mises à jour en base permanentes et que toute transaction non commise n'ait aucune de ses mises à jour en base permanentes. Il faut pour cela un matériel spécifique capable d'exécuter toutes les écritures à la fois et de garantir l'atomicité de ces écritures multiples.

8.4. REPRISE À FROID ET CATASTROPHE

La reprise après une panne de mémoire secondaire est plus difficile. Ce type de reprise est souvent appelé **reprise à froid**. Une telle reprise est exécutée lorsqu'une partie des données est perdue ou lorsque la base est devenue incohérente. Dans ce cas, une sauvegarde cohérente de la base ainsi que le journal des activités qui ont suivi sont utilisés afin de reconstruire la base actuelle. Il suffit pour cela d'appliquer les images après à partir de la sauvegarde et du point de reprise associé. Lorsque le journal a été parcouru en avant jusqu'à la fin, la reprise à froid enchaîne en général sur une reprise à chaud.

Une **panne catastrophique** survient quand tout ou partie du journal sur mémoire secondaire est perdu. Certains systèmes, tel système R, permettent de gérer deux copies du journal sur des mémoires secondaires indépendantes afin de rendre très peu probable un tel type de panne. Dans le cas où cependant une telle panne survient, il n'existe guère d'autre solution que d'écrire des transactions spéciales chargées de tester la cohérence de la base en dialoguant avec les administrateurs et de compenser les effets des mises à jour malheureuses.

9. LA MÉTHODE ARIES

ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*) [Mohan92] est une des méthodes de reprise de transactions des plus efficaces. Elle est à la base des algorithmes de reprises implémentés dans de nombreux systèmes, dont ceux des SGBD d'IBM. Réalisée dans le cadre du projet de SGBD extensible d'IBM Starburst, cette méthode s'est imposée comme une des meilleures méthodes intégrées.

9.1. OBJECTIFS

Les objectifs essentiels initiaux de la méthode étaient les suivants [Mohan92] :

– **Simplicité**. Vu la complexité du sujet, il est essentiel de viser à la simplicité de sorte à être applicable en vraie grandeur et à garantir des algorithmes robustes. C'est d'ailleurs vrai pour tous les algorithmes, au moins en système.

– **Journalisation de valeur et d'opération**. Classiquement, ARIES permet de défaire et refaire une mise à jour avec les images avant et après. Au-delà, la méthode intègre la possibilité de journaliser des opérations logiques, comme incrément et décrément d'un compte. Cela permet le verrouillage sémantique avec des modes d'opérations sophistiqués, pour supporter des exécutions à opérations commutatives non sérialisables au niveau des lectures et écritures. Ceci est important pour permettre une meilleure concurrence, comme vu ci-dessus. Une des difficultés dans ce type de journalisation par opération est de bien associer un enregistrement du journal avec l'état de la page qui lui correspond : ceci est accompli astucieusement avec un numéro de séquence d'enregistrement journal (*Log Sequence Number, LSN*) mémorisé dans la page. Une autre difficulté est due aux pannes pendant la reprise : il faut pouvoir savoir si l'on a refait ou non une opération. Ceci est accompli en journalisant aussi les mises à jour effectuées pendant la reprise par des enregistrements de compensation (*Compensation Log Records, CRLs*).

– **Gestion de mémoire stable et du cache flexible**. La mémoire stable est gérée efficacement. Différentes stratégies de report du cache sont utilisables. Reprise et verrouillage sont logiques par nature ; cela signifie que la reprise reconstruit une base cohérente qui n'est pas forcément physiquement identique à la base perdue (pages différentes), mais aussi que les verrouillages sont effectués à des niveaux de granularités variables, en utilisant le verrouillage d'intention vu ci-dessus.

– **Reprise partielle de transactions**. Un des objectifs de la méthode est de pouvoir défaire une transaction jusqu'au dernier point de sauvegarde. C'est important pour pouvoir gérer efficacement les violations de contraintes d'intégrité et l'utilisation de données périmées dans le cache.

– **Reprise granulaire orientée page**. La méthode permet de reprendre seulement une partie de la base, la reprise d'un objet (une table par exemple) ne devant pas impliquer la reprise d'autres objets. La reprise est orientée page en ce sens que la granularité de reprise la plus fine est la page : si une seule page est endommagée, il doit être possible de la reconstruire seule.

– **Reprise efficace et rapide**. Il s'agit bien sûr d'avoir de bonnes performances, à la fois pendant l'exécution normale de transactions et pendant la reprise. L'idée est de réduire le nombre de pages à réécrire sur disques et le temps unité central nécessaire à la reprise.

Outre ces quelques objectifs, ARIES en a atteint de nombreux autres tels que l'espace disque minimal nécessaire en dehors du journal, le support d'objets multi-pages, la

prise de points de reprise système efficaces, l'absence de verrouillage pendant la reprise, le journal limité en cas de pannes multiples lors de la reprise, l'exploitation du parallélisme possible, etc. [Mohan92].

9.2. LES STRUCTURES DE DONNÉES

ARIES gère **un journal unique**. Le format d'un enregistrement est représenté figure XVI.43.

LSN	*Log Sequence Number* : Adresse relative du premier octet de l'enregistrement dans le log, utilisé comme identifiant de l'enregistrement (non mémorisé).
TYPE	*Record type* : Indique s'il s'agit d'un enregistrement de compensation, d'un enregistrement de mise à jour normal, d'un enregistrement de validation à deux étapes (prêt, commise) ou d'un enregistrement système (*checkpoint*).
PrevLSN	*Previous LSN* : LSN de l'enregistrement précédent pour la transaction s'il y en a un.
TransID	*Transaction ID* : Identifiant de la transaction qui a écrit l'enregistrement.
PageID	*Page ID* : Identifiant de la page à laquelle les mises à jour de l'enregistrement s'appliquent, si elle existe. Présent seulement pour les enregistrements de mise à jour ou de compensation.
UndoNxtLSN	*Undo next LSN* : LSN de l'enregistrement suivant à traiter pendant la reprise arrière. Présent seulement pour les enregistrements de compensation.
Data	*Data :* Données décrivant la mise à jour qui a été effectuée, soit l'image avant et l'image après, soit l'opération et son inverse si elle ne s'en déduit pas. Présent seulement pour les enregistrements de mise à jour ou de compensation qui n'ont que les informations pour refaire.

Figure XVI.43 : Format des enregistrements du journal

Pour chaque page de données, ARIES nécessite **un seul champ par page nommé page_LSN** qui mémorise le numéro LSN du dernier enregistrement du journal qui concerne cette page.

En plus, ARIES gère une **table des transactions** utilisée pendant la reprise pour mémoriser l'état de validation des transactions actives (Prête ou Non dans le protocole à deux phases), et enfin une **table des pages sales**, c'est-à-dire modifiée en cache et ne correspondant plus à la version sur mémoire stable.

9.3. APERÇU DE LA MÉTHODE

ARIES garantit l'atomicité et la durabilité des transactions en cas de panne de processus, de transaction, du système ou de mémoire secondaire. Pour cela, ARIES utilise un journal et applique les règles de **journalisation avant écriture** (*Write-Ahead Loging, WAL*) et de **validation après écriture** (*Write Before Commit, WBC*) vues ci-dessus. ARIES journalise les mises à jour effectuées par les transactions sur chaque page. Il journalise aussi les états prêts et validés des transactions gérées par un protocole de validation à deux étapes, et les points de reprise système (*system checkpoints*).

En plus, ARIES journalise les mises à jour effectuées pour défaire les transactions, à la fois pendant le fonctionnement normal du système (par exemple, suite à une violation d'intégrité) et pendant la reprise. Ces enregistrements spéciaux de restauration sont appelés des **enregistrements de compensation** (*Compensation Log Records CLR*). Comme tous les enregistrements, chacun pointe sur l'enregistrement précédent du journal pour la même transaction (PrevLSN). Chaque enregistrement de compensation pointe aussi par UndoNxtLSN sur le prédécesseur de l'enregistrement juste défait. Par exemple, si une transaction a écrit les enregistrements normaux 1, 2, 3, puis 3' et 2' pour compenser respectivement 3 et 2, alors 3' pointe sur 2 et 2' sur 1. Cela permet de retrouver ce qui a déjà été défait. Si une panne survient pendant une reprise arrière d'une transaction, par exemple après exécution de 2', grâce au journal on retrouvera 2' puis 1 et l'on saura qu'il suffit de défaire 1. Ainsi, les enregistrements de compensation CLR gardent trace des progrès des reprises de transaction et permettent d'éviter de défaire ce qui a déjà été défait. Ils permettent aussi de refaire des opérations logiques grâce à la trace gardée.

ARIES marque chaque page avec le dernier LSN lui correspondant dans le journal. Ainsi, il est possible de retrouver très rapidement le dernier enregistrement du journal concernant cette page. Cela permet de connaître précisément l'état de la page, particulièrement de savoir que toutes les mises à jour ayant un LSN plus petit ont été correctement enregistrées. Cela est un point clé pour les performances de la reprise : on évite ainsi de ré-appliquer des mises à jour non perdues.

ARIES utilise des **points de reprise système périodique** marquant des points de bon fonctionnement dans le journal. Un enregistrement point de reprise identifie les transactions actives, leurs états, le LSN de leur plus récent enregistrement dans le journal, et aussi les pages modifiées et non reportées sur disques dans le cache (pages sales).

La procédure de reprise détermine le dernier point de reprise et, dans **une première passe d'analyse**, balaye le journal. Grâce aux LSN des pages sales figurant dans l'enregistrement point de reprise, le point de reprise exact du journal pour la passe suivante de réfaction est déterminé. L'analyse détermine aussi les transactions à défaire. Pour chaque transaction en cours, le LSN de l'enregistrement le plus récent du journal est déterminé. Il est mémorisé dans la table des transactions introduites ci-dessus.

Une deuxième passe de reconstruction est ensuite effectuée, durant laquelle ARIES répète l'histoire mémorisée dans le journal et non enregistrée sur mémoire stable. Ceci est fait pour toutes les transactions, y compris celles qui n'avaient pas été validées au moment de la panne. L'objectif est de rétablir l'état de la base au moment de la panne. Un enregistrement du journal est appliqué en avant (refait) sur une page si son LSN est supérieur à celui de la page, ce qui réduit le nombre de réfactions nécessaires. Durant cette phase, les verrous pour les transactions prêtes à valider au moment de la panne sont restaurés, afin de pouvoir les relancer.

Une troisième passe de réparation consiste à défaire les transactions perdantes, c'est-à-dire celles non validées ou non prêtes à valider au moment de la panne. Ceci s'effectue par parcours du journal en arrière depuis le plus grand LSN restant à traiter pour les transactions perdantes. Si l'enregistrement du journal est un enregistrement normal, les données d'annulation (image avant ou opération inverse) sont appliquées. L'enregistrement suivant à traiter pour la transaction est alors déterminé en regardant le champ PrevLSN. S'il s'agit d'un enregistrement de compensation, il est simplement utilisé pour déterminer l'enregistrement suivant à traiter mémorisé dans UndoNxtLSN, puisque les enregistrements de compensation ne sont jamais compensés.

En résumé, la méthode comporte donc trois passes : analyse, reconstruction en avant et réparation en arrière. Grâce aux chaînages intelligents des enregistrements du journal et à la mémorisation du numéro de séquence du dernier enregistrement pertinent au niveau de chaque page, la méthode évite de refaire et défaire des mises à jour inutilement. Elle intègre aussi la possibilité d'annulation logique ou physique d'opérations, ce qui permet des verrouillages en modes variés exploitant la commutativité. La méthode a été implémentée dans de nombreux systèmes industriels.

10. LES MODÈLES DE TRANSACTIONS ÉTENDUS

Nous avons ci-dessus étudié les techniques implémentées dans les SGBD pour assurer l'atomicité des transactions. Ces techniques sont aujourd'hui bien connues et opérationnelles pour les transactions en gestion. Cependant, les techniques de gestion de transactions atomiques sont trop limitées dans le cas de transactions longues que l'on rencontre en conception (CAO, CFAO), en production de documents (PAO) et plus généralement dans les domaines techniques nécessitant des objets complexes. Aussi différents modèles de transactions ont-ils été proposés, cherchant à permettre une plus grande flexibilité que le tout ou rien. Ces modèles étendent plus ou moins les propriétés ACID vues ci-dessus.

10.1. LES TRANSACTIONS IMBRIQUÉES

Le modèle des **transactions imbriquées** est dû à [Moss85]. Il étend le modèle plat, à un seul niveau, à une structure de transaction à niveaux multiples. Chaque transaction peut être découpée en sous-transactions, et ce récursivement.

Notion XVI.31 : Transaction imbriquée (*Nested transaction*)

Transaction récursivement décomposée en sous-transactions, chaque sous-transaction et la transaction globale étant bien formée, c'est-à-dire commençant par BEGIN et se terminant par COMMIT.

On obtient ainsi un arbre de transactions (voir figure XVI.44). Une transaction fille est lancée par sa mère par exécution d'une commande *BEGIN* ; elle rend un compte-rendu à sa mère qui se termine après toutes ses filles. De manière classique, chaque (sous-)transaction se termine par *COMMIT* ou *ABORT* et est une unité atomique totalement exécutée ou pas du tout. Chaque (sous-)transaction peut être refaite (*REDO*) ou défaite (*UNDO*) par un journal hiérarchisé.

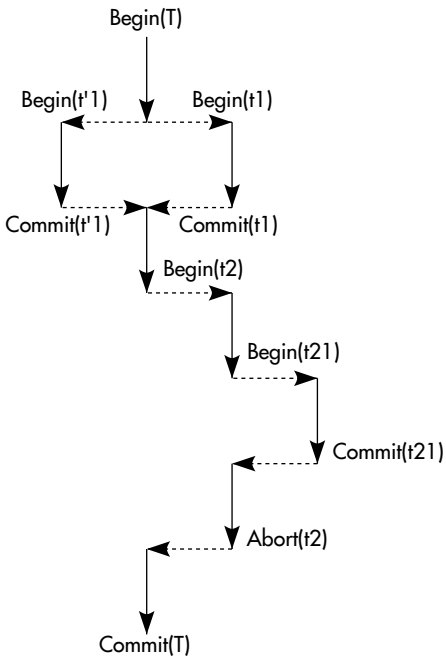


Figure XVI.44 : Exemple de transactions imbriquées

Les règles de validation et d'annulation résultent du découpage logique d'un travail en sous-tâche :

– L'annulation d'une transaction mère implique l'annulation de toutes ses filles, et ce récursivement.

– La validation d'une transaction fille est conditionnée par celle de ses ancêtres, et ce récursivement.

En conséquence, bien que chaque transaction se termine par *COMMIT* ou *ABORT,* la validation n'est confirmée que lorsque tous les ancêtres ont validé. Le **modèle fermé** [Moss85] garantit l'atomicité de la transaction globale. Donc un échec d'une sous-transaction implique une annulation de la mère. Ceci est très limitatif, mais conserve les propriétés ACID entre transactions globales. Au contraire, le **modèle ouvert** relâche l'atomicité de la transaction globale.

Du point de vue de la concurrence, chaque (sous-)transaction applique le verrouillage deux phases. Les verrous sont hérités dans les deux sens : lorsqu'une transaction fille réclame un verrou tenu par un de ses ancêtres, elle n'est pas bloquée ; lorsqu'une transaction fille valide, les verrous qu'elle a acquis sont transmis à sa mère. Deux (sous-)transactions peuvent donc se bloquer voir s'interbloquer si le verrou n'est pas tenu par un ancêtre commun. Ceci complique les détections de verrous mortels.

Le **modèle ouvert** relâche donc la nécessité d'atomicité des transactions globales. Alors, une (sous-)transaction peut être annulée alors que sa mère est validée. Le travail est donc seulement partiellement réalisé. Ceci nécessite l'introduction de **transactions de compensation** [Korth90]. Par exemple, la compensation d'une transaction achetant de la marchandise consiste à rendre cette marchandise. Une (sous-)transaction de compensation défait donc une (sous-)transaction précédemment exécutée. Une variante est une **transaction de complément**. Ces transactions de complément viendront plus tard refaire les portions de travail non validées. Par exemple, le complément d'une transaction de mise à jour d'un ouvrage qui a échoué sur le chapitre II est une transaction qui met à jour le chapitre II. Finalement, dès que l'on relâche l'atomicité, il est nécessaire d'introduire de telles transactions dépendant de la sémantique de l'application.

Ces principes ont conduit à définir un modèle de transactions imbriquées ouvert très flexible [Weikum92]. Outre l'atomicité, ce modèle relâche en plus le principe d'isolation en laissant les résultats des sous-transactions commises visibles aux transactions concurrentes. La reprise nécessite alors de gérer des sphères d'influence de transactions, généralisation de la notion de traînée déjà vue. La sémantique des opérations est aussi intégrée pour gérer les commutativités d'opérations typées. De tels modèles commencent à être implémentés dans les SGBD classiques. Leurs limites et intérêts restent encore mal connus.

10.2. LES SAGAS

Les sagas [Garcia-Molina87] ont été introduites pour supporter des transactions longues sans trop de blocages. Une saga est une séquence de sous-transaction ACID {T1,…, Tn} devant s'exécuter dans cet ordre, associée à une séquence de transactions

de compensation {CT1,…, CTn}, Cti étant la compensation de Ti. La saga elle-même est une transaction longue découpée en sous-transactions. Tout abandon d'une sous-transaction provoque l'abandon de la saga dans sa totalité. La figure XVI.45 représente deux exécutions possibles d'une saga.

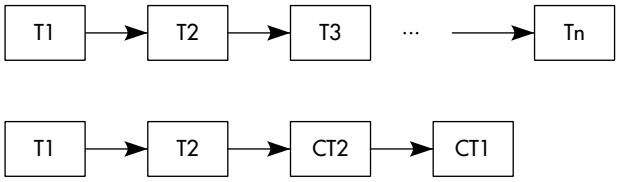


Figure XVI.45 : Exemples d'exécution de sagas

L'intérêt des sagas est de pouvoir relâcher le principe d'isolation. En effet, chaque transaction composante relâche ses verrous dès qu'elle est terminée. Une autre saga peut alors voir les résultats. L'annulation de la saga doit donc provoquer l'annulation de l'autre saga. Pour cela, il suffit d'enchaîner les transactions de compensation.

10.3. LES ACTIVITÉS

La généralisation des sagas conduit au **modèle d'activités**, proche des modèles de *workflow*. L'idée est d'introduire un langage de contrôle de transactions permettant de définir des travaux sous la forme d'une collection d'étapes avec enchaînements conditionnels, les étapes étant des transactions et les travaux des activités. Les propriétés souhaitables d'une activité sont de pouvoir garder un contexte d'exécution persistant, progresser par le biais de transactions ou reculer par le biais de compensation, essayer des alternatives et plus généralement introduire un flot de contrôle dépendant des succès et échecs des transactions composantes. Il faut aussi pouvoir être capable de différencier les échecs système des échecs de programmes, les premiers étant corrigés par le système, les seconds par l'activité.

Un exemple typique d'activité est la réservation de vacances. Cette activité est par exemple composée de trois transactions, la première T1 ayant une alternative T1' :

1. T1 : réservation avion alternative T1' : location voiture
2. T2 : réservation hôtel
3. T3 : location voiture

De plus, chaque transaction est associée à une transaction de compensation CT1, CT2 et CT3 permettant d'annuler son effet.

Cette idée a conduit à concevoir des modèles et des langages de contrôle d'activités [Dayal91, Wachter92] et à implémenter des moniteurs d'activités. Un tel langage per-

met de contrôler les transactions par des ordres du type *If abort, If commit, Run alternative, Run compensation*, etc. Le moniteur contrôle les transactions et exécute les programmes de ce langage. On obtient ainsi des systèmes de gestion de la coopération très flexibles, proches des *workflows* [Rusinkiewicz95].

10.4. AUTRES MODÈLES

Un modèle basé sur des **transactions multi-niveaux** a été introduit par [Weikum91, Beeri88]. Ce modèle est une variante du modèle des transactions imbriquées dans lequel les sous-transactions correspondent à des opérations à des niveaux différents d'une architecture en couches. Le modèle permet de considérer des opérations au niveau N qui se décompose en séquences d'opérations au niveau N – 1. Par exemple, le crédit d'un compte en banque se décompose au niveau inférieur (SQL) en une sélection (SELECT) et une mise à jour (UPDATE). Ceux-ci se décomposent à leur tour en lectures et écritures de pages au niveau système. Il est alors possible de décomposer une exécution simultanée de transactions en exécution (ou histoire) au niveau N, N – 1, etc. La méthode traditionnelle de gestion de concurrence introduite ci-dessus ne considère que le niveau le plus bas, des lectures et écritures de pages. La sérialisabilité d'une exécution simultanée multi-niveaux est alors définie comme la sérialisabilité à chaque niveau avec des ordres de sérialisation compatibles entre eux. En considérant la sémantique des opérations à chaque niveau, il devient possible d'exploiter la commutativité pour tolérer des exécutions simultanées qui ne seraient pas valides si l'on ne considérait que le niveau le plus bas. Des contrôleurs de transactions peuvent être définis aux différents niveaux. Il en résulte moins de reprise de transactions.

Le modèle *split-join* [Pu88] est une autre variante des transactions imbriquées. À l'image des *split* et *join* de processus sur Unix, il est possible avec ce modèle de diviser une transaction dynamiquement en deux sous-transactions ou de réunir deux sous-transactions en une. Des règles strictes sont imposées sur les sous-transactions pour garantir la sérialisabilité globale.

L'introduction d'un **arbre de versions d'objet** introduit une autre dimension aux modèles de transaction (voir figure XVI.46). La gestion de versions autorise les mises à jour simultanées d'un objet. Pour manipuler les versions, deux opérations sont introduites : *Check-out* pour sortir un objet de la base dans un espace privé et *Check-in* pour le réintroduire dans la base. Les objets versionnables sont verrouillés pendant tout le temps de la sortie de la base dans des modes spécifiques **dérivation partagée** ou **dérivation exclusive**. La dérivation exclusive ne permet que la sortie d'une version à un instant donné alors que la dérivation partagée permet plusieurs sorties simultanées. La figure XVI.47 donne les compatibilités de ces modes.

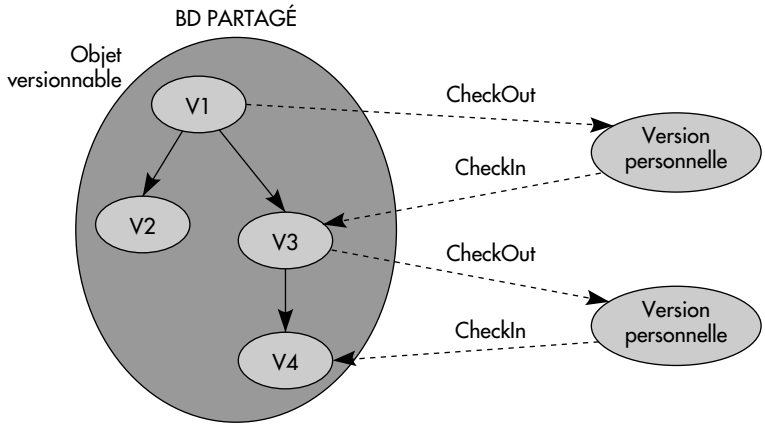


Figure XVI.46 : Exemple d'objet versionnable et d'évolution

	Lecture	**Ecriture**	**Dérivation Partagée**	**Dérivation Exclusive**
Lecture	1	0	1	1
Écriture	0	0	0	0
Dérivation Partagée	1	0	1	0
Dérivation Exclusive	1	0	0	0

Figure XVI.47 : Compatibilité des opérations de lecture, écriture et dérivation

Lors de la réinsertion d'une version dans la base (*Check-in*), le graphe de versions est mis à jour et la nouvelle version ajoutée comme une fille de celle dont elle dérive. Les versions sont généralement maintenues en différentiel, c'est-à-dire que pour un nœud donné de l'arbre, seuls les attributs ou pages modifiés par rapport au nœud père sont maintenus. Le problème qui se pose est de faire converger les versions. Si deux versions dérivant d'un ancêtre commun n'ont pas de données communes modifiées, une fusion automatique peut être réalisée par intégration de toutes les mises à jour effectuées à l'ancêtre. Sinon, il y a nécessité de définir des procédures de réconciliation de versions, voire d'une intervention manuelle effectuant des choix.

Au-delà de ces modèles fort nombreux, des formalismes de compréhension et description ont été proposés. ACTA [Chrysanthis90] est un cadre pour spécifier et raisonner sur la structure des transactions et leur comportement. Il permet d'exprimer la séman-

tique des interactions entre transactions en termes d'effets sur la validation ou l'annulation des autres transactions, et sur l'état des objets et des synchronisations nécessaires. Il permet ainsi de décrire et de raisonner sur un modèle d'activité ou de transaction.

11. SÉCURITÉ DES DONNÉES

Dans cette section, nous abordons brièvement les problèmes de sécurité, de plus en plus importants avec, par exemple, l'utilisation des bases de données pour le commerce électronique. Soulignons que les techniques de sécurité [Russell91] ne doivent cesser de progresser pour ne pas s'exposer à être dépassées par les escrocs.

11.1. IDENTIFICATION DES SUJETS

Un système de bases de données doit assurer la sécurité des données qu'il gère, c'est-à-dire que les opérations non autorisées ou mal intentionnées doivent être rejetées. Assurer la sécurité nécessite tout d'abord une identification des utilisateurs, encore appelés **sujets**. Les sujets effectuent des **opérations** sur les **objets** protégés qui les subissent. Dans les systèmes de base de données, les sujets sont les utilisateurs devant les terminaux et les objets les données protégées avec différents niveaux de granularité possibles. Les sujets doivent tout d'abord être **identifiés**.

Notion XVI.32 : Identification (*Identification*)

Procédé consistant à associer à un sujet un nom ou un numéro qui le désigne de manière unique.

Un premier moyen de violer la sécurité des données est pour un sujet de se faire passer pour un autre. Afin d'éviter cela, un procédé d'**authentification** des sujets est introduit.

Notion XVI.33 : Authentification (*Authentification*)

Procédé permettant de vérifier qu'un sujet est bien qui il prétend être.

Le procédé d'authentification le plus courant est l'utilisation de **mots de passe**. Un mot de passe est une chaîne de caractères en principe connue du sujet et de l'objet seuls, que le sujet doit fournir pour être authentifié. Les mots de passe ne doivent pas être facilement retrouvables, par exemple à partir d'un dictionnaire par essais exhaus-

tifs. Il est donc souhaitable de choisir des mots de passe longs (plus de 7 caractères), comportant un mixage de caractères numériques, alphanumériques et de contrôle. D'autres procédés plus sophistiqués et plus sûrs sont l'utilisation de questionnaires, l'exécution d'algorithmes connus seulement par l'objet et le sujet, l'utilisation de badges, de cartes à puces, d'empreintes digitales ou d'empreintes de la rétine. Ces derniers procédés nécessitent un périphérique spécialisé.

La définition des sujets ne pose a priori pas de problème : tout utilisateur est un sujet. Cependant, il est utile de considérer des **groupes** d'utilisateurs.

Notion XVI.34 : Groupe d'utilisateurs (*User group*)

Ensemble d'utilisateurs ayant chacun nom et mot de passe, désigné globalement par un nom, pouvant intervenir comme sujet dans le mécanisme d'autorisation.

La notion de groupe peut être étendue de manière hiérarchique, avec des sous-groupes. Dans ce cas, un groupe hérite de toutes les autorisations de ses antécédents hiérarchiques. Il est aussi possible de superposer plusieurs hiérarchies [Fernandez80]. Egalement, un sujet peut être un ensemble de transactions cataloguées ; l'accès à ces transactions doit alors être aussi protégé.

Un utilisateur peut joindre ou quitter un groupe. La dynamique des groupes lors des départs de personnes dans une entreprise peut être un problème : il faut par exemple enlever un utilisateur et le remplacer par un autre. Attribuer ou enlever de multiples droits à des groupes devient rapidement difficile lors des changements de fonctions. Pour éviter ces problèmes, les SGBD ont introduit la notion de **rôle**.

Notion XVI.35 : Rôle (*Role*)

Ensemble de droits sur les objets caractérisé par un nom, pouvant intervenir comme sujet dans le mécanisme d'autorisation.

Ainsi, des droits sur les objets sont attribués aux rôles. Ceux-ci peuvent être ensuite affectés aux utilisateurs ou aux groupes d'utilisateurs. Les SGBD modernes sont capables de gérer à la fois des utilisateurs, des groupes et des rôles qui sont donc les sujets des mécanismes de contrôle d'autorisations.

11.2. LA DÉFINITION DES OBJETS

Le choix des objets à protéger est un des problèmes importants lors de la conception d'un module de protection pour un SGBD. Les systèmes anciens tels CODASYL et IMS restreignent les objets protégeables aux types d'objets : CODASYL permet de protéger par des clés n'importe quel niveau de type d'objet décrit dans le schéma (donnée élémentaire, article, fichier), alors qu'IMS permet de protéger des sous-

ensembles de champs d'un segment. Au contraire, les systèmes relationnels permettent de protéger des ensembles d'occurrences de tuples définis par des prédicats, donc dépendants du contenu. Les systèmes objet permettent souvent de protéger à la fois des types et des instances.

Dans les systèmes relationnels, les **vues** jouent un rôle essentiel dans la définition des objets à protéger [Chamberlin75]. Tout d'abord, lorsqu'une vue est un objet autorisé à un usager, celui-ci ne peut accéder qu'aux tuples de cette vue. Ensuite, il est possible avec la notion de vue de définir des objets protégés à granularité variable. Une vue peut être une table de la base : un droit d'accès est alors attribué à un usager sur une relation entière. Une vue peut aussi être un ou plusieurs tuples extraits dynamiquement d'une ou de plusieurs tables : un droit d'accès est alors attribué sur une relation virtuelle résultant de l'évaluation d'une question. La technique de modification de questions [Stonebraker75] étudiée au chapitre IX traitant des vues permet alors de protéger efficacement les tuples ou attributs de tuples composant la vue. Une autre approche est d'utiliser le même langage pour définir les objets sur lesquels on autorise des opérations que celui pour exprimer les questions. Cette approche est celle de QBE où les questions et les autorisations sont exprimées de la même manière. Cela revient au même que l'utilisation de vues comme objets de protection, puisqu'une vue est définie comme une question.

Dans les systèmes objet, il est souvent souhaitable de protéger les types et même individuellement les opérations des types, mais aussi les instances. Les objets sont organisés en hiérarchie d'héritage : les instances d'une classe peuvent être perçues comme le dernier niveau de la hiérarchie. Il doit être possible d'ajouter des prédicats pour partitionner plus finement l'extension d'une classe en objets autorisés ou non. Généralement, un utilisateur d'une sous-classe hérite des droits de la classe de base. Dans le cas où des autorisations positives (droits d'accès) et négatives (interdictions d'accès) seraient attribuées, déterminer les autorisations d'une sous-classe devient difficile, notamment en cas d'héritage multiple [Rabitti91]. Aujourd'hui, les SGBD objet restent pour la plupart encore assez faibles du point de vue de la sécurité.

11.3. ATTRIBUTION ET CONTRÔLE DES AUTORISATIONS : LA MÉTHODE DAC

La méthode DAC (*Discretionary Access Control*) est la plus utilisée dans les SGBD. Elle consiste à attribuer aux sujets des droits d'opérations sur les objets et à les vérifier lors des accès. Un sujet authentifié peut exécuter certaines opérations sur certains objets selon les droits d'exécution accordés par les administrateurs du système ou plus généralement par d'autres sujets. Plus précisément, nous définirons le concept **d'autorisation** déjà utilisé informellement.

Notion XVI.36 : Autorisaton (*Authorization*)

Droit d'exécution d'une opération par un sujet sur un objet.

Les autorisations considérées sont en général positives : ce sont des accords de droits. Cependant, la défense américaine a aussi introduit dans ses standards des possibilités d'autorisations négatives, qui sont des interdictions d'accès. Cela pose des problèmes de conflits, les interdictions dominant en général les autorisations. Dans la suite, nous ne considérons que les autorisations positives.

L'attribution d'une autorisation peut dépendre :

– du sujet, par exemple de ses privilèges d'accès ou du terminal qu'il utilise,

– de l'objet, par exemple de son nom, son état actuel, son contenu ou sa valeur,

– de l'opération effectuée, par exemple lecture ou mise à jour,

– d'autres facteurs, par exemple de l'heure du jour ou de la date.

Il existe plusieurs manières de mémoriser les autorisations. Une première est l'utilisation de **matrices d'autorisations**. Une matrice d'autorisations est une matrice dont les lignes correspondent aux sujets et les colonnes aux objets, définissant pour chaque couple sujet-objet les opérations autorisées.

Considérons par exemple deux relations décrivant les objets NOM et RESULTAT d'étudiants et des sujets étudiants, secrétaires et professeurs. Les opérations que peuvent accomplir les sujets sur les objets nom et résultat sont lecture et écriture. Les autorisations peuvent être codées par deux bits, droit d'écriture puis droit de lecture. La figure XVI.48 représente la matrice correspondant à cet exemple, 0 signifiant accès interdit et 1 accès autorisé.

	NOM	**RÉSULTAT**
ÉTUDIANT	0 1	0 1
SECRÉTAIRE	1 1	0 1
PROFESSEUR	1 1	1 1

Figure XVI.48 : Exemple de matrice d'autorisations

Dans les SGBD, les définitions des sujets, objets et autorisations sont mémorisées dans la méta-base ou catalogue du système. En pratique, la matrice d'autorisation peut être stockée :

– par ligne – à chaque sujet est alors associée la liste des objets auxquels il peut accéder ainsi que les droits d'accès qu'il possède ;

– par colonne – à chaque objet associée la liste des sujets pouvant y accéder avec les droits d'accès associés ;
– par élément – à chaque couple sujet-objet sont associés les droits d'accès du sujet sur l'objet.

C'est cette dernière technique qui est retenue dans les SGBD relationnels. Les autorisations sont mémorisées dans une table DROITS(<Sujet>, <Objet>, <Droit>, <Donneur>), le donneur permettant de retrouver la provenance des droits.

L'**attribution de droits** aux sujets sur les objets est un autre problème important. Plus précisément, il est nécessaire de définir qui attribue les droits. Deux approches sont possibles. Soit, comme dans SQL2, le créateur d'un objet (une relation ou une vue par exemple) devient son propriétaire et reçoit tous les droits. Afin de passer et retirer les droits qu'il possède, il doit alors disposer de commandes du type [Griffiths76] :

- **GRANT** < types opérations > **ON** < objet >
  TO < sujet >
  [**WITH GRANT OPTION**]

  avec :

  ```
  <type opération> ::= ALL PRIVILEGES | <action>+
  <action>         ::= SELECT | INSERT
                     | UPDATE [(<nom de colonne>+)]
                     | REFERENCE [(<nom de colonne>+)]
  <sujet> ::=      PUBLIC | <identifiant d'autorisation>
  ```

- **REVOKE** < types opérations > **FROM** < objet > **TO** < sujet >.

Soulignons que PUBLIC désigne l'ensemble des utilisateurs, alors qu'un identifiant d'autorisation peut désigner un utilisateur, un groupe ou un rôle. L'option WITH GRANT OPTION permet de passer aussi le droit de donner le droit.

Soit un groupe de sujets particuliers (les administrateurs des bases de données) possèdent tous les droits et les allouent aux autres par des primitives du même type. La première approche est décentralisée alors que la seconde est centralisée. Des approches intermédiaires sont possibles si l'on distingue plusieurs groupes de sujets avec des droits a priori [Chamberlin78].

11.4. ATTRIBUTION ET CONTRÔLE DES AUTORISATIONS : LA MÉTHODE MAC

Un moyen plus simple de contrôler les autorisations consiste à associer un **niveau d'autorisation** aux sujets et aux objets.

Notion XVI.37 : Niveau d'autorisation (*Authorization level*)

Fonction mappant l'ensemble des sujets et utilisateurs sur les entiers [0-N] permettant à un sujet d'accéder à un objet si et seulement si : Niveau(Sujet) ≥ Niveau(Objet).

Un niveau d'autorisation est un nombre associé à chaque objet ou à chaque sujet. Il caractérise des niveaux de sensibilité des informations (Top secret, secret, confidentiel, non classé) et des niveaux de permissions des sujets vis à vis des objets. Un accès est autorisé si et seulement si le niveau du sujet accédant est supérieur ou égal au niveau de l'objet accédé. Le sujet domine alors le type d'information auquel il accède.

Considérons par exemple les sujets et objets introduits ci-dessus avec les niveaux d'autorisation : Etudiant (1), Secrétaire (2), Enseignant (3), Nom (1), Résultat (3). La matrice d'autorisation équivalente est représentée figure XVI.49.

	NOM	**RÉSULTAT**
ÉTUDIANT	1 1	0 1
SECRÉTAIRE	1 1	0 0
PROFESSEUR	1 1	1 1

Figure XVI.49 : Matrice d' autorisation équivalente aux niveaux indiqués

L'inconvénient de la solution niveau d'autorisation est que l'on perd la notion d'opération. Cependant, cette solution est simple à implanter et de plus elle permet de combiner les niveaux. En effet, si un sujet de niveau S_1 accède à travers une procédure ou un équipement de niveau S_2, on associera au sujet le niveau $S = \min (S_1, S_2)$. Par exemple, s'il existe un terminal en libre accès de niveau 1 et un terminal situé dans un bureau privé de niveau 3, un enseignant ne conservera ses privilèges que s'il travaille à partir du terminal situé dans le bureau. De plus, la méthode peut être étendue, avec des classes de niveaux partiellement ordonnées, vers un modèle plus complet de contrôle de flots d'informations. Elle peut aussi être combinée avec la méthode MAC.

11.5. QUELQUES MOTS SUR LE CRYPTAGE

Les données sensibles dans les bases peuvent être cryptées. Le cryptage est fondé sur des algorithmes mathématiques qui permettent de transformer les messages en rendant très difficile la découverte des transformations inverses. Plus précisément, on utilise en général des fonctions non facilement réversibles basées sur des divisions par des produits de grands nombres premiers. La recherche de la fonction inverse nécessite des décompositions en facteurs premiers très longues à réaliser. Les algorithmes de codage et décodage demandent donc l'attribution de clés de cryptage et de décryptage.

Les algorithmes symétriques utilisent **une seule clé secrète** pour le cryptage et le décryptage. La clé est par exemple un nombre de 128 bits. Un décodage d'un message crypté sans connaître la clé nécessite en moyenne quelques années avec des machines

parallèles les plus puissantes. Le mécanisme est donc sûr, mais le problème est la gestion des clés qui doivent être échangées. L'application d'un tel algorithme aux bases de données nécessite son implémentation dans le SGBD et au niveau de l'utilisateur. Il faut en effet pouvoir décrypter au niveau du SGBD pour effectuer les recherches. Rechercher sur des données cryptées est très difficile puisque les chaînes de données codées dépassent en général un attribut. Le SGBD devra donc garder un catalogue des clés de cryptage/décryptage. Les utilisateurs doivent pouvoir accéder au catalogue pour retrouver leur clé secrète. Celui-ci devient un point faible du système.

Les algorithmes à **clés publiques et privées** proposent deux clés différentes, inverses l'une de l'autre. L'une permet de coder, l'autre de décoder. Les clés doivent être plus longues qu'avec les algorithmes à clés secrètes, car elles sont plus facilement cassables, c'est-à-dire qu'à partir d'une clé publique de 40 bits, on peut déduire la clé privée en l'essayant sur un message compréhensible en quelques heures de calcul d'une machine puissante. Il faut donc des clés de 512 bits et plus pour obtenir une sécurité totale, ce qui peut poser des problèmes de législation. Avec de telles clés, le SGBD peut garder ses clés privées et publier ses clés publiques. L'utilisateur code les données avec une clé publique et les insère dans la base. Le SGBD décode avec la clé privée correspondante. Les résultats peuvent être envoyés codés avec une clé publique de l'utilisateur qui décode alors avec sa clé privée. Ce schéma est très sûr si les clés sont assez longues. Il est illustré figure XVI.50. Les algorithmes asymétriques tels que Diffie-Hellmann et RSA permettent ainsi des solutions très sûres, où SGBD et utilisateurs gardent leurs clés privées.

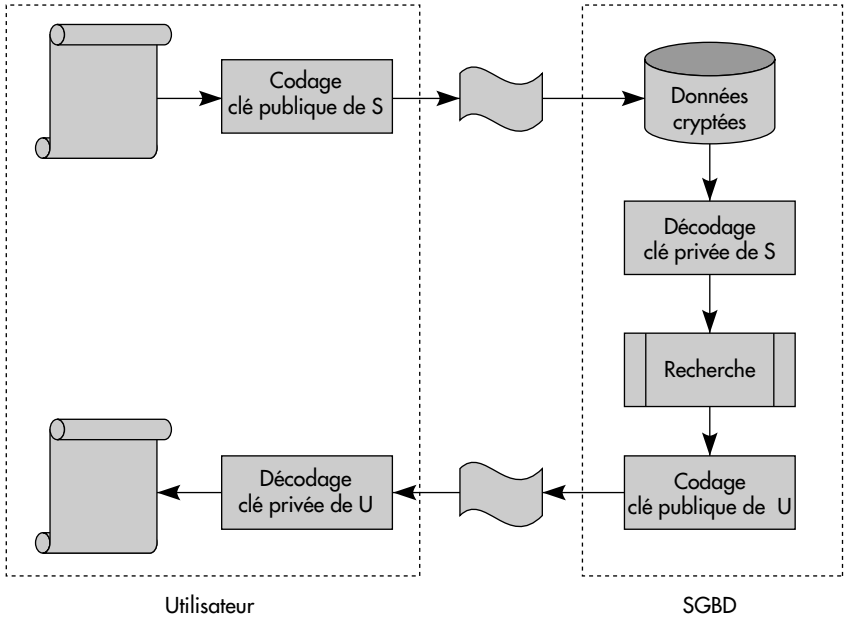


Figure XVI.50 : Fonctionnement avec clés asymétriques
(U désigne l'utilisateur et S le SGBD)

12. CONCLUSION ET PERSPECTIVES

Dans ce chapitre, nous avons abordé les problèmes de gestion de transactions. Nous avons étudié la théorie de la concurrence puis les algorithmes de contrôle. Ensuite, nous avons abordé la résistance aux pannes avec ses aspects validation et reprise. La méthode ARIES couplée au verrouillage deux phases et à la validation en deux étapes constitue une bonne référence que nous avons présentée.

Nous avons aussi abordé les modèles de transactions étendues, tels que les transactions imbriquées. Sur le contrôle de concurrence et sur les transactions étendues, de nombreux articles ont été publiés. Il est difficile d'y voir clair, mais nous avons synthétisé l'essentiel. Finalement, au-delà du verrouillage deux phases, de la validation en deux étapes et des procédures de reprise type ARIES, peu de choses sont implémentées dans les systèmes industriels. Une des raisons est sans doute la complexité des algorithmes très sensibles pour l'exploitation réelle des bases de données.

Plusieurs problèmes n'ont cependant pas été abordés, en particulier celui de la concurrence dans les index et celui de la sécurité dans les bases de données statistiques.

Les algorithmes de concurrence au niveau des index nécessitent des approches spécifiques du fait que l'index est un point de passage obligé pour toute les transactions accédant sur clé. On ne peut donc bloquer l'index pendant la durée de vie d'une transaction. La prise en compte des commutativités d'opérations telles que l'insertion et l'insertion avec reprise par des opérations logique comme la suppression est une direction de solution. Une autre voie repose sur une granularité de verrouillage très fine correspondant à une entrée d'index. Avec une telle granularité, dans un arbre B, il est possible de relâcher les verrous sur les entrées supérieures dès qu'une entrée inférieure est verrouillée et que l'on est sûr de ne pas devoir modifier l'entrée supérieure. Verrouiller efficacement des index nécessite à la fois une granularité fine et la prise en compte des commutativités d'opérations. Ceci complique la méthode de reprise. Notez qu'ARIES supporte ces points. Pour une introduction plus complète sur le contrôle de concurrence au niveau des index, vous pouvez consulter [Besancenot97].

L'objectif des bases de données statistiques est de fournir des résumés sur une population sans permettre à l'utilisateur de déduire des informations sur des individus particuliers. Ce type de protection est important, notamment dans le domaine médical et plus généralement pour le décisionnel. Il apparaît très difficile d'éviter les possibilités de déduction d'informations spécifiques à un individu en corrélant plusieurs résumés. Ainsi le chasseur de secrets reste-t-il une menace pour les bases de données statistiques [Denning80]. La seule manière de le perturber est sans doute de lui mentir un peu. Une vue d'ensemble du domaine est présentée dans un ouvrage assez ancien [Demillo78]. Le sujet n'est plus guère d'actualité, mais il peut le redevenir.

13. BIBLIOGRAPHIE

[Baer81] Baer J.-L., Gardarin G., Girault C., Roucairol G., « The Two-Step Commitment Protocol : Modeling, Specification and Proof Methodology », *5th Intl. Conf. on Software Engineering*, IEE Ed., San Diego, 1981.

Cet article modélise le protocole de validation en deux étapes par des réseaux de Petri. Il prouve partiellement la correction du protocole, c'est-à-dire que tous les sites prennent la même décision pour une transaction.

[Barghouti91] Barghouti N. S., Kaiser G. E., « Concurrency Control in Advance Database Applications » *ACM Computing Survey*, vol. 23, n° 3, p. 270-317, Sept. 1991

Cet article fait le point sur les techniques de contrôle de concurrence avancées. Il rappelle les techniques traditionnelles vues ci-dessus et résume le verrouillage altruiste, la validation par clichés, les transactions multi-niveaux, le verrouillage sémantique, les sagas, etc.

[Bancilhon85] Bancilhon F., Korth H., Won Kim, « A Model of CAD Transactions », *11th Int. Conf. on Very Large data Bases*, Stockholm, Suède, Août 1985.

Cet article propose un modèle de transactions longues pour les bases de données en CAO. Il fut l'un des précurseurs des modèles de transactions imbriqués développés par la suite.

[Beeri88] Beeri C., Scheck H-J., Weikum G., « Multi-Level Transaction Management, Theoretical Art or Practical Need ? », *Proc. Intl. Conf. on Extending Database Technology* (EDBT), p. 134-155, Venise, Mars 1988.

Cet article introduit la sérialisabilité à niveaux multiples présentée ci-dessus.

[Bernstein80] Bernstein P.A., Goodman N., « Timestamp-based Algorithm for Concurrency Control in Distributed Database Systems », *5th Intl. Conf. On Very Large data Bases*, Montreal, Oct. 1980.

Cet article introduit différents algorithmes d'estampillage de transactions dans le contexte de systèmes répartis.

[Bernstein87] Bernstein P.A., Hadzilacos V., Goodman N., *Concurrency Control and Recovery in Database Systems*, Addison-Weley, 1987.

Cet excellent livre de 370 pages fait le tour des problèmes de gestion de transactions dans les SGBD centralisés et répartis. Il présente notamment la théorie de base, le verrouillage, la certification, le multi-version, les protocoles de validation à deux et trois phases, la gestion de données répliquées.

[Bernstein90] Bernstein P.A., Hsu M., Mann B., « Implementing Recoverable Requests Using Queues », *ACM SIGMOD Int. Conf.*, ACM Ed., SIGMOD Record V. 19, n° 2, June 1990.

Cet article propose un protocole résistant aux pannes pour gérer les flots de requêtes de transactions entre clients et serveurs. Il discute une implémentation en utilisant des files d'attente persistantes récupérables après panne.

[Besancenot97] Besancenot J., Cart M., Ferrié J., Guerraoui R., Pucheral Ph., Traverson B., *Les Systèmes Transactionnels : Concepts, Normes et Produits*, Ed. Hermès, Paris, 1997.

Ce remarquable livre sur les systèmes transactionnels couvre tous les aspects du sujet : concepts de base, algorithmes de reprise, transactions réparties, duplication, modèles de transactions étendus, normes et standards, transactions dans les SGBD relationnels et objet.

[Bjork72] Bjork L.A., Davies C.T., « The Semantics of the Preservation and Recovery of Integrity in a Data System », *Technical Report TR-02.540,* IBM, Dec. 1972.

Un des tout premiers articles introduisant un concept proche de celui de transaction.

[Cart90] Cart M., Ferrié J., « Integrating Concurrency Control into an Object-Oriented Database System », *Proc. Intl. Conf. on Extending Database Technology*, EDBT, LCNS n° 416, p. 367-377, Venise, Mars 1990.

Cet article présente l'intégration d'algorithmes de contrôle de concurrence avec prise en compte de la commutativité des opérations dans un SGBD objet.

[Chamberlin75] Chamberlin D.D., Gray J.N., Traiger I.L., « Views, Authorizations and Locking in a Relational Data Base System », *Proc. of ACM National Computer Conf.,* p. 425-430, 1975

Cet article discute de l'utilisation des vues comme mécanisme d'autorisation et de verrouillage par prédicat dans un SGBD.

[Chamberlin78] Chamberlin D.D., Gray J.N., Griffiths P.P., Mresse M., Traiger I.L., Wade B.W., « Data Base System Authorization », in *Foundations of Secure Computation* [Demillo78], p. 39-56.

Cet article discute des mécanismes d'autorisation dans un SGBD relationnel.

[Chrysanthis90] Chrystandis P.K., Ramamritham K., « ACTA : A Framework for Specifying and reasoning about Transaction Structure and Behavior », *ACM SIGMOD Intl. Conf. on Management of Data, SIGMOD Record* vol. 19, n° 2, p. 194-203, Atlantic City, NJ, Juin 1990.

Cet article présente ACTA, un formalisme permettant de spécifier structure et comportement des transactions, en exprimant notamment la sémantique des interactions.

[Dayal91] Dayal U., Hsu M., Ladin R., « A Transactional Model for Long-Running Activities », Proc. of the 17th Intl. *Conf. on Very Large Data Bases*, Morgan Kaufmann Ed., p. 113-122, Bacelonne, Sept. 1991.

Cet article décrit un modèle d'activités composées récursivement de sous-activités et de transactions. Le modèle définit la sémantique des activités et décrit une implémentation en utilisant des files résistantes aux pannes pour chaîner les activités.

[Denning80] Denning D.E., Schlörer J., « A Fast Procedure for Finding a Tracker in a Statistical Database », *ACM Transactions on Database Systems*, vol. 5, n° 1, p. 88-102, Mars 1980.

Cet article présente une procédure rapide pour découvrir les caractéristiques d'un individu à partir de requêtes sur des ensembles d'individus.

[Demillo78] DeMillo R.A., Dobkin D.P., Jones A.K., Lipton R.J., *Foundations of Secure Computation*, Academic Press, 1978.

Ce livre présente un ensemble d'articles sur la sécurité dans les BD statistiques et les systèmes opératoires.

[Eswaran76] Eswaran K.P., Gray J.N., Lorie R.A., Traiger L.L., « The Notion of Consistency and Predicates Locks in a Database System », *Comm. of the ACM*, vol. 19, n° 11, p. 624-633, Nov. 1976.

Un des articles de base sur la notion de transaction et le principe du verrouillage deux phases. La notion de sérialisabilité et le verrouillage par prédicat sont aussi introduits par les auteurs qui réalisent alors la gestion de transactions dans le fameux système R.

[Fernandez80] Fernandez E.B., Summers R.C., Wood C., *Database Security and Integrity, The Systems Programming Series*, 1980.

Ce livre de 320 pages fait le tour des techniques d'integrité et de sécurité au sens large dans les bases de données.

[Garcia-Molina87] Garcia-Molina H., Salem K., « Sagas » *Proc. ACM SIMOD Intl. Conf. on Management of Data*, p. 249-259, San Fransisco, 1987.

Cet article présente le modèle des sagas introduit ci-dessus.

[Gardarin76] Gardarin G., Spaccapietra S., « Integrity of Databases : A General Locking Algorithm with Deadlock Detection », *IFIP Intl. Conf. on Modelling in DBMS*, Freudenstadt, January 1976.

Cet article présente un algorithme de verrouillage multi-mode et un algorithme de détection du verrou mortel dans ce contexte.

[Gardarin77] Gardarin G., Lebeux P., « Scheduling Algorithms for Avoiding Inconsistency in Large Data Bases », *3rd Intl. Conf. on Very Large Data Bases*, IEEE Ed., Tokyo, 1977.

Cet article introduit pour la première fois la commutativité des opérations comme solution pour tolérer plus d'exécution sérialisable. Il présente un algorithme de verrouillage à modes multiples prenant en compte les commutativités possibles.

[Gardarin78] Gardarin G., « Résolution des Conflits d'Accès simultanés à un Ensemble d'Informations – Applications aux Bases de Données Réparties », Thèse d'État, Paris VI, 1978.

Cette thèse introduit différents algorithmes de verrouillage et de détection du verrou mortel, dont certains avec prise en compte de la commutativité des opérations, d'autres basés sur l'ordonnancement total ou partiel par estampillage. Ces deux résultats étaient nouveaux à cette époque. La méthode basée sur la commutativité des opérations avait été publiée auparavant au VLDB 1977 [Gardarin77]. Les algorithmes d'estampillage ont été un peu plus tard plus clairement présentés avec M. Melkanoff dans le rapport INRIA n° 113 et en parallèle par P. Bernstein [Bernstein80].

[Gray78] Gray J.N., « Notes on Database Operating Systems », in *Operating Systems – An Advanced Course*, R. Bayer Ed., Springer-Verlag, 1978.

Cet article est un autre tutorial sur la gestion de transactions, basé sur la réalisation du gestionnaire de transaction du système R.

[Gray81] Gray J., « The Transaction Concept : Virtues and Limitations », *Proc. of the 7th Intl. Conf. on Very Large Data Bases*, IEEE Ed., p. 144-154, 1981.

Un autre tutorial sur la notion de transaction, discutant verrouillage, validation et procédures de reprise.

[Gray91] Gray J. Ed., *The Benchmark Handbook* , Morgan & Kaufman Pub., San Mateo, 1991.

Le livre de base sur les mesures de performances des SGBD. Composé de différents articles, il présente les principaux benchmarks de SGBD, en particulier le fameux benchmark TPC qui permet d'échantillonner les performances des SGBD en transactions par seconde. Les conditions exactes du benchmark définies par le « Transaction Processing Council » sont précisées. Les benchmarks de l'université du Madisson, AS3AP et Catell pour les bases de données objets, sont aussi présentés.

[Gray93] Gray J.N., Reuter A., *Transaction Processing : Concepts and Techniques*, Morgan Kaufman Ed., 1993.

Une bible de 1070 pages qui traite en détail tous les aspects des transactions.

[Griffiths76] Griffiths P.P., Wade B.W., « An Authorization Mechanism for a Relational Database System », *ACM Transactions on Database Systems*, vol. 1, n° 3, p. 242-255, Sept. 1996.

Cet article décrit le mécanismes des droits d'accès avec opérations GRANT et REVOKE pour la première fois. L'implémentation du système R réalisée par les auteurs fut la première.

[Holt72] Holt R.C., « Some Deadlock Properties of Computer Systems », *ACM Computing Surveys*, vol. 4, n° 3, p. 179-196, Sept. 1972.

Cet article discute des problèmes de deadlock et introduit notamment le graphe d'allocation des ressources avec des algorithmes de détection associés.

[Kaiser95] Kaiser G.E., « Cooperative Transactions for Multiuser Environments », *in Modern Database Systems*, Won Kim Ed., ACM Press, p. 409-433, 1995.

Cet article est un tutorial sur les transactions coopératives de longue durée. Il passe en revue le modèle des versions, split-join, de transactions imbriquées et de groupes.

[Korth90] Korth H.F., Levy E., Silberschatz A., « A Formal Aproach to Recovery by Compensating Transactions », Proc. of 16th *Intl. Conf. on Very Large Data Bases*, Morgan Kaufman Ed., p. 95-106, Brisbane, Australia, August 1990.

[Kung81] Kung H.T., Robinson J.T., « On Optimistic Method for Concurrency Control », *ACM Transaction on Database Systems* (TODS), vol. 6, n° 2, p. 213-226, Juin 1981.

Cet article présente deux familles d'algorithmes optimistes de contrôle de concurrence basés sur la certification.

[Lampson76] Lampson B., Sturgis H., « Crash Recovery in Distributed Data Storage System », *Xerox Technical Report*, Palo Alto, Xerox Research Center, 1976.

Cet article de base présente la première implémentation du protocole de validation en deux étapes. Lampson et Sturgis sont connus comme étant les inventeurs de ce fameux protocole.

[Lorie77] Lorie R.A., « Physical Integrity in a Large Segmented Database », ACM Transactions on Database Systems, vol. 2, n° 1, p.91-104, Mars 1977.

Cet article décrit le système de stockage du système R et sa gestion de transaction et reprise.

[Mohan86] Mohan C., Lindsay B., Obermark R., « Transaction Management in the R* Distributed Database Management System, *ACM Trans. On Database Syst.*, vol. 11, n° 4, Dec. 1986.

R est le prototype de SGBD réparti développé par IBM au début des années 80. Cet article présente la gestion de transactions distribuées, notamment les protocoles de validation à deux phases.*

[Mohan92] Mohan C., Haderle D., Lindsay B., Pirahesh H., Schwarz P., « ARIES : A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging », *ACM Transactions on Database Systems*, vol. 17, n° 1, p. 94-162, Mars 1992.

Cet article présente la méthode ARIES décrite ci-dessus.

[Moss82] Moss J.E.B., « Nested transactions and Reliable Distributed Computing », *2nd Symposium on Reliability in Distributed Software and Database Systems,* p. 33-39, IEEE Ed., Pittsburgh, 1982.

Cet article introduit les transactions imbriquées, inventées par Moss dans son PhD. Les transactions imbriquées, au départ fermées, ont été ouvertes (non-atomicité globale) et son implémentées dans des systèmes de plus en plus nombreux.

[Moss85] Moss J.E.B., *Nested Transactions : An Approach to Reliable Distributed Computing*, The MIT Press, Cambridge, Mass., 1985.

Ce livre, version améliorée du PhD de Moss, introduit et étudie en détail la théorie et la pratique des transactions imbriquées.

[Murphy68] Murphy J.E., « Ressource Allocation with Interlock Detection in a Multi-Task system », *Proc of AFIPS-FJCC Conf.*, vol. 33, n° 2, p. 1169-1176, 1968.

Cet article introduisit les graphes d'attente et l'un des premiers algorithmes de détection de deadlock.

[Özsu91] Özsu M.T., Valduriez P., *Principles of Distributed Database Systems,* Prentice Hall, Englewood Cliffs, New Jersey, 562 p., 1991.

Cet ouvrage est le livre de référence en anglais sur les bases de données réparties. Il couvre en particulier les aspects architecture, conception, contrôle sémantique, optimisation de requêtes, gestion de transactions, fiabilité et concurrence, bases de données fédérées. Chaque aspect est traité de manière très complète. Les algorithmes sont esquissés et une formalisation minimale est souvent introduite.

[Ozsu94] Ozsu T., « Transaction Models and Transaction Management in Object-Oriented Database Management Systems », in *Advances in Object-Oriented Database Systems*, p. 147-184, A. Dogac et. Al. Ed., NATO ASI Series, Springer Verlag, Computer and System Sciences, 1994.

Ce remarquable article est un autre tutorial sur la gestion de transactions. Il couvre particulièrement bien les modèles de transactions étendus et l'influence de la commutativité des opérations.

[Pu88] Pu C., Kaiser G.E., Hutchinson N., « Split-Transactions for Open-Ended Activities » 14th *Intl. Conf. on Very Large data Bases*, p. 26-37, Morgan Kaufman Ed., Los Angelès, 1998.

Cet article présente le modèle dynamique split et join évoqué ci-dessus pour les transactions imbriquées.

[Rabitti91] Rabitti F., Bertino E., Kim W., Woelk D., « A Model of Authorization for Next Generation Database Systems », *ACM Trans. On Database Systems*, vol. 16, n° 1, p. 88-131, Mars 1991.

Cet article développe un modèle formel de sécurité de type DAC pour un SGBD objet. Il s'appuie sur l'expérience conduite avec le SGBD ORION.

[Reed79] Reed D.P., « Implementing Atomic Actions », *Proc. of the 7th ACM SIGOPS Symposium on Operating Systems Principles*, ACM Ed., Dec. 1979.

L'auteur a proposé un algorithme d'ordonnancement multi-version qu'il utilise pour implémenter des actions atomiques que l'on appelle aujourd'hui des transactions.

[Rosenkrantz78] Rosenkrantz D., Stearns R., Lewis P., « System Level Concurrency Control for Distributed Database Systems », *ACM Transactions on Database Systems*, vol. 3, n° 2, p. 178-198, Juin 1978.

Cet article présente les techniques de contrôle de concurrence WAIT-DIE et WOUND-WAIT décrites ci-dessus.

[Rusinkiewicz95] Rusinkiewicz M., Sheth A., « Specification and Execution of Transactional Workflows », in *Modern Database Systems*, Won Kim Ed., ACM Press, p. 592-620, 1995.

Cet article discute les applications du concept de transaction aux activités de type workflow qui nécessitent la coordination de tâches multiples.

[Russell91] Russell D., Gangemi Sr. G.T., *Computer Security Basics*, O'Reilly & Associates, Inc., 1991.

Un livre de base sur les techniques de sécurité, couvrant notamment les contrôles d'accès, les virus, le livre orange de l'armée américaine (Orange Book) et la sécurité dans les réseaux.

[Skeen81] Skeen D., « Nonblocking Commit Protocols », *ACM SIGMOD Intl. Conf. on Management of Data*, Ann Arbor, p. 133-142, ACM Ed., Mai 1981.

Des protocoles non bloquants, en particulier le protocole en trois étapes décrit ci-dessus, sont introduits dans cet article.

[Stonebraker75] Stonebraker M., « Implémentation of Integrity Constraints and Views by Query Modification », *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San José, CA 1975.

Cet article propose de modifier les questions au niveau du source par la définition de vues pour répondre aux requêtes. La technique est formalisée avec le langage QUEL d'INGRES, où elle a été implémentée.

[Verhofstad78] Verhofstad J.S.M, « Recovery Techniques for database Systems, *ACM Computing Surveys*, vol. 10, n° 2, p. 167-195, Juin 1978.

Un état de l'art sur les techniques de reprise dans les bases de données en réseaux et relationnelles.

[Wachter92] Wachter H., Reuter A., « The Contract Model », *in Database Transaction Models for Advanced Applications*, Morgan Kaufman Pub., 1992.

Cet article introduit les contrats comme un ensemble de transactions dirigées par un script. Il fut l'un des précusseurs des workflows.

[Weikum91] G. Weikum, « Principles and Realization Strategies of Multi-Level Transaction Management », *ACM Transactions on Database Systems*, vol. 16, n° 1, 1991.

Cet article décrit un protocole de gestion de transactions imbriquées et distribuées, avec des sphères d'influence définies pour chaque transaction : sphère de validation des sous-transactions devant être validées, sphère d'annulation des sous-transaction devant être annulées, ceci en cas de validation ou d'annulation de la transaction maître.

[Weihl88] Weihl W.E., « Commutativity-based Concurrency Control for Abstract Data Types », *IEEE Transactions on Computers*, vol. 37, n° 12, p. 1488-1505, 1988.

Cet article propose des méthodes et algorithmes de contrôle de concurrence prenant en compte la commutativité des opérations pour gérer des types abstraits de donnés.

Chapitre XVII

CONCEPTION DES BASES DE DONNÉES

1. INTRODUCTION

Une des tâches essentielles des développeurs de bases de données est la conception du schéma des bases. L'objectif est de structurer le domaine d'application de sorte à le représenter sous forme de types et de tables. La représentation doit être juste pour éviter les erreurs sémantiques, notamment dans les réponses aux requêtes. Elle doit aussi être complète pour permettre le développement des programmes d'application souhaités. Elle doit enfin être évolutive afin de supporter la prise en compte rapide de nouvelles demandes.

Le concepteur, ou plutôt l'administrateur de base, effectue également le choix du placement des tables sur disques et le choix des index, choix essentiels pour les performances. En exagérant un peu, on peut dire qu'il n'y a pas de mauvais SGBD, mais de mauvais concepteurs responsables des erreurs sémantiques ou des mauvaises performances. Les choix de structures physiques sont dépendants des programmes qui manipulent la base, particulièrement des types et fréquences des requêtes d'interrogation et de mise à jour.

Traditionnellement, la démarche de conception s'effectue par abstractions successives, en descendant depuis les problèmes de l'utilisateur vers le SGBD. Nous proposons de distinguer cinq étapes :

1. **Perception du monde réel et capture des besoins**. Cette étape consiste à étudier les problèmes des utilisateurs et à comprendre leurs besoins. Elle comporte des entretiens, des analyses des flux d'information et des processus métier. Des démarches de type BPR (*Business Process Reengineering*) [Hammer93] de reconception des processus métiers existants en les dirigeant vers le client peuvent être un support pour cette étape. La génération de modèles de problèmes est aussi une technique courante à ce niveau [DeAntonellis83]. Comme il est difficile de comprendre le problème dans son ensemble, les concepteurs réalisent des études de cas partiels. Le résultat se compose donc d'un ensemble de vues ou schémas externes qu'il faut intégrer dans l'étape suivante. Ces vues sont exprimées dans un modèle de type entité-association ou objet, selon la méthode choisie.
2. **Élaboration du schéma conceptuel**. Cette étape est basée sur l'intégration des schémas externes obtenus à l'étape précédente. Chaque composant est un schéma entité-association ou objet. Il résulte d'un modèle de problème représentant une partie de l'application. La difficulté est d'intégrer toutes les parties dans un schéma conceptuel global complet, non redondant et cohérent. Des allers et retours avec l'étape précédente sont souvent nécessaires.
3. **Conception du schéma logique**. Cette étape réalise la transformation du schéma conceptuel en structures de données supportées par le système choisi. Avec un SGBD relationnel, il s'agit de passer à des tables. Avec un SGBD objet-relationnel, il est possible de générer des types et des tables, les types étant réutilisables. Avec un SGBD objet, il s'agit de générer des classes et des associations. Cette étape peut être complètement automatisée, comme nous le verrons.
4. **Affinement du schéma logique**. Une question qui se pose est de savoir si le schéma logique obtenu est un « bon » schéma. À titre de première approximation, un « bon schéma » est un schéma sans oublis ni redondances d'informations. Pour caractériser plus précisément les « bons » schémas, le modèle relationnel s'appuie sur la théorie de la normalisation, qui peut être avantageusement appliquée à ce niveau. En relationnel, l'objectif est de grouper ou décomposer les tables de manière à représenter fidèlement le monde réel modélisé.
5. **Élaboration du schéma physique**. Cette étape est nécessaire pour obtenir de bonnes performances. Elle nécessite la prise en compte des transactions afin de déterminer les patterns d'accès fréquents. À partir de là, il faut choisir les bonnes structures physiques : groupage ou partitionnement de tables, index, etc. C'est là que se jouent pour une bonne part les performances des applications.

Dans ce chapitre, nous étudions essentiellement les étapes 2, 3 et 4 qui font largement partie du domaine des bases de données. La partie 1 appartient plutôt au génie logiciel, voire à l'économie ou la psychologie. Nous ne l'aborderons guère. Nous détaillons surtout la partie 3 où toute une théorie s'est développée à la fin des années 70 et au début des années 80 pour les bases relationnelles.

Dans la section qui suit, nous abordons le problème de la conception du schéma conceptuel. C'est l'occasion de présenter le langage de modélisation UML, plus préci-

sément les constructions nécessaires à la modélisation de BD. Nous discutons aussi des techniques d'intégration de schémas. La section 3 développe les règles pour passer d'un schéma conceptuel UML à un schéma relationnel. Elle propose aussi quelques pistes pour passer à l'objet-relationnel. La section 4 présente les approches pour l'affinement du schéma logique. La théorie de la normalisation, qui peut être intégrée au cœur de l'affinement, est l'objet des trois sections qui suivent. La section 5 discute les principales techniques d'optimisation du schéma physique. Nous concluons en résumant et discutant les voies d'évolution.

2. ÉLABORATION DU SCHÉMA CONCEPTUEL

Dans cette section, nous traitons des techniques permettant de définir un schéma conceptuel. Nous procédons par modélisation entité-association ou objet en construisant des diagrammes basés sur UML, le langage de modélisation unifié standardisé par l'OMG.

2.1. PERCEPTION DU MONDE RÉEL AVEC E/R

Le monde des applications informatiques peut être modélisé à l'aide d'entités qui représentent les objets ayant une existence visible, et d'associations entre ces objets [Benci76, Chen76]. Le modèle entité-association (*Entity Relationship, E/R*) a eu un très grand succès pour représenter des schémas externes d'un domaine de discours particulier, autrement dit des parties d'une application. Comme nous l'avons vu au chapitre II, ce modèle repose sur des **entités** encore appelées individus, des **associations** ou relations entre entités, et des **attributs** ou propriétés. Il est à la base de Merise [Tardieu83] et de nombreuses autres méthodes. Une entité modélise un objet intéressant perçu dans le réel analysé, ayant une existence propre. Un attribut est une information élémentaire qui caractérise une entité ou une association et dont la valeur dépend de l'entité ou de l'association considérée. Une association est un lien sémantique entre deux entités ou plus. Définir une vue du réel analysé par le modèle entité-association nécessite d'isoler les types d'entités, d'associations et d'attributs.

Différents **diagrammes** ont été introduits pour représenter les schémas entité-association. Au chapitre II, nous avons introduit les notations originelles de Chen. Dans la suite, nous utilisons la représentation proposée dans le langage universel de modélisation UML [Rational98]. En effet, ce langage devient le standard pour la conception dans les entreprises. Poussé par l'OMG, il a le mérite d'être complet, clair et sans

doute résistant à l'usure du temps, comme tous les standards. La construction de base dérivée du langage UML pour représenter des entités est symbolisée figure XVII.1. À titre d'exemple, nous avons représenté des voitures avec les attributs numéro de véhicule (NV), marque, type, puissance et couleur.

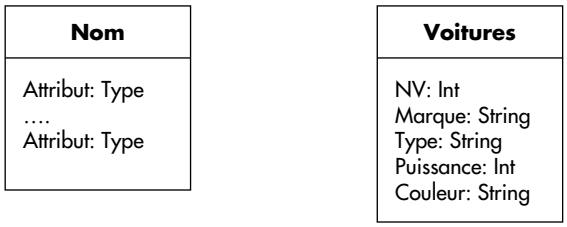


Figure XVII.1 : Représentation d'entités

La construction de base permettant de représenter des **associations** est symbolisée figure XVII.2. Nous représentons là une association binaire générique avec attributs. L'association entre les entités Entité1 et Entité2 est représentée par un trait simple. Le nom de l'association apparaît au-dessus du trait. Les attributs sont représentés par une entité sans nom accrochée par un trait en pointillé à l'association. Si les données participent elles-mêmes à des associations, il est possible de leur donner un nom : on a alors une véritable entité associative possédant une valeur pour chaque instance de l'association binaire. La représentation d'associations ternaires est possible avec UML : on utilise alors un losange où convergent les traits associatifs et de données. Cependant, nous déconseillons l'emploi de telles associations difficiles à lire : les associations n-aires peuvent toujours se représenter par une classe associative en ajoutant une contrainte qui exprime que les associations sont instanciées ensemble. Par exemple, une vente associe simultanément un client, un produit et un vendeur. Dans la suite, nous considérons souvent des associations binaires, plus faciles à manipuler et à comprendre.

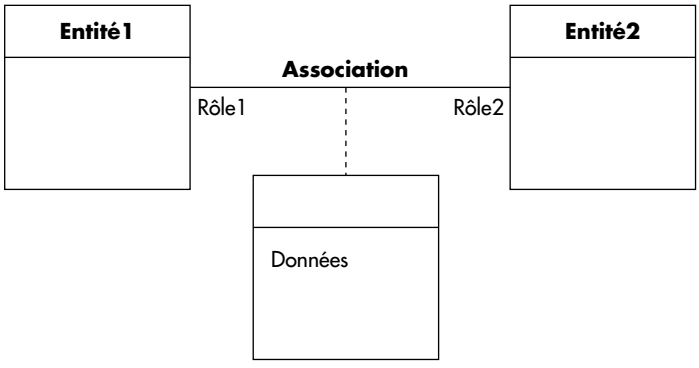


Figure XVII.2 : Représentation d'associations

Dans une association, chaque entité participante joue un **rôle**. Celui-ci peut être explicitement nommé, comme indiqué figure XVII.2. Mais ceci n'est pas obligatoire. Les associations sont caractérisées par des **cardinalités** : la cardinalité [m,n] attachée à une entité indique les nombres minimal et maximal d'instance d'associations pour une instance de cette entité.

Notion XVII.1 : Cardinalités d'association (*Relationship cardinalities*)

Cardinalités minimale et maximale associées à chacun des rôles de l'association, indiquant le nombre minimal et maximal d'instances d'association auxquelles participe une instance de l'entité du rôle.

Une cardinalité se lit donc dans le sens entité vers association. Il faut se demander pour une instance d'entité (ou de classe) combien d'instances d'association lui sont attachées ? Avec des associations binaires, cela revient à indiquer le nombre d'instances de l'autre entité pour une instance de celle à laquelle est attachée la cardinalité. UML propose les notations indiquées figure XVII.3 pour les cardinalités. Notez que 1 signifie à la fois un minimum de 1 et un maximum de 1. La notation {ord} signifie que l'ordre d'apparition des entités dans l'association est important.

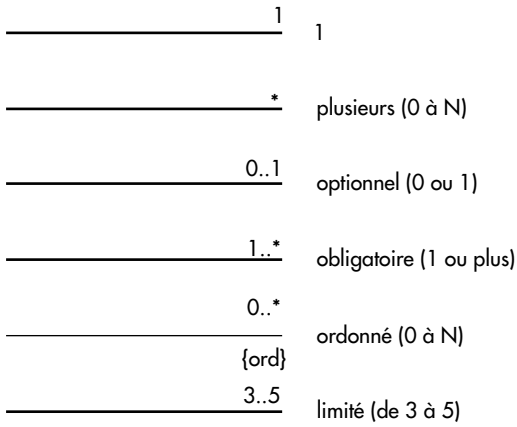


Figure XVII.3 : Notation des cardinalités d'associations

À titre d'exemple, soient une base modélisant des entités « personne » et « voiture », et le type d'association « possède » qui traduit le fait qu'une personne est propriétaire d'une ou plusieurs voitures. Une personne est caractérisée par un numéro de Sécurité Sociale (NSS), un nom, un prénom et une date de naissance alors qu'une voiture est caractérisée par les attributs déjà vus NV, MARQUE, TYPE, PUISSANCE et COULEUR. Chaque personne est identifiée par une occurrence du numéro de Sécurité Sociale (NSS), alors que chaque voiture est identifiée par un numéro de véhicule

(NV). À chaque occurrence d'association correspond par exemple une date d'achat (DATE) et un prix d'achat (PRIX). La figure XVII.4 représente le schéma externe correspondant décrit avec les notations UML réduites aux entités, associations et attributs. Les cardinalités indiquent qu'une personne peut posséder de O à N voitures alors qu'une voiture est possédée par une et une seule personne.

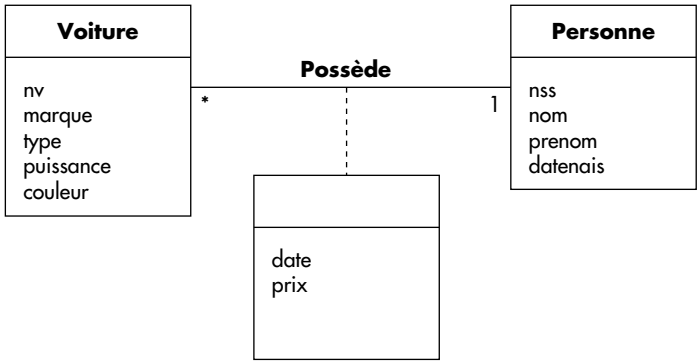


Figure XVII.4 : Exemple d'association entre voiture et personne

Nous présentons figure XVII.5 l'exemple classique des buveurs, des vins et de l'association boire caractérisée par une date et une quantité. Les cardinalités sont donc portées par les rôles : le rôle Estbu porte la cardinalité 1..*, ce qui signifie qu'à un buveur est associé entre 1 et N abus ou vins si l'on préfère. * est une notation raccourcie pour 0..*. Le rôle Estbu porte cette cardinalité, ce qui signifie qu'un vin est bu par 0 à N buveurs. Tout cela, aux notations près, est bien connu mais souvent confus, les cardinalités étant interprétées de différentes manières selon les auteurs. Nous avons choisi ces notations pour assurer la compatibilité avec l'approche objet et UML que nous allons maintenant développer un peu plus.

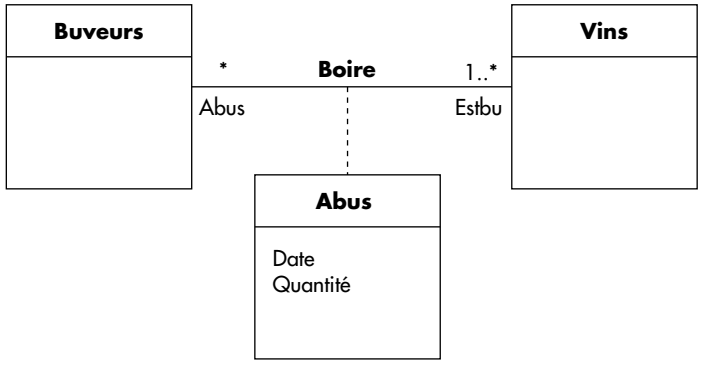


Figure XVII.5 : Représentation de l'association entre buveur et vin

2.2. PERCEPTION DU MONDE RÉEL AVEC UML

UML (*Universal Modelling Language*) est donc le langage qui se veut universel pour la modélisation objet. Nous l'avons déjà souvent approximativement utilisé pour représenter des objets. UML a été développé en réponse à l'appel à proposition lancé par l'OMG (*Object Management Group*). Il existe de nombreux ouvrages sur UML et nous nous contenterons des constructions utiles pour modéliser les bases de données. Le lecteur désirant en savoir plus pourra se reporter à [Bouzeghoub97], [Muller98] ou encore [Kettani98]. UML présente beaucoup d'autres diagrammes que ceux utilisés, en particulier pour représenter les cas d'utilisation, les séquences, les transitions d'états, les activités, les composants, etc. Nous utilisons essentiellement les diagrammes de classe, d'association, d'héritage et d'agrégation.

Une **classe** est une extension du concept d'entité avec des opérations, comme le montre la figure XVII.6. Nous donnons en exemple la classe Voiture avec les opérations Démarrer(), Accélérer(), Rouler() et Freiner(). UML distingue aussi les attributs privés précédés de – et publics notés +. D'autres niveaux de visibilité sont possibles. Pour l'instant, et par défaut, nous supposons tous les attributs publics. Cela n'est pas très conforme à l'objet, mais les spécialistes des bases de données sont des briseurs d'encapsulation bien connus, car ils s'intéressent avant tout aux données !

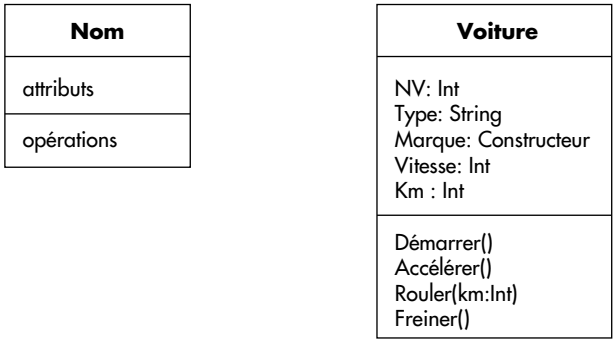


Figure XVII.6 : Représentation d'une classe en UML

La découverte des classes, comme celle des entités, nécessite d'isoler les types d'objets du monde réel qui ont un cycle de vie propre. Dans une description en langage naturel, les classes comme les entités correspondent souvent à des noms. À partir des objets, il faut abstraire pour découvrir les propriétés, attributs et méthodes. Une réflexion sur le cycle de vie de l'objet et sur ses collaborations avec les autres objets permet de préciser les méthodes, et par là les attributs manipulés par ces méthodes. UML fournit des outils pour représenter cycle de vie et collaboration : ce sont les diagrammes d'état et de collaboration, dont l'étude dépasse le cadre de cet ouvrage.

La découverte des classes conduit à découvrir les liens de **généralisation** et de **spécialisation** entre classes. Dans une description en langage naturel, les objets sont alors

reliés par le verbe `être` (relation *is a*). UML permet la représentation de l'héritage comme indiqué figure XVII.7. S'il est possible de grouper les deux flèches en une seule, cela n'a pas de signification particulière. Si les deux sous-classes sont disjointes, une contrainte {Exclusive} peut être explicitement notée. De même, il est possible de préciser {Inclusive} si tout objet se retrouve dans toutes les sous-classes. Un nom discriminant peut être ajouté sur l'arc de spécialisation, pour distinguer différentes spécialisations d'une classe.

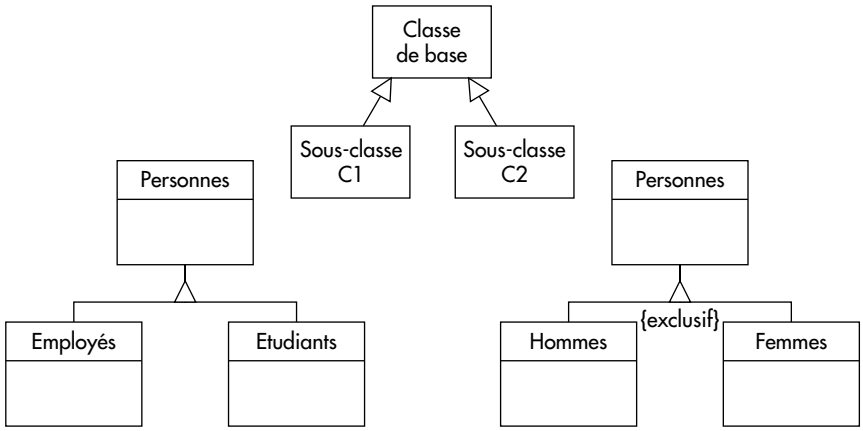


Figure XVII.7 : Représentation et exemples de spécialisation en UML

L'**agrégation** est utilisée pour représenter les situations où une classe est composée d'un ou plusieurs composants. UML permet de distinguer l'**agrégation indépendante** de l'**agrégation composite.** La première est une association particulière qui relie un objet à un ou plusieurs objets composants ; les deux classes sont deux classes autonomes. La seconde permet de représenter des objets composites résultant de l'agrégation de valeurs. Elle se distingue de la première par un losange plein. Les diagrammes représentant ces deux types d'associations sont symbolisés figure XVII.8.

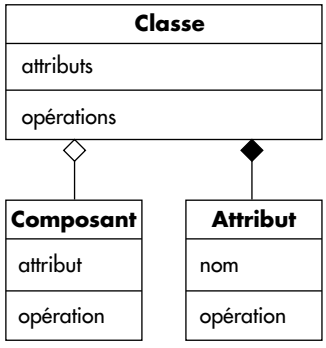


Figure XVII.8 : Représentation d'agrégations en UML

Les figures XVII.9 et XVII.10 illustrent les constructions introduites par deux études de cas conduisant à l'élaboration de schémas externes ou vues, ou encore **paquetages** (un paquetage UML est un ensemble de composants objets qui peut comporter beaucoup d'autres éléments). Chaque paquetage est représenté par un rectangle étiqueté contenant ses composants. UML permet d'ajouter des notes à tous les niveaux. Pour les paquetages, nous définissons dans la note associée la situation correspondante en français.

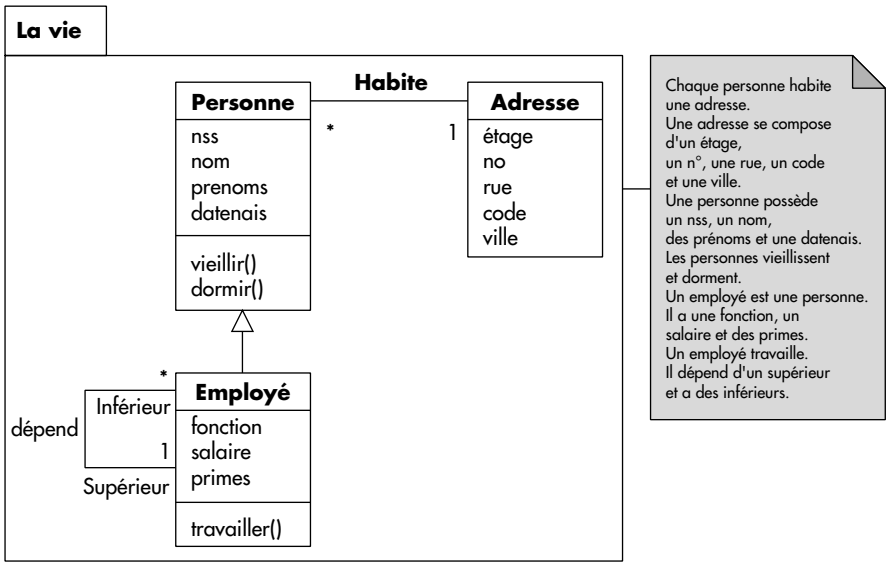


Figure XVII.9 : Exemple de schéma externe en UML

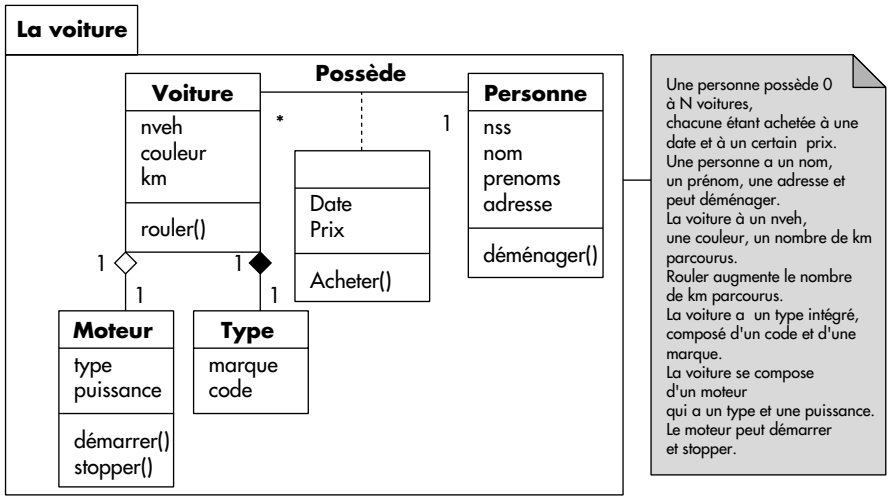


Figure XVII.10 : Un autre exemple de schéma externe en UML

2.3. INTÉGRATION DE SCHÉMAS EXTERNES

Les schémas externes ou vues ou encore paquetages permettent donc de représenter des sous-ensembles du schéma conceptuel de la base de données. Construire le schéma externe nécessite d'intégrer ces différentes parties, ce qui n'est pas une tâche simple. Les difficultés proviennent des recouvrements et des liens sémantiques entre parties. De nombreux travaux ont été effectués pour intégrer des schémas objet, non seulement dans le domaine de la conception [Batini86, Navathe84], mais aussi dans celui des bases de données objet fédérées [WonKim95].

Les conflits peuvent concerner les noms des classes et associations (synonymes et homonymes), les structures (attributs manquants, associations regroupées), les définitions d'attributs (conflits, inclusion), les contraintes (cardinalités), etc. La figure XVII.11 propose une liste plus ou moins exhaustive des conflits possibles [WonKim95].

Classe ↔ Classe
- Noms différents pour des classes équivalentes
- Noms identiques pour des classes différentes
- Inclusion de l'une dans l'autre
- Intersection non vide
- Contraintes entre instances

Attribut ↔ Attribut
- Noms différents pour des attributs équivalents
- Noms identiques pour des attributs différents
- Types différents
- Compositions différentes

Classe ↔ Attribut
- Significations identiques
- Compositions similaires

Association ↔ Association
- Noms différents pour des associations équivalentes
- Noms identiques pour des associations différentes
- Cardinalités différentes
- Données différentes
- Compositions de plusieurs autres

Agrégation ↔ Agrégation
- Agrégation composite versus non composite
- Cardinalités différentes

Association ↔ Attribut
- Association versus référence
- Cardinalités incompatibles

Figure XVII.11 : Quelques types de conflits entre schémas

Le premier problème est d'isoler les conflits. Cela nécessite le passage par un dictionnaire unique des noms, voire par une **ontologie**. Une ontologie est une définition complète des concepts, avec leurs relations sémantiques. L'utilisation d'une ontologie spécifique au domaine permet de ramener les concepts à un référentiel unique et de mesurer la distance et le recouvrement entre eux [Métais97]. On peut ainsi isoler les conflits potentiels.

Pour chaque cas, des solutions doivent être envisagées, telles que :

– Changement de dénomination de classes, d'associations ou d'attributs
– Ajout d'attributs ou remplacement par des opérations
– Définition de classes plus générales ou plus spécifiques
– Transformation d'agrégation en objets composites et vice versa
– Redéfinition de types plus généraux
– Transformation de représentation
– Conversion et changement d'unités.

Certains conflits ne sont solubles que manuellement. Un outil graphique d'aide à l'intégration peut être avantageusement utilisé.

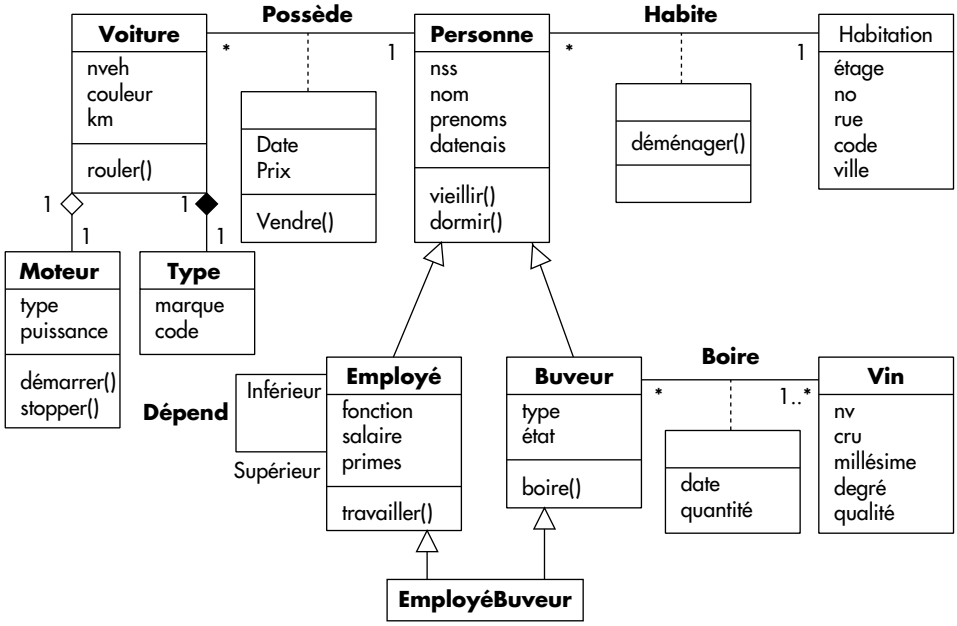


Figure XVII.12 : Exemple de schéma intégré UML

En résumé, après des transformations de schémas automatiques ou manuelles, les schémas externes peuvent être intégrés afin d'obtenir un schéma global cohérent avec

un minimum de redondance. À titre d'illustration, la figure XVII.12 (ci-avant) propose un schéma intégré résultant de l'intégration des vues « La vie » et « La voiture » avec nos éternels buveurs (« La fête »). Vous pouvez repérer les transformations effectuées, qui sont assez réduites.

3. CONCEPTION DU SCHÉMA LOGIQUE

Cette section explique comment obtenir un schéma relationnel ou objet-relationnel à partir d'un schéma objet représenté en UML. Nous proposons deux méthodes, la première pour passer de l'objet (et donc d'entité-association qui en est un cas particulier) au relationnel appelée UML/R, la seconde pour passer de l'objet à l'objet-relationnel, appelée UML/OR ou UML/RO selon que la dominance est donnée à l'objet ou au relationnel.

3.1. PASSAGE AU RELATIONNEL : LA MÉTHODE UML/R

3.1.1. Cas des entités et associations

Examinons tout d'abord comment **traduire des entités et associations** simples. Le modèle relationnel se prête bien à la représentation des entités et des associations. Les règles sont les suivantes :

R1. Une **entité** est représentée par une relation (table) de même nom ayant pour attributs la liste des attributs de l'entité.

R2. Une **association** est représentée par une relation de même nom ayant pour attributs la liste des clés des entités participantes et les attributs propres de l'association.

Pour appliquer la règle 2, chaque table résultant de la règle 1 doit posséder une clé primaire, c'est-à-dire un groupe d'attributs (1, 2 ou 3 au plus sont conseillés) qui détermine à tout instant un tuple unique dans la table. S'il n'en est pas ainsi, il faut ajouter une clé qui est un numéro de tuple (une séquence attribuée par le système en SQL2). Les tables résultant des associations ont pour clés la liste des clés des entités participantes avec éventuellement, en cas d'association multivaluée, une ou plusieurs données propres de l'association.

Par exemple, les entités PERSONNE et VOITURE de la figure XVII.4 seront respectivement représentées par les relations :

```
PERSONNE (NSS, NOM, PRENOM, DATENAIS)
VOITURE (NV, MARQUE, TYPE, PUISSANCE, COULEUR).
```

Les clés sont respectivement NSS et NV. En conséquence, l'association POSSÈDE sera représentée par la relation :

```
POSSEDE (NSS, NV, DATE, PRIX).
```

Les transformations proposées donnent autant de relations que d'entités et d'associations. Il est possible de regrouper certaines relations et associations dans les cas particuliers où un tuple d'une table référence un et un seul tuple de l'association. Une telle association est dite bijective avec l'entité : en effet, tout tuple de la table correspond à un tuple de l'association et réciproquement. La règle est la suivante :

R3. Une **association bijective**, c'est-à-dire de cardinalités minimale et maximale 1, peut être regroupée en une seule table avec la relation attachée par jointure sur la clé de l'entité.

Cette règle est illustrée figure XVII.13. Dans le cas où l'association est 1..1 des deux côtés, la règle peut être appliquée à droite ou à gauche, et si les deux entités ne sont reliées à aucune autre association, elles peuvent même être regroupées en une seule table.

Pour implémenter un modèle objet, il n'est pas nécessaire d 'avoir une BD objet. Au-delà des entités et associations, tous les concepts d'un modèle objet peuvent être implémentés avec un SGBD relationnel. Alors que les tables mémorisent l'état des objets, les méthodes apparaissent comme des attributs calculés. Elles seront généralement implémentées sous forme de procédures stockées. Nous allons maintenant examiner le passage d'un modèle UML au relationnel, sachant que ce que nous avons dit pour les associations restent vrai dans le contexte objet. La plupart des règles décrites ci-dessous ont été implémentées dans le système expert SECSI d'aide à la conception de bases de données [Bouzeghoub85].

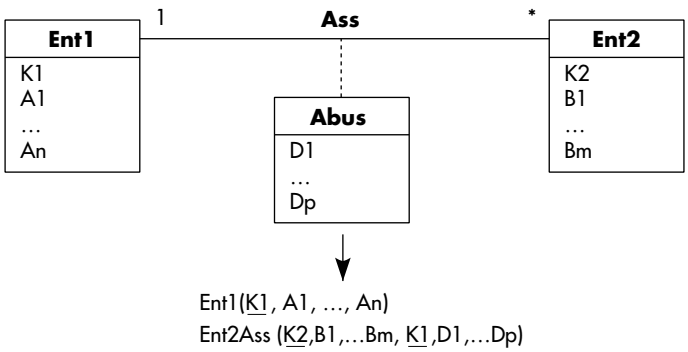


Figure XVII.13 : Translation d'association 1..1

3.1.2. Cas des généralisations avec héritage

Les systèmes relationnels ne connaissent pas les concepts de généralisation et d'héritage. Il faut donc réaliser statiquement ce dernier lors de la transformation du schéma. Pour les données, cela ne pose pas trop de problèmes et plusieurs solutions sont pos-

sibles, consistant toutes à aplatir les hiérarchies de spécialisation. Pour les méthodes, le polymorphisme doit être réalisé dans le corps de la méthode par des tests (CASE). Nous examinons ici la transformation des données, comme il se doit pour une base de données.

La solution la plus naturelle pour traduire une hiérarchie de généralisations de classes C1, C2…Cn vers une classe C est d'appliquer la règle suivante, en plus de la règle R1 qui conduit à traduire chaque classe (ou entité) comme une table avec une clé primaire, la règle suivante :

R4.a Une **spécialisation** d'une classe C en plusieurs classes C1, C2…Cn est traduite par répétition de la clé de la table représentant C au niveau de chacune des tables représentant C1, C2…Cn.

Cette solution conduit à une table par classe (voir figure XVII.14(a)). Lorsqu'on souhaite retrouver un attribut hérité dans une classe dérivée, il faut effectuer une jointure avec la table représentant la classe de base. L'héritage doit donc être accompli par les programmes d'application. La définition d'une vue jointure des tables dérivées et de la table de base (par exemple C|X|C1) permet d'automatiser l'héritage.

D'autres solutions sont possibles pour traduire des spécialisations, comme le montre la figure XVII.14(b) et (c). La solution (b) consiste à faire une table par classe feuille de la hiérarchie en appliquant la règle suivante :

R4.b Une **spécialisation** d'une classe C en plusieurs classes C1, C2…Cn est traduite par répétition des attributs représentant C au niveau de chacune des tables représentant C1, C2…Cn et par transformation de C en une vue dérivée de C1, C2…Cn par union des projections sur les attributs de C.

Le problème avec cette solution survient lorsque les classes C1, C2, Cn ne sont pas exclusives et contiennent des objets communs. Les attributs de C sont alors répétés dans chacune des tables C1, C2…Cn, ce qui pose des problèmes de cohérence. Cette règle sera donc seulement appliquée dans le cas d'héritage exclusif. Par exemple, il est intéressant de représenter la hiérarchie d'héritage `FEMMES, HOMMES` ➔ `PERSONNES` de la figure XVII.7 par les tables `FEMMES` et `HOMMES`, la table `PERSONNES` pouvant être dérivée par une vue. Au contraire, la même technique utilisée pour la hiérarchie d'héritage `EMPLOYES, ETUDIANTS` ➔ `PERSONNES` conduirait à dupliquer les employés étudiants. On préférera alors appliquer la règle R4.a conduisant à trois tables `EMPLOYES`, `ETUDIANTS` et `PERSONNES`, chacune ayant pour clé le numéro de sécurité sociale de la personne.

Une autre solution encore possible consiste à implémenter une seule table comme illustré figure XVII.14(c) :

R4.c Une **spécialisation** d'une classe C en plusieurs classes C1, C2…Cn est traduite par une table unique comportant la traduction de la classe C complétée avec les attributs de C1, C2…Cn, les tables correspondant à C1, C2…Cn étant des vues dérivées de C par projection sur les attributs pertinents.

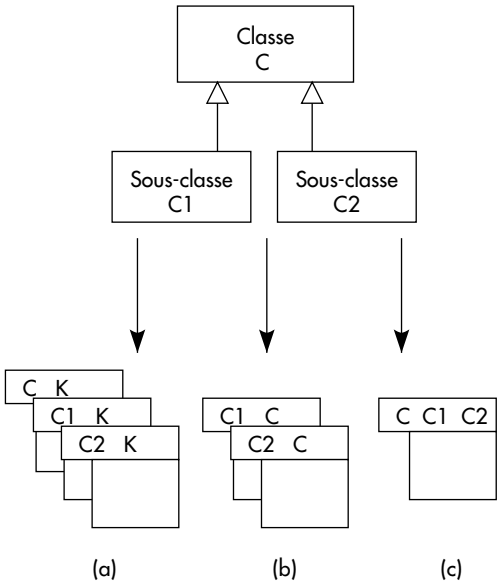


Figure XVII.14 : Traduction de hiérarchie d'héritage en relationnel

Le problème avec cette solution survient lorsqu'un objet de la classe C n'a pas de spécialisation dans une sous classe Ci : dans ce cas, tous les attributs de la classe Ci apparaissent comme des valeurs nulles ! Par exemple, pour la hiérarchie `FEMMES, HOMMES` ➔ `PERSONNES`, les attributs spécifiques aux femmes seront nuls pour chaque homme dans la table `PERSONNES` générale (par exemple, le nom de jeune fille). Pour la hiérarchie `EMPLOYES, ETUDIANTS` ➔ `PERSONNES`, tout employé non étudiant aura les attributs d'étudiants nuls et tout étudiant non employé aura les attributs spécifiques aux étudiants nuls. Cette solution n'est donc bonne que dans le cas d'héritage complet, où chaque objet de la classe de base est membre de la sous-classe.

En résumé, les différents cas sont illustrés figure XVII.14. Bien sûr, ils peuvent être mixés et il faut réfléchir pour chaque sous-classe. Certaines peuvent par exemple être regroupées avec la classe de base, d'autres implémentées de manière autonome.

3.1.3. Cas des agrégations et collections

Comme nous l'avons vu ci-dessus, UML propose deux cas d'agrégations : les agrégations indépendantes et les agrégations composites.

L'**agrégation indépendante** n'est rien de plus qu'un cas particulier d'association : elle sera donc traduite en appliquant les règles des associations. Un problème peut être que les agrégations n'ont pas de nom en général : il faut en générer un, par exemple

par concaténation des noms des entités participantes. On ajoutera un nom de rôle si plusieurs agrégations indépendantes relient deux classes.

L'**agrégation composite** correspond à un groupe d'attributs (et méthodes) imbriqués dans l'objet composite. Dans le cas de bijection (cardinalité 1..1), tous les attributs de la classe cible (le composant) sont simplement ajoutés à la table représentant la classe source (le composé). La classe cible est représentée par une vue. La règle est la suivante :

R5. Une classe dépendant d'une autre par une agrégation **composite monovaluée** est représentée par des attributs ajoutés à la table représentant l'objet composite et si nécessaire transformée en une vue, sinon omise.

Cette règle est appliquée figure XVII.15. Au-delà, le relationnel pur ne permet pas de traduire les agrégations composites multivaluées par une seule table. On procède alors comme avec une agrégation indépendante et plus généralement une association. Des contraintes d'intégrité additionnelles peuvent être ajoutées, comme nous le verrons ci-dessous.

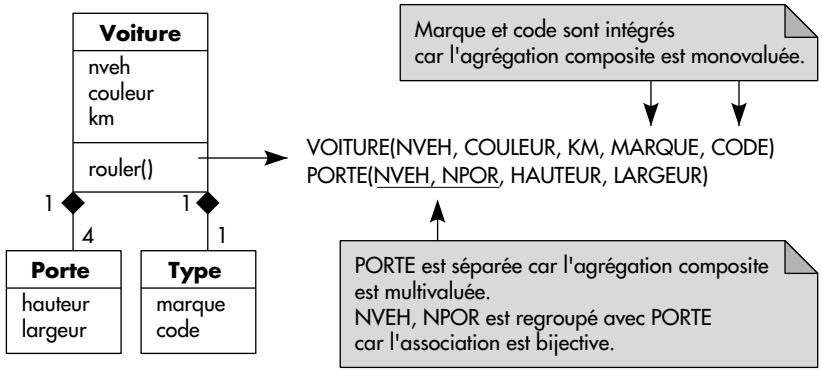


Figure XVII.15 : Traduction d'agrégations composites

Les **attributs multivalués** peuvent être intégrés directement dans une classe par le biais de collections SET<X>, LIST<X>, BAG<X>, etc. Une bonne modélisation UML traduit de tels attributs en agrégations composites. Cependant, il est permis d'utiliser des *templates* ; alors le problème de la traduction en relationnel se pose.

Deux cas sont à considérer pour traduire en relationnel. Si le nombre maximal N de valeurs est connu et faible (< 5 par exemple), il est possible de déclarer N colonnes, de nom A1, A2,, etc., où A est le nom de l'attribut. Cette solution manque cependant de flexibilité et conduit à des valeurs nulles dès qu'il y a moins de N valeurs. Une solution plus générale consiste à isoler la clé K de la table résultant de la classe ayant un attribut collection, et à créer une table répertoriant les valeurs de la collection associée à la clé. La table a donc pour schéma AS (K, A) et donne les valeurs de A pour chaque valeur de K. Par exemple, si une collection a trois valeurs v1, v2, v3 pour la clé 100, (100-v1),

(100-v2) et (100-v3) seront trois tuples de la table AS. Les collections seront reconstituées par des jointures lors des interrogations. Cette solution est connue sous le nom de **passage en première forme normale** et nous y reviendrons ci-dessous.

3.1.4. Génération de contraintes d'intégrité

Au-delà des tables, le passage d'un modèle objet exprimé en UML au relationnel permet de générer des contraintes d'intégrité référentielles. Les associations sont particulièrement utiles pour cela. Voici deux règles applicables :

R6. Toute **association E1→R→E2** représentée par une table R non intégrée à E1 ou E2 donne naissance à deux contraintes référentielles : R.K(E1) référence E1 et R.K(E2) référence E2, K(Ei) désignant la clé de Ei.

R7. Toute **association E1→R→E2 de cardinalité minimale 1** sur E2 représentée par une table non intégrée à E1 donne naissance à une contrainte référentielle additionnelle : E1.K(E1) référence R.K(E1).

Ces règles sont illustrées figure XVII.16 sur l'association `Boire` entre `Buveurs` et `Vins`. La contrainte de `Buveurs` vers `Abus` résulte du fait que la cardinalité minimale de 1 signifie que pour un objet quelconque, il existe au moins une instance d'association (règle 7). Les contraintes référentielles de la table associative `Abus` vers `Buveurs` et `Vins` proviennent du fait que l'association implique l'existence des objets associés (règle 6). En théorie, les associations sont donc très contraignantes pour le modèle relationnel sous-jacent. On omet parfois les contraintes résultant de la règle 6, ce qui signifie qu'on tolère des associations entre objets non existants.

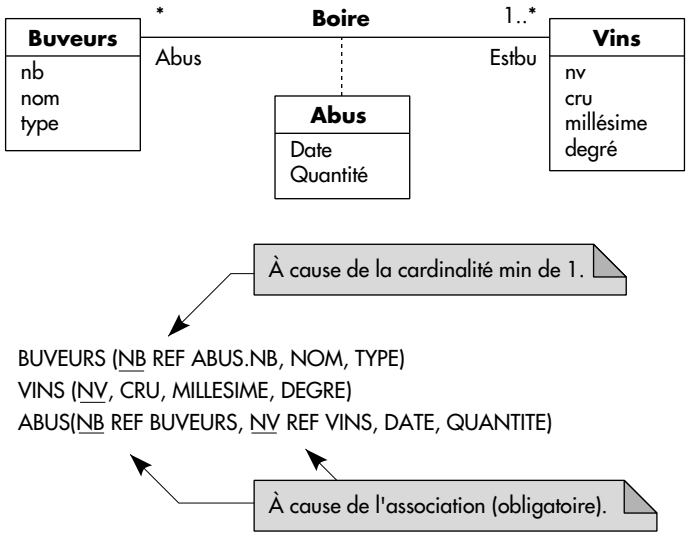


Figure XVII.16 : Génération de contraintes référentielles

3.2. PASSAGE À L'OBJET-RELATIONNEL : UML/RO OU UML/OR?

Les SGBD deviennent objet-relationnel avec SQL3, comme nous l'avons montré au chapitre XIII. Passer d'un modèle UML à un modèle objet-relationnel est à la fois plus simple et plus compliqué. C'est plus simple en théorie car l'objet-relationnel supporte directement les associations, l'héritage et les collections. C'est plus complexe car il y a maintenant des types et des tables, et tous les types possibles en objet ne sont pas possibles en objet-relationnel. De plus, la représentation relationnelle reste possible. Alors que faire ? Deux approches au moins sont possibles : l'une étendant simplement l'approche relationnelle avec des types utilisateurs, l'autre beaucoup plus proche de l'objet.

L**'approche relationnelle étendue** (notée **UML/RO**, le R venant en premier) consiste à faire une première passe sur le schéma objet afin d'isoler les types intéressants. Ceux-ci peuvent être des groupes d'attributs qui apparaissent de manière répétitive dans différentes classes ou des groupes d'attributs supports de méthodes. On s'attachera à chercher des types significatifs pour l'entreprise ou le métier. Les types resteront réduits à quelques attributs, et couvriront rarement une classe entière. L'objectif est de rester dans une démarche relationnelle, simplement en ajoutant quelques types fondamentaux aux types de base SQL entier, réel, caractères et date. Une fois isolés, ces types seront définis comme tels, des attributs typés remplaceront les groupes isolés, et la démarche UML/R précédente sera appliquée pour générer les tables. Cette méthode présente l'avantage de continuité et laisse possible l'application des techniques de normalisation que nous allons étudier ci-dessous, en considérant les instances des types utilisés comme atomiques.

La **démarche objet** (notée **UML/OR**, le O venant en premier) au contraire va tout transformer en type et voir les tables comme des extensions de type. Une technique peut consister à procéder systématiquement comme suit :

1. Pour chaque classe, générer le type SQL3 correspondant par la commande CREATE TYPE. Utiliser l'héritage de type pour traduire les spécialisations et l'agrégation de type pour les agrégations composites.
2. Si une classe n'est pas cible d'une agrégation composite ou d'une généralisation, alors l'implémenter comme une table d'objets du type associé. Cela conduit :
   - À implémenter toutes les classes feuille des hiérarchies d'héritage en incluant les attributs hérités, comme dans le cas (b) de la figure XVII.14. D'autres choix sont possibles, comme cela a déjà été signalé ci-dessus.
   - À respecter les agrégations composites en ce sens que les instances de la classe cible figureront dans la table associée à l'autre classe.
3. Implémenter toutes les associations par des attributs références mono ou multivalués (en respectant les cardinalités maximum de l'association) d'une table vers une autre, éventuellement dans les deux sens. L'utilisation de références dans les deux

sens permet les parcours de chemins dans les deux sens. C'est utile si les requêtes le nécessitent.

En fait, aujourd'hui bien peu de SGBD objet-relationnel supporteront une telle démarche pour une grosse base (quelques centaines de classes). Elle conduit en effet à gérer beaucoup de types et beaucoup de références. De plus, le schéma résultat ne peut être normalisé simplement et peut présenter des difficultés d'évolutions, les types étant souvent liés par héritage ou par agrégation. Nous conseillerons donc plutôt l'approche UML/RO plus compatible avec la théorie héritée du relationnel que nous étudions ci-dessous.

4. AFFINEMENT DU SCHÉMA LOGIQUE

Cette section justifie la nécessité d'une étape d'affinement des schémas relationnels et introduit les approches possibles pour réduire les problèmes soulevés par une mauvaise perception du réel.

4.1. ANOMALIES DE MISE À JOUR

Une mauvaise conception des entités et associations représentant le monde réel modélisé conduit à des relations problématiques. Imaginons par exemple que l'on isole une entité unique PROPRIETAIRE contenant tous les attributs des trois relations PERSONNE, VOITURE et POSSEDE. Ainsi, nous pourrions représenter toutes les informations modélisées par une seule table. La figure XVII.17 représente une extension possible de cette table.

NV	Marque	Type	Puiss.	Coul.	NSS	Nom	Prénom	Date	Prix
672RH75	Renault	RME8	8	Rouge	142032	Martin	Jacques	10021998	70 000
800AB64	Peugeot	P206A	7	Bleue	142032	Martin	Jacques	11061999	90 000
686HK75	Citroën	BX20V	9	Verte	158037	Dupond	Pierre	200499	120 000
720CD63	Citroën	2CV8	2	Verte	158037	Dupond	Pierre	200278	5 000
400XY75	Renault	RCL5	4	Verte	275045	Fantas	Julie	110996	20 000
–	–	–	–	–	280037	Schiffer	Claudia	–	–
963TX63	Renault	P306B	7	Bleue	–	–	–	–	–

Figure XVII.17 : Extension de la relation Propriétaire

La relation représentée figure XVII.17 souffre de plusieurs types d'anomalies [Codd72, Fagin81] :

1. Tout d'abord, des données sont redondantes : par exemple, MARTIN Jacques et DUPOND Pierre apparaissent deux fois ; plus généralement, une personne apparaît autant de fois qu'elle possède de voitures.
2. Ces redondances conduisent à des risques d'incohérences lors des mises à jour. Par exemple, si l'on s'aperçoit que le prénom de DUPOND n'est pas Pierre mais Jean, il faudra veiller à mettre à jour les deux tuples contenant DUPOND, sous peine de voir apparaître un DUPOND Pierre et un DUPOND Jean.
3. Il est nécessaire d'autoriser la présence de valeurs nulles dans une telle relation afin de pouvoir conserver dans la base des voitures sans propriétaire ou des personnes ne possédant pas de voitures.

En résumé, une relation qui ne représente pas de « vraies » entités ou associations semble donc souffrir de la présence de données redondantes et d'incohérences potentielles, et nécessite le codage de valeurs nulles. En réalité, un fait élémentaire est enregistré plusieurs fois dans une relation résultant de la jointure de plusieurs entités et association. De plus, les faits élémentaires sont artificiellement associés si bien qu'ils ne peuvent être insérés indépendamment. Il y a tout intérêt à éliminer ces **anomalies d'insertion, de mise à jour et de suppression** afin de faciliter la manipulation des relations.

4.2. PERTE D'INFORMATIONS

Une analyse simpliste des exemples précédents pourrait laisser croire que les relations problématiques sont celles ayant trop d'attributs. Une méthode simple pour éviter les problèmes pourraient être de n'utiliser que des relations binaires à deux colonnes. Malheureusement, des faits élémentaires associent plus de deux valeurs d'attributs. Cela se traduit par le fait qu'un découpage par projections d'une table peut conduire à ne plus être capable de retrouver les informations du monde réel représentées par jointure : on dit qu'il y a **perte d'informations.** Un exemple est représenté figure XVII.18.

VIN1	NV	CRU
	100	Volnay
	200	Chablis
	300	Chablis
	400	Volnay
	500	Sancerre

VIN2	CRU	MILL	DEGRE
	Volnay	1992	11.5
	Chablis	1997	12.3
	Chablis	1999	12.1
	Sancerre	2002	12.0

Figure XVII.18 : Exemple de perte d'informations

L'entité `VIN` a été représentée par deux tables `VIN1` et `VIN2`. En interrogeant par des jointures et projections, et plus généralement en SQL, il est impossible de retrouver précisément le degré d'un vin ou la qualité d'un cru millésimé. Il y a perte de sémantique car la jointure naturelle des deux tables `VIN1` et `VIN2` sur l'attribut commun `CRU` ne permet pas de retrouver les vins de départ, avec un degré unique (par exemple pour les Chablis).

4.3. L'APPROCHE PAR DÉCOMPOSITION

L'approche par décomposition à la conception des schémas relationnels tend à partir d'une relation composée de tous les attributs, appelée la **relation universelle**, et à la décomposer en sous-relations ne souffrant pas des anomalies précédemment signalées.

Notion XVII.2 : Relation universelle (*Universal relation*)

Table unique dont le schéma est composé par union de tous les attributs des tables constituant la base.

La définition de cette relation universelle suppose préalablement une nomination des attributs telle que deux attributs représentant le même concept aient le même nom et deux attributs représentant des concepts distincts aient des noms différents.

Le processus de décomposition est un processus de raffinement successif qui doit aboutir (du moins on l'espère) à isoler des entités et des associations élémentaires, ou si l'on préfère canoniques, du monde réel. Il doit être réalisé à partir d'une bonne compréhension des propriétés sémantiques des données. Cette approche est illustrée par la figure XVII.19. La compréhension de la théorie de la décomposition des relations nécessite la bonne connaissance des deux opérations élémentaires de manipulation de relations que sont la projection et la jointure. En effet, nous allons décomposer par projection et recomposer par jointure.

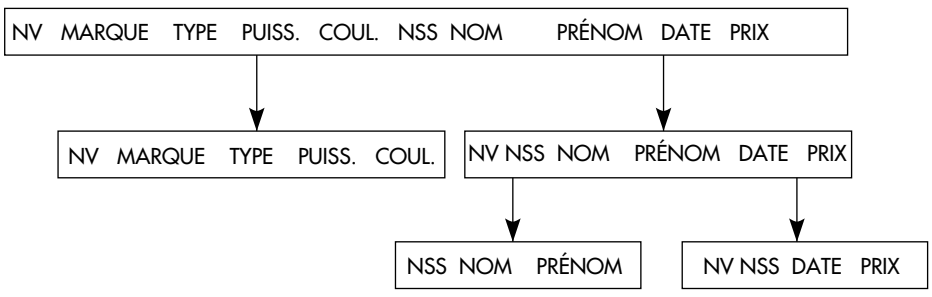


Figure XVII.19 : Le processus de décomposition

Il est maintenant possible d'introduire plus précisément la notion de **décomposition** [Ullman88].

Notion XVII.3 : Décomposition (*Decomposition*)

Remplacement d'une relation R (A_1, A_2… A_n) par une collection de relations R1, R2, … Rn obtenues par des projections de R sur des sous-ensembles d'attributs dont l'union contient tous les attributs de R.

Par suite, lors d'une décomposition, le schéma de relation R (A_1, A_2… A_n) est remplacé par une collection de schémas dont l'union des attributs est (A_1, A_2… A_n). La jointure naturelle R1 |X| R2 |X| Rn constitue donc une relation de même schéma que R, mais dont les tuples ne sont pas forcément les mêmes que ceux de R. À titre d'illustration, la figure XVII.20 propose deux décompositions possibles pour la relation `VOITURE`.

NV	MARQUE	TYPE	PUISS.	COUL.
872RH75	Peugeot	P206A	7	Bleue
975AB80	Peugeot	P206A	7	Rouge

Décomposition 1

R1	NV	TYPE	COUL.
	872RH75 975AB80	P206A P206A	Bleue Rouge

R2	TYPE	MARQ.	PUISS.
	P206A	Peugeot	7

Décomposition 2

V1	NV	TYPE
	872RH75 975AB80	P206A P206A

V2	TYPE	PUISS.	COUL.
	P206A P206A	7 7	Bleue Rouge

V3	TYPE	MARQ.
	P206A	Peugeot

Figure XVII.20 : Deux décompositions possibles de la relation `Voiture`

Si l'on admet qu'à un type de véhicule sont associées une seule marque et une seule puissance (ce qui est vrai pour les tuples figurant sur des cartes grises), la décomposi-

tion 1 est plus plaisante que l'autre : elle permet de retrouver toutes les informations par jointure, alors que la décomposition 2 ne permet pas de retrouver la couleur d'un véhicule ; la jointure V1 $\bowtie$ V2 $\bowtie$ V3 est différente de la relation initiale VOITURE. D'où la notion de **décomposition sans perte** (d'information).

Notion XVII.4 : Décomposition sans perte (*Lossless join decomposition*)

Décomposition d'une relation R en R1, R2… Rn telle que pour toute extension de R, on ait :

R = R1 $\bowtie$ R2 $\bowtie$ … Rn.

Le problème de la conception des bases de données relationnelles peut donc être perçu comme celui de décomposer la relation universelle composée de tous les attributs en sous-relations ne souffrant pas des anomalies vues ci-dessus, de sorte à obtenir une décomposition sans perte. La décomposition a été introduite par Codd [Codd71] et a donné lieu a de nombreux travaux à la fin des années 70 pour « casser en morceaux les relations ». Nous allons ci-dessous étudier les principales méthodes proposées pour effectuer une telle décomposition, qui devrait permettre de déterminer des entités et associations canoniques du monde réel, donc en fait de générer un schéma conceptuel. Ces méthodes ont été développées dans [Rissanen73] puis généralisées dans [Fagin77] et [Zaniolo81].

En pratique, on ne part généralement pas de la relation universelle, mais plutôt du schéma logique obtenu par la modélisation entité-association ou objet, puis par application des règles de passage au relationnel étudiées ci-dessus. Ce processus conduit de fait à une première décomposition intuitive. Si elle est trop fine, il se peut que des informations soient perdues.

4.4. L'APPROCHE PAR SYNTHÈSE

L'**approche par synthèse** [Bernstein76] procède par recomposition des relations à partir d'un ensemble d'attributs indépendants. Fondée sur les propriétés sémantiques des attributs et des liens entre eux, les relations sont composées progressivement de façon à ne pas souffrir des anomalies précédemment mentionnées (voir figure XVII.21 page suivante). Les approches par synthèse s'appuient souvent sur un graphe représentant les liens inter-attributs. Nous verrons un algorithme de synthèse plus loin.

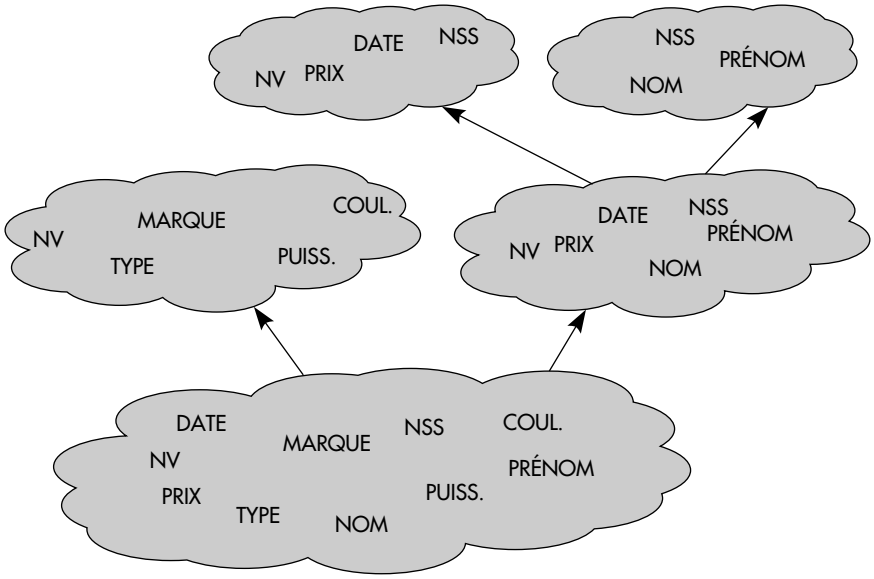


Figure XVII.21 : Illustration de l'approche par synthèse

5. DÉPENDANCES FONCTIONNELLES

Dans cette section, nous étudions les liens sémantiques entre attributs, particulièrement les liens fonctionnels.

5.1. QU'EST-CE QU'UNE DÉPENDANCE FONCTIONNELLE ?

La notion de **dépendance fonctionnelle** fut introduite dès le début du relationnel par CODD afin de caractériser des relations pouvant être décomposées sans perte d'informations.

Notion XVII.5 : Dépendance fonctionnelle (*Functional dependency*)

Soit R (A1, A2… An) un schéma de relation, et X et Y des sous-ensembles de {A1, A2, … An}. On dit que X → Y (X détermine Y, ou Y dépend fonctionnellement de X) si pour toute extension r de R, pour tout tuple t1 et t2 de r, on a : $\Pi_X(t1) = \Pi_X(t2) \Rightarrow \Pi_Y(t1) = \Pi_Y(t2)$

Plus simplement, un attribut (ou groupe d'attributs) Y dépend fonctionnellement d'un attribut (ou groupe d'attributs) X, si, étant donné une valeur de X, il lui correspond une valeur unique de Y (quel que soit l'instant considéré).

À titre d'exemple, dans la relation `VOITURE`, les dépendances fonctionnelles suivantes existent :

```
NV → COULEUR
TYPE → MARQUE
TYPE → PUISSANCE
(TYPE, MARQUE) → PUISSANCE
```

Par contre, les dépendances fonctionnelles suivantes sont inexistantes :

```
PUISSANCE → TYPE
TYPE → COULEUR
```

Il est essentiel de bien remarquer qu'une dépendance fonctionnelle (en abrégé, DF) est une assertion sur toutes les valeurs possibles et non pas sur les valeurs actuelles : elle caractérise une intention et non pas une extension d'une relation. Autrement dit, il est impossible de déduire les DF d'une réalisation particulière d'une relation. La seule manière de déterminer une DF est de regarder soigneusement ce que signifient les attributs car ce sont des assertions sur le monde réel qui lient les valeurs possibles des attributs entre elles. Les DF devraient ainsi être déclarées par l'administrateur d'entreprise au niveau du schéma conceptuel et un bon SGBD devrait les faire respecter.

5.2. PROPRIÉTÉS DES DÉPENDANCES FONCTIONNELLES

Les DF obéissent à plusieurs règles d'inférences triviales. Les trois règles suivantes composent les axiomes des dépendances fonctionnelles et sont connues dans la littérature sous le nom d'axiomes d'Armstrong [Armstrong74] :

– **Réflexivité** : $Y \subseteq X \Rightarrow X \rightarrow Y$; tout ensemble d'attributs détermine lui-même ou une partie de lui-même.
– **Augmentation** : $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$; si X détermine Y, les deux ensembles d'attributs peuvent être enrichis par un même troisième.
– **Transitivité** : $X \rightarrow Y$ et $Y \rightarrow Z \Rightarrow X \rightarrow Z$; cette règle est moins triviale et provient du fait que le composé de deux fonctions dont l'image de l'une est le domaine de l'autre est une fonction. Par exemple, des dépendances NV → TYPE et TYPE → PUISSANCE, on déduit NV → PUISSANCE.

Plusieurs autres règles se déduisent de ces axiomes de base :

– **Union** : $X \rightarrow Y$ et $X \rightarrow Z \Rightarrow X \rightarrow YZ$.
– **Pseudo-transitivité** : $X \rightarrow Y$ et $WY \rightarrow Z \Rightarrow WX \rightarrow Z$.

– **Décomposition** : $X \rightarrow Y$ et $Z \subseteq Y \Rightarrow X \rightarrow Z$.

À partir de ces règles, il est possible d'introduire la notion de **dépendance fonctionnelle élémentaire** [Zaniolo81].

Notion XVII.6 : Dépendance fonctionnelle élémentaire (*Elementary functional dependancy*)

Dépendance fonctionnelle de la forme $X \rightarrow A$, où A est un attribut unique n'appartenant pas à X ($A \notin X$) et où il n'existe pas $X' \subset X$ tel que $X' \rightarrow A$.

La seule règle d'inférence qui s'applique aux dépendances fonctionnelles élémentaires est la transitivité.

5.3. GRAPHE DES DÉPENDANCES FONCTIONNELLES

Soit un ensemble F de dépendances fonctionnelles élémentaires. Si tous les attributs gauches sont uniques, il est possible de visualiser cet ensemble de DF par un graphe appelé **graphe des dépendances fonctionnelles**. À titre d'exemple, nous considérons les dépendances fonctionnelles entre les attributs de la relation VOITURE figure XVII.22.

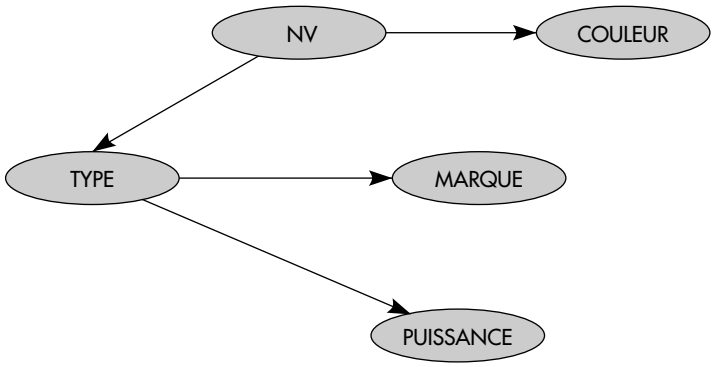


Figure XVII.22 : Exemple de graphe des DF

Il n'est pas toujours possible de représenter les DF d'une relation par un graphe simple : si une partie gauche d'une DF comporte plus d'un attribut, il faut introduire des arcs représentant une association de plusieurs sommets vers un sommet. Nous pouvons alors utiliser la notation des réseaux de Pétri pour représenter les dépendances (voir figure XVII.23).

En effet, la relation :

```
REDUCTION (CRU, TYPE, CLIENT, REMISE)
```

dans laquelle :

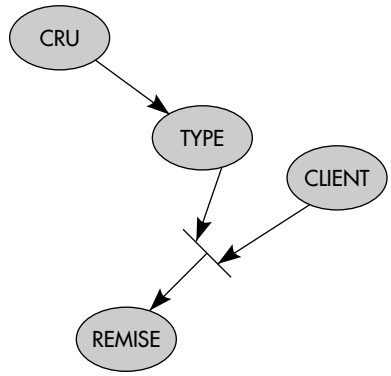
(a) un cru possède un type associé et un seul,

(b) les réductions sont effectuées selon le type et le client,

comporte une dépendance à partie gauche multiple :

```
TYPE, CLIENT → REMISE.
```

CRU	TYPE	CLIENT	REMISE
CHENAS	A	C1	3 %
MEDOC	A	C2	5 %
JULIENAS	B	C1	4 %



CRU → TYPE
TYPE, CLIENT → REMISE

Figure XVII.23 : Exemple de réseau de DF

De même, la relation CODE POSTAL (CODE, VILLE, RUE) comporte les DF :

```
(VILLE, RUE) → CODE
CODE → VILLE.
```

La dépendance (VILLE,RUE) → CODE nécessite un réseau.

5.4. FERMETURE TRANSITIVE ET COUVERTURE MINIMALE

À partir d'un ensemble de DF élémentaires, on peut composer par transitivité d'autres DF élémentaires. On aboutit ainsi à la notion de fermeture transitive d'un ensemble F de DF élémentaires : c'est l'ensemble des DF élémentaires considérées enrichi de toutes les DF élémentaires déduites par transitivité.

Par exemple, à partir de l'ensemble de DF :

```
F =     { NV → TYPE ; TYPE → MARQUE ; TYPE → PUISSANCE ;
        NV → COULEUR}
```

on déduit la fermeture transitive :

```
F+ = F ∪ {NV → MARQUE ; NV → PUISSANCE}
```

Le graphe correspondant à F+ est représenté figure XVII.24.

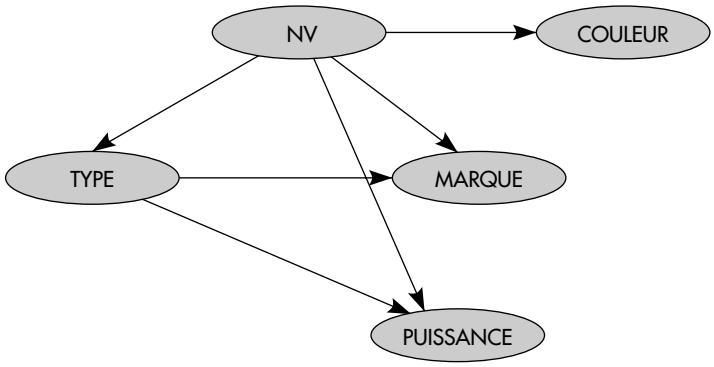


Figure XVII.24 : Exemple de fermeture transitive

À partir de la notion de fermeture transitive, il est possible de définir l'équivalence de deux ensembles de DF élémentaires : deux ensembles sont équivalents s'ils ont la même fermeture transitive. Par suite, il est intéressant de déterminer un sous-ensemble minimal de DF permettant de générer toutes les autres. C'est la **couverture minimale** d'un ensemble de DF.

Notion XVII.7 : Couverture minimale (*Minimal cover*)

Ensemble F de DF élémentaires associé à un ensemble d'attributs vérifiant les propriétés suivantes :

1. Aucune dépendance dans F n'est redondante, ce qui signifie que pour toute DF f de F, F – f n'est pas équivalent à F.

2. Toute DF élémentaire des attributs est dans la fermeture transitive de F (notée F+).

Il a été montré [Delobel73] que tout ensemble de DF a une couverture minimale qui n'est en général pas unique. Par exemple :

```
F =     {NV → TYPE ; TYPE → MARQUE ; TYPE → PUISSANCE ;
        NV → COULEUR
```

est une couverture minimale pour l'ensemble des DF de `VOITURE`. La couverture minimale va constituer un élément essentiel pour composer des relations sans perte d'informations directement à partir des attributs.

5.5. RETOUR SUR LA NOTION DE CLÉ DE RELATION

La notion de **clé** de relation est un concept de base du modèle relationnel. Bien que la notion intuitive de clé soit bien connue, il est possible d'en donner une définition plus formelle à partir de celle de dépendance fonctionnelle, comme suit.

Notion XVII.8 : Clé de relation (*Relation key*)

Sous-ensemble X des attributs d'une relation R (A1, A2..., A2) tel que :

1. X → A1 A2 ... An.

2. Il n'existe pas de sous-ensemble Y ∈ X tel que Y → A1 A2 ... An.

En clair, une clé est un ensemble minimal d'attributs qui détermine tous les autres. Un ensemble d'attributs qui inclut une clé est appelé **superclé**. Par exemple, NV est une clé de la relation VOITURE, alors que (NV, TYPE) n'est pas une clé mais une superclé. Il peut y avoir plusieurs clés pour une même relation : on en choisit en général une comme **clé primaire**. On parle parfois de **clé candidate** pour désigner une clé quelconque.

6. LES TROIS PREMIÈRES FORMES NORMALES

Les trois premières formes normales ont pour objectif de permettre la décomposition de relations sans perdre d'informations, à partir de la notion de dépendance fonctionnelle [Codd72]. L'objectif de cette décomposition est d'aboutir à un schéma conceptuel représentant les entités et les associations canoniques du monde réel.

6.1. PREMIÈRE FORME NORMALE

La première forme normale permet simplement d'obtenir des tables rectangulaires sans attributs multivalués irréguliers.

Notion XVII.9 : Première forme normale (*First normal form*)

Une relation est en première forme normale si tout attribut contient une valeur atomique.

Cette forme normale est justifiée par la simplicité et l'esthétique. Elle consiste simplement à éviter les domaines composés de plusieurs valeurs. Plusieurs décompositions sont possibles, comme vu ci-dessus. Par exemple, la relation PERSONNE(NOM, PRENOMS) pourra être décomposée en PERSONNE1(NOM, PRENOM1) et PERSONNE2(NOM, PRENOM2) si l'on sait que les personnes n'ont pas plus de deux prénoms. Plus généralement, nous avons montré ci-dessus qu'une relation de clé K avec un attribut multivalué A* pouvait être décomposée en deux relations par élimina-

tion de l'attribut multivalué et génération d'une table de schéma (K, A) donnant les valeurs élémentaires de A associées aux valeurs de la clé. La règle de décomposition en première forme normale n'est rien d'autre que l'application systématique de cette transformation. Si la relation n'a pas d'autre attribut que la clé et l'attribut multivalué, elle est simplement désimbriquée comme pour l'exemple de la figure XVII.25.

PERSONNE	NOM	PROFESSION
	DUPONT	Ingénieur, Professeur
	MARTIN	Géomètre

Une telle relation doit être décomposée en répétant les noms pour chaque profession (Opération UNNEST)

Figure XVII.25 : Exemple de relation non en première forme normale

Soulignons que la première forme normale est une question de définition de domaine : chaque valeur d'un domaine est en effet un atome du point de vue du modèle relationnel. Par suite, rien n'empêche de considérer une date ou une figure géométrique comme atomique si les domaines de valeur sont les dates et les figures géométriques. C'est une question de point de vue et de niveau de décomposition.

6.2. DEUXIÈME FORME NORMALE

La **deuxième forme normale** permet d'assurer l'élimination de certaines redondances en garantissant qu'aucun attribut n'est déterminé seulement par une partie de la clé.

Notion XVII.10 : Deuxième forme normale (*Second normal form*)

Une relation R est en deuxième forme normale si et seulement si :

1. Elle est en première forme.

2. Tout attribut n'appartenant pas à une clé ne dépend pas d'une partie d'une clé.

Le schéma typique d'une relation qui n'est pas en 2e forme normale est représenté figure XVII.26. K1 et K2 désignent deux parties de la clé K. Le problème est que K2 à lui seul détermine Y : K2 → Y. Donc Y dépend d'une partie de la clé. Comme nous le verrons plus loin, une telle relation doit être décomposée en R1(K1,K2,X) et R2(K2,Y).

Par exemple, considérons la relation :

```
FOURNISSEUR (NOM, ADRESSE, ARTICLE, PRIX)
```

La clé est le couple (NOM, ARTICLE). Il existe les DF :

```
(NOM, ARTICLE) → PRIX et NOM → ADRESSE.
```

Par suite, une partie de la clé (NOM) détermine un attribut n'appartenant pas à la clé. Cette relation n'est donc pas en deuxième forme normale. Elle pourra être décomposée en deux relations :

```
FOURNISSEUR (NOM, ADRESSE)
PRODUIT (NOM, ARTICLE, PRIX)
```

qui, quant à elles, sont en deuxième forme normale.

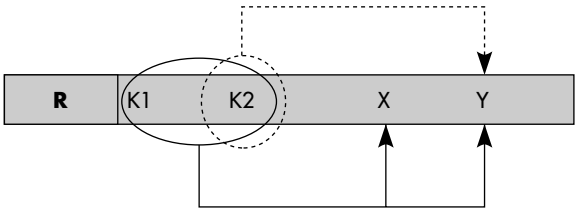


Figure XVII.26 : Schéma de relation non en 2e forme normale

6.3. TROISIÈME FORME NORMALE

La **troisième forme normale** permet d'assurer l'élimination des redondances dues aux dépendances transitives.

Notion XVII.11 : Troisième forme normale (*Third normal form*)

Une relation R est en troisième forme normale si et seulement si :

1. Elle est en deuxième forme.

2. Tout attribut n'appartenant pas à une clé ne dépend pas d'un autre attribut non clé.

Soulignons qu'une partie de clé n'est pas une clé. En conséquence, une relation en 3e forme est automatiquement en 2e : la condition 1 est automatiquement vérifiée, mais figure par souci de clareté. Soulignons aussi que si la relation possède plusieurs clés candidates, la définition doit être vérifiée pour chacune d'elles successivement.

Le schéma typique d'une relation qui n'est pas en 3e forme normale est représenté figure XVII.27. K est la clé de R. Le problème est que X à lui seul détermine Z : X → Z. Donc Z dépend d'un attribut non clé. Comme nous le verrons plus loin, une telle relation doit être décomposée en R1(K1,K2,X) et R2(K2,Y).

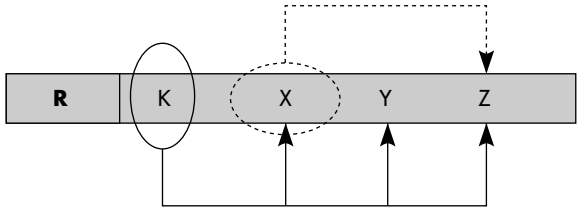


Figure XVII.27 : Schéma de relation non en 3e forme normale

À titre d'illustration, la relation :

```
VOITURE (NV, MARQUE, TYPE, PUISSANCE, COULEUR)
```

n'est pas en troisième forme normale. En effet, l'attribut non clé TYPE détermine MARQUE et aussi PUISSANCE. Cette relation peut être décomposée en deux relations :

```
VOITURE (NV, TYPE, COULEUR)
MODELE (TYPE, MARQUE, PUISSANCE).
```

Si la relation possède une seule clé primaire, il est possible de donner une définition équivalente comme suit. Une relation R est en troisième forme normale si et seulement si :

1) elle est en deuxième forme normale ;

2) tout attribut n'appartenant pas à la clé ne dépend pas transitivement de la clé.

Par exemple, dans la relation VOITURE, l'attribut MARQUE dépend transitivement de la clé ainsi que l'attribut PUISSANCE :

```
NV → TYPE → PUISSANCE,
NV → TYPE → MARQUE.
```

6.4. PROPRIÉTÉS D'UNE DÉCOMPOSITION EN TROISIÈME FORME NORMALE

Les dépendances fonctionnelles sont des règles indépendantes du temps que doivent vérifier les valeurs des attributs. Il est nécessaire qu'une décomposition préserve ces règles.

Notion XVII.12 : Décomposition préservant les dépendances fonctionnelles (*Dependencies preserving decomposition*)

Décomposition {R1, R2… Rn} d'une relation R telle que la fermeture transitive des DF de R est la même que celle de l'union des DF de {R1, R2… Rn}.

À titre d'exemple, la décomposition de la relation `VOITURE(NV, MARQUE, TYPE, PUISSANCE, COULEUR)` en `R1(NV, TYPE, COULEUR)` et `R2(TYPE, MARQUE, PUISSANCE)` préserve les DF, alors que la décomposition en `V1(NV, TYPE)`, `V2(TYPE, PUISSANCE, COULEUR)` et `V3(TYPE, MARQUE)` ne les préserve pas, comme le montre la figure XVII.28.

La troisième forme normale est importante. En effet, toute relation a au moins une décomposition en troisième forme normale telle que :

(a) la décomposition préserve les DF ;

(b) la décomposition est sans perte.

Cette décomposition peut ne pas être unique. Nous allons dans la suite étudier un algorithme permettant de générer une telle décomposition.

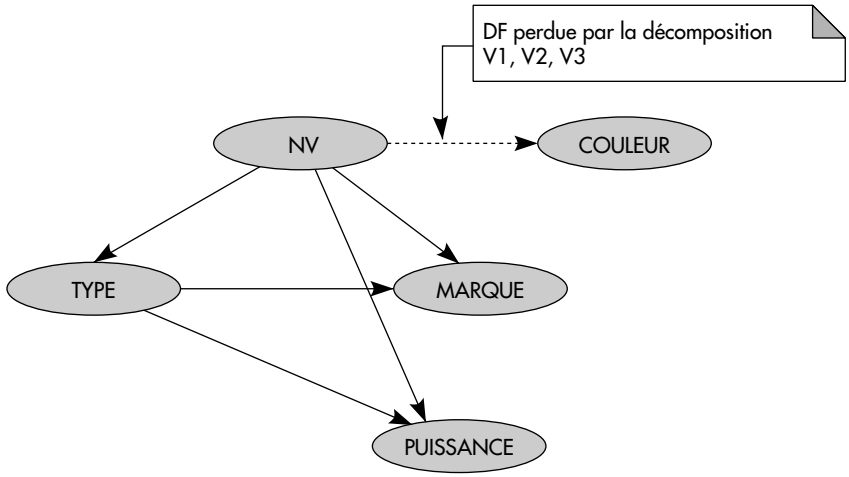


Figure XVII.28 : DF perdue par la décomposition {V1, V2, V3}

6.5. ALGORITHME DE DÉCOMPOSITION EN TROISIÈME FORME NORMALE

Pour toute relation, y compris la relation universelle, il existe donc au moins une décomposition en troisième forme normale préservant les DF et sans perte. Le but

d'un algorithme de décomposition en 3e forme normale est de convertir un schéma de relation qui n'est pas en 3e FN en un ensemble de schémas en 3e FN. Le principe consiste simplement à appliquer récursivement les règles de décomposition énoncées ci-dessus, afin de décomposer jusqu'à obtenir des relations en 3e FN.

Revenons quelque peu sur ces règles afin de montrer leur validité. Soit donc une relation de schéma R(K1, K2, X, Y) qui n'est pas en 2e FN. La figure XVII.29 donne le graphe de DF associé. Les cercles correspondant aux relations décomposées R1(K1,K2, X) et R2(K2,Y) montrent simplement que l'union des DF de R1 et R2 est bien l'ensemble des DF : cette décomposition préserve les DF. D'un autre côté, par jointure sur K2, on retrouve bien la relation initiale. Donc, la décomposition est sans perte. Les deux relations sont bien en 2e FN. Cette règle de décomposition a donc toutes les bonnes propriétés.

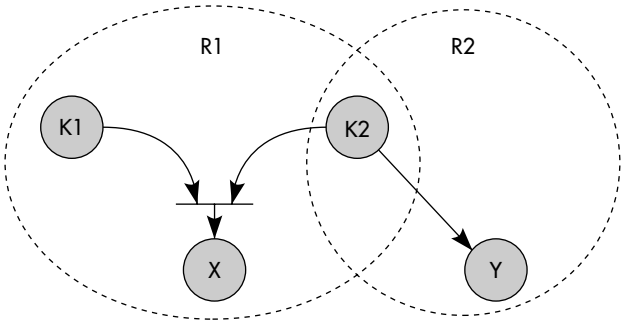


Figure XVII.29 : Décomposition d'une relation non en 2e forme normale

On vérifie de manière similaire les bonnes propriétés de la décomposition en 3e FN, comme illustrée figure XVII.30.

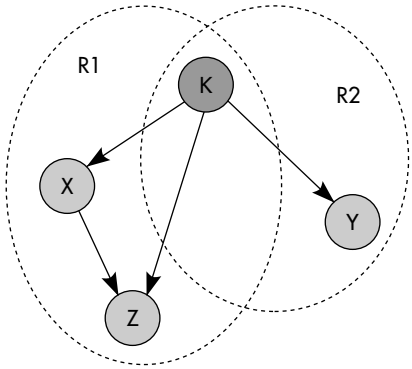


Figure XVII.30 : Décomposition d'une relation non en 3e forme normale

6.6. ALGORITHME DE SYNTHÈSE EN TROISIÈME FORME NORMALE

Une décomposition en 3e FN peut être générée par un algorithme de synthèse ayant pour entrées l'ensemble des attributs ainsi que les DF. Le principe d'un tel algorithme [Bernstein76] consiste à construire tout d'abord une couverture minimale F des DF élémentaires. Ensuite, la couverture F doit être partitionnée en groupes Fi tels que les DF dans chaque Fi ait le même ensemble d'attributs à gauche. Chaque groupe produit une relation en 3e FN.

Pour produire un groupe, on recherche le plus grand ensemble X d'attributs qui détermine d'autres attributs A_1, A_2… An (avec n ≥1), et l'on extrait la relation (X1, A1, A2 … An). Une telle relation de clé X est bien en 3e forme normale car X détermine tous les autres attributs et il ne peut exister de dépendances transitives $X \rightarrow Ai \rightarrow Aj$ du fait que l'on est parti d'une couverture minimale (sinon, $X \rightarrow Aj$ ne serait pas dans cette couverture). Les DF $X \rightarrow A_1$, $X \rightarrow A_2$…, $X \rightarrow An$ sont alors éliminées de la couverture minimale, ainsi que les attributs isolés créés (non source ou cible de DF). L'algorithme est ensuite appliqué itérativement jusqu'à ce qu'il ne reste plus de groupe. S'il reste des attributs isolés, on les sort dans une table qui est bien en 3e FN puisqu'elle n'a pas de DF entre ses attributs. Cet algorithme est schématisé figure XVII.31.

```
Procédure Synthèse({Ai}, {DF}) { // Normalisation par synthèse
   Trouver une couverture minimale F de {DF} ;
   Conserver les seules dépendances élémentaires ;
   Tant que « il reste des DF » Faire { ProduireGroupe() } ;
   Editer la relation composée des attributs restants ;
}
// Production d'une relation de clé maximale à partir de F
Procédure ProduireGroupe {
   X = Rechercher(), // rechercher le plus grand ensemble
               // d'attributs X qui en détermine d'autres ;
   A = {Ai | Ai → X} ;
   Editer la relation (X, A) ; //Génération de relation en 3e FN
   Pour chaque Ai de A Faire { // Réduction de F
      Eliminer toute DF incluse dans (X,Ai) ;
      Si Ai est isolé Alors Eliminer Ai de F ;} ;
   } ;
```

Figure XVII.31 : Algorithme de synthèse de relations en 3e forme normale

6.7. FORME NORMALE DE BOYCE-CODD

La 2e forme normale élimine les anomalies créées par des dépendances entre parties de clé et attributs non clé. La 3e forme normale élimine les anomalies créées par des

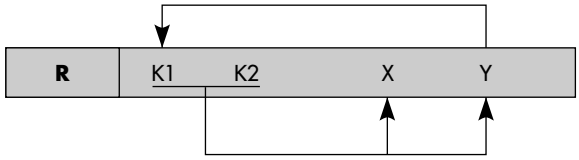
dépendances entre les attributs non clés. Quid des dépendances de parties de clés entre elles ou d'attributs non clé vers une partie de clé ? Eh bien, la 3e FN est insuffisante.

Afin d'éliminer les redondances crées par des dépendances entre parties de clés et celles déjà éliminées par la 3e FN, Boyce et Codd ont introduit la forme normale qui porte leur nom (en abrégé BCNF) [Codd74].

Notion XVII.13 : Forme normale de Boyce-Codd (*Boyce-Codd normal form*)

Une relation est en BCNF si et seulement si les seules dépendances fonctionnelles élémentaires sont celles dans lesquelles une clé entière détermine un attribut.

Cette définition a le mérite d'être simple : pas de dépendance autre que K→A, K étant la clé et A un attribut non clé. Il a été montré que toute relation a une décomposition en BCNF qui est sans perte. Par contre, une décomposition en BCNF ne préserve en général pas les DF. La figure XVII.32 illustre le cas typique où un attribut non clé détermine une partie de clé, et indique le schéma de décomposition associé.



Une telle relation doit être décomposée en R1(K1, K2, X) et R2(Y, K1)

Figure XVII.32 : Schéma de relation en 3e forme normale mais non en BCNF

Considérons par exemple la relation :

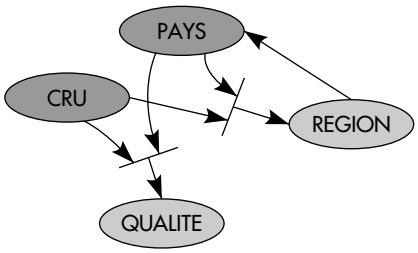
```
LOCALISATION (CRU, PAYS, REGION, QUALITE)
```

avec les dépendances :

```
CRU, PAYS → REGION
CRU, PAYS → QUALITE
REGION → PAYS.
```

Cette relation n'est pas en BCNF bien qu'en 3e forme puisque le cru ne détermine pas la région (il y a du Chablis en Bourgogne mais aussi en Californie ; notez que la qualité dépend bien du pays !). Une instance, le réseau de DF et la décomposition souhaitable sont représentés figure XVII.33. La DF CRU, PAYS → REGION est perdue. La décomposition est cependant sans perte.

CRU	PAYS	REGION	QUALITE
Chenas	France	Beaujolais	Excellent
Juliénas	France	Beaujolais	Excellent
Morgon	France	Beaujolais	Bon
Chablis	États-Unis	Californie	Moyen
Brouilly	États-Unis	Californie	Médiocre
Chablis	France	Beaujolais	Excellent



Décomposition :
CRUS (CRU, PAYS, QUALITE)
REGIONS(REGION, PAYS)

Figure XVII.33 : Exemple de relation non en BCNF et décomposition

7. QUATRIÈME ET CINQUIÈME FORMES NORMALES

La BCNF n'est pas suffisante pour éliminer complètement les redondances. Pour aller au-delà, il faut introduire des dépendances plus lâches. Nous allons en voir de plusieurs types.

7.1. DÉPENDANCES MULTIVALUÉES

Un exemple permet de comprendre la nature des redondances. Considérons la relation :

```
ETUDIANT (NE, COURS, SPORT)
```

NE est le numéro d'étudiant, COURS les cours suivis et SPORT les sports pratiqués. Une extension de cette relation est représentée figure XVII.34. NE, COURS et SPORT constituent la clé composée. En effet, NE ne détermine ni cours ni sport car il est conseillé de suivre plusieurs cours et de pratiquer plusieurs sports (c'est le cas de l'étudiant 100 ci-dessous).

NE	COURS	SPORT
100	Bases de données	Tennis
100	Bases de données	Football
100	Réseaux	Tennis
100	Réseaux	Football
200	Réseaux	Vélo

Figure XVII.34 : Relation en troisième forme normale avec redondance

Ses redondances apparaissent clairement dans cette relation. Cependant, en raison de l'absence de dépendances fonctionnelles, elle est jusque-là non décomposable.

L'exemple précédent montre l'insuffisance de la notion de dépendance fonctionnelle : elle ne permet pas de saisir l'indépendance qui existe entre des attributs comme COURS suivi et SPORT pratiqué. Pour cela, on généralise la notion de DF en introduisant celle de **dépendance multivaluée** (DM) [Fagin77, Zaniolo81].

Notion XVII.14 : Dépendance multivaluée (*Multivalued dependency*)

Soit R (A1, A2… An) un schéma de relation, et X et Y des sous-ensembles de A1, A2, … An. On dit que X $\twoheadrightarrow$ Y (X multidétermine Y, ou il y a une dépendance multivaluée de Y sur X) si, étant donné des valeurs de X, il y a un ensemble de valeurs de Y associées et cet ensemble est indépendant des autres attributs Z = R – X – Y de la relation R.

Une dépendance multivaluée caractérise donc une indépendance entre deux ensembles d'attributs (Y et Z) corrélés par un même troisième X. Plus formellement, on a :

$$(X \twoheadrightarrow Y) \Leftrightarrow \{(x\ y\ z) \text{ et } (x\ y'\ z') \in R \Rightarrow (x\ y'\ z) \text{ et } (x\ y\ z') \in R\}$$

où x, y, z, y', z' désignent des occurrences des attributs X, Y, Z, Y, Z.

Il faut souligner que les DF sont des cas particuliers de DM. En effet :

$$(X \rightarrow Y) \Rightarrow \{(x\ y\ z) \text{ et } (x\ y'\ z') \in R \Rightarrow y = y'\}.$$

Donc :

$$(X \rightarrow Y) \Rightarrow \{(x\ y\ z) \text{ et } (x\ y'\ z') \in R \Rightarrow (x\ y'\ z) \text{ et } (x\ y\ z') \in R.$$

Par suite :

$$(X \rightarrow Y) \Rightarrow (X \twoheadrightarrow Y).$$

Comme avec les dépendances fonctionnelles, il est possible d'effectuer des inférences à partir des dépendances multivaluées. Les axiomes d'inférence des DM sont les suivants, en considérant une relation composée d'un ensemble d'attributs R [Beeri79] :

1. **Complémentation** : $(X \twoheadrightarrow Y) \Rightarrow (X \twoheadrightarrow R - X - Y)$
2. **Augmentation** : $(X \twoheadrightarrow Y)$ et $(V \subseteq W) \Rightarrow (XW \twoheadrightarrow YV)$
3. **Transitivité** : $(X \twoheadrightarrow Y)$ et $(Y \twoheadrightarrow Z) \Rightarrow (X \twoheadrightarrow Z - Y)$

Des règles additionnelles s'en déduisent, telles que celle d'union :

4. **Union** : $(X \twoheadrightarrow Y)$ et $(Y \twoheadrightarrow Z) \Rightarrow (X \twoheadrightarrow YZ)$

À partir des axiomes précédents, il est possible d'introduire la notion de **dépendance multivaluée élémentaire**, ceci afin d'éliminer les dépendances déduites trivialement d'un ensemble de dépendances de base [Zaniolo81].

Notion XVII.15 : Dépendance multivaluée élémentaire (*Elementary multivalued dependency*)

Dépendance multivaluée $X \twoheadrightarrow Y$ d'une relation R telle que :

1. Y n'est pas vide et est disjoint de X.

2. R ne contient pas une autre DM du type $X' \twoheadrightarrow Y'$ telle que $X' \subset X$ et $Y' \subset Y$.

Ainsi, une dépendance multivaluée élémentaire a, à la fois, un côté droit et un côté gauche minimaux, comme une dépendance fonctionnelle élémentaire.

Afin d'illustrer plus en détail, nous donnerons deux autres exemples de DM. Soit la relation :

```
VOL (NV, AVION, PILOTE)
```

où NV est un numéro de vol. On suppose disposer d'un ensemble d'avions et d'un ensemble de pilotes. Tout pilote est conduit à piloter tout avion sur n'importe quel vol. Ainsi, les avions et les pilotes sont indépendants. D'où les deux DM élémentaires :

```
NV ↠ AVION
NV ↠ PILOTE
```

Soit encore la relation :

```
PERSONNE (N°SS, PRENOM-ENFANT, N° VEHICULE)
```

Il est clair que l'on a les DM élémentaires :

```
N°SS ↠ PRENOM-ENFANT
N°SS ↠ N°VEHICULE
```

7.2. QUATRIÈME FORME NORMALE

La quatrième forme normale est une généralisation de la forme normale de Boyce-Codd destinée à décomposer les relations ayant des DM élémentaires.

Notion XVII.16 : Quatrième forme normale (*Fourth normal form*)

Une relation est en quatrième forme normale si et seulement si les seules dépendances multivaluées élémentaires sont celles dans lesquelles une superclé détermine un attribut.

Rappelons qu'une superclé est un ensemble d'attributs contenant une clé. Donc, une relation R n'est pas en 4e FN si l'on peut trouver une dépendance de la forme X $\twoheadrightarrow$ Y où X n'inclut pas une clé de R. Comme une dépendance fonctionnelle est un cas particulier de dépendance multivaluée, il apparaît qu'une relation en quatrième forme normale est en forme normale de Boyce-Codd et donc en troisième forme normale.

À titre d'exemple, la relation ETUDIANT (NE, COURS, SPORT) n'est pas en quatrième forme normale : la clé est l'ensemble des attributs et il existe des DM élémentaires entre des attributs participants à la clé :

NE $\twoheadrightarrow$ COURS
NE $\twoheadrightarrow$ SPORT.

Il a été montré que toute relation a une décomposition (pas forcément unique) en quatrième forme normale qui est sans perte [Fagin77]. Par exemple, la relation ETUDIANT peut être décomposée en deux relations (NE, COURS) et (NE, SPORT) qui sont bien en quatrième forme normale.

7.3. DÉPENDANCES DE JOINTURE

La notion de dépendance multivaluée a conduit à décomposer les relations en quatrième forme normale. Est-ce suffisant pour éliminer les problèmes de redondances et anomalies ? [Nicolas78] et [Fagin79] ont montré que non.

Considérons par exemple la relation BUVCRU représentée figure XVII.35 ; cette relation modélise des vins bus par des buveurs, d'un cru donné et commandés à un producteur produisant ce cru.

BUVEUR	CRU	PRODUCTEUR
Ronald	Chablis	Georges
Ronald	Chablis	Nicolas
Ronald	Volnay	Nicolas
Claude	Chablis	Nicolas

Figure XVII.35 : Relation en quatrième forme normale avec redondance

Cette relation est bien en quatrième forme normale. En effet, il n'existe pas de dépendance multivaluée d'après l'extension représentée ci-dessus :

– BUVEUR $\twoheadrightarrow$ CRU est faux car par exemple le tuple (Ronald Volnay Georges) n'existe pas.
– CRU $\twoheadrightarrow$ PRODUCTEUR est faux car par exemple le tuple (Claude Chablis Georges) n'existe pas.
– PRODUCTEUR $\twoheadrightarrow$ BUVEUR est faux car par exemple le tuple (Claude Volnay Nicolas) n'existe pas.

Autrement dit, si l'on considère les projections R1, R2, R3 de la relation BUVCRU sur deux attributs (voir figure XVII.36), on constate que l'on a :

– BUVCRU $\neq$ R1 $\bowtie$ R2,
– BUVCRU $\neq$ R1 $\bowtie$ R3,
– BUVCRU $\neq$ R2 $\bowtie$ R3.

Cependant, la relation représentée figure XVII.35 présente bien des redondances : on apprend deux fois que Ronald boit du Chablis et que Nicolas produit du Chablis. Elle n'est cependant pas décomposable en deux relations.

R1	BUVEUR	CRU
	Ronald	Chablis
	Ronald	Volnay
	Claude	Chablis

R2	BUVEUR	PRODUCTEUR
	Ronald	Georges
	Ronald	Nicolas
	Claude	Nicolas

R2	CRU	PRODUCTEUR
	Chablis	Georges
	Chablis	Nicolas
	Volnay	Nicolas

Figure XVII.36 : Trois projections de la relation BUVCRU

L'exemple précédent montre l'insuffisance de la notion de dépendance multivaluée pour éliminer les redondances. Le problème vient du fait que jusqu'à présent, nous avons essayé de décomposer une relation seulement en deux relations. Ainsi, la notion de dépendance multivaluée capture la possibilité de décomposer une relation en deux ; la relation (XYZ) dans laquelle X $\twoheadrightarrow$ Y est en effet décomposée en (XY) et (XZ) puisque Y et Z sont indépendants par rapport à X. Comme nous allons le voir, il existe des relations non décomposables en deux mais décomposables en trois, quatre ou plus généralement N relations. Ce phénomène a été découvert par [Aho79] et [Nicolas78].

À titre d'exemple de relation décomposable en trois relations et non décomposable en deux, supposons que la relation BUVCRU de la figure XVII.35 obéisse à la contrainte d'intégrité assez plausible :

« Tout buveur ayant bu un cru et ayant commandé à un producteur produisant ce cru a aussi commandé ce cru à ce producteur ».

Cette contrainte s'écrit plus formellement :

$(b,c) \in R1$ et $(b,p) \in R2$ et $(c,p) \in R3 \Rightarrow (b,c,p) \in R$.

Dans ce cas, R sera la jointure de R1, R2 et R3 :

$R = R1 \bowtie R2 \bowtie R3$

Cette contrainte est bien vérifiée par l'extension de la relation BUVCRU représentée figure XVII.35 en considérant ses projections R1, R2 et R3 représentées figure XVII.36.

Plus généralement, [Rissanen78] a introduit la notion de **dépendance de jointure (DJ)** afin de décomposer des relations en plusieurs.

Notion XVII.17 : Dépendance de jointure (*Join dependency*)

*Soit R (A1, A2 … An) un schéma de relation et R1, R2… Rm des sous-ensembles de {A1, A2… An}. On dit qu'il existe une dépendance de jointure *{R1, R2… Rm} si R est la jointure de ses projections sur R1, R2… Rm, c'est-à-dire si* $R = \Pi_{R1}(R) \bowtie \Pi_{R2}(R) \ldots \bowtie \Pi_{Rm}(R)$.

En d'autres termes, la dépendance de jointure *{R1, R2… Rm} est valide si R1, R2…Rp est une décomposition sans perte de R. En conséquence, une relation de schéma R satisfait la dépendance de jointure *{R1, R2… Rm} quand la condition suivante est valide pour toute instance r de R :

Si $t1 \in \Pi_{R1}(r)$ $t2 \in \Pi_{R2}(r) \ldots \ldots tm \in \Pi_{Rm}(r)$, alors t1*t2*..tm ∈ R en notant * la jointure naturelle des relations Ri concernées.

Par exemple, la relation de schéma BUVCRU(BUVEUR, CRU, PRODUCTEUR) obéit à la dépendance de jointure :

```
*{(BUVEUR,CRU),(BUVEUR,PRODUCTEUR),(CRU, PRODUCTEUR)}
```

Elle est donc décomposable en trois relations R1(BUVEUR,CRU), R2(BUVEUR, PRODUCTEUR) et R3(CRU,PRODUCTEUR) comme représentée figure XVII.36. Si (b,c), (b,p) et (c,p) sont respectivement des tuples de R1, R2 et R3, alors (b,c,p) est un tuple de BUVCRU.

Les dépendances multivaluées sont bien sûr des cas particuliers de dépendances de jointures. En effet, une relation R (X,Y,Z) vérifiant la dépendance multivaluée $X \twoheadrightarrow Y$ (et donc $X \twoheadrightarrow Z$) satisfait la dépendance de jointure *{(XY), (XZ)}.

7.4. CINQUIÈME FORME NORMALE

La forme normale de projection jointure, parfois appelée cinquième forme normale, est une généralisation de la quatrième à partir de la notion de dépendance de jointure.

Sa définition nécessite d'étudier les dépendances de jointures comme nous l'avons fait pour les DF ou les DM. Soit une relation R et *{R1, R2…Rp} une dépendance de jointure. Une telle dépendance de jointure est triviale si l'une des relations Ri est la relation R elle-même.

Il nous est maintenant possible de définir la **forme normale de projection-jointure**, encore appelée 5e forme normale.

Notion XVII.18 : Forme normale de projection-jointure (*Project-join normal form*)

Une relation R est en forme normale de projection-jointure si et seulement si toute dépendance de jointure est triviale ou tout Ri est une superclé de R (c'est-à-dire que chaque Ri contient une clé de R).

L'idée simple est que si la DJ est impliquée par les clés, la décomposition n'éliminera pas de redondance et est sans intérêt. Si elle contient R, elle ne sert à rien puisque R demeure. Dans les autres cas, il est possible de décomposer par projection selon les schémas de la DJ ; l'expression de jointures dérivées de la JD permet de recomposer la relation R. Par suite, la décomposition d'une relation non en 5e forme suit les DJ et est sans perte. Par contre, elle ne préserve en général pas les DF, comme la BCNF. Notons aussi que la 5e forme normale est une généralisation directe de la BCNF et de la 4e ; donc une relation en 5e forme est en 4e et bien sûr en 3e.

Ainsi la relation BUVCRU n'est pas en 5e forme normale puisque la seule clé candidate (`BUVEUR, CRU, PRODUCTEUR`) n'implique pas la DJ *{(`BUVEUR CRU`), (`CRU PRODUCTEUR`), (`BUVEUR PRODUCTEUR`)}. Elle doit donc être décomposée en ces trois relations afin d'éviter les anomalies de mise à jour.

[Fagin79] a démontré le résultat essentiel suivant. Toute relation en 5e forme normale ne peut plus être décomposée sans perte d'informations (excepté par les décompositions basées sur les clés qui sont sans intérêt) si l'on ne considère que la décomposition par projection et la recomposition par jointure. La 5e forme normale est donc un point final à la décomposition par projection-jointure. Voilà pourquoi Fagin a proposé d'appeler cette forme « **forme normale de projection-jointure** » (JD/NF).

La 5e forme n'est cependant pas la forme ultime de décomposition si l'on accepte aussi des décompositions horizontales, c'est-à-dire en partitionnant la table en sous-tables comportant chacune un ensemble de tuples avec tous les attributs. Il est possible d'introduire des **dépendances algébriques** du style $R \subset E(R)$ où E est une expression de l'algèbre relationnelle avec union, projection et jointure [Yannakakis80]. Les **dépendances d'inclusion** constituent une forme plus restreinte de dépendances qui unifient les FD et les JD [Abiteboul95]. Mais tout cela n'est pas d'une grande utilité pour concevoir une BD.

8. CONCEPTION DU SCHÉMA INTERNE

Nous abordons dans cette partie les problèmes plus pratiques de passage au niveau interne, c'est-à-dire d'implémentation du schéma logique sur un SGBD particulier.

8.1. LES PARAMÈTRES À PRENDRE EN COMPTE

La conception du schéma interne, encore appelée conception physique, vise non plus à résoudre les problèmes sémantiques, mais les problèmes de performances. L'objectif est, pour une charge applicative donnée, de trouver le meilleur schéma physique pour optimiser les temps d'accès et de calcul, plus généralement le débit en transactions et les temps de réponse.

Quels sont les paramètres nécessaires ? Tout d'abord, le schéma logique de la base issu des étapes précédentes est connu. Pour chaque relation, il faut aussi connaître les tailles en nombre de tuples et le profil moyen d'un tuple (attributs et tailles). En plus, il faut avoir un modèle de l'application. Plus précisément, pour chaque transaction, il est souhaitable de connaître :

– la fréquence, voire éventuellement une distribution de fréquence pendant la journée ;
– les requêtes exécutées, avec les types (SELECT, UPDATE, etc.), les critères et un nombre de fois si la requête est répétée ;
– le mode d'exécution, c'est-à-dire requête interactive ou compilée.

À partir de ces paramètres, un modèle de l'application peut être élaboré. Il peut être analytique ou simulé. Mais il faut aussi modéliser le SGBD sauf si l'on choisit un modèle simulé effectivement construit sur le SGBD. Les paramètres intéressants à prendre en compte au niveau du SGBD sont les configurations des disques, la taille du cache, la taille des pages, les temps de lecture et d'écriture d'une page, les types d'index, etc. C'est en fait très complexe.

Le **modèle analytique** conduit à une formule de coût paramétrée (par exemple la présence ou non d'un index est un paramètre) qu'il s'agit d'optimiser. En général, le coût va être composé du temps d'entrées-sorties et du temps unité centrale pondéré par un coefficient unificateur. Les formules de l'optimiseur de requêtes sont à intégrer dans un tel modèle. Le problème est généralement que le nombre de paramètres est trop grand pour permettre une recherche d'optimum.

Le **modèle simulé** est plus crédible. Il conduit à réaliser une base réduite avec des corps de transactions comprenant essentiellement les requêtes. Un générateur de transactions doit alors être réalisé pour simuler la charge. Un tel modèle permet de faire des mesures effectives en faisant varier tel ou tel paramètre, par exemple en ajoutant

ou en supprimant un index. Les modèles à base d'outils généraux basés sur les files d'attente sont aussi utilisables.

Au-delà du modèle qui est un problème en soi, nous examinons ci-dessous les paramètres sur lequel l'administrateur système peut jouer. Ceux-ci sont souvent dépendant du SGBD.

8.2. DÉNORMALISATION ET GROUPEMENT DE TABLES

D'un point de vue logique, pour éviter anomalies et pertes d'opérations, nous avons vu ci-dessus qu'il était souhaitable de décomposer les relations par projection, la recomposition étant faite par jointure. D'un point de vue performance, ce n'est pas toujours optimal. En effet, la normalisation conduit, si l'application le nécessite, à recalculer les jointures. D'où l'idée d'une étape de **dénormalisation** possible.

Notion XVII.19 : Dénormalisation (*Denormalisation*)

Technique consistant à implémenter la jointure de deux relations (ou plus) à la place des relations individuelles initiales.

Bien sûr, la dénormalisation conduit à des redondances et éventuellement à des valeurs nulles qui doivent être prises en compte par les programmes d'application. Soulignons aussi que les jointures intéressantes suivent en général les contraintes référentielles, qui sont les plus fréquemment utilisées. Quand une dénormalisation est-elle souhaitable ? Pratiquement, le gain est pour les recherches, la jointure des deux tables étant remplacée par une sélection sur la table résultant de la jointure. La perte affecte les mises à jour qui s'appliquent sur la table jointure en plusieurs points, mais aussi les sélections sur les tables individuelles qui doivent maintenant s'appliquer sur la jointure. Seuls un calcul analytique ou une simulation permettent de calculer la différence et de conclure.

Une variante plus performante de la dénormalisation est le groupement, permis dans certains SGBD.

Notion XVII.20 : Groupement (*Clustering*)

Technique permettant de regrouper dans une même page ou des pages voisines les tuples de deux tables (ou plus) selon un critère de jointure.

Cette technique est connue depuis longtemps sous le nom de **placement à proximité** dans le modèle réseau. Elle permet par exemple de placer les lignes de commandes dans la page de l'en-tête. Ainsi, la jointure ne nécessite qu'une entrée-sortie par commande. Il n'y a cette fois pas de duplication et donc pas de problème introduit en mise à jour. C'est donc une excellente technique qui peut cependant pénaliser les sélections sur l'une et l'autre des tables.

8.3. PARTITIONNEMMENT VERTICAL ET HORIZONTAL

Le **partitionnement vertical** consiste à diviser la table en deux par projection en répétant les clés.

Notion XVII.21 : Partitionnement vertical (*Vertical partitionning*)

Technique consistant à implémenter deux projections ou plus d'une table sur des schémas R1, R2… en répétant la clé dans chaque Ri pour pouvoir recomposer la table initiale par jointure sur clé.

Cette technique, qui prolonge en quelque sorte la normalisation, permet d'éloigner d'une relation tous les attributs peu fréquemment utilisés : ceux-ci sont reportés dans la deuxième table. La table fréquemment sélectionnée R1 devient plus petite, ce qui améliore les performances. La technique est donc très intéressante si un groupe d'attributs longs est rarement interrogé. Là encore, calculs analytiques et simulations aideront à prendre les bonnes décisions.

Le **partitionnement horizontal** consiste au contraire à diviser la table en sous-tables de même schéma, chacune conservant une partie des tuples.

Notion XVII.22 : Partitionnement horizontal (*Horizontal partitionning*)

Technique consistant à diviser une table en N sous-tables selon des critères de restriction σ1, σ2…σn.

Tout tuple respectant les contraintes d'intégrité doit appartenir à l'une des partitions. Un exemple typique est la création de partitions sur le mois pour la table des commandes. Ainsi, douze partitions sont créées, une par mois. Cette technique est particulièrement intéressante si les requêtes indiquent souvent le mois de la commande, donc en général un des σi. Elle permet surtout de diviser les grosses tables notamment pour le chargement et la sauvegarde. De nombreux SGBD l'implémentent de manière invisible au développeur d'application. Le partitionnement horizontal correspond de fait à une décomposition par union.

8.4. CHOIX DES INDEX

Le choix des index n'est pas un problème simple. En effet, les index permettent des gains impressionnants en interrogation, mais des pertes significatives en mise à jour. De plus, il existe différents types d'index, les plus fréquents étant les arbres B et les index bitmap pour l'OLAP.

Alors que faire ? Quelques règles simples sont utiles :

1. Toute clé primaire doit être indexée par un arbre B dès que la relation dépasse quelques pages. Cela évite les problèmes pour la vérification d'intégrité, la

recherche sur clé, etc. Les SGBD du marché automatisent de plus en plus ce type d'indexation.

2. Un attribut sur lequel figure un prédicat conjonctif avec égalité sera indexé :
   - par un arbre B si le nombre de valeurs possibles dépasse quelques centaines ;
   - par un index bitmap sinon.

Il ne faut cependant pas abuser de l'indexation pour les tables fréquemment mises à jour ou avec insertions nombreuses. Dans ce cas, un modèle analytique ou une simulation permettront d'y voir plus clair. On pourra consulter [Cardenas75, Schkolnick85] pour de tels modèles.

8.5. INTRODUCTION DE VUES CONCRÈTES

Mentionnons pour finir que l'introduction de vues concrètes pré-calculant les réponses aux questions fréquentes est une technique très intéressante pour l'optimisation. Elle est d'ailleurs de plus en plus utilisée dans les environnements OLAP, comme nous l'avons vu au chapitre IX traitant des vues. Bien sûr, les répercussions des mises à jour des relations de base sur la vue concrète peuvent être difficiles. Une vue concrète sera donc particulièrement intéressante lorsque les relations de base sont peu fréquemment mises à jour.

9. CONCLUSION

En résumé, concevoir une base de données nécessite tout d'abord d'élaborer un schéma conceptuel. Le modèle objet et le langage graphique de modélisation UML nous semble un bon véhicule pour ce faire. À partir de là, des règles précises permettent d'obtenir un schéma logique relationnel. Faut-il ensuite normaliser les schémas de tables ? Cela peut être utile en cas de mauvaise isolation des entités. Cependant, la normalisation est un processus long qui nécessite d'analyser les dépendances qu'on n'applique finalement qu'à l'exception. Trop systématiquement appliquée, elle conduit à éclater les tables en molécules de quelques attributs. Il apparaît alors des myriades de tables qu'il faut regrouper. L'optimisation est finalement beaucoup plus importante pour les performances, mais très difficile à maîtriser. C'est un métier de spécialiste de système.

La conception reste encore plus un art qu'une science, surtout avec le modèle objet. En effet, on est à peu près incapable de dire ce qu'est un bon schéma objet. De plus, le passage d'un schéma objet à un schéma « logique » objet-relationnel reste une étape mal maîtrisée. Les approches par un modèle sémantique de plus haut niveau sont très

intéressantes [Bouzeghoub91], mais restent théoriques. Il y a donc toujours de grands besoins en recherche sur la conception, notamment avec l'avènement des bases de données objet-relationnelles. La mode est plutôt aujourd'hui aux méthodes de conception globale pour les applications objet. Reste qu'il faut bien générer le schéma de la base. Les *patterns*, bibliothèques de cas typiques paramétrables, sont sans doute une voie d'avenir [Gamma97].

10. BIBLIOGRAPHIE

[Abiteboul95] Abiteboul S., Hull R., Vianu V., *Foundations of Databases*, Addison-Wesley, 1995.

Ce livre sur les fondements des bases de données couvre particulièrement bien la théorie des dépendances fonctionnelles, de jointure et d'inclusion, et bien d'autres aspects.

[Aho79] Aho A.V., Beeri C., Ullman J-D., « The Theory of Joins in Relational Databases », *ACM Transactions on Database Systems*, Vol.4, n° 3, p. 297-314, Sept. 1979.

Cet article étudie la décomposition d'une relation par projection et donne des algorithmes efficaces pour déterminer si la jointure de relations est sans perte en présence de dépendances fonctionnelles et multivaluées.

[Armstrong74] Amstrong W.W., « Dependency Structures of Database Relationships », *IFIP World Congress*, North-Holland Ed., p. 580-583, 1974.

Cet article introduit les fameux axiomes de Armstrong.

[Batini86] Batini C., Lenzerini M., « A Methodology for Data Schema Integration in the Entity-Relationship Model », *IEEE Transactions on Software Engineering*, vol. 10, n° 6, p.650-664, Nov. 1984.

Les auteurs présentent une méthodologie pour intégrer les schémas entité-association.

[Beeri79] Beeri C., Bernstein P.A., « Computational Problems Related to the Design of Norma Form Schemas », *ACM Transactions on Database Systems*, Vol.4, n° 1, Mars 1979.

Cet article décrit un algorithme linéaire pour tester si une dépendance fonctionnelle est dans la fermeture d'un ensemble de DF et une implémentation optimisée de l'algorithme de synthèse de Bernstein.

[Benci76] Benci E., « Concepts for the Design of a Conceptual Schema », *IFIP Conference on Modelling in Database Management Systems*, North-Holland Ed., p. 181-200, 1976.

Cet article collectif introduit une méthode de type entité-association pour modéliser des données au niveau conceptuel. C'est un des articles fondateurs de MERISE.

[Bernstein76] Bernstein P.A., « Synthesizing Third Normal Form Relations from Functional Dependencies », *ACM Transactions on Database Systems*, Vol.1, n° 4, p. 277-298, 1976.

L'auteur présente l'algorithme de synthèse de relations en 3e FN et ses les fondements.

[Bouzeghoub85] Bouzeghoub M., Gardarin G., Métais E., « SECSI: An Expert System for Database Design », *11th Very Large Data Base International Conference*, Morgan Kaufman Pub., Stockolm, Suède, 1985.

Cet article décrit le système SECSI basé sur un modèle sémantique appelé MORSE. MORSE supporte l'agrégation, la généralisation, l'association et l'instanciation. SECSI est construit selon une architecture système expert. C'est un outil d'aide à la conception de bases de données relationnelles qui transforme le modèle sémantique en relations normalisées. Pour passer au relationnel, SECSI applique les règles de transformation du modèle objet vues ci-dessus.

[Bouzeghoub91] Bouzeghoub M., Métais E., « Semantic Modeling of Object Oriented Databases », *Proc. of the 17th Intl. Conf. on Very large Data Bases*, Morgan Kaufman Ed., p. 3-14, Sept. 1991.

Cet article propose une méthode de conception de BD objet basée sur un réseau sémantique. La méthode a été implémentée dans un outil CASE.

[Bouzeghoub97] Bouzeghoub M., Gardarin G., Valduriez P., *Les Objets*, Éditions Eyrolles, Paris, 1997.

Ce livre couvre tous les aspects de l'objet et détaille les principales méthodologies de conception objet.

[Chen76] Chen P.P., « The Entity-Relationship Model – Toward a Unified View of Data », *ACM Transactions on Database Systems*, Vol.1, n° 1, p. 9-36, Mars1976.

Le fameux article introduisant le modèle E/R.

[Codd71] Codd E.F., « Normalized Database structure : A Brief Tutorial », *ACM SIGFIDET Workshop on Data Description, Access and Control*, p. 1-17, Nov. 1971.

Un état de l'art sur les trois premières formes normales.

[Codd72] Codd E.F., « Further Normalization of the Data Base Relational Model », in *Data Base Systems*, R. Rusin ed., Prentice-Hall, 1972.

Cet article introduit la BCNF. L'intérêt de cette forme normale fut plus tard contesté par Bernstein.

[Codd74] Codd E.F., « Recent Investigations in Relational Database Systems », *IFIP World Congress*, North Holland Ed., p. 1017-1021, 1974.

Un autre tutorial sur le relationnel et les formes normales.

[DeAntonellis83] De Antonellis V., Demo B., « Requirements Collection and analysis » in *Methodology and Tools for Database Design*, S. Ceri ED., North-Holland, 1983.

Cet article fait le tour de méthodes en matière de capture des besoins des utilisateurs.

[Delobel73] Delobel C., Casey R.G., « Decomposition of a Data Base and the Theory of Boolean Switching Functions », *IBM Journal of research and Devlopment*, vol. 17, n° 5, p. 374-386, 1973.

Cet article étudie les dépendances fonctionnelles comme des fonctions booléennes. Il introduit en particulier la notion de couverture minimale.

[Fagin77] Fagin R., « Multivalued Dependencies and a New Normal Form for Relational Databases », *ACM Transactions on Database Systems*, Vol.2, n° 3, p. 262-278, Sept. 1977.

Cet article introduit les dépendances multivaluées et la 4^e forme normale.

[Fagin79] Fagin R., « Normal Forms and Relational Database Operators », *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Boston, p.153-160, Juin 1979.

Cet article introduit les dépendances de jointures et la forme normale par projection jointure.

[Fagin81] Fagin R., « A Normal Form for Relational Databases Based on Domains and Keys », *ACM Transactions on Database Systems*, vol. 6, n° 3, p. 387-415, Sept. 1981.

Cet article introduit une nouvelle forme normale basée sur des définitions de domaines, les déclarations de clés et des contraintes généralisées exprimées par des règles logiques. Cette forme généralise les précédentes.

[Gamma97] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns*, Addisson-Wesley, Professional Computing Series, 1997.

Ce livre définit ce qu'est un pattern objet pour la conception et présente plus d'une vingtaine de patterns utiles.

[Hammer93] Hammer M., Champy J., *Le Reengineering*, Dunod, 1993.

Ce livre détaille les techniques de reengineering, en particulier le BPR qui permet de modéliser une société depuis le client jusqu'au processus de fabrication avec des diagrammes de flux d'information.

[Kettani98] Kettani N., Mignet D., Paré P., Rosenthal-Sabroux C., *De Merise à UML*, Éd. Eyrolles, Paris, 998

Ce livre présente UML, le langage de modélisation universel, le compare avec MERISE et propose une démarche de transition.

[Métais97] Métais E., Kedad Z., Comyn-Wattiau I., Bouzeghoub M., « Using Linguistic Knowledge in View Integration : Towards a Third Generation of Tools », *Data and Knowledge Engineering*, Nort-Holland, 1997.

Cette article propose une méthodologie d'intégration de vues fondée sur des connaissances linguistiques. Un outil d'intégration pouvant utiliser une ontologie en est dérivé.

[Muller98] Muller P.A., *Modélisation Objet avec UML*, Éd. Eyrolles, Paris, 1998.

Ce livre présente l'objet et le langage de modélisation associé UML. Il propose quelques exemples de modèles.

[Navathe84] Navathe S., El-Masri R., Sahidar T., « Relationship Merging in Schema Integration », *Proc. 10th Intl. Conf. on Very Large Data Bases*, Morgan Kaufman Ed., p. 78-90, Florence, Août 1984.

Cet article propose des méthodes d'intégration de schémas, notamment de fusion d'associations dans un modèle E/R.

[Nicolas78] Nicolas J-M., « Mutual Dependencies and Some Reults on undecomposable relations », *Proc. 5th Intl. Conf. on Very Large Data Bases*, IEEE Ed., p. 360-367, Berlin, 1978.

Cet article propose des dépendances mutuelles intermédiaires entre les dépendances multivaluées et de jointure. Il montre que des relations non décomposables en 2 peuvent l'être en 3.

[Rational98] Rational Soft., UML 1.1, Documentation du langage UML, http://www.rational.com/uml/.

La documentation en HTML et PDF de UML.

[Rissanen73] Rissanen J., Delobel C., « Decomposition of Files – A Basis for Data Storage and Retrieval », *IBM Research Report RJ 1220*, San José, Ca., Mai 1973.

Quelques-uns des premiers algorithmes de décomposition en 3e forme normale.

[Rissanen78] Rissanen J., « Theory of Relations for Databases – A Tutorial Survey », *7th Symposium on Math. Foundations of Computer Science*, Springer– Verlag ED., LCNS n° 64, p. 537-551, 1978.

Un tutorial sur la décomposition des relations, de la 1e à la 4e forme normale.

[Tardieu83] Tardieu H., Rochefeld A., Colletti R., *La Méthode Merise*, Éd. d'Organisations, 1983.

Le livre de base sur la méthode MERISE.

[Ullman88] Ullman J.D., *Principles of database and Knowledge-base Systems*, vol. I, Computer Science Press, Rockville, MD, 1988.

Le livre de Ullman couvre particulièrement bien la théorie de la décomposition et les dépendances généralisées par des tableaux.

[WonKim95] Won Kim, Choi I., Gala S., Scheevel M., « On Resolving Schematic Heterogeneity in Multidatabase systems », in *Modern Database Systems*, Won Kim ED., ACM Press, 1995.

Cet article est un tutorial des techniques de résolution conflits entre schémas à intégrer. Il discute particulièrement de l'intégration de schémas objet et relationnels dans un contexte de BD réparties fédérées.

[Yannakakis80] Yannakakis M., Papadimitriou C.H., « Algebraic dependencies », *Foundations of Computer Science Conference*, IEEE Ed., p. 328-332, 1980.

Cet article introduit les dépendances algébriques qui généralisent les dépendances de jointures.

[Zaniolo81] Zaniolo C., Melkanoff M.A., « On the design of relational Database Schemata », *ACM Transactions on Database Systems*,Vol.6, n° 1, p. 1-47, 1981.

Cet article introduit la 4^e forme normale et des algorithmes de normalisation basés sur des hypergraphes.

Chapitre XVIII

CONCLUSION ET PERSPECTIVES

1. INTRODUCTION

Trois générations de systèmes de bases de données ont déjà vu le jour. La première génération des années 70 correspondait aux modèles hiérarchique et réseau. Elle était représentée par des produits tels TOTAL, IDS II, IMS et Socrate. La seconde génération des années 80 était basée sur le modèle relationnel et fut conduite par les produits Oracle, DB2, Ingres, Informix et Sybase. La 3e génération des années 90 a vu l'intégration de l'objet aux systèmes de 2e génération. Bien que des systèmes purs objets tels O2 ou Object Store aient montré le chemin, l'industrie a procédé par extension, si bien que d'un point de vue industriel Oracle, DB2, Informix et SQL Server restent les produits phares maintenant de 3e génération.

Les principes des systèmes de bases de données sont souvent venus de la recherche au cours de ces trente dernières années. L'enjeu aujourd'hui est le développement de la génération de SGBD de l'an 2000. Celle-ci devrait voir l'intégration efficace du décisionnel aux systèmes transactionnels, le support transparent de l'Internet et bien sûr la possibilité de recherche par le contenu des objets multimédias sur le Web vu comme une grande base de données. Tous ces travaux sont déjà bien engagés dans les laboratoires de recherche et chez les constructeurs de SGBD, notamment aux USA. Nous résumons ci-dessous quelques aspects des recherches en cours qui nous paraissent essentiels pour le futur.

2. LES BD ET LE DECISIONNEL

La décennie 90 a vu le développement des entrepôts de données (*Datawarehouse*). Un entrepôt de données est un ensemble de données historisées variant dans le temps, organisé par sujets, agrégé dans une base de données unique, géré dans un environnement de stockage particulier, aidant à la prise de décision dans l'entreprise. Trois fonctions essentielles sont prises en compte par ces nouveaux systèmes décisionnels : la collecte de données à partir de bases existantes et le chargement de l'entrepôt, la gestion et l'organisation des données dans l'entrepôt, l'analyse de données pour la prise de décision en interaction avec les analystes.

La figure XVIII.1 illustre les composants d'un entrepôt de données. Le moniteur-adapteur est chargé de la prise en compte des mises à jour des sources de données locales, de la préparation de tables différentielles (les deltas) pour envois à l'entrepôt et du transfert des deltas périodiquement vers le médiateur. Ce dernier assure la fusion des sources et la mise en forme des données pour la base de l'entrepôt. Autour du *datawarehouse*, les outils OLAP (*On Line Analysis Processing*) permettent l'analyse des données historisées. Les outils de *data mining* permettent l'extraction de règles et de modèles à partir des données.

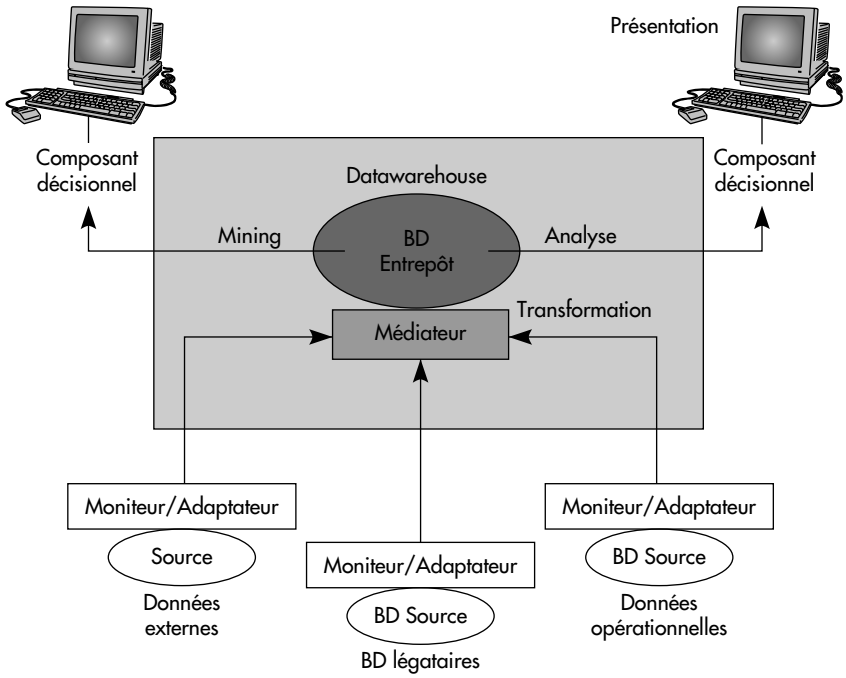


Figure XVIII.1 : Architecture d'un entrepôt de données

2.1. L'ANALYSE INTERACTIVE MULTIDIMENSIONNELLE (OLAP)

Le développement de l'analyse interactive de données (*OLAP*) a été basé sur l'utilisation de cubes multidimensionnels [Gray96]. Un cube permet la visualisation et l'analyse d 'une mesure selon trois axes. Par exemple, un chiffre d'affaire sera représenté dans un espace 3-D, en fonction du temps, des produits vendus et de la géographie. Ce cube de données (*datacube*) peut être manipulé par des opérations basées sur une algèbre des cubes de données composée d'opérateurs de tranches, extensions et agrégats (*slice, dice, rollup, drilldown).*

Les problèmes de performance dans le cadre de larges bases de données historisées sont nombreux : Que concrétiser dans l 'entrepôt ? Comment gérer des vues redondantes pour faciliter le calcul des cubes ? Comment passer du relationnel au multidimensionnel ? Et plus généralement comment concevoir la base de l'entrepôt ? Comment choisir les résumés stockés, les vues concrétisées, maintenir les méta-données [Mumick97]?

2.2. LA FOUILLE DE DONNÉES (*DATA MINING*)

Au-delà de l'analyse interactive multidimensionnelle, les techniques de fouille de données (*Data mining*) se sont répandues. Il s'agit d'un ensemble de techniques d'exploration de larges bases de données afin d'en tirer les liens sémantiques significatifs et plus généralement des règles et des modèles pour la compréhension et l'aide à la décision. Les domaines d'applications sont nombreux, par exemple l'analyse de risque, le marketing direct, la grande distribution, la gestion de stocks, la maintenance, le contrôle de qualité, le médical, l'analyse financière. L'approche consiste souvent à induire des règles avec des coefficients de vraisemblance à partir de large ensemble de données. Les techniques de base sont issues de l 'IA et de l'analyse de données (analyse statistique, modèles fonctionnels, réseaux de neurones, recherche de règles associatives, classification, segmentation, etc.). La problématique BD est le passage à l 'échelle, c'est-à-dire être capable de traiter quelques giga octets de faits ! Par exemple des indexes spécialisés (*bitmap*) et des échantillonnages contrôlés ont été proposés. Les techniques de découvertes de règles associatives ont été particulièrement développées [voir par exemple Agrawal93, Gardarin98]

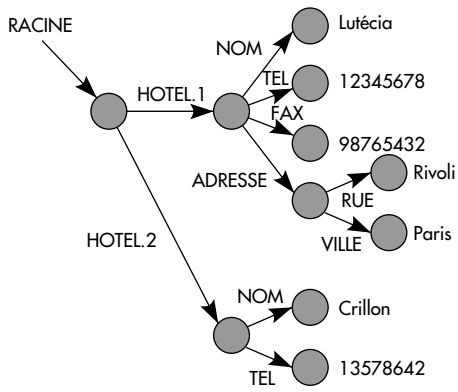
3. BD ET WEB

Le Web s'est développé comme un hypertexte sur le réseau Internet, pour permettre facilement l'accès à des fichiers chaînés. Rapidement, le besoin de couplage avec les

bases de données est apparu. Pourquoi coupler ? Trois raisons au moins motivent ce besoin : l'introduction du client-serveur à présentation universelle (architectures 3-tiers), la génération de sites Web dynamiques composés à partir de *templates* Html et de données extraites de bases, et le commerce électronique, qui nécessite la gestion de catalogues et de transactions en bases de données.

Il existe déjà de nombreuses solutions industrielles, plus ou moins issues de la recherche, telles Oracle Web, Web SQL de Sybase, LiveWire de Netscape, Visual Interdev de Microsoft, O2 Web, etc. Ces outils réalisent une intégration faible de deux modèles de données (le relationnel-objet et le modèle semi-structuré abstrait de Html). Ils sont insuffisants car ils ne permettent guère la transcription automatique de résultats de requêtes en Html et vice-versa.

De nombreux projets de recherche (par exemple, Tsimmis à Stanford [Abiteboul97], Strudel à ATT [Fernandez97], MiroWeb au PriSM en collaboration avec Osis et l'Inria) tendent à permettre le stockage direct d'hypermédia dans la base. XML paraît le standard adapté pour les bases de données semi-structurées. Les principaux projets proposent des modèles de représentation de documents XML et des langages d'interrogation associés. Les bases de données semi-structurées [Abiteboul96, Buneman97] modélisent les données par un graphe étiqueté, chaque nœud feuille pouvant correspondre à un objet externe, structuré ou non. Les étiquettes correspondent aux tags XML. La figure XVIII.2 illustre un document semi-structuré représenté sous la forme d'un graphe et en XML.




```
<RACINE>
    <HOTEL>
        <NOM> Lutécia </NOM>
        <TEL> 12345678 </TEL>
        <FAX> 98765432 </FAX>
        <ADRESSE>
            <RUE> Rivoli </RUE>
            <VILLE> Paris </VILLE>
        </ADRESSE>
    </HOTEL>
    <HOTEL>
        <NOM> Crillon </NOM>
        <TEL> 13578642 </TEL>
    </HOTEL>
</RACINE>
```

Figure XVIII.2 : Exemple de document semi-structuré

Le semi-structuré paraît bien adapté au Web. Il permet en effet la prise en compte de documents HTML/XML et facilite l'interrogation et la recomposition dynamique de partie de documents multimédias. Il autorise aussi la navigation intelligente dans une base de documents. Un document distribué est représenté par un réseau sémantique d'objets liés. Les liens correspondent à des arcs de composition ou d'association. La

grammaire des documents (DTD) peut être maintenue comme une nouvelle sorte de contrainte d'intégrité. La question d'intégrer le semi-structuré aux SGBD existants reste posée. Elle nécessite en particulier la capacité à découvrir des structures répétitives dans les documents, afin de les représenter sous forme d'extensions de types. Les langages SQL ou OQL doivent aussi être étendus pour manipuler les graphes d'objets semi-structurés, par exemple avec des parcours de chemins et des expressions régulières [Abiteboul97, Arocena98, Consens93, Fernandez97, Konopnicki95, Mendelzon96].

La réalisation d'adaptateurs capables de générer une vue semi-structurée de sources de données structurées ou non est aussi un problème d'actualité. Il faut non seulement comprendre la sémantique des sources afin de découvrir un peu de structure mais aussi assurer le passage du structuré au semi-structuré. Des vues partielles de sources sous forme de graphes étiquetées doivent être générées sur demande à partir de requêtes.

L'optimisation de requêtes mixant des données structurées et semi-structurées pose aussi des problèmes nouveaux. Il faudrait pouvoir disposer d'un modèle de coût pour données semi-structurées. La gestion de documents distribués (sur Intranet ou Internet) nécessite aussi des techniques d'optimisation originale pouvant allier les approches *push* et *pull*.

4. BD MULTIMÉDIA

Le multimédia est à la mode [Subrahmanian96]. Il est reconnu qu'une BD multimédia doit posséder cinq caractéristiques :

1. gérer des types de données multimédias incluant texte libre, géométrie, image, son, vidéo ;
2. offrir les fonctionnalités des bases de données, c'est-à-dire l'existence d'un langage d'interrogation non procédural permettant les recherches par le contenu, la persistance, la concurrence et la fiabilité ;
3. assurer la gestion de larges volumes de données, pouvant atteindre les péta-bases (10**15) ;
4. supporter des structures de stockage efficaces comme les *Quadtree*, les Rtree et leurs variantes, pour permettre la recherche rapide par le contenu ;
5. être capable de récupérer des informations à partir de sources hétérogènes.

L'interrogation d'objets multimédias passe par la recherche classique dans une BD structurée à partir d'attributs décrivant les objets (*exact-match retrieval*), mais surtout par la recherche basée sur le contenu des objets. Une requête typique est la recherche

des k objets les plus similaires à un objet donné. Là, il n'y a pas de garantie sur la correction et la précision des résultats. On récupère un ensemble de résultats classés par ordre de pertinence et interrogeable à nouveau (raffinement). Dans cet esprit, une extension de SQL3 avec des types de données abstraits spécifiques au multimédia est en cours de conception par un sous-groupe de l'ISO : SQL multimédia (SQL/MM). Il s'agit d'un projet international de standardisation dont l'objectif est de développer une librairie de types SQL pour les applications multimédias. SQL/MM est composé de diverses parties amenées à évoluer : *Full text*, *Graphic still*, *Animation*, *Image still*, *Full motion video*, *Audio*, *Spatial* 2D et 3D, *Music*. Tous ces types de données devraient être standardisés, en conjonction avec d'autres efforts (MPEG 7 par exemple).

Les thèmes de recherche liés au multimédia sont nombreux et sortent souvent du strict domaine des bases de données. Il s'agit en particulier de l'indexation automatique [Salton88], de l'extraction de caractéristiques (*features*), de l'évaluation de distances combinées entre objets, de la gestion de proximité sémantique, du développement de structures de stockage efficaces. L'optimisation de requêtes, les modèles de coûts, la formulation et l'évaluation de requêtes mixtes, l'intégration aux SGBD existants, la distribution et la recherche sur Internet/Intranet devraient aussi contribuer à la constitution de musées virtuels interrogeables par le contenu sur les grands réseaux.

5. CONCLUSION

Les bases de données ont connu une évolution douce vers l'objet, le standard étant aujourd'hui le relationnel-objet et SQL3. Les domaines les plus actifs sont le décisionnel, l'intégration avec le Web et le multimédia. Nous développons ces thèmes dans un ouvrage complémentaire. Les bases de données mobiles ne sont pas non plus à négliger et la conception, notamment d'entrepôt de données et de BD actives, restent des problèmes ouverts. Il ne faut pas non plus négliger les thèmes de recherche traditionnels où il y a encore beaucoup à faire (méthodes d'accès, concurrence, réparti, intégrité et vues, parallélisme).

Un sondage effectué récemment auprès de 20 chercheurs reconnus (comité de programme de CIKM) sur leurs domaines d'intérêt a donné les résultats indiqués figure XVIII.3. La note est un poids entre 0 et 20 (intérêt ou non).

Si l'on analyse les thèmes des articles des derniers VLDB et SIGMOD, on obtient des résultats sensiblement différents comme indiqués figure XVIII.4. Tout ceci montre à la fois la multiplicité, la diversité et l'ouverture de la recherche en bases de données. Les enjeux économiques sont très grands, ce qui explique la croissance du nombre de chercheurs (400 articles soumis au dernier VLDB, 220 à CIKM).

Semi-structured and Web Database	20
Heterogeneous and Distributed System	16
Query Languages and Query Processing	16
Application	15
DataWarehousing and Mining	14
Multimedia Databases	11
Database Design	10
Active & Rule Databases	9
Object Storage Methods	8
User and Application Interfaces	8
Imprecise and Uncertain Information	6
Parallel database systems	5
Transaction and Reliability	4
Tuning, Benchmarking and Performance	3
Privacy and Security Issues	3

Figure XVIII.3 : Sondage d'intérêt auprès de chercheurs

Multimedia Databases	19
DataWarehousing and Mining	18
Query Languages and Query Processing	9
Semi-structured and Web DB	7
Heterogeneous an Distributed Systems	7
Object Storage Methods	7
Transaction and Reliability	7
Active & Rule Databases	5
Tuning, Benchmarking and Performance	4
Application	3
Database Design	3
User and Application Interfaces	2
Parallel database systems	2
Imprecise and Uncertain Information	1
Privacy and Security Issues	1

Figure XVIII.4 : Analyse des thèmes des articles des derniers SIGMOD et VLDB.

Pour terminer, nous voudrions souligner la vertu des prototypes et des applications en bases de données. La réalisation de systèmes plus ou moins exploratoires (Socrate,

Syntex, Sabre, O2) a permis par le passé de constituer des équipes à masse critique de pointe. Elle permet aussi d'acquérir des connaissances, de les et maintenir et de le passer à de nouveaux chercheurs au sein d'un système. Il ne faudrait pas que la réduction et la dispersion des crédits conduisent à abandonner les grands projets au profit de publications souvent secondaires.

La réalisation d'applications à l'aide de prototypes avancés permet de cerner les véritables besoins et de découvrir des sujets neufs. De grandes expériences couplées au réseau (Intranet ou Internet) sont déjà en cours par exemple au Musée du Louvre et à Bibliothèque Nationale. Une meilleure participation de la recherche dans ces projets est souhaitable.

6. BIBLIOGRAPHIE

[Abiteboul97] Abiteboul S., « Querying semi-structured data », *Proc. Of the International Conference on Database Theory* (ICDT), Jan. 1997.

Une mise en perspective théorique des bases de données semi-structurées.

[Abiteboul97] Abiteboul S., Quass D., McHugh J., Widom J., Weiner J., « The Lorel Query Language for Semi-structured Data », *Journal of Digital Libraries*, vol. 1, n° 1, p. 68-88, April 1997.

Cet article présente le langage LOREL, extension de OQL pour les données semi-structurées.

[Agrawal93] Agrawal R., Imielinski T., Swami A. N., « Mining Association Rules between Sets of Items in Large Databases », *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, D.C., 1993 , p. 207-216.

Le premier article sur les règles associatives. Il propose la méthode Apriori pour extraire les règles associatives de grandes bases de données.

[Arocena98] Arocena G., Mendelzon A., « WebOQL : Restructuring documents, databases and Webs », in *Proc. IEE ICDE 98*, Orlando, Florida, Feb. 1998.

Cet article propose un langage d'interrogation pour le Web construit à partir d'OQL.

[Buneman97] Buneman P., « Semi-structured data », *Proc. ACM PODS'97*, Tucson, Arizona, USA, p. 117-121, 1997.

Cet article présente un tutoriel sur les données semi-structurées. En particulier le langage UnQL et les premières techniques d'optimisation, en particulier pour les expressions régulières, sont décrits.

[Consens93] Consens M., Mendelzon A., « Hy+ : A hygraph-based query and Vizualisation System », *SIGMOD Record, vol. 22*, n° 2, p. 511-516, 1993.

Un des premiers langages d'interrogation pour hypertexte.

[Gardarin98] Gardarin G., Pucheral P., Wu F., « Bitmap Based Algorithms For Mining Association Rules », rapport PriSM, *Proc. BDA'98*, Tunis, octobre 1998.

Un algorithme de découverte de règles associatives basé sur des index spécialisés, de type bitmap.

[Gray96] Gray J., Bosworth A., Layman A., Pirahesh H., « Data Cube : A relational aggregation operator generalizing group-by, cross-tab, and sub-total », *Proc. of the 12th IEEE Data Engineering Conf.*, p. 152-159, New-Orleans, Feb. 1996.

Un des premiers articles formalisant le datacube et l'introduisant comme une construction en SQL.

[Konopnicki95] Konopnicki D., Shmueli O., « W3QS : A Query System for the World Wide Web », *Proc. Of the 21st VLDB*, Zurich, Switzerland, 1995.

Un autre langage d'interrogation pour le Web.

[Mendelzon96] Mendelzon, Mihaila G., Milo T., « Querying the World-Wide Web », *Proc. Of 1st Int. Conf. On Parallel and Distributed Information Systems*, p. 80-91, Dec. 1996.

Une formalisation de langages d'interrogation pour le Web.

[Mumick97] Mumick S.I., Quass D., Mumick B.S., « Maintenance of Data Cubes and Summary Tables in a Warehouse », *in Proc. of ACM SIGMOD'97*, Sigmod Record n° 26, vol. 26, n° 2, p. 100-111, 1997.

Cet article propose une méthode pour maintenir des vues avec agrégats dans un entrepôt de données. Les mises à jour sont envoyées périodiquement sous forme de tables différentielles ou deltas. Tout d'abord, un algorithme efficace de propagation des mises à jour sur une vue est proposée. Puis, il est montré comment un ensemble de vues support de cubes de données peut être maintenu efficacement.

[Salton88] Salton G., Buckley C., « Term-weighting approaches in automatic text retrieval », *Information Processing & Management*, vol. 24, p. 513-523, 1988.

Cet article présente une méthode basée sur des matrices de fréquence de termes pour interroger les bases de données textuelles.

[Subrahmanian96] Subrahmanian V.S., Jajodia S. editors, *Multimedia Database Systems*, Springer-Verlag, 323 pages, Berlin, 1996.

Cet excellent livre sur le multimédia donne une vue d'ensemble des techniques proposées pour les BD images, textes, vidéo et audio. Il fait aussi un tour des techniques de stockage et de publications de documents multimédia.

EXERCICES

Préparés en collaboration avec

Yann Vièmont

Laboratoire PRiSM, UVSQ

Ces textes représentent un ensemble de sujets de partiels et d'examens écrits par Georges Gardarin et Yann Viémont soumis aux étudiants à Paris VI puis à Versailles entre 1980 et 2000. Nous nous excusons auprès de ces nombreux étudiants (ils sont aujourd'hui pour la plupart ingénieurs dans un domaine porteur) pour la difficulté et parfois le manque de clarté de certains de ces textes, mais les bases de données, ce n'est pas si facile...

Des versions de ces textes ont souvent été relues par des collègues que nous tenons ici à remercier, dont M. Bouzeghoub, M. Cheminaud, B. Finance, R. Gomez, M. Jouve, J. Madelaine, E. Métais, I. Paumelle, P. Pucheral, E. Simon, P. Testemale, J-M. Thévenin, P. Valduriez et K. Zeitouni.

Ces sujets ont parfois été repris comme texte de Travaux Dirigés (TD) et développés par ailleurs. Des corrigés de quelques-uns de ces exercices étendus et modifiés pourront être trouvés dans la série d'ouvrages *Les Bases de Données en Questions* aux éditions Hermès, par Mokrane Bouzeghoub, Mireille Jouve et Philippe Pucheral. Cependant, les annales qui suivent ont été adaptées et modernisées pour mieux correspondre aux chapitres qui précédent.

1. DISQUES MAGNETIQUES (chapitres I et III)

Le parallélisme entre un traitement par l'unité centrale et une entrée-sortie (E/S) disque permet la prise en compte de nouvelles requêtes pendant l'exécution d'une requête plus ancienne. Ainsi, devant chaque unité d'échange, le système est amené à gérer des files d'attente de requêtes. Les requêtes en attente peuvent être planifiées afin de minimiser les temps d'E/S disques. Nous considérons une unité de disques inamovibles à tête mobile des plus courantes.

Soit une file d'attente composée des cinq instructions de lecture suivantes, arrivées dans cet ordre :

1. Piste 20 – secteurs 11 à 14
2. Piste 30 – secteurs 41 à 42
3. Piste 40 – secteurs 40 à 43
4. Piste 30 – secteurs 05 à 08
5. Piste 53 – secteurs 03 à 07.

Question 1

Rappelez le fonctionnement d'une E/S disque et calculez le temps nécessaire en fonction de la vitesse de rotation et du temps de déplacement de bras unitaire. Fixez les paramètres à des valeurs correspondant aux technologies courantes.

Question 2

Quel est le temps nécessaire pour exécuter les requêtes en attente suivant leur ordre d'arrivée (stratégie FIFO) ?

Question 3

Déterminez les quatre meilleures permutations possibles des requêtes minimisant le temps d'E/S disques. Parmi ces quatre séquences, quelle est la meilleure ? Pourquoi ?

Question 4

Donnez le principe d'un algorithme permettant de trouver la séquence optimale. Quel doit être son temps de réponse maximal pour qu'il puisse être utilisé ?

Question 5

Calculez le temps nécessaire pour exécuter les cinq instructions d'E/S en attente en choisissant à chaque fois d'exécuter la requête la plus proche des têtes de lecture-écriture.

Question 6

L'utilisation de disques RAID permet d'accélérer les E/S disques. Discutez des différents types de RAID et de leur intérêt pour accélérer les cinq requêtes.

2. METHODES DE HACHAGE (chapitre III)

On considère un fichier haché de P paquets contenant N articles au total, décrivant des produits selon le format (Numéro, Nom, Type, Prix, Fournisseur, Quantité). Les articles sont composés de 80 octets en moyenne.

Question 1

Discutez le choix de la taille du paquet.

Proposez quelques fonctions de hachage afin de déterminer le numéro de paquet à partir du numéro de produit. Discutez de l'intérêt de ces fonctions selon le mécanisme d'attribution des numéros de produits.

Question 2

Lorsqu'un paquet est plein, un produit devant être inséré dans ce paquet est écrit en débordement. Proposez différentes solutions pour la gestion des débordements. Pour chacune d'elles, calculez le nombre d'entrées-sorties moyen nécessaires pour écrire un nouvel article et lire un article existant.

Question 3

On utilise le hachage extensible. Rappelez l'algorithme d'écriture d'un nouvel article et de lecture d'un article. Calculez le nombre d'entrées-sorties nécessaires pour écrire un nouvel article et lire un article existant.

Question 4

Même question avec le hachage linéaire.

3. MÉTHODES D'INDEXATION (chapitre III)

Les méthodes d'accès indexées des systèmes actuels sont toutes basées sur les arbres B et B+. L'objectif de cet exercice est d'étudier plus en détail ces méthodes.

Question 1

Rappelez la définition d'un arbre B et d'un arbre B+. Illustrez ces définitions en construisant un arbre B et un arbre B+ pour stocker l'alphabet dans des arbres d'ordre 2. Quel est l'intérêt d'un arbre B+ par rapport à un arbre B ?

Question 2

Calculez le nombre de comparaisons de clés nécessaires à la recherche d'une lettre dans un arbre B et B+ d'ordre 2. Ce nombre est-il différent pour un arbre d'ordre 3 ?

Généralisez les résultats à un arbre d'ordre m contenant n clés.

Question 3

Il est possible de comprimer les clés dans un arbre B ou B+ en enregistrant pour chaque clé seulement la différence par rapport à la précédente. Préciser alors le format de chaque entrée dans l'arbre B. Donnez les algorithmes de recherche et d'insertion d'une clé dans un arbre B ou B+ en prenant en compte la compression.

Question 4

VSAM implémente les arbres B+ en essayant de les calquer à la structure des disques. Discutez de l'intérêt de cette approche et de ses inconvénients. Calculez le nombre d'entrées-sorties nécessaires pour lire un article dans un fichier avec un seul niveau d'index maître. Même question pour les insertions d'articles.

Question 5

Une recherche dans un fichier simplement indexé par un arbre B sur une clé non discriminante (clé secondaire) peut être coûteuse. Pourquoi ? Proposez des approches pour réduire le temps d'entrées-sorties disques nécessaire.

4. ARBRES B+ (chapitre III)

On désire étudier les propriétés des arbres B+ pour les clés alphabétiques, pour les attributs de type chaîne de caractères de longueur variable (varchar). On définit ici un arbre B+ un peu particulier comme suit :

– les articles sont tous stockés dans les feuilles,
– lors d'un éclatement, la clé médiane est recopiée au niveau supérieur,
– le nombre d'articles dans une feuille ou le nombre de couples clé/pointeur dans un sommet intermédiaire est contrôlé par le taux de remplissage (en octets) qui doit rester compris entre 50 % et 100 %,
– la racine admet un taux de remplissage quelconque,
– la racine est soit une feuille soit un sommet non-feuille qui possède alors au moins deux fils.

On notera que le taux minimal de remplissage n'est qu'à peu près de 50 % à cause des problèmes de parité du nombre des articles lors des éclatements et à cause de la longueur variable des articles.

On supposera que les pages font 50 octets, qu'un pointeur (sur un sommet) est codé sur 4 octets, que les clés sont de longueur variable, que les articles se composent de la

clé + une information associée sur 10 octets + 2 octets qui servent de séparateurs et que les couples clé/pointeur n'utilisent qu'un séparateur. On négligera dans les calculs l'information de contrôle en tête des pages qui permet de connaître le nombre d'articles ou de couples clé/pointeur, etc.

Par exemple l'article de clé "Marcel" est représenté par "Marcel#xxxxxxxxxx#" soit 18 octets. Le couple clé/pointeur ("Marc", sommet yyyy) est représenté par "Marc#yyyy" soit 9 octets.

Question 1

Donnez l'arbre obtenu en partant d'une racine vide après insertion des articles de clé :

{Marcel, Marius, Martine, Maurice, Marcelle, Maude, Marc, Marguerite, Mathilde, Maxime, Marielle, Martial, Mariette, Marina, Matthieu, Mattias} dans l'ordre indiqué.

Question 2

Donnez l'arbre obtenu si la liste des articles est triée selon l'ordre des clés :

{Marc, Marcel, Marcelle, Marguerite, Marielle, Mariette, Marina, Marius, Martial, Martine, Mathilde, Matthieu, Mattias, Maude, Maurice, Maxime}.

Question 3

Proposez une procédure de construction de l'arbre B+ par construction directe des feuilles et des sommets intermédiaires à partir d'un fichier trié des articles.

Question 4

On remarque que dans un arbre B+ les clés dans les sommets non-feuilles ne servent que d'aiguilleurs dans la recherche d'un article de clé donnée. On utilise cette remarque pour augmenter le nombre de couples clé/pointeur dans les sommets non-feuilles en remplaçant les clés par un préfixe de ces clés.

Donnez la règle qui permet de déterminer le plus petit préfixe possible qui conserve la propriété d'aiguiller correctement la recherche dans l'arbre.

Question 5

Donnez l'arbre obtenu en utilisant des préfixes par insertions répétées comme dans la question 1.

Question 6

Discutez les avantages et les inconvénients de cette méthode pour de petits fichiers et pour de grands fichiers.

5. UTILISATION D'INDEX SECONDAIRES (chapitre III)

Un index secondaire est un index sur une clé de recherche (dite clé secondaire) construit de telle sorte que l'information associée à la clé soit l'adresse de l'enregistrement (ou tuple en relationnel) et non pas l'enregistrement lui-même. Cette adresse peut être soit directement une adresse relative de tuple dans le fichier le contenant, soit indirectement la clé (dite primaire) primaire d'un autre index permettant de retrouver ce tuple. Dans ce dernier cas, que nous utiliserons ici, une recherche sur clé secondaire consiste à d'abord rechercher la clé secondaire dans l'index secondaire, à obtenir l'information associée c'est-à-dire la clé primaire, puis à rechercher le tuple dans l'index primaire à l'aide de la valeur trouvée.

Un tel index est dit non-plaçant car les tuples correspondants à deux valeurs successives de clé secondaire dans l'index ne se trouvent pas en séquence dans le fichier les contenant, ni non plus en général dans les mêmes pages de disques (contrairement à l'index primaire dit plaçant).

Soit R (A, B, C, D, E) une relation à cinq attributs dont A est une clé unique. On suppose que la relation est placée, par un index primaire, sur l'attribut A en utilisant un arbre B^+ à quatre niveaux (dont le dernier niveau contient les tuples).

Deux index secondaires sur C et E sont disponibles. Ces index sont également organisés en arbre B^+. Chaque index secondaire est composé de trois niveaux.

R comprend 1 000 000 de tuples et donc autant de valeurs différentes pour A. Les tuples font en moyenne 100 octets, les pages de disques 1 K octets. Avec un taux de remplissage de 75 %, le dernier niveau de l'arbre B^+ occupe donc environ 130 000 pages de disque. Il y a 10 000 valeurs différentes pour C et 1 000 pour E. On suppose de plus que toutes les distributions des attributs X sont (1) uniformes entre une valeur MIN_X et MAX_X et (2) indépendantes.

Question

Quelle est la meilleure méthode et le nombre correspondant d'entréees/sorties disques nécessaires pour répondre aux recherches suivantes :

Q1. (A = "valeur")

Q2. ("valeur" < A)

Q3. (C = "valeur")

Q4. (B = "valeur")

Q5. ("valeur 1" < C < "valeur 2").

Q6. (C = "valeur 1") ET (E = "valeur 2")

6. OPTIMISATION ET INDEX SECONDAIRES (chapitre III)

La recherche dans les bases de données s'effectue selon des critères multiples, du type :

```
<critère simple> {AND | OR } <critère simple> ...
```

où chaque critère simple est lui-même du type :

```
<attribut> <comparateur> <valeur>.
```

Les comparateurs peuvent être =, <, > , ≥, ≤. Les expressions de AND et OR peuvent être encadrées de parenthèses pour indiquer les priorités.

Par exemple, on recherchera dans un fichier de vins tous les articles satisfaisant :

```
(CRU = "Volnay" OR CRU = "Chablis")
AND DEGRÉ ≥ 12 AND MILLÉSIME = 2000.
```

Question 1

On suppose le fichier des vins indexés par un arbre B sur la clé secondaire `DEGRÉ`. Proposez un format d'entrée pour un tel index. Justifiez votre solution.

Les degrés étant nombreux car continus entre 0 et 20, comment peut-on réduire leur nombre ?

Proposez un algorithme permettant de répondre aux requêtes précisant `DEGRÉ` > $v, où $v désigne un réel compris entre 0 et 20.

Question 2

Ce fichier est aussi indexé sur les clés secondaires `CRU` et `MILLÉSIME`. Proposez trois méthodes pour résoudre des requêtes du type `CRU` = $c `AND MILLÉSIME` = $m, l'une utilisant les deux index, les deux autres un seul ($c et $m sont deux constantes). Essayer d'estimer le coût en entrées-sorties de chacune d'elles. Donnez des heuristiques simples pour choisir l'une ou l'autre.

Question 3

De manière générale, proposez un algorithme capable d'évaluer efficacement des recherches avec critères multiples conjonctifs (AND).

Question 4

Étendre l'approche au cas de critères multiples connectés avec des AND et des OR.

Question 5

Un fichier haché peut aussi être indexé selon plusieurs attributs. Comment peut-on améliorer l'algorithme précédent pour supporter aussi des fichiers hachés ?

7. HACHAGE MULTI-ATTRIBUTS (chapitre III)

Le hachage multi-attributs consiste à combiner plusieurs fonctions de hachage H1, H2, ... Hn sur des champs différents A1, A2, ..., An des articles d'un fichier, ceci afin de calculer l'adresse virtuelle d'un paquet. En version statique simple, le numéro du paquet est simplement égal à H1(A1) || H2(A2) ... Hn(An), où || désigne l'opérateur de concaténation. En version extensible, les bits de poids faible de cette chaîne peuvent être retenus pour adresser le répertoire des paquets. D'autres solutions plus sophistiquées sont possibles, par exemple en permutant sur les fonctions de hachage pour prélever des bits constituant l'adresse du répertoire. Dans la suite, on suppose un fichier haché statique avec adressage des paquets par concaténation simple des fonctions de hachage.

Question 1

On désire placer un fichier de lignes de commandes de format (`NC`, `NP`, `TYPE`, `QUANTITÉ`, `PRIX`) par hachage multi-attributs. NC désigne le numéro de commande, NP le numéro de produit et TYPE le type du produit. Sachant que 50 % des recherches se font sur NC, 20% sur NP et 30% sur le type de produit, proposez des fonctions de hachage optimum afin de placer un tel fichier dans 100 paquets.

Question 2

Proposez un algorithme général pour répondre aux requêtes multicritères précisant deux des attributs hachés parmi les trois, par exemple NC et TYPE.

Question 3

Généralisez l'algorithme pour traiter des requêtes multicritères avec AND et OR, comme vu à l'exercice précédent.

Question 4

Les fichiers hachés peuvent aussi être indexés par des arbres B. Discutez des formats d'adresses d'articles intéressants pour gérer les index en conjonction au hachage. Intégrez les résultats de l'exercice précédent à l'algorithme de la question 3. En déduire un algorithme général pour traiter des requêtes multicritères sur des fichiers hachés sur plusieurs attributs et indexés.

8. INDEX BITMAPS (chapitre III)

Les index *bitmap* sont adaptés aux attributs ayant un nombre limité de valeurs (a_0, a_1, ...a_n) pour un attribut A. Un index *bitmap* est une table de bits à deux dimensions. Le bit (i, j) est à 1 si le i^e article du fichier à la valeur a_i pour l'attribut A.

Question 1

Précisez le format d'un index *bitmap* et des structures associées. Calculez la taille d'un tel index. Proposez des méthodes de compression pour un tel index.

Question 2

Donnez les algorithmes d'insertion et suppression d'un enregistrement dans un fichier avec index *bitmap*.

Question 3

Un index *bitmap* sur un attribut A est idéal pour accélérer des recherches sur critère de la forme :

$$A = \$a_0 \text{ OR } A = \$a_1 \text{ OR } \ldots A = \$a_n.$$

Précisez l'algorithme de recherche.

Question 4

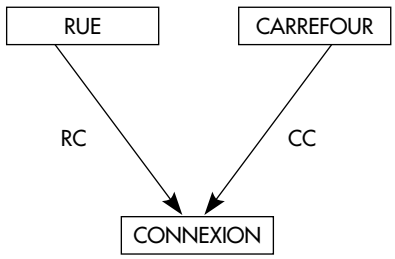
On peut gérer plusieurs index *bitmap* pour un même fichier, sur des attributs A, B, etc. Précisez les types de critères pouvant être traités par des index *bitmap*.

Question 5

On considère un fichier avec index *bitmap* et index secondaires sous la forme d'arbre B. Que doit référencer une entrée dans un index secondaire pour être combinables avec les index *bitmap* ? Donnez un algorithme général de recherche multicritères prenant en compte les index secondaires et les index *bitmap*.

9. MODÈLE RÉSEAU (chapitre IV)

On désire faire circuler des camions encombrants dans une ville pour livrer un client. Soit le schéma suivant de la base de données urbaine dans le modèle réseau :



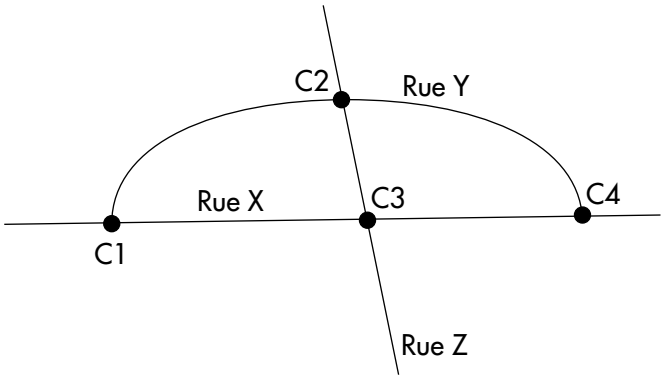
Une rue possède un nom, une longueur, une largeur minimum, un revêtement et un nombre de voies. Un carrefour possède un nom et un indicateur de présence de feu tricolore. Une connexion indique la présence éventuelle d'un sens interdit.

Question 1

Donnez le schéma CODASYL correspondant à ce diagramme.

Question 2

Donnez le graphe des occurrences correspondant au plan simplifié, à trois rues (X, Y, Z) et quatre carrefours (C1, C2, C3, C4), suivant :



Question 3

Donnez le programme en DML CODASYL permettant de trouver tous les carrefours avec feu de la rue Albert Einstein.

Question 4

Donnez le programme en DML CODASYL permettant de trouver toutes les rues qui sont directement accessibles depuis la rue Albert Einstein (donc qui l'intersectent où s'y raccordent). Une rue n'est pas accessible si elle est en sens interdit.

Question 5

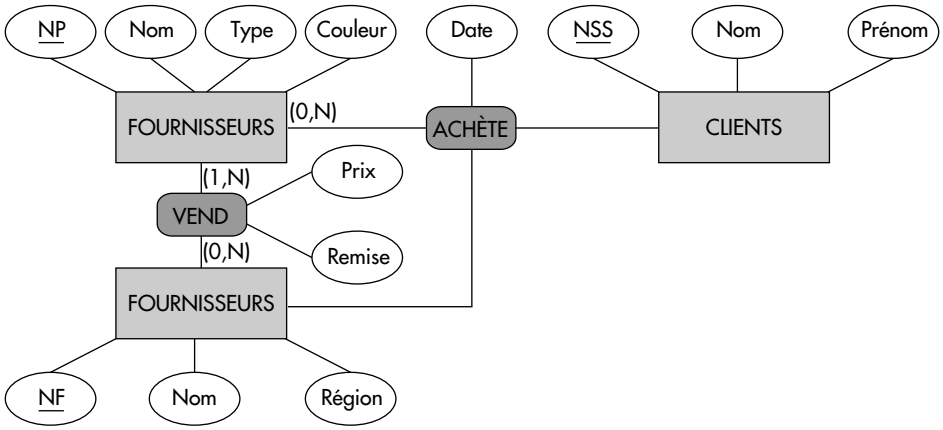
Sachant que les poids lourds partent d'un dépôt situé dans la rue A, doivent rejoindre le client situé rue B et ne doivent circuler que dans des rues de largeur supérieure à 10 mètres, donner le programme en DML CODASYL et pseudo-code qui recherche tous les itinéraires possibles en ne les donnant qu'une fois avec leur longueur et le nombre de feux. On précise que la partie pseudo code devra rester à un niveau de quelques blocs de commentaires et devra résoudre le problème de la détection des boucles.

Question 6

Proposez un placement des types d'article dans un ou plusieurs fichiers pour accélérer les performances. Donnez le nombre d'entrées-sorties disque nécessaires pour traiter la question 4.

10. DU RÉSEAU AU RELATIONNEL (chapitre IV et VI)

On considère la base de données dont le schéma entité-association est représenté ci-dessous :



Cette base décrit des produits vendus par des fournisseurs à des clients.

Question 1

Donnez le schéma réseau en DDL CODASYL correspondant au diagramme entité-association représenté figure 1. On précise que les entités `PRODUITS` et `FOURNISSEURS` sont rangées dans un même fichier `FOU` via le set `VEND`, alors que les instances de `CLIENTS` sont rangées dans le fichier `CLI`.

Question 2

Écrire les programmes en DML CODASYL correspondant aux requêtes suivantes :

1. Nom des fournisseurs vendant des produits de type « informatique ».
2. Nom des clients ayant acheté de tels produits.

3. Noms des produits et liste des fournisseurs associés ayant vendu au client de N°SS 153300017012.
4. Chiffre d'affaires total réalisé avec le client Dupond Jules.

Utiliser du pseudo-Pascal ou pseudo-C si nécessaire.

Question 3

Donnez un schéma relationnel pour la base de données représentée.

Question 4

Exprimer en SQL les requêtes correspondant aux questions étudiées au 2.

Question 5

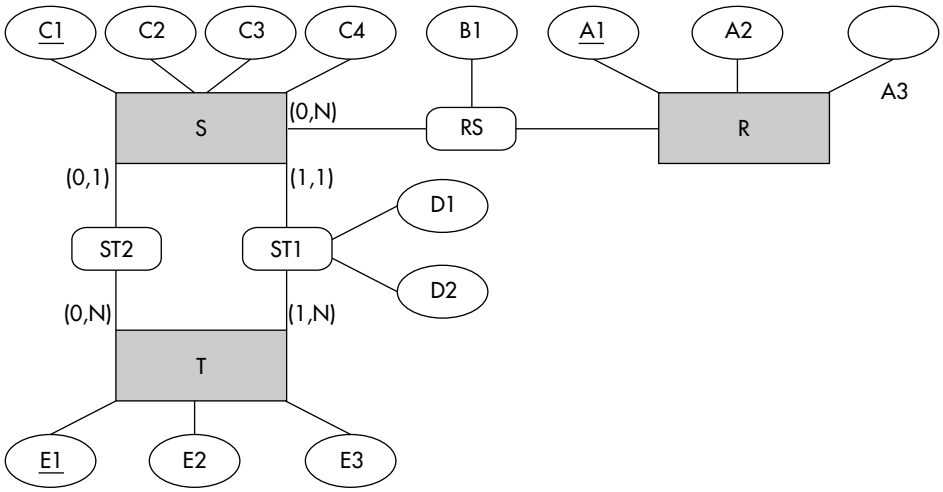
Pour les schémas représentés, proposez un algorithme de traduction automatique de requête SQL de type restriction-projection-jointure en programme DML CODASYL. Pour cela, on traduira chaque requête en un arbre d'opérations de l'algèbre relationnelle optimisé de manière adéquate ; ensuite, on traduira des groupes d'opérateurs choisis en programmes DML avec du pseudo-PASCAL ou pseudo-C.

Question 6

Étudiez le ré-engineering de la base réseau proposée en base relationnelle. Discutez des difficultés. Proposez une démarche.

11. CALCULS DE DOMAINES ET DE TUPLES (chapitre V)

Soit le schéma entité-association représenté figure 1, modélisant une base de données décrivant des entités R, S et T reliées par les associations RS, ST1 et ST2. Tous les attributs sont de type entier. Les clés des entités sont soulignées ; elles correspondent simplement au premier attribut des entités. Les cardinalités minimum et maximum des associations sont indiquées sur les branches correspondantes. Ainsi, un tuple de R est associé par l'association RS à au moins 0 et au plus n tuples de S ; réciproquement, un tuple de S est associé par l'association RS à au moins 0 et au plus n tuples de T ; un tuple de T correspond par l'association ST1 à un et un seul tuple de S ; etc.



Question 1

Donnez le schéma de la base de données relationnelle représentant directement ce diagramme entité-association (une entité générant une relation, une association générant une relation), avec les clés et les contraintes référentielles nécessaires.

Question 2

Exprimer en calcul de tuples les requêtes suivantes :

1. Donnez tous les attributs de S pour les tuples dont l'attribut C3 est compris entre 100 et 200.
2. Donnez les attributs C1 et C2 de S, E1 et E2 de T tels que les tuples de S et T soient associés par un tuple de ST1 d'attribut D1 supérieur à 10.
3. Même question, mais on souhaite en plus qu'il existe un tuple de R correspondant à chaque tuple de S sélectionné, via l'association RS, ayant un attribut A2 positif.
4. Donnez les tuples de T associés par RS à tout tuple de R et au moins à un tuple de T par ST1 ou ST2.

Question 3

Exprimer ces mêmes requêtes en calcul de domaines.

Question 4

Exprimer ces mêmes requêtes en QBE.

Question 5

Exprimer ces mêmes requêtes en SQL.

12. ALGEBRE RELATIONNELLE (chapitre VI)

Soit la base de données **Sécurité Routière** comprenant les tables :

```
PERSONNE    (N°PERS, NOM, PRÉNOM, ADRESSE)
VÉHICULE    (N°VEH, MARQUE, TYPE)
CONDUCTEUR  (N°PERS, N°VEH, NBACC)
ACCIDENT    (N°ACC, DATE, DÉPT)
VÉHPART     (N°ACC, N°VEH, N°COND)
BLESSÉ      (N°ACC, N°PERS, N°VEH, GRAVITÉ)
```

En définissant le schéma, les hypothèses suivantes ont été faites :

– Les clés sont soulignées.

– Les relations PERSONNE et VÉHICULE ont les significations évidentes.

– La relation CONDUCTEUR associe les personnes et les véhicules et mémorise le nombre d'accidents auxquels a participé un conducteur donné au volant d'un véhicule donné.

– La relation ACCIDENT donne les informations globales d'un accident.

– La relation VÉHPART (véhicule participant) est définie de sorte que chaque véhicule impliqué dans un même accident donne un tuple avec le même numéro d'accident. L'attribut N°COND est le numéro de la personne conduisant le véhicule au moment de l'accident.

– La relation BLESSÉ indique tous les blessés d'un accident (un par tuple), y compris le conducteur si celui-ci a été blessé.

– L'attribut GRAVITÉ peut prendre les valeurs 'Bénigne', 'Légère', 'Sérieuse', 'Grave', 'Fatale'.

Question 1

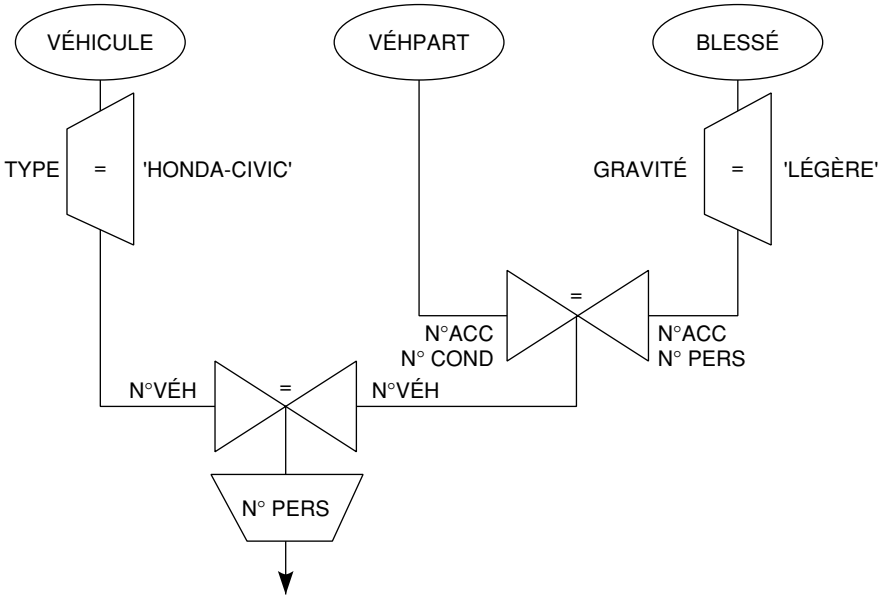
Exprimer sous la forme d'un arbre de calcul de l'algèbre relationnelle les requêtes suivantes :

a) Donnez le nom, le prénom, la date de l'accident pour chaque personne blessée fatalement dans un accident du département 75 dans une voiture Citroën.

b) Trouver les personnes qui ont été blessées dans tous les accidents où elles étaient conductrices.

Question 2

Que retrouve l'arbre algébrique suivant ?



Question 3

Calculez le coût en entrées/sorties disques de l'arbre de la question précédente avec les hypothèses suivantes :

– La relation VÉHICULE comporte 1 000 000 tuples dont 850 sont des Honda Civic; elle est placée par hachage statique sur le N°VÉH ; elle possède un index secondaire sur le TYPE qui est un arbre B^+ à 3 niveaux.

– Les index secondaires du système utilisé retournent la clé primaire (ou clé de placement) et non pas directement l'adresse physique.

– La relation VÉHPART comporte 100 000 tuples ; elle est placée sous forme d'arbre B^+ sur le N°ACC (index primaire).

– La relation BLESSÉ comporte 40 000 tuples dont 10 000 sont des blessés légers ; elle est placée en séquentiel avec un facteur de blocage (moyen) de 20 tuples par page.

– Les jointures sont faites par l'algorithme des boucles imbriquées.

– Tous les résultats intermédiaires tiennent en mémoire centrale ; une page de mémoire correspond à un bloc de disque.

Question 4

Que devient le coût des entrées/sorties si la mémoire est limitée à *m* pages ?

13. OPERATEURS RELATIONNELS (chapitre VI)

On désire réaliser un évaluateur d'opérations de l'algèbre relationnelle sur un gestionnaire de fichiers avec index secondaires. Les opérations de l'algèbre relationnelle considérées sont les opérations classiques (restriction, projection, jointure, union, différence, intersection) avec en plus deux opérateurs spécialisés pour les calculs d'agrégats (groupage et calcul de fonction). Les expressions d'attributs sont aussi étendues avec les calculs arithmétiques (par exemple A*B+C est utilisable en argument de projection, A, B et C étant des attributs simples).

L'opérateur de groupage (nommé GROUP ou NEST) a pour argument un ensemble d'attributs X sur lequel est réalisé le groupage et un ensemble d'attributs à grouper Y. Il s'écrit formellement $\nu_{X/Y}(R)$. L'opération de groupage des attributs Y de la relation R sur les attributs X construit une relation non en première forme normale (NF^2) de schéma XY ; pour chaque valeur de X, l'ensemble des valeurs de Y correspondant à cette valeur de X est répertorié. Par exemple, soit la relation :

R	A	B
	1	5
	1	7
	2	8
	3	8

$\nu_{A/B}(R)$ est la relation suivante :

$\nu_{A(B^*)}(R)$	A	{B}
	1	{5,7}
	1	{8}
	2	{8}

L'opérateur de calcul de fonction applique simplement une fonction sur ensemble à une colonne d'une relation valuée par un ensemble. Les fonctions considérées sont MIN, MAX, AVG, SUM, COUNT qui sont respectivement le minimum, le maximum,

la moyenne, la somme et le compte. Par exemple, l'application de l'opérateur MIN_B à la relation précédente conduit au résultat suivant :

$\nu_{A(B^*)}(R)$	A	{B}
	1	{5,7}
	1	{8}
	2	{8}

Afin d'illustrer, on considère une base de données décrivant un stock de jouets de type A, B, C ou D livrés par des fabricants en une certaine quantité à une certaine date. Le schéma de la base est le suivant :

```
JOUETS(NJ,NOMJ,TYPE,PRIX)
FABRIQUANTS(NF,NOMF,VILLE,ADRESSE)
LIVRAISONS(NF,NJ,DATE,QUANTITE)
```

Question 1

Exprimez en algèbre relationnelle étendue les questions suivantes sur la base de données des jouets :

a) Donnez la liste des fabricants qui ont livré au moins un jouet de type A et de prix supérieur à 1 000 F ainsi que le nom des jouets correspondants livrés.

b) Donnez la liste des fabricants qui n'ont pas livré de jouets.

c) Donnez la somme des quantités livrées par chaque fabricant (caractérisé par `NOMF` et `ADRESSE`).

Question 2

On se propose d'optimiser l'algorithme d'intersection de deux relations R1 et R2. Proposez trois algorithmes permettant de réaliser cet opérateur dans les cas sans index, respectivement basés sur :

a) le produit cartésien des deux relations ;

b) le tri des deux relations ;

c) le hachage des relations avec tableaux de bits.

En supposant qu'il existe trois tampons d'une page en mémoire, que les relations comportent respectivement r1 et r2 pages ($r1 < r2$) et que la relation intersection est composée de i pages, calculez approximativement le coût en E/S de chaque algorithme. On supposera que les tableaux de bits tiennent en mémoire et qu'ils permettent d'éliminer un pourcentage b des tuples de la deuxième relation.

Question 3

On se propose d'étudier l'algorithme de groupage d'une relation R. Proposez trois algorithmes permettant de réaliser cet opérateur dans les cas sans index, respectivement basés sur :

a) la comparaison des tuples de la relation (proche du produit cartésien) ;

b) le tri de la relation ;

c) le hachage de la relation avec comparaisons internes aux paquets.

En supposant qu'il existe trois tampons d'une page en mémoire, que la relation comporte r pages et que la relation groupée est composée de g pages, calculez approximativement le coût en E/S de chaque algorithme.

Question 4

Exprimer la question suivante en SQL sur la base de données des jouets :

Donnez les chiffres d'affaires de chaque fabricant (caractérisé par son NOMF et son ADRESSE) de poupées (NOMJ LIKE « Poupée »).

Exprimer cette question en algèbre relationnelle étendue.

Donnez un arbre d'opérateurs d'algèbre relationnelle optimisé permettant de calculer la réponse à cette question. On ajoutera pour cela l'opérateur de groupage (noté par un rectangle) aux opérateurs classiques.

14. LANGAGES DE REQUÊTES (chapitres V, VI et VII)

Soit la base suivante décrivant un supermarché (les clés sont soulignées) :

```
RAYON (NOMR, ÉTAGE)
ARTICLE(RÉFÉRENCE, TYPE, DESCRIPTION, COULEUR)
DISPONIBILITÉ(NOMR, RÉFÉRENCE, QUANTITÉ)
EMPLOYÉ(NUMERO, NOME, SALAIRE, RAYON, RESPONSABLE)
```

Les rayons identifiés par la clé NOMR sont situés à un étage donné. Les articles sont référencés par l'entier RÉFÉRENCE et décrit par un type, une description et une couleur. Ils sont disponibles en une certaine quantité à chaque rayon. Un employé travaille à un rayon et a pour responsable un autre employé. L'attribut responsable représente donc un numéro d'employé.

Question 1

Proposez sous la forme d'un graphe un ensemble de clés étrangères pour cette base de données.

Question 2

Exprimer en algèbre relationnelle puis en SQL les requêtes suivantes :

a) références des articles de type 'électroménager' en rayon au second étage,

b) nom des employés travaillant dans un rayon proposant des articles de type jouet pour une quantité totale supérieure à 1 000.

Question 3

Exprimer en SQL les requêtes suivantes :

c) nom des employés qui gagnent plus que leur responsable,

d) étages qui ne vendent que des vêtements.

Question 4

Exprimer en calcul de tuple puis en SQL la requête suivante :

e) nom et salaire du responsable le mieux payé.

15. LANGAGES DE REQUÊTES (chapitres V, VI et VII)

Soit le schéma relationnel suivant du Windsurf-Club de la Côte de Rêve (WCCR) :

```
SPOT (NUMSPOT, NOMSPOT, EXPOSITION, TYPE, NOTE)
PLANCHISTE (NUMPERS, NOM, PRENOM, NIVEAU)
MATOS (NUMMAT, MARQUE, TYPE, LONGUEUR, VOLUME, POIDS)
VENT (DATE, NUMSPOT, DIRECTION, FORCE)
NAVIGUE (DATE, NUMPERS, NUMMAT, NUMSPOT)
```

La base du WCCR comprend cinq relations et les clés sont soulignées. Les `SPOTS` sont les bons coins pour faire de la planche, avec un numéro, leur nom, l'exposition principale par exemple 'Sud-Ouest', le type par exemple 'Slalom' ou 'Vague', et une note d'appréciation globale. Les `PLANCHISTE`s sont les membres du club et les invités, les niveaux varient de 'Débutant' à 'Compétition'. Le `MATOS` (jargon planchiste pour matériel) comprend la description des planches utilisées (pour simplifier les voiles, ailerons, etc., ne sont pas représentés). Le `VENT` décrit la condition moyenne d'un spot pour une date donnée. Enfin `NAVIGUE` enregistre chaque sortie d'un plan-

chiste sur un spot à une date donnée avec le matos utilisé. Pour simplifier, on suppose qu'un planchiste ne fait qu'une sortie et ne change pas de matos ni de spot dans une même journée.

Question 1

Indiquez les clés étrangères éventuelles de chaque relation.

Question 2

Donnez les expressions de l'algèbre relationnelle qui permettent de calculer les requêtes suivantes :

a) Nom des planchistes de niveau 'Confirmé' qui ont navigué le '20/07/99' sur un spot où le vent moyen était supérieur à force 4 sur une planche de moins de 2,75 m.

b) Nom des planchistes qui ne sont pas sortis le '20/07/99'

c) Nom des planchistes qui ont essayé tous les types de planches de la marque 'FANA-BIC'. On suppose que tous les types sont dans la relation MATOS.

Question 3

Donnez les expressions de calcul de tuples correspondant aux trois requêtes précédentes.

Question 4

Donnez les ordres SQL correspondant aux requêtes suivantes :

c) Nom des planchistes qui ont essayé tous les types de planches de la marque 'FANA-BIC'. On suppose que tous les types sont dans la relation MATOS.

d) Pour chaque spot de la base, en indiquant son nom, donner le nombre de jours de vent au moins de force 4 pour l'année 94.

e) Pour chaque marque de matériel représentée par au moins 10 planches, indiquer le nombre d'utilisateurs 'Confirmé' ou 'Expert'.

16. LANGAGE SQL2 (chapitre VII)

Soit la base de données touristique suivante (les clés sont soulignées) :

```
STATION (NUMSTA, NOMSTA, ALTITUDE, REGION)
HOTEL (NUMHOT, NOMHOT, NUMSTA, CATEGORIE)
CHAMBRE (NUMHOT, NUMCH, NBLITS)
RESERVATION (NUMCLI, NUMHOT, NUMCH, DATADEB, DATEFIN, NBPERS)
CLIENT (NUMCLI, NOMCLI, ADRCLI, TELCLI)
```

Les clients réservent des chambres dans des hôtels en station. On note que pour une réservation de plusieurs personnes (couple ou famille) un seul nom de client est enregistré. De plus une réservation porte sur une seule chambre (pour une famille dans deux chambres il faudra deux tuples dans réservation).

Exprimer en SQL les questions suivantes :

Question 1

Donnez le nom des clients et le nombre de personnes correspondant pour les réservations de l'hôtel "Bellevue" à "Courchevel".

Question 2

Pour chaque station de Haute-Savoie, donner le nombre de lits en catégorie "trois étoiles".

Question 3

Pour chaque station de Haute-Savoie, donner le nombre de chambres réservées le samedi 11/02/95.

Question 4

Quels sont les noms des hôtels de catégorie "deux étoiles" de "Méribel" qui sont complets la semaine du 12/02/2000 au 18/02/2000 ?

Question 5

Quelles sont les régions dont toutes les stations sont à plus de 1500 m d'altitude ?

Question 6

Quels sont les clients qui sont allés dans toutes les stations du "Jura" ?

17. SQL2 ET LES JOINTURES (chapitre VII)

Soit la base de données suivante :

```
FREQUENTE (Buveur, Bar)
SERT (Bar, Vin)
AIME (Buveur, Vin)
```

Tous les attributs sont de type chaînes de caractères notamment un bar, un buveur ou un vin.

Exprimer les questions suivantes en SQL :

1) Donnez les buveurs qui fréquentent un bar servant du Beaujolais nouveau.
2) Donnez les buveurs qui aiment au moins un vin servit dans un bar qu'ils fréquentent.
3) Donnez les buveurs qui n'aiment aucun des vins servis dans les bars qu'ils fréquentent.

18. SQL2 ET LES AGRÉGATS (chapitre VII)

On considère la base de données relationnelle représentée figure 1 correspondant approximativement au benchmark TPC/D. Elle stocke des commandes de numéro NUMCO passées par des clients de numéro NUMCLI décrits dans la table CLIENTS. Chaque commande est caractérisée par un état, un prix, une date de réception, une priorité et un responsable. Une commande est composée de lignes, chaque ligne étant caractérisée par un numéro NUMLIGNE et correspondant à un produit de numéro NUMPRO, commandé à un fournisseur NUMFOU. Les produits sont commandés en une certaine quantité pour un prix total PRIX, avec une remise en pourcentage (REMISE) et un taux de TVA (taxe). Les produits sont définis par un nom, une marque, un type et une forme, ainsi qu'un prix unitaire de base. Fournisseurs et clients sont des personnes décrites de manière standard. PRODFOURN est une table associative associant produit (NUMPRO) et fournisseur (NUMFOU) en indiquant pour chaque lien la quantité disponible et un commentaire libre.

```
COMMANDES (NUMCO, NUMCLI, ETAT, DATE, PRIORITÉ, RESPONSABLE)
LIGNES(NUMCO, NUMLIGNE, NUMPRO, NUMFOU, QUANTITÉ, PRIX, REMISE, TAXE)
PRODUITS (NUMPRO, NOM, MARQUE, TYPE, FORME, PRIXUNIT)
FOURNISSEURS (NUMFOU, NOM, ADRESSE, NUMPAYS, TELEPHONE, COMMENTAIRE)
PRODFOURN (NUMPRO, NUMFOU, DISPONIBLE, COMMENTAIRE)
CLIENTS (NUMCLI, NOM, ADRESSE, NUMPAYS, TELEPHONE, COMMENTAIRE)
PAYS (NUMPAYS, NOM, CONTINENT)
```

Exprimer en SQL les questions suivantes :

a) Calculez le prix TTC de chaque commande.
b) Calculez le nombre de produits (rubriques) par chaque fournisseur.
c) Donnez les produits vendus par tous les fournisseurs.
d) Donnez les produits vendus par tous les fournisseurs dans tous les pays.
e) Donnez les noms et adresses des clients allemands ayant passé des commandes de produits de type "CD" à des fournisseurs français.
f) Calculez les recettes effectuées au travers de ventes de fournisseurs à des clients du même pays, c'est-à-dire les recettes résultant de ventes internes à un pays, et ceci pour tous les pays d'Europe pendant une année commençant à la date D1.

19. SQL2 ET OPTIMISATION (chapitres VII et X)

Soit le schéma relationnel de la Société Française d'Archéologie (fictive) :

```
OBJET (NUM-OBJ, DESCRIPTION, TYPE, DATATION, NUM-VILLE, NUM-SITE,
   NUM-MUSEE)
VILLE (NUM-VILLE, ANCIEN-NOM, NOM-ACTUEL)
MUSEE (NUM-MUSEE, NUM-VILLE, NOM)
SITE (NUM-VILLE, NUM-SITE, DESCRIPTION, CIVILISATION)
PUBLICATION (NUM-PUB, TITRE, DATE, EDITEUR)
AUTEUR (NUM-AUT, NOM, PRENOM)
COOPERATION (NUM-AUT, NUM-PUB)
REFERENCE (NUM-PUB, NUM-OBJ)
```

Cette base gère des objets archéologiques et des publications sur ces objets. La relation `OBJET` décrit les objets proprement dits avec l'indication de leur type (par exemple "vase"), de leur datation qui est une année, du site où ils ont été découverts et du musée où ils se trouvent actuellement. La relation `VILLE` comprend deux noms pour simplifier (ce qui par exemple exclut le cas `BIZANCE-CONSTANTINOPLE-ISTAMBUL`). La relation `SITE` indique la ville à laquelle se rattache le site et un numéro qui est un numéro d'ordre pour cette ville : toutes les villes ont un site 1, un site 2, etc. La civilisation du site est une grande catégorie comme "romaine" ou "crétoise". Les clés sont soulignées.

Question 1

Exprimer en SQL les requêtes suivantes sur la base :

Q1. Quelles sont les frises du troisième siècle ? Donnez leur numéro et leur description.

Q2. Même question avec en plus le nom du musée où elles sont exposées.

Q3. Noms et prénoms des auteurs d'ouvrage(s) référençant des objets de la civilisation Dorienne.

Q4. Quelles statues sont exposées dans la ville où elles ont été découvertes ? Donnez le numéro et la description.

Q5. Donnez le nom actuel des villes (s'il en existe) dont le nom actuel est le même que le nom ancien d'une *autre* ville.

Q6. Donnez le nombre de statues trouvées dans chaque site de la civilisation phénicienne.

Q7. Quels sont les sites d'Athènes qui ont fourni plus d'objets que le site 5 n'en a fourni ?

Q8. Noms et prénoms des auteurs des ouvrages qui référencent tous les objets trouvés dans le site 2 de Thèbes.

Question 2

Soit la requête SQL suivante :

```
SELECT P.TITRE, P.DATE
FROM   PUBLICATION P, AUTEUR A, COOPERATION C,
       REFERENCE R, OBJET O, MUSEE M
WHERE  A.NOM = 'Vieille'
AND    A.PRENOM = 'Pierre'
AND    A.NUM-AUT = C.NUM-AUT
AND    C.NUM-PUB = P.NUM-PUB
AND    P.EDITEUR = 'Éditions archéologiques modernes'
AND    P.NUM-PUB = R.NUM-PUB
AND    R.NUM-OBJ = O.NUM-OBJ
AND    O.TYPE = 'Mosaïque'
AND    O.NUM-MUSEE = M.NUM-MUSEE
AND    M.NOM = 'Le Louvre'
```

Proposez deux arbres algébriques différents pour exécuter cette requête. Le premier arbre sera optimisé au mieux en utilisant les heuristiques de restructuration algébrique et le second sera le pire possible.

Question 3

On suppose qu'il y a dans la base 10 000 publications, 100 éditeurs distincts, 1 000 auteurs de noms différents (pas d'homonymes), 2000 coopérations, 100 000 objets, 200 types d'objets différents, 1 000 000 de références et 100 musées de noms différents (pas d'homonyme).

On suppose de plus que toutes les distributions sont uniformes et indépendantes.

En prenant comme unité de coût la comparaison pour les sélections et les jointures, en supposant qu'il n'y a pas d'index et en admettant que toutes les jointures se font par produit cartésien, calculer le coût d'exécution des deux arbres que vous avez proposé.

Question 4

Si vous étiez autorisé à créer un seul index pour accélérer cette requête, lequel choisiriez-vous ? Pourquoi ?

20. INTÉGRITÉ DES DONNÉES (chapitre VIII)

L'objet de l'étude est de concevoir un composant logiciel capable de contrôler les contraintes d'intégrité lors des mises à jour d'une base de données relationnelles. Le composant doit être intégré dans un SGBD qui gère des relations différentielles. Lors de l'exécution des commandes de mise à jour (insérer, modifier, supprimer), deux

relations différentielles sont associées à chaque relation R : la relation des tuples supprimés R^- et celle des tuples insérés R^+. Un tuple modifié donne naissance à deux tuples, l'un dans R^-, l'autre dans R^+. En fin de transaction validée, les tuples de R^- sont enlevés de R et ceux de R^+ ajoutés à R. Le SGBD réalise : $R = (R - R^-) \cup R^+$.

En guise d'illustration, on utilisera la base composée des tables suivantes :

```
PLAGE (NP, NOMP, TYPE, REGION, POLLUTION)
NAGEUR (NN, NOM, PRENOM, QUALITE)
BAIGNADE (NN, NP, DATE, DUREE).
```

La relation PLAGE modélise les plages de France, de nom `NOMP`, de type `TYPE` (galets, sable ou rocher) ayant un taux de pollution donné (attribut `POLLUTION`). La relation `NAGEUR` mémorise les nageurs de qualité excellente, bonne ou médiocre. La relation `BAIGNADE` décrit les bains effectués par les nageurs.

Question 1

Définir en SQL2 les contraintes d'intégrité suivantes :

1. La qualité d'un nageur est excellente, bonne ou médiocre (contrainte de domaine).
2. Toute baignade a été effectuée par un nageur existant dans la base sur une plage existante (contraintes référentielles).
3. Le type et la région d'une plage déterminent sa pollution de manière unique (dépendance fonctionnelle).
4. La somme des durées des baignades d'un nageur par jour ne doit pas excéder deux heures.

Question 2

Précisez quels types de contraintes d'intégrité peuvent être vérifiées après chaque ordre de mise à jour (`INSERT`, `DELETE`, `UPDATE`) ceux qui nécessitent d'attendre la fin de transaction.

Question 3

Proposez des algorithmes pour traiter chaque type de contraintes d'intégrité, soient exécutés après la mise à jour, soient en fin de transaction. Montrer qu'il est possible de combiner les deux types de vérification.

Question 4

Les contraintes avec agrégats comme la somme des durées des baignades sont coûteuses à vérifier à chaque mise à jour. Proposez une méthode basée sur des données redondantes afin de réduire ce coût.

21. VUES ET TRIGGERS (chapitres VIII et IX)

Soit la base de données CINÉMA suivante :

```
FILM(NUMF, TITRE, DATE, LONGUEUR, BUDGET, RÉALISATEUR, SALAIRER)
GÉNÉRIQUE(FILM, ACTEUR, ROLE, SALAIRE)
PERSONNE (NUMP, FNOM, LNOM, DATENAIS, SEXE, NATIONALITÉ, ADRESSE,
        TÉLÉPHONE)
ACTEUR (NUMA, AGENT, SPÉCIALITÉ, TAILLE, POIDS)
CINÉMA(NUMC, NOM, ADRESSE, TÉLÉPHONE, COMPAGNIE)
PASSE (NUMF, CINÉMA, SALLE, DATEDEB, DATEFIN, HORAIRE, PRIX)
SALLE (CINÉMA, NUMS, TAILLEÉCRAN, PLACES)
```

NUMF, NUMP, NUMA, NUMC, NUMS, sont des identifiants uniques (clés primaires) pour respectivement : FILM, PERSONNE, ACTEUR, CINÉMA, SALLE.

Tout nom de relation utilisé comme attribut est une clé étrangère qui renvoie à l'identifiant (clé primaire) de la relation correspondante, par exemple dans GÉNÉRIQUE, FILM correspond à NUMF de FILM et est défini sur le même domaine.

RÉALISATEUR dans FILM et NUMA dans ACTEUR sont définis sur le domaine des NUMP.

Le numéro de salle NUMS est un numéro local pour chaque cinéma (Salle 1, 2, 3, …).

Question 1

Donnez la définition de FILM en SQL2 en précisant les contraintes de domaine, de clé (primaire ou candidate), et de clé étrangère.

Question 2

Même question pour GÉNÉRIQUE si on suppose qu'un acteur peut jouer plusieurs rôles dans un même film.

Question 3

Exprimer la contrainte généralisée suivante : "Le budget d'un film doit être supérieur à la somme des salaires des acteurs jouant dans ce film"

Question 4

Écrire un déclencheur qui met à jour automatiquement le numéro de film dans toutes les relations où il est utilisé si ce numéro est modifié dans la relation FILM.

On ajoute un attribut NBSALLES a FILM qui indique le nombre de salles où le film est actuellement visible.

Question 5

Utiliser des déclencheurs pour gérer automatiquement ce nombre de salles.

22. TRIGGERS ET VUES CONCRÈTES (chapitre VIII et IX)

On considère la base de données relationnelles :

```
PRODUITS (NP, NOMP, QTES, PRIXU)
VENTES (NP, NC, NV, QTEV)
```

La relation produit décrit des produits en stock de numéro NP, de nom NOMP, en quantité QTES, dont le prix de vente unitaire est PRIXU. La relation VENTES décrit les ventes réalisées ; celles-ci sont numérotées par client. QTEV est la quantité vendue pour le produit NP au client NC lors de la vente NV. Pour simplifier, on supposera que les prix des produits ne changent pas.

Question 1

Définir les vues `VENTEPRO` (`NC,NP,NV`, `QTEV`, `PRIXU`) et `VENTETOT` (`NC`, `PRIXT`) spécifiées comme suit :

– `VENTEPRO` donne pour chaque client, pour chaque produit et pour chaque vente la quantité de produit vendu et le prix unitaire du produit correspondant.

– `VENTETOT` donne pour chaque client le montant total en francs (`PRIXT`) des ventes effectuées.

On écrira les questions SQL permettant de créer ces vues à partir des relations de base.

Montrer que la vue `VENTETOT` peut être définie à partir de la vue `VENTEPRO`. Donnez sa définition en SQL à partir de `VENTEPRO`.

Question 2

On introduit un opérateur **partitionnement** représenté par un rectangle préparant l'application du GROUP BY sur une relation, utilisable dans les arbres relationnels ; cet opérateur partitionne horizontalement la relation sur les attributs paramètres en effectuant simplement un tri sur ces attributs : il s'agit donc en fait d'un opérateur de tri. En addition, on permet l'usage de fonctions agrégats et arithmétiques dans les projections. Appliquées aux relations partitionnées suite à l'opérateur précédent, les fonctions agrégats accomplissent les calculs d'agrégats selon le dernier partitionnement effectué.

Donnez l'arbre relationnel optimisé permettant de retrouver les clients ayant acheté pour plus de 10 000 F de produits avec la liste des produits qu'ils ont achetés.

Question 3

La modification de questions sur des vues (avec ou sans agrégat) ne permettant pas toujours d'optimiser, une autre méthode d'implantation de vues peut être basée sur la matérialisation de la vue comme une relation implantée sur disque (vue concrète). Le problème qui se pose est alors de mettre à jour la vue concrète lors des mises à jour des relations de base. Une technique possible consiste à générer des déclencheurs (*triggers*).

Exprimer en SQL les déclencheurs permettant de maintenir les relations `VENTEPRO` et `VENTETOT` lors d'insertion d'un nouveau produit ou d'une nouvelle vente dans la base.

De manière générale, préciser les règles à générer pour maintenir une vue lors d'insertion dans une relation de base. On pourra considérer les cas suivants :

1. La vue est sans agrégat, c'est-à-dire issue de projection, jointure et restriction des relations de base.
2. La vue est avec agrégat.

Question 4

Donnez les règles permettant de maintenir les relations `VENTEPRO` et `VENTETOT` lors d'une suppression d'un produit ou d'une vente dans la base.

De manière générale, préciser les règles à générer pour maintenir une vue sans agrégat, puis avec agrégat, lors d'une suppression dans une relation de base. Discutez selon les cas. Préciser les attributs qu'il est nécessaire de rajouter à la vue afin d'être capable de gérer correctement les suppressions.

23. GESTION DE VUES VIRTUELLES (chapitre IX)

Une vue est un ensemble de relations déduites d'une base de données, par composition des relations de la base. Dans la norme SQL la notion de vue a été réduite à une seule relation déduite. Les mises à jour sont généralement limitées aux vues dont les projections portent sur une seule relation. Soit la base de données `VITICOLE` composée des relations suivantes :

```
BUVEURS (NB, NOM, PRENOM, VILLE, AGE)
VINS(NV, CRU, REGION, MILLESIME, DEGRE)
ABUS(NB, NV, DATE, QUANTITE)
```

Celles-ci décrivent des buveurs identifiés par l'attribut numéro `NB`, des vins identifiés par l'attribut numéro `NV` et des consommations (`ABUS`) de vins identifiés par le numéro de buveur, le numéro de vin et la date de consommation.

Question 1

Donnez les commandes SQL permettant de créer les vues suivantes :

```
BUVEURSB (NB, NOM, PRENOM, NV, DATE, QUANTITE)
```

décrivant les buveurs de Beaujolais,

```
VINSB (NV, CRU, MILLESIME, DEGRE)
```

décrivant les vins de Beaujolais bus par au moins un buveur.

Il est précisé que la vue `VINSB` est mettable à jour et que l'on souhaite la vérification des tuples insérés dans la vue par rapport au reste de la base.

Question 2

Un utilisateur ayant droit d'interroger à partir des vues précédentes pose la question suivante :

« Donnez le nom des buveurs ayant bu du Beaujolais de millésime 1983 en quantité supérieure à 100, le même jour ».

Exprimez cette question telle que doit le faire l'utilisateur en SQL. Exprimez également la question modifiée portant sur les relations `BUVEURS`, `VINS` et `ABUS` que traitera le système.

Question 3

De manière générale, proposez en quelques boîtes, sous la forme d'un organigramme, un algorithme permettant de transformer une question posée sur une vue en une question exprimée sur la base.

Question 4

À partir de l'exemple, découvrir et énoncer quelques problèmes soulevés par la mise à jour à travers une vue référençant plusieurs tables :

1. ajout d'un tuple dans BUVEURSB ;
2. suppression d'un tuple dans BUVEURSB.

24. LANGAGES ET OPTIMISATION (chapitres V, VI et X)

Soit la base de données composée des relations :

```
VILLE(NOMV, POP, NDEP)
DEPARTEMENT(NDEP, NOMD, REGION)
SPECIALITE(NOMV, NOMP, TYPE)
```

où:

NOMV = nom de ville,
POP = population,
NDEP = numéro de département,
NOMD = nom de département,
REGION = nom de région,
NOMP = nom de produit,
TYPE = type de produit.

Soit l'expression algébrique suivante référençant les relations de cette base de données ($\bowtie$ représente la jointure naturelle) :

$\pi_{NOMV,NOMP}(\sigma_{REGION="Auvergne"}(\sigma_{POP>10000}$ (VILLE $\bowtie$ DEPARTEMENT $\bowtie$ SPECIALITE))) –

$\pi_{NOMV,NOMP}(\sigma_{NOMP="Fromage"}$ (VILLE $\bowtie$ DEPARTEMENT $\bowtie$ SPECIALITE)))

Question 1

Exprimez cette requête :

1) en calcul relationnel de domaine;

2) en calcul relationnel de tuple;

3) en SQL.

Question 2

Représentez la question sous forme d'un arbre d'opérations algébriques.

Proposez un arbre optimisé par restructuration algébrique pour cette question.

Question 3

Soit v, d et s les tailles respectives en tuples des relations VILLE, DEPARTEMENT et SPECIALITE. En supposant l'uniformité des distributions des valeurs d'attributs, calculez une approximation de la taille du résultat de la question précédente en nombre de tuples. On pourra introduire tout paramètre supplémentaire jugé nécessaire.

25. OPTIMISATION ET CHOIX D'INDEX (chapitre X)

Soit une base de données relationnelle composée des trois relations :

```
R1 (A1,A2,A3,A4,A5)
R2 (B1,B2,B3,B4,B5)
R3 (C1,C2,C3,C4,C5)
```

On suppose pour simplifier qu'il n'existe pas d'index.

On considère la requête SQL :

```
SELECT A2,B3,C4
FROM R1, R2, R3
WHERE A1 = B1
AND B2 = C1
AND A3 = a
AND C3 = c
```

où a et c désignent des constantes.

Question 1

Donnez les arbres algébriques relationnels optimisés par les heuristiques classiques des BD relationnelles (descente des projections et restrictions) permettant d'exécuter cette question.

Question 2

On choisit l'arbre qui effectue les jointures dans l'ordre R1 avec R2 puis avec R3. En considérant l'utilisation d'un algorithme de jointure par produit Cartésien (boucles imbriquées), calculez le temps d'exécution en nombre d'E/S de cet arbre en fonction de la taille en page des relations R1, R2 et R3, du nombre moyen de tuples par page T, du nombre de valeurs distinctes et non distinctes des attributs utilisés. On supposera pour cela une distribution uniforme des valeurs d'attributs. Tout paramètre supplémentaire jugé nécessaire pourra être introduit.

Question 3

Proposez un ensemble d'index plaçant et non plaçant optimaux pour la base de données et la question considérée.

Calculez alors le temps d'exécution de la question.

On pourra supposer pour simplifier que tous les index sont à deux niveaux et on négligera les temps de calcul.

26. OPTIMISATION ET JOINTURES (chapitre X)

Soit une base de données relationnelle composée des tables suivantes :

```
R3 (U1, U2, U3)
R2 (V1, V2)
R1 (T1)
```

U1 et V1 sont respectivement les clés uniques de R3 et R2. Tous les attributs sont des entiers tirés aléatoirement.

Question 1

On considère alors la question :

```
SELECT U2,V2
FROM R1, R2, R3
WHERE R3.U1 = R2.V2 AND R2.V1 = R1.T1 AND R3.U3 = 100.
```

Donnez tous les arbres relationnels effectuant les projections dès que possible permettant d'exécuter cette question.

Question 2

Calculez le coût en nombre d'entrée-sortie des trois arbres effectuant la restriction d'abord dans le cadre d'un SGBD relationnel implémentant les restrictions par balayage et les jointures par parcours d'index ou boucles imbriquées. On précise qu'il existe un index sur les clés U1 de R3 et V1 de R2 et pas d'autre index. La taille de chaque relation R1, R2 et R3 est respectivement r1, r2 et r 3. On supposera une distribution uniforme des valeurs d'attributs. Le nombre de valeurs distinctes d'un attribut est obtenu par la fonction DIST (par exemple, DIST(U2) est le nombre de valeurs distinctes de U2). Ce nombre peut être calculé.

Question 3

Même question avec des jointures par hachage.

Question 4

Avec un algorithme par hachage ou par boucles imbriquées, il est possible de commencer une jointure avant d'avoir fini la précédente. Expliquez ces algorithmes dits de type *pipeline*.

Dans quel cas est-il intéressant d'utiliser un algorithme de type *pipeline* ?

Question 5

On généralise ce type de requêtes à N relations Rn, Rn-1,...R1. Donnez la forme générale d'une requête. Calculez le nombre de plans possibles pour une telle requête.

27. INDEX DE JOINTURES (chapitre X)

Soit deux relations U(U1, U2, …) et V(V1, V2, …). Afin d'accélérer les équi-jointures de U et V sur les attributs U1 et V1, on propose de gérer un index de jointures. Celui-ci peut être vu comme une relation UV(IDU, IDV) où IDU représente l'identifiant d'un tuple de U et IDV l'identifiant d'un tuple de V. Chaque tuple de UV donne donc les identifiants de deux tuples de U et V qui appartiennent à la jointure. Un identifiant est une adresse invariante qui permet de retrouver un tuple d'une relation en une entrée-sortie. Afin de permettre un accès rapide aux tuples de UV à partir des identifiants des tuples de U ou de V, l'index de jointure est organisé comme un fichier grille (*Grid file*) sur les attributs IDU et IDV (c'est-à-dire qu'il est organisé par hachage extensible multi-attributs sur IDU et IDV, l'adresse de hachage étant obtenue en tournant sur les bits des deux fonctions de hachage sur IDU et sur IDV).

Question 1

Proposez un algorithme en pseudo-code, utilisant au mieux l'index de jointure, permettant de répondre à la question (bêta est une constante et il n'y a pas d'index sur V2) :

```
SELECT U.U2
FROM U, V
WHERE U1 = V1 AND V2 = beta
```

Calculez le coût en nombre d'entrée-sortie de cet algorithme en fonction des tailles de U et de V (respectivement u et v tuples), de la sélectivité de la jointure (j), de la taille de l'identifiant (i octets), de la taille du paquet de hachage pour UV (1 paquet = 1 page de p octets) et du taux de remplissage moyen du paquet (dénoté t).

Question 2

On suppose que le système gère un index de U sur U1 et de V sur V1 à la place de l'index de jointure. Détailler l'algorithme pour répondre à la question précédente en utilisant la jointure par index. Calculez le coût d'une telle jointure. On précise que :

1. le nombre d'octets occupés par chaque valeur de U1 ou U2 est dénoté a ;
2. le nombre de valeurs distinctes de U1 est DIST(U1), celui de U2 est DIST(U2) ;
3. les index sont des arbres B+ à deux niveaux (feuille + racine).

Comparer à la solution index de jointure vue en 1.

28. DU RELATIONNEL A L'OBJET (chapitre XI)

L'hôpital X a besoin de votre aide pour exprimer des requêtes sur leur base de données. On rappelle le schéma de leur base relationnelle :

```
SERVICE          (CODE-S, NOM-S, BATIMENT, DIRECTEUR)
SALLE            (CODE-S, NUM-SALLE, SURVEILLANT, NB-LITS)
EMPLOYEE         (NUM-E, NOM-E, PRENOM-E, ADR, TEL)
DOCTEUR          (NUM-D, SPECIALITE)
INFIRMIER        (NUM-I, CODE-S, ROTATION, SALAIRE)
MALADE           (NUM-M, NOM-M, PRENOM-M, ADR, TEL, MUTUELLE)
HOSPITALISATION (NUM-M, CODE-S, NUM-SALLE, LIT, DIAGNOSTIC)
SOIGNE           (NUM-D, NUM-M)
```

En définissant le schéma les hypothèses suivantes ont été faites :

– Les clés sont soulignées.

– Un service possède deux clés, `CODE-S` et `NOM-S`.

– Le directeur d'un service est un docteur identifié par son `NUM-D`.

– Le numéro de salle est local à un service (i.e., chaque service possède une salle numéro 1).

– Un(e) surveillant(e) est un(e) infirmier(ère) identifié(e) par son `NUM-I`.

– Docteurs et infirmiers(ères) sont des employés, `NUM-D` et `NUM-I` sont donc également des `NUM-E` et les tuples correspondants doivent exister dans `EMPLOYEE`.

– Un(e) infirmier(ère) est affecté(e) à un service et à un seul.

– Les docteurs ne sont pas affectés à un service particulier.

– La relation `HOSPITALISATION` ne concerne que les malades hospitalisés à l'état courant.

– Un malade non hospitalisé peut toujours être suivi par son (ses) médecin(s) comme patient externe.

Question 1

Donnez en UML la définition d'un schéma objet équivalent à cette base relationnelle. Exprimez ce schéma en ODL.

Question 2

Puisque DOCTEUR et INFIRMIER sont des EMPLOYEE, proposez un nouveau schéma objet utilisant l'héritage pour la base hospitalière.

Question 3

Exprimez en OQL la requête suivante : “Nom et prénom des cardiologues soignant un malade Parisien (dont l’adresse contient ‘Paris’) dans le service de ‘Chirurgie générale”.

Question 4

Essayez de simplifier le schéma objet en introduisant des attributs multivalués. Proposez des requêtes OQL utilisant des collections dépendantes.

29. BD OBJETS (chapitre XI)

La société REVE désire informatiser la gestion des pères noël. Pour cela, elle a acquis un SGBD objet ou objet-relationnel, supportant le langage OQL ou SQL3 (au choix). L’application est décrite par les phases suivantes :

– Toute personne est caractérisée par un nom, un prénom, une date de naissance, et une adresse composée d’un nom de ville, d’un numéro et nom de rue, ainsi que d’une localisation géographique 2D (X,Y).
– Les pères noël effectuent des tournées, au cours desquelles ils visitent des enfants.
– Les pères noël sont des personnes (on ne peut même plus rêver), ainsi que les enfants.
– Une tournée est composée d’une liste d’enfants à visiter, chaque point correspondant à un ou plusieurs enfants.
– À chaque tournée est affectée une hotte contenant des jouets étiquetés par le nom et l’adresse des enfants. La société enregistrera aussi le prix de chaque jouet.
– La transaction Effectuée_Tournée consiste à parcourir une tournée et à ajouter à chaque enfant à visiter dans la tournée les jouets qui lui reviennent, par insertion dans l’association nommée “possède”.

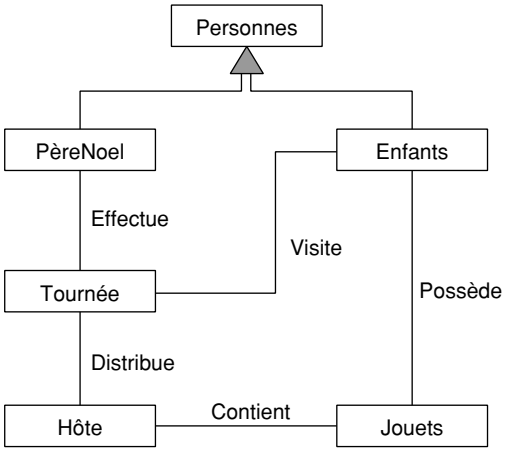
Une ébauche de schéma UML de la base est représentée page suivante.

Question 1

Discutez de l’intérêt de choisir un SGBD objet ou objet-relationnel. Effectuez un choix.

Question 2

Complétez la représentation UML de la base en précisant les cardinalités des associations.



Question 3

Définir le schéma de la base en C++. Bien lire le texte et la question 4 afin de prévoir les structures et méthodes capables d'y répondre.

Question 4

Écrire des programmes C++ naviguant dans la base et répondant aux requêtes suivantes :

Q1 : Nom du père noël qui vous a visité et prix total des jouets que vous avez reçus à Noël (on suppose que vous êtes un enfant).

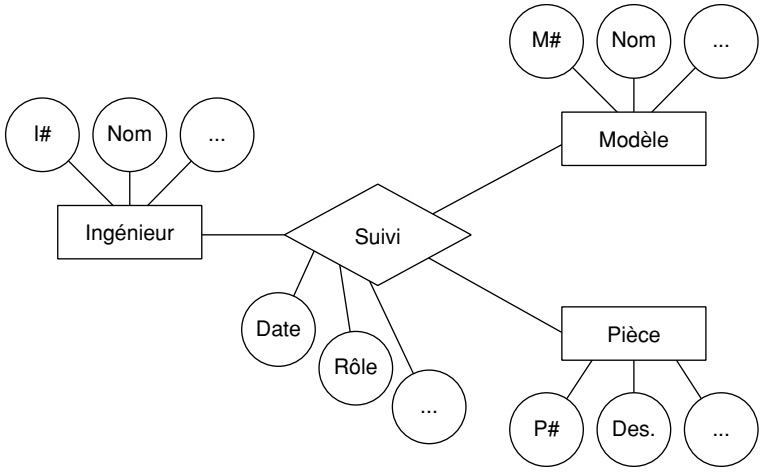
Q2 : Distance parcourue par le père noël Georges Gardarin et liste des jouets distribués.

U1 : Mettre à jour la base par enregistrement d'une nouvelle tournée pour un père noël donné.

30. BD OBJETS (chapitre XI)

Soit le schéma entité / association page suivante.

Il correspond au suivi de pièces entrant dans la composition de modèles d'avion d'une compagnie aéronautique. On remarque que cette association ternaire ne peut pas être décomposée car (1) une même pièce peut être utilisée dans plusieurs avions, (2) un ou plusieurs ingénieurs sont responsables du suivi de cette pièce et (3) le ou les ingénieurs responsables sont définis de façon différente pour chaque modèle où une pièce particulière est utilisée.



Question 1

Proposez une représentation objet de ce schéma en utilisant UML. Discutez les propriétés de votre représentation. Donnez la définition en ODL correspondante.

Question 2

Écrire les méthodes nécessaires pour faire une affectation d'une pièce à un modèle d'avion avec l'indication de la liste des ingénieurs chargés du suivi. Les méthodes seront écrites en Smalltalk, C++ ou Java persistant (ODMG) ou en pseudo-code pour ceux qui ne sont pas à l'aise avec les concepts de l'un de ces langages objet. Attention à l'encapsulation.

Question 3

Certaines pièces sont en fait des modules composés d'autres pièces ou d'autres modules. Proposez une modification du schéma pour `Pièce`.

Question 4

Écrire une méthode qui donne la liste de tous les ingénieurs impliqués dans le suivi d'un module, soit directement soit au titre d'un composant. On pourra utiliser la méthode prédéfinie `union` de la classe générique `Set` qui prend un autre `Set` en paramètre.

31. ALGEBRE D'OBJETS COMPLEXES (chapitre XI)

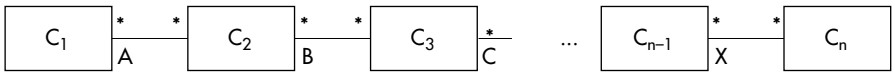
Une algèbre d'objets complexes permet d'exprimer des requêtes sur une base de données objet sous forme d'expressions algébriques. Elle résulte en général d'une extension de l'algèbre relationnelle.

Question 1

Rappelez les extensions nécessaires à l'algèbre relationnelle pour exprimer les requêtes objets. Proposez des exemples génériques de requêtes illustrant chaque extension.

Question 2

Soit une succession de classes T1, T2, ...Tn reliées par des associations [M-N] via des attributs de rôle nommés A, B, ...X, comme représenté ci-dessous :



Quels algorithmes proposeriez-vous pour traiter efficacement la requête SQL3 de parcours de chemins :

```
SELECT *
FROM C1, C2, ... Cn
WHERE C1.A = OID(C2)
AND C2.B = OID(C3)
...
AND Cn-1.X = OID(Cn) ?
```

On pourra aussi exprimer cette requête en OQL.

32. ODMG, ODL ET OQL (chapitre XII)

On considère une base de données géographiques dont le schéma est représenté de manière informelle ci-dessous :

```
REGIONS (ID CHAR(5), PAYS STRING, NOM STRING, POPULATION INT,
   carte GEOMETRie)
VILLES (NOM STRING, POPULATION INT, REGION ref(REGIONS), POSITION
   GEOMETRie)
FORETS (ID CHAR(5), NOM STRING, carte GEOMETRie)
```

```
ROUTES (ID CHAR(4), NOM STRING, ORIGINE ref(VILLES), EXTREMITE
   ref(VILLES), carte GEOMETRIE)
```

Cette base représente des objets RÉGIONS ayant un identifiant utilisateur (ID), un pays (PAYS), un nom (NOM) une population et une carte géométrique. Les objets VILLES décrivent les grandes villes de ces régions et sont positionnés sur la carte par une géométrie. Les forêts (FORETS) ont un identifiant utilisateur, un nom et une géométrie. Les routes (ROUTES) sont représentées comme des relations entre villes origine et extrémité ; elles ont une géométrie.

Une géométrie peut être un ensemble de points représentés dans le plan, un ensemble de lignes représentées comme une liste de points ou un ensemble de surfaces, chaque surface étant représentée comme un ensemble de lignes (en principe fermées). Sur le type GEOMETRIE, les méthodes Longueur, Surface, Union, Intersection et DRAW (pour dessiner) sont définies.

Question 1

Proposez une définition du schéma de la base en ODL.

Question 2

Exprimer les questions suivantes en OQL:

Q1. Sélectionner les noms, pays, populations et surfaces des régions de plus de 10 000 km^2 et de moins de 50 000 habitants.

Q2. Dessiner les cartes des régions traversées par la RN7.

Q3. Donnez le nom des régions, dessiner régions et forêts pour toutes les régions traversées par une route d'origine PARIS et d'extrémité NICE.

Question 3

Rappelez les opérations d'une algèbre d'objets complexes permettant d'implémenter le langage OQL. Donnez les arbres algébriques correspondants aux questions Q1, Q2 et Q3.

33. SQL3 (chapitre XIII)

On considère une base de données composée des entités élémentaires suivantes (les clés sont soulignées lorsqu'elles sont simples) :

```
CLIENT
NoCl = Numéro du client,
Nom = Nom du client,
```

```
Rue = Numéro et rue du client,
Ville = Ville d'habitation du client,
CP = Code postal du client,
Région = Région d'habitation du client.

FOURNISSEUR
NoFo = Numéro du fournisseur,
Nom = Nom du fournisseur,
Rue = Numéro et rue du fournisseur,
Ville = Ville d'habitation du fournisseur,
CP = Code postal du fournisseur,
Région = Région d'habitation du fournisseur.

COMMANDE
NoCo = Numéro de commande,
NoCl = Numéro du client,
Date = Date de la commande,
Resp = Nom du responsable de la suivie de la commande.

LIGNE
NoCo = Numéro de commande,
NoPro = Numéro du produit,
NoLi = Numéro de ligne dans la commande,
Qtté = Quantité de produit commandé,
Prix = Prix total Toutes Taxes Comprises de la ligne,

PRODUIT
NoPro = Numéro du produit,
Nom = Nom du produit,
Type = Type du produit,
PrixU = Prix unitaire Hors Taxe,
NoFo = Numéro du fournisseur.
```

On se propose d'étudier le passage en objet-relationnel de cette base de données et l'interrogation en SQL3. Pour cela, les questions suivantes seront considérées :

a) Noms, types des produits et adresses des fournisseurs offrant ces produits.

b) Noms des clients ayant commandé un produit de type « Calculateur ».

c) Noms des fournisseurs et des clients habitant une région différente, tels que le fournisseur ait vendu un produit au client.

d) Chiffre d'affaires toutes taxes comprises (somme des prix TTC des produits commandés) de chacun des fournisseurs.

Question 1

Proposez un schéma UML pour cette base de données. On ajoutera les méthodes appropriées pour illustrer. On supprimera tous les pointeurs cachés (clés de jointures) qui seront remplacés par des associations. Définir ce schéma directement en SQL3, en utilisant des références et des collections imbriquées.

Question 2

Exprimer en SQL3 les questions (a), (b), (c) et (d).

Question 3

Comparer l'implémentation objet-relationnelle avec une implémentation directe en relationnel.

Question 4

Reprendre les questions 1, 2 et 3 en utilisant ODL et OQL à la place de SQL3. Comparer les solutions.

34. INTERROGATION D'OBJETS (chapitres XII, XIII et XIV)

Reprendre la base de données de la société RÉVE concernant les pères noël, vue ci-dessus.

Question 1

Définir cette base en ODL puis en SQL3.

Question 2

Exprimer les requêtes suivantes en OQL puis SQL3 :

Q1 : Nom du père noël qui vous a visité et prix total des jouets que vous avez reçu à Noël (on suppose que vous êtes un enfant).

Q2 : Distance parcourue par le père Noël Georges Gardarin et liste des jouets distribués.

Question 3

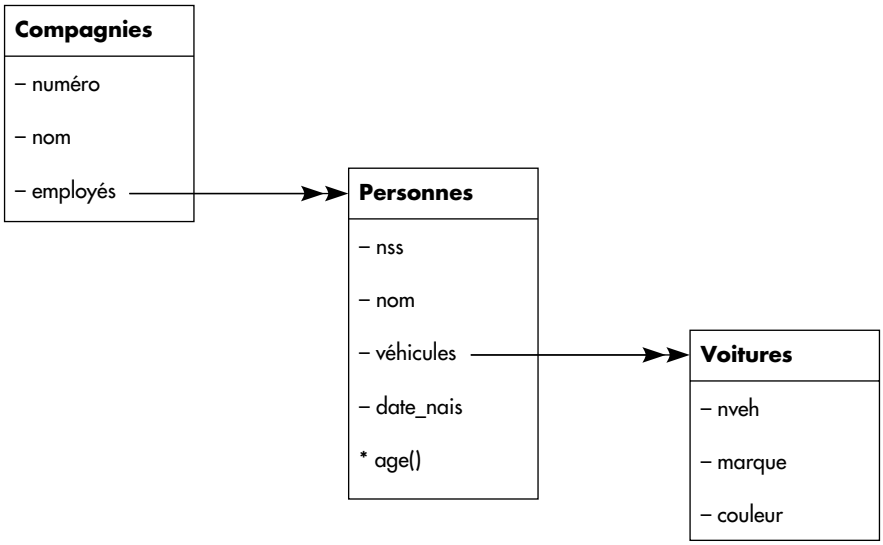
Pour chacune des questions précédentes, proposez un arbre algébrique optimisé permettant de l'exécuter.

Question 4

Rappelez la définition d'un index de chemin. Proposez deux index de chemins permettant d'accélérer l'exécution des questions Q1 et Q2.

35. OPTIMISATION DE REQUÊTES OBJET (chapitre XIV)

Soit la base de données objet dont le schéma est représenté ci-dessous. Elle modélise des compagnies dont les employés sont des personnes. Ceux-ci possèdent 0 à N véhicules. Les doubles flèches représentent des attributs de liens de type listes de pointeurs.



Question 1

Exprimer en OQL les questions suivantes:

Q1: ensemble des numéros de sécurité sociale et noms de personnes âgées de plus de 40 ans possédant au moins un véhicule de couleur rouge.

Q2: liste des noms de compagnies employant des personnes de plus de 40 ans possédant une voiture rouge.

Question 2

Donnez un arbre d'opérations d'algèbre d'objets complexes optimisé par heuristique simple permettant d'exécuter la question Q2.

Question 3

La jointure de deux classes C1 et C2, telles que Personnes et Voitures reliées par une association orientée multivaluée du type A—>>C1 peut être réalisée par un algorithme de parcours de pointeurs ou de jointure sur comparaison d'identifiant clas-

sique. Nous les appellerons « jointure avant JV » (par pointeur) et « jointure arrière JR » (par valeur). Détailler ces deux algorithmes en pseudo-code.

Question 4

On constate que la jointure de N classes reliées par des associations peut être réalisée par un seul opérateur de jointure n-aire, traversant le graphe en profondeur d'abord, appelé ici DFF. Détailler cet algorithme en pseudo-code.

Question 5

En dénotant page(Ci) et objet(Ci) le nombre respectif de pages et d'objets d'une classe Ci, par fan(Ci,Cj) le nombre moyen de pointeurs par objet de Ci vers Cj, en supposant un placement séquentiel sans index des objets, calculez le coût de chacun des algorithmes JR, JV et DFF. On pourra introduire d'autres paramètres si nécessaire.

36. RÉCURSION ET ALGÈBRE RELATIONNELLE (chapitre XV)

Soit le schéma relationnel suivant :

```
PERSONNE (NUMÉRO, NOM, PRÉNOM, EMPLOI, PÈRE, MÈRE)
```

`PÈRE` et `MÈRE` sont des clés étrangères sur la relation PERSONNE elle-même, ce sont donc des numéros de personne. Cette relation permet donc de représenter un arbre généalogique complet.

Question 1

Donnez l'arbre algébrique correspondant à la requête : Quels sont les enfants de M. Martin, de numéro 152487.

Question 2

Donnez l'arbre algébrique correspondant à la requête : Quels sont les petits enfants de M. Martin, de numéro 152487.

Question 3

Donnez l'arbre algébrique étendu correspondant à la requête : Quels sont les descendants de M. Martin, de numéro 152487.

Pour cette dernière question on étendra les arbres algébriques de la façon suivante :

– Un nœud supplémentaire de test permettra de spécifier un branchement conditionnel ;

– Une boucle dans l'arbre sera autorisée ;
– Un test pourra porter sur le résultat de l'opération précédente, par exemple (résultat = vide ?).

Attention à la condition d'arrêt du calcul.

37. DATALOG ET MÉTHODES NAÏVES (chapitre XV)

PARTIE 1

Soit le programme DATALOG :

p1(X,X) ← p2(X,Y)
p2(X,Y) ← p3(X,X), p4(X,Y)

et la base de données :

p3(a,b), p3(c,a), p3(a,a), p3(c,c), p4(a,d), p4(a,b), p4(b,c), p4(c,a).

Question 1

Donnez une expression relationnelle équivalente à ce programme pour le calcul des faits instances des prédicats p1(X,X) et p2(X,Y).

Question 2

Donnez les faits instanciés de symbole de prédicat p1 déduits de ce programme et de la base de données.

PARTIE 2

Soit le programme DATALOG :

anc(X,Y) ← par(X,Y)
anc(X,Y) ← anc(X,Z), par(Z,Y)

la base de données :

par(baptiste,léon), par(léon,lucie), par(janine,léon), par(étienne,cloé), par(lucie,cloé), par(baptiste,pierre), par(janine,pierre), par(pierre,bernard), par(bernard,nicolas), par(bernard,lucien), par(bernard,noémie), par(lucien,éric).

et la requête :

? anc(bernard,Y)

Question 1

Calculez les réponses à cette requête en appliquant la méthode naïve. Donnez les faits déduits à chaque itération.

Question 2

Même question avec la méthode semi-naïve.

Question 3

Écrire le programme modifié obtenu par la méthode des ensembles magiques.

Évaluez ce programme modifié sur la base, en montrant à chaque itération les faits déduits, et en précisant pour chaque fait déduit, par quelle règle il a été déduit.

38. OPTIMISATION DE RÈGLES LINÉAIRES (chapitre XV)

On considère la base de données relationnelles :

```
EMPLOYE (NSS, NSER, NOM, SALAIRE)
SERVICE (NSER, NCHEF)
```

`NSS` est le numéro de sécurité sociale de l'employé, `NSER` son numéro de service et `NCHEF` le numéro de sécurité sociale du chef.

Question 1

Écrire un programme DATALOG calculant la relation :

```
DIRIGE(NSSC, NSSD)
```

donnant quel employé dirige directement ou indirectement quel autre employé (le numéro de sécurité sociale NSSC dirige le numéro de sécurité sociale NSSD), ceci pour tous les employés.

Question 2

Transformer le programme DATALOG obtenu à la question 1 en appliquant la méthode des ensembles magiques pour trouver les employés dirigés par Toto.

Question 3

Modifier le programme trouvé à la question 1 pour calculer la relation

```
DIRIGE(NSSC, NSSD, NIVEAU)
```

NIVEAU est le niveau hiérarchique du chef (1 chef direct, 2 chef du chef direct, etc.). Dans quel cas ce programme a-t-il un modèle infini ?

Question 4

Donnez le graphe Règle-But et le graphe d'opérateurs relationnels correspondant à la question `? DIRIGE(Toto, x, 2)` pour le programme trouvé en 3.

Question 5

Écrire en DATALOG avec ensemble la question permettant de calculer la table

```
STATISTIQUE (NSER, CHEF, SALMAX, SALMIN, SALMOY)
```

donnant pour chaque service, le nom du chef, le salaire maximum, le salaire minimum et le salaire moyen, ceci pour tous les services dont le salaire minimum dépasse 7 000 F.

39. DATALOG ET RÈGLES REDONDANTES (chapitre XV)

Soit une base de données relationnelle composée de trois relations binaires B1, B2 et B3. Le but de l'exercice est d'optimiser le programme DATALOG suivant pour une question précisant le premier argument de R :

$R(x,y) \leftarrow B1(x,y)$
$R(x,y) \leftarrow B3(x,z),R(z,t), B2(t,u), R(u,y)$
$R(x,y) \leftarrow R(x,v), B3(v,z), R(z,t), B2(t,u), R(u,y)$

On considère donc la question ? R("a",y), où a est une constante.

Question 1

En utilisant les ensembles magiques, donnez le programme de règles transformées permettant de calculer la réponse à cette question.

Question 2

Simplifier le programme obtenu par élimination des règles redondantes. Peut-on proposer une méthode générale permettant d'éviter de transformer des règles redondantes ?

40. DATALOG ET OBJETS (chapitre XV)

On considère une base de données décrivant des lignes de chemins de fer dont le schéma est représenté figure 1. Cette base représente des objets VILLES ayant nom (NOM) une population (POPULATION) et une carte géométrique (PLACE). Certains objets villes sont reliés par des tronçons de voies ferrées de type 1 (normal) ou 2 (T.G.V.) représentés par une géométrie et ayant une longueur (LONG) exprimée en km. On précise que l'identifiant de tout objet est obtenu par la méthode OID() ; celle-ci peut donc être vue comme un attribut implicite calculé au niveau de chaque classe.

```
CLASS VILLES (NOM STRING, POPULATION INT, PLACE GEOMETRIE)
CLASS TRONCONS (ID CHAR(4), TYPE INT, ORIGINE Obj(VILLE),
                EXREMITE obj(VILLE), CARTE GEOMETRIE)
```

Une géométrie peut être un ensemble de points représentés dans le plan, un ensemble de lignes représentées comme une liste de points ou un ensemble de surfaces, chaque surface étant représentée comme un ensemble de lignes (en principe fermées). Sur le type GEOMETRIE, les méthodes LONG, UNION, INTERSECTION et DRAW (pour dessiner) sont définies. LONG est supposée calculer la longueur d'une géométrie en km.

Question 1

Donnez un programme DATALOG avec fonctions (utiliser en particulier les fonctions OID et DRAW) permettant de générer et tracer tous les parcours allant de la ville de nom A à la ville de nom B en parcourant moins de K km.

Discutez de l'optimisation de ce programme.

Question 2

On désire calculer le coût d'un billet de train. Le coût est proportionnel au nombre de km parcourus. Il existe des billets normaux, des billets réduits, des billets T.G.V., et des billets T.G.V. réduits. La majoration sur les lignes T.G.V. est un coût de réservation initial plus un coût proportionnel à la longueur du trajet. La réduction est un pourcentage du prix appliqué au seul coût proportionnel à la distance parcourue. Proposez une organisation en classes et sous-classes pour les billets de train. Programmer en pseudo C++ une méthode TARIF qui calcule le prix de chaque billet. On précise qu'il est possible d'appeler une méthode d'une classe (qui peut être une super-classe) par la syntaxe classe::méthode (…).

Question 3

Proposez une intégration de la méthode TARIF dans le programme réalisé à la question 1 afin de calculer le prix de chaque itinéraire de moins de x km permettant d'aller de A à B et de choisir le moins coûteux.

41. CONTRÔLE DE CONCURRENCE (chapitre XVI)

Soit une base composée de 4 granules A, B, C, D et une exécution de 6 transactions T1 à T6. Les accès suivants ont été réalisés sur les granules (Ei signifie écriture par la transaction i et Lj lecture par la transaction j du granule indiqué) :

```
A : E2 E3 L5
B : L2 L4 L1
C : E5 L1 L3 E4
D : L6 L2 E3
```

Question 1

Dessiner le graphe de précédence. Que peut-on en conclure ?

Question 2

On suppose que les demandes se font dans l'ordre global suivant :

```
E2(A) L2(B) L6(D) E5(C) E3(A) L5(A) L1(C) L2(D) L3(C) E4(C) E3(D)
L4(B) L1(B)
```

Aucune transaction n'a relâché de verrous. L'algorithme utilisé est le verrouillage deux phases. Donnez le graphe d'attente. Que se passe-t-il ?

Question 3

On rappelle que pour l'algorithme d'estampillage multiversion, les accès doivent être effectués dans l'ordre des estampilles des transactions, sauf les lectures qui peuvent se produire en retard. Une lecture en retard obtient la version qu'elle aurait dû lire si elle était arrivée à son tour. Les écritures en retard provoquent par contre un abandon de transaction.

Toujours avec l'ordre des accès de Q2, que se passe-t-il en estampillage multiversion ?

Question 4

Pour gérer le multiversion on suppose que l'on dispose d'une table en mémoire pouvant contenir 100 000 versions avec les estampilles correspondantes (pour l'ensemble des granules). De plus on conserve sur disque la dernière version produite de chaque granule.

On se propose de modifier l'algorithme d'estampillage multiversion de la façon suivante : chaque nouvelle version d'un granule mis à jour est écrite sur disque tandis que la version précédente est recopiée dans la table en mémoire. Cette table est gérée comme une file circulaire : chaque nouvelle version entrée chasse la version la plus ancienne (tous granules confondus). Lors d'une lecture on recherche dans la table (ou

sur disque) la version nécessaire si elle est disponible. Une lecture peut donc échouer et conduire à un abandon.

Donnez en pseudo-code les algorithmes `LIRE` et `ÉCRIRE` correspondants.

42. CONCURRENCE SÉMANTIQUE (chapitre XVI)

Soit une classe C++ persistante Réel définie comme suit :

```
Class Réel: Pobject {
   private:
         Réel Valeur;
   public:
         operator+…;
         operator*…}.
```

Les actions de base indivisibles pour le contrôle de concurrence à étudier sont les opérations sur objets, c'est-à-dire + et * sur l'exemple.

On considère trois transactions travaillant sur des réels nommés A et B :

```
T1: { A+1 --> A; B+1 --> B }
T2: { A+2 --> A: B+2 --> B}
T3: { A*3 --> A: B*3 --> B}
```

Question 1

Donnez un exemple d'exécution simultanée des transactions T1 et T3 non sérialisable. Tracer le graphe de précédence associé.

Question 2

Montrer que la commutativité de l'opération d'addition garantie la correction de toute exécution simultanée de T1 et T2.

Question 3

La commutativité des opérations permet donc d'accepter des exécutions simultanées non à priori sérialisable. On se propose de munir chaque classe persistante d'un contrôleur de concurrence utilisant une matrice de commutativité des opérations et un vecteur d'opérations en cours par transaction.

Définir les primitives nécessaires à un tel contrôle de concurrence.

Préciser les algorithmes sous-jacents.

43. TRANSACTIONS (chapitre XVI)

Soit le programme suivant écrit en C avec des requêtes SQL imbriquées (ESQLC de INGRES) :

```
#include <stdio.h>;
main (){
   exec sql    begin declare section;
      char nombase [20];
      char instr [150];
   exec sql    end declare section;
   int i;
   char rep;
   printf ("Nom de la base : ");
   scanf ('%s", nombase);
   exec sql whenever sqlerror stop;
   exec sql connect :nombase;
   exec sql whenever sqlerror goto erreur;
   for (i=1, rep = 'O'; rep = 'O'; i++){
      printf ("\n\n Instruction %d : ", i);
      scanf ("%s, instr);
      exec sql execute immediate :instr;
      exec sql commit;
   suite :
      printf ("\n\n Encore ?");
      scanf ("%c", &rep);
      }
   printf ("\n\n Au revoir ...");
   exec sql disconnect;
   return;
erreur :
   exec sql whenever sqlerror continue;
   Affic_Erreur ();   procédure de gestion des erreurs
   exec sql rollback;
   exec sql whenever sqlerror goto erreur;
   goto suite;
}
```

Question 1

Que fait ce programme ? Précisez le début et la fin de la (des) transaction(s).

Question 2

Que se passe-t-il si on supprime l'instruction exec sql commit ?

Question 3

Précisez les effets du rollback.

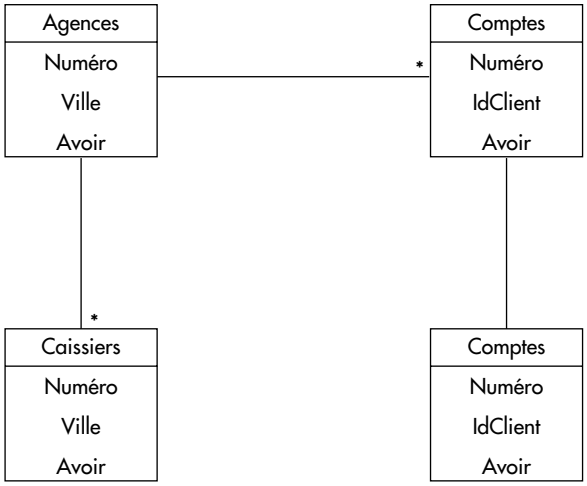
Question 4

On lance plusieurs instances simultanées du programme précédent. Une transaction ne peut pas lire (et encore moins modifier) les données mises à jour par une transaction non validée. Pourquoi ?

Précisez les complications des procédures de reprise qu'impliqueraient la levée de cette limitation, c'est-à-dire le non respect de la propriété d'isolation des mises à jour.

44. VALIDATION DE TRANSACTIONS (chapitre XVI)

Soit la base de données issue du banc d'essai TPC/A dont le schéma entité-association est représenté ci-dessous (en UML) :



L'objectif de l'étude est d'étudier la résistance aux pannes dans un contexte centralisé puis réparti sur une telle base de données implémentée en relationnel.

Question 1

Donnez le schéma relationnel de la base. Écrire le code de la transaction débit/crédit sous la forme de requêtes SQL.

Question 2

Le système démarre après un point de reprise et exécute quatre instances de la transaction débit/crédit notées T1, T2, T3, T4. T1 crédite 1 000 F sur votre compte avec

succès. T2 tente de créditer 2 000 F mais échoue, T3 crédite 3 000 F avec succès et T4 débite 4 000 F avec succès. Donnez le format du journal (images après et avant comme dans ARIES) après exécution de ces quatre transactions.

Ce journal est géré en mémoire. Quand est-il écrit sur disque ?

Question 3

Le système tombe en panne après validation (*commit*) de T4. Que se passe-t-il lors de la procédure de reprise ?

Question 4

Chaque agence bancaire possède un serveur gérant sa base de données. Écrire le code d'une transaction s'exécutant depuis un client C et effectuant le transfert de m francs depuis un compte d'une agence A sur le compte d'une agence B. Préciser les messages échangés entre client et serveurs dans le cas d'exécution avec succès d'une telle transaction, puis dans le cas où le site A ne répond plus lors de la première phase du commit.

Question 5

Le site client lance la transaction de transfert qui est bien exécutée sur A et B. Malheureusement, le site client se bloque juste avant l'envoi de la demande de validation (COMMIT). Préciser l'état du système (messages échanger et journaux). Que se passe-t-il alors ?

Afin de sortir des situations de blocage, proposez une extension au protocole de validation à deux phases en ajoutant une phase et en introduisant un état intermédiaire basculant automatiquement en échec de la transaction après un délai d'attente.

45. GESTION DES AUTORISATIONS (chapitre XVI)

Lorsqu'un usager crée une relation, tous les droits (lecture, insertion, suppression, modification) lui sont accordés sur cette relation. Cet usager a également le droit de transmettre ses droits à d'autres usagers à l'aide de la commande `GRANT`. Un usager ayant transmis des droits à un autre peut les lui retirer par la commande `REVOKE`.

Question 1

Écrire la commande permettant de transmettre les droits de lecture et insertion sur la relation `VINS` à l'usager `FANTOMAS`, en lui garantissant le droit de transmettre ces droits.

Écrire la commande permettant de supprimer le droit précédent.

Question 2

Afin de rendre plus souple l'attribution de droits, on introduit la notion de rôle. Précisez les commandes nécessaires à la gestion des rôles.

Question 3

Les droits donnés aux usagers sont mémorisés dans une relation `DROITS`. Proposez un schéma pour la relation `DROITS`.

Question 4

Décrivez en quelques lignes ou par un organigramme les principes des algorithmes d'accord de droit (`GRANT`), de suppression de droits (`REVOKE`) et de vérification d'un droit.

46. NORMALISATION DE RELATIONS (chapitre XVII)

PARTIE 1

Soit la relation R (A, B, C, D, E, F, G, H) avec la liste de dépendances fonctionnelles suivantes :

DF = { A→B, A→C, D→E, F→A, F→C, F→G, AD→C, AD→H }

Question 1

Proposez une couverture minimale DF^{-} pour DF.

Question 2

Dessiner le graphe des dépendances fonctionnelles.

Question 3

Proposez une décomposition en troisième forme normale pour R.

PARTIE 2

Soit la liste suivante de données élémentaires utilisées pour construire une base de données `Commandes` :

NUMCOM : numéro de commande	Identifiant unique
DATECOM : date de la commande	
NUMCLI : numéro de client	Identifiant unique
NOMCLI : nom du client	Doubles possibles
ADCLI : adresse client	Doubles possibles
NOMVEND : nom du vendeur	Identifiant unique du vendeur qui passe la commande
NUMPRO : numéro du produit	Identifiant unique
NOMPRO : nom du produit	Doubles possibles
PRIX : prix unitaire produit	
QTE : quantité commandée d'un produit	
MONTANT : montant pour la quantité commandée du produit	
TOTAL : prix total commande	Montant total de la commande pour tous les produits commandés

On suppose que l'on commence par former une unique relation `Commandes` avec les données précédentes pour attributs.

Question 1

Proposez une liste de dépendances fonctionnelles entre ces attributs en respectant les hypothèses.

Question 2

Dessiner le graphe des dépendances fonctionnelles en utilisant des arcs en traits pleins pour la couverture minimale et des arcs en pointillé pour ceux qui peuvent se déduire par transitivité.

Question 3 :

Proposez une décomposition en troisième forme normale de la relation.

47. INTEGRITE DES BD OBJETS (chapitre XVII)

Soit le schéma de base de données objet suivant, donné dans un langage proche de C++ utilisant une définition de classe de grammaire `CLASS` <nom de classe> { [<type> <nom d'attribut>] …}, un type pouvant être simple (exemple: integer) ou complexe (exemple: `SET`<…>). Le constructeur de collection « `SET` » est utilisé pour préciser un ensemble d'objets du type indiqué entre crochets (exemple `SET<PISTE*>` désigne un ensemble de références à des pistes).

```
CLASS STATION { CHAR(20) NOMS, INTEGER ALTITUDE,
INTEGER HAUTEUR_NEIGE, SET<PISTE*> PISTES,
CHAR(10) REGION, CHAR(10) PAYS}

CLASS PISTE{ CHAR(12) NOMP, INTEGER LONG,
INTEGER DENIVELEE, SET<BOSSE*> DIFFICULTES}

CLASS BOSSE { INTEGER POSITION, CHAR CATEGORIE}
```

On précise que:

1. Il n'existe pas deux stations de même nom (`NOMS`)
2. Une région appartient à un pays unique.
3. `ALTITUDE` et `HAUTEUR_NEIGE` sont déterminés par le nom de station `NOMS`.
4. Deux pistes de stations différentes peuvent avoir le même nom.
5. Une piste dans une station possède une longueur et une dénivelée unique.
6. La position d'une bosse la détermine de manière unique sur une piste donnée.

Question 1

En considérant chaque classe, donnez le graphe des dépendances fonctionnelles entre attributs. Ajouter au graphe des arcs représentant les dépendances hiérarchiques de 1 vers N représentés par des arcs à tête multiple (du style –->>). Compléter ce graphe par les dépendances interclasses fonctionnelles ou hiérarchiques.

Question 2

À partir du graphe précédent, déduire un schéma relationnel en 3^{e} forme normale permettant de représenter la base de données objet initiale.

Question 3

Préciser les contraintes d'unicité de clé du schéma relationnel. Spécifier ensuite les contraintes référentielles permettant d'imposer l'existence d'une station pour insérer une piste puis l'existence d'une piste pour insérer une bosse.

Question 4

Proposez un langage de définition de contraintes permettant d'exprimer les mêmes contraintes dans le monde objet. Illustrez sur la base objet initiale.

48. CONCEPTION COMPLÈTE D'UNE BD (chapitre XVII)

L'association (fictive) des Amateurs de Vins de Monde Entier (AVME) est une organisation a but non lucratif dont le but principal est de promouvoir l'entente entre les peuples par l'échange et la promotion des meilleurs vins du monde entier. Ses membres sont principalement des dégustateurs professionnels et des amateurs éclairés (qui bien sûr dégustent aussi). Ils désirent constituer une base de données recensant les vins, les dégustateurs (les membres) et tous les événements de dégustations. Ils ont demandé votre aide pour la conception du schéma relationnel, qui ne doit présenter ni anomalie ni redondance. La liste des données élémentaires qui doivent figurer dans la base est la suivante :

NUM_VIN	Numéro d'un vin
NUM_PRO	Numéro d'un producteur de vin
PRODUCTEUR	Par exemple "Château Talbot" ou "Burgess Cellars"
DESIGNATION	Désignation additionnelle pour un vin, comme "Réserve"
MILLESIME	Année de production d'un vin
CEPAGE	Comme "Cabernet-Sauvignon" ou "Chardonnay"
NOTE_MOY	Note moyenne d'un millésime d'un vin
PAYS	Pays d'origine
REGION	Grande région de production, comme "Californie" aux USA, "Rioja" en Espagne ou "Bourgogne" en France
CRU	Petite région de production dont les vins ont un caractère marqué, comme "Napa Valley" ou "Haut-Médoc"
QUALITE	Qualité moyenne d'un cru pour un millésime
NUM_MEM	Numéro d'un membre
NOM	Nom d'un membre
PRENOM	Prénom d'un membre
DATE	Date d'un événement de dégustation
LIEU	Lieu d'un événement de dégustation
NOTE	Note attribuée à un millésime d'un vin par un dégustateur (membre) lors d'un événement dégustation
RESP_REG	Membre responsable d'une région viticole

À titre d'explication et/ou de simplification on notera que :

– les dégustations se font lors d'événements mettant en jeu de nombreux membres et de nombreux vins ;
– il n'y a jamais deux événements de dégustation à la même date ;
– un même vin est goûté de nombreuses fois lors d'un événement et reçoit une note pour chaque dégustateur qui l'a goûté, un dégustateur ne goûtant pas tous les vins ;
– deux régions de production n'ont jamais le même nom ;
– un même vin donne lieu à plusieurs millésimes (un par an) et tous les vins de la base sont des vins millésimés ;
– le producteur est le même pour tous les millésimes du même vin.

Question 1

Donnez la liste des dépendances fonctionnelles non triviales entre données élémentaires sous la forme d'un graphe. S'assurer qu'il ne reste aucune dépendance fonctionnelle qui puisse se déduire des autres par transitivité (les enlever s'il en reste).

Question 2

Proposez un schéma normalisé en 3FN pour la base `DÉGUSTATION` à partir de l'ensemble des dépendances fonctionnelles et à l'aide de l'algorithme vu en cours.

Question 3

Exprimez en SQL les requêtes suivantes sur le schéma proposé :

a) Quels sont les vins de 1991 de la région du Piémont (en Italie) qui ont reçu au moins une note supérieure à 17 lors de la dégustation du 14/05/1994 ?
b) Quelle est la note moyenne décernée tous vins confondus par chaque membre (dégustateur) ?
c) Quels sont les crus qui n'ont pas reçu de note inférieure à 15 pour le millésime 1989 ?
d) Quels sont les dégustateurs qui ont participé à tous les événements ?

Question 4

Donnez une expression des requêtes a) c) et d) en algèbre relationnelle (en utilisant de préférence la notation graphique vue en cours).

Question 5

Proposez un schéma physique pour le schéma relationnel précédent. On suppose que l'on peut placer en séquentiel (avec index primaire non plaçant), par des index primaires plaçant de type arbre B+ ou par hachage classique. Choisir un mode de placement pour chaque relation en le justifiant.

Question 6

Proposez les index secondaires les mieux à même d'accélérer la manipulation de la base. Indiquer les organisations d'index retenues dans chaque cas et donner les justifications des choix réalisés.

TRANSPARENTS

Pour chacun des chapitres, une série de transparents peut être obtenue pour l'enseignement public en contactant le site Web du laboratoire PRiSM de l'université de Versailles :

www.prism.uvsq.fr/~gardarin/home.html

INDEX

F

G

H

I

Q

R

S

Mise en page : EDIE
N° d'éditeur : 6917