Extension orientée objet d'un SGBD relationnel

François Exertier

THESE

présentée par

François EXERTIER

pour obtenir le titre de

Docteur de l'Université
Joseph Fourier - Grenoble 1

(arrêté ministériel du 23 novembre 1988)

Spécialité : INFORMATIQUE

Extension orientée objet d'un SGBD relationnel

Thèse soutenue devant la commission d'examen le :
11 décembre 1991

Sacha Krakowiak	Président
Jean Ferrié	Rapporteur
Georges Gardarin	Rapporteur
Michel Adiba	Directeur
Roland Balter	
Mauricio Lopez	

Thèse préparée au sein du Laboratoire de Génie Informatique et à l'Unité Mixte Bull-IMAG

Je tiens à remercier

Michel Adiba et *Sacha Krakowiak*, Professeurs à l'Université Joseph Fourier (Grenoble I), pour la confiance qu'ils m'ont accordée pendant ces trois années et pour avoir accepté de partager la responsabilité de diriger mon travail.

Jean Ferrié, Professeur à l'Université de Montpellier, qui a accepté d'être rapporteur de ce travail. Je le remercie pour l'intérêt qu'il lui a porté et pour l'évaluation qu'il en a fait.

Georges Gardarin, Professeur à l'Université Paris VI, pour avoir accepté d'évaluer ce travail. Je tiens à le remercier particulièrement, ainsi que son équipe, pour la collaboration que nous avons eue durant ce travail.

Roland Balter, Directeur du laboratoire "Unité Mixte Bull-IMAG", pour m'avoir accueilli au sein de son équipe, et pour avoir accepté de participer au jury.

Mauricio Lopez, Ingénieur au Centre de Recherche Bull de Grenoble (notre chef d'équipe BD), pour avoir encadré ce travail, et dont les conseils sont toujours de très grande valeur.

Samer Haj Houssain, Ingénieur au Centre de Recherche Bull de Grenoble, avec qui une grande partie du travail présenté ici a été pensée et réalisée, et qui s'avère un adversaire coriace au tennis.

Christian Lenne, Ingénieur au Centre de Recherche Bull de Grenoble, mon "conseiller technique", qui a apporté beaucoup d'humour à ces trois années de travail et qui a eu le mérite d'essayer de m'apprendre la brasse. Je le remercie aussi pour avoir relu avec attention ce document.

Adriana Danes, pour sa contribution à la dernière version du module de gestion d'objets, pour sa sympathie et pour l'intérêt qu'elle a porté à ce travail.

André Freyssinet et *Xavier Rousset de Pina* pour l'effort de relecture qu'ils ont bien voulu me consacrer.

Je remercie l'ensemble des membres des projets Guide et Aristote avec qui j'ai apprécié de travailler et que je regrette de ne pouvoir tous citer.

Pour finir, je remercie tous les sportifs du laboratoire avec qui j'ai pris plaisir à courir, et à qui je dédie mon diplôme du BRA 1991.

Extension orientée objet d'un SGBD relationnel

Résumé

L'objectif de ce travail est la conception et la réalisation d'un Système de Gestion de Base de Données Relationnel (SGBDR) intégrant les concepts et la technologie "objets". Le principe de notre approche est d'étendre les domaines relationnels aux types abstraits (ADT) et revient à coupler de façon relativement faible les concepts et mécanismes objets au modèle et à un système relationnels. Cela introduit des problèmes de modélisation et d'optimisation nouveaux qui restent à étudier.

Dans un premier temps, le modèle de données et les caractéristiques de l'extension sont définis. La notion de type abstrait est introduite pour exprimer de nouveaux domaines : un ADT définit une structure de données et un ensemble de méthodes (fonctions) qui constituent son unique interface de manipulation. Un mécanisme d'héritage simple est offert. Des constructeurs sont disponibles pour définir la structure de données d'un type ; on introduit ainsi la notion d'objet complexe. Le concept de partage, associé à l'identité d'objet, est un apport important de ce travail. Le langage associé au modèle est une extension de SQL appelée ESQL ; le langage d'écriture des méthodes actuellement disponible est une extension de C.

La mise en œuvre d'un tel système consiste à développer les composants nécessaires au support d'objets et à les intégrer à un noyau de SGBDR existant. Elle permet de mettre en évidence trois modules principaux. Le *gestionnaire de types* est un complément du gestionnaire de catalogue relationnel qui gère les définitions d'ADT. Le *gestionnaire de méthodes* regroupe un ensemble de fonctions allant de la compilation à l'exécution. Le *gestionnaire d'objets* assure le stockage et la manipulation des objets complexes (instances d'ADT) ; cette partie a notamment permis d'étudier des techniques évoluées de stockage d'objets.

Mots clés

Base de données, encapsulation, héritage, méthode, objet complexe, partage, relationnel, type abstrait

Object-Oriented Extension of a Relational DBMS

Abstract

The objective of this work is the specification and development of a Relational Database Management System (RDBMS) which incorporates "Object" technology and concepts. The principle of our approach is to extend relational domains with Abstract Data Types (ADT) and consists in loosely coupling object concepts and mechanisms to the relational model and system. This introduces new modelling and optimization problems requiring further study.

First, the data model and the features of the extension are defined. The concept of Abstract Data Type is introduced in order to express new domains: an ADT specifies a data structure together with a set of methods (functions) which constitutes its manipulation interface. A simple inheritance mechanism is provided. Constructors are supplied to define the data structure of a type, introducing the notion of complex object. The concept of sharing, based on object identity, is an important contribution of this work. The language which materializes the model is a SQL extension, named ESQL; methods are presently written using a C extension.

The realization of such a system consists in the implementation of different components providing the object support and in the integration of these components to an existing RDBMS kernel. There are three main modules. The *type manager* is a part of the relational catalogue manager which handles ADT definitions. The *method manager* provides different functions like compilation and execution. The *object manager* supplies object storage and manipulation capabilities; this part provided the opportunity to study advanced object storage techniques.

Keywords

Database, encapsulation, inheritance, method, complex object, sharing, relational, abstract data type

Chapitre I

Introduction

I.1 Le contexte et les motivations

Les systèmes de gestion de bases de données (SGBD) relationnels ont pris une place importante sur le marché des SGBD. Ils répondent de façon satisfaisante aux besoins des applications classiques de gestion (administration, comptabilité, banques, etc.). Des domaines d'application moins classiques sont apparus, comme la CAO (Conception Assistée par Ordinateur), la bureautique, le génie logiciel, les applications géographiques, etc.. Ces applications gèrent des données beaucoup plus complexes, qu'il est difficile de représenter avec le modèle relationnel. Les limites du modèle sont d'une part un manque de capacité de représentation structurelle, et d'autre part l'impossibilité de représenter pleinement la sémantique des entités de la base. La seule structure de données disponible dans le modèle relationnel est le tableau à deux dimensions (qui matérialise une relation n-aire entre plusieurs domaines de valeurs). Ces structures plates ne permettent pas de représenter directement des objets complexes comprenant plusieurs niveaux d'imbrication. Il faut alors décomposer ces objets en plusieurs relations. Cette décomposition est visible à l'utilisateur qui doit formuler des requêtes permettant de reconstituer de tels objets (jointures). D'autre part les données stockées sont purement descriptives ; l'essentiel de la sémantique associée à ces données se trouve dans les programmes de manipulation. Seules les contraintes d'intégrité permettent de représenter une certaine sémantique associée aux données. Ces lacunes sont justement comblées par certaines approches de l'*orientation objet*. Les recherches menées dans le domaine des bases de données se sont donc inspirées de l'approche *objet*[1]. On peut regrouper les SGBD de nouvelle génération ("post-relationnels") en trois familles :

- Les extensions de SGBD relationnels constituent une première famille. Des SGBD comme POSTGRES [48] et SABRINA-OBJET [40] en sont des résultats.
- Les SGBD Orientés Objet (SGBDOO) sont une autre famille de systèmes. Ils sont conçus pour supporter un modèle de données à objets et offrent au programmeur un langage de programmation à objets. Des produits issus de la

1 Il n'existe pas de terminologie stricte pour nommer les modèles, les langages et les systèmes qui s'inspirent du concept d'*objet*. Nous n'avons pas non plus fixé de vocabulaire précis, afin de conserver les termes utilisés le plus couramment suivant le contexte. Nous utilisons donc aussi bien les termes *"à objets"*, "*orientés objet"* ou simplement "*objets"* pour qualifier les systèmes et langages évoqués.

recherche sont déjà sur le marché : O2 [24], ONTOS et ITASCA [4] (à partir d' ORION [8]) en sont des exemples.

- La dernière famille est celle des SGBD dits "extensibles". Ces derniers proposent des systèmes adaptables aux types d'applications qu'ils doivent supporter. Ils fournissent un support système "ouvert", dans le sens où il est possible d'y introduire de nouveaux mécanismes, ainsi que d'adapter ceux qui existent. EXODUS [19], GENESIS [11] et Starburst [35] font partie de cette catégorie.

Notre travail se situe dans le cadre de la première famille de propositions, tout en tenant compte des résultats et techniques des deux autres familles. L'objectif est d'apporter une extension *orientée objet* à un SGBD relationnel. Les aspects modèle, langage et mise en œuvre[2] sont concernés. La mise en place d'une telle extension fait appel à de nombreux concepts et mécanismes issus des SGBDOO, ainsi qu'aux techniques de mise en œuvre des SGBD extensibles. Mais les concepts que nous abordons ne sont pas tous spécifiques du domaine des SGBD. On les retrouve notamment dans les systèmes et langages à objets. Cela comprend les systèmes d'exploitation et les environnements de programmation à objets comme Guide [7] [39] ou Emerald [16], ainsi que les systèmes de stockage spécialisés comme Cricket [54] ou GEODE [14]. Il est clair que les systèmes d'exploitation, les langages et les SGBD ont tendance à se rapprocher de plus en plus. Au plus bas niveau les SGBD utilisent des mécanismes qui pourraient être fournis par le système d'exploitation, si ce dernier tenait compte de leurs besoins spécifiques. La notion de persistance des données est introduite dans les langages et les SGBD ont de plus en plus besoin d'un langage de programmation qui intègre les manipulations ensemblistes et associatives pour écrire les applications. L'orientation objet semble accélérer le processus de rapprochement de ces trois entités. Nous tentons au cours de ce travail de tirer profit des enseignements apportés par tous ces domaines.

I.2 Les objectifs

L'objectif essentiel de la thèse est la fusion du modèle et de la technologie à objets avec le modèle et la mécanique relationnels. Ce travail a été réalisé dans le cadre du projet Esprit EDS (European Declarative System, Esprit II Project 2025). Un des objectifs du projet EDS est la conception d'un Système de Gestion de Bases de Données (SGBD) relationnel étendu (objet et déductif) sur une machine parallèle. Ce SGBD se présente sous la forme d'un serveur auquel les applications désirant accéder aux données de la base doivent de connecter. L'originalité de l'approche se situe dans trois domaines :

- Dans beaucoup d'expériences précédentes, l'introduction des concepts objets n'est que partielle. Certaines approches introduisent les objets comme

2 La mise en œuvre comprend l'architecture, les mécanismes particuliers et le support système.

des valeurs d'attribut de relation non partageables, d'autres se contentent de proposer la notion d'objet construit (ou structuré) sans comportement (méthodes ou fonctions) associé. Ces différentes approches sont détaillées dans l'état de l'art du chapitre II. Nous proposons d'intégrer simultanément la notion d'identité d'objet (de partage), la notion d'objet construit et la notion d'encapsulation. Dans notre cas, un objet est non seulement structuré, mais il est aussi partageable entre plusieurs relations ou objets, et dispose d'un certain nombre de méthodes associées qui constituent son interface d'accès.

- L'environnement technologique est un deuxième facteur d'innovation, puisque le cadre de réalisation du serveur relationnel étendu est une machine parallèle à mémoires de grande taille ; ces paramètres sont pris en compte dans la mise en œuvre du gestionnaire de stockage et des mécanismes d'exécution.
- Enfin la dernière particularité réside dans l'approche compilée, adoptée pour le serveur relationnel. Les requêtes ne sont pas interprétées comme dans beaucoup de SGBD relationnels, mais compilées. Le résultat d'une compilation est constitué d'un ensemble de programmes (code machine) dont l'exécution est lancée sur les différents nœuds de la machine parallèle. Cette technique a une influence sur les mécanismes d'édition de lien et d'exécution des méthodes.

L'extension objet a des répercussions au niveau modèle, langage et système qui ont fait l'objet de notre étude.

I.2.1 Définition du modèle

La première étape consiste à intégrer le concept d'objet dans le modèle relationnel. Ce dernier est défini sur trois concepts clés : la relation, l'attribut et le domaine. Un domaine représente un ensemble de valeurs. Une relation est un sous-ensemble du produit cartésien de *n* domaines, elle peut être considérée comme une table à *n* colonnes. Un attribut est une colonne particulière, tandis qu'un n-uplet est une ligne de cette table. Le domaine définit l'ensemble des valeurs que peut prendre un attribut.

La notion de domaine dans les systèmes actuels est habituellement limitée à quelques types de base (entier, réel, caractère, chaîne, etc.). Nous proposons ici d'y intégrer des types définis par l'utilisateur, propres à son application. Ceci est réalisé en utilisant la notion de type abstrait de données (ADT pour "Abstract Data Type"). Il est donc possible de définir de nouveaux types, sous forme d'ADT, et de les utiliser dans la définition des attributs. Cette approche est dite "par extension de domaine". Cela présente l'avantage de ne nécessiter que des modifications minimales au modèle et au langage : les entités de premier plan restent les relations et

les attributs, tandis que les concepts objets viennent simplement s'*ajouter* aux concepts existants, sans les redéfinir ; le système résultant reste compatible avec les applications relationnelles existantes, ce qui est un impératif du projet EDS. L'inconvénient majeur réside dans le fait que le modèle présente deux niveaux d'abstraction : le niveau relation et le niveau objet, ce qui pose des problèmes de modélisation. D'autres approches sont présentées dans le chapitre II, mais elles perdent en général la "compatibilité relationnelle". Les différents points caractérisant l'extension que nous avons apportée sont présentés ci-dessous.

Les types abstraits

Nous avons repris le terme de *type abstrait*, souvent utilisé dans les extensions de SGBD, mais issu des langages ([18], [25]), pour désigner une notion particulière de type de données. Un tel type décrit une structure de données assortie d'un ensemble d'opérateurs qui permettent de la manipuler. L'ensemble de ces opérateurs constitue l'interface de manipulation de ce type ; il n'est pas possible d'accéder à la structure de données autrement que par cette interface, on parle alors d'*encapsulation* . Les opérateurs associés à un ADT sont aussi appelés *méthodes*. Les exemplaires d'un type sont appelés *instances*.

L'héritage

La notion d'héritage entre types abstraits est définie. C'est un héritage simple, un type ne possède qu'un seul supertype. Un type hérite des méthodes et de la structure de données de son supertype. Il peut redéfinir des méthodes existantes, en définir de nouvelles et spécialiser la structure de données.

Les objets complexes

Un des objectifs consiste à fournir à l'utilisateur la notion d'objet complexe (ou construit). Des modèles d'objets complexes ont déjà été définis par ailleurs, mais ils se présentent comme des sur-ensembles du modèle relationnel. Comme nous l'avons vu plus haut, nous désirons conserver les structures relationnelles au premier plan et n'introduire les objets complexes qu'au niveau de l'extension de domaine. Ceci est réalisé en introduisant un certain nombre de constructeurs qui permettent de spécifier la structure de données d'un ADT. Ces constructeurs ont en fait le statut de types abstraits génériques et sont actuellement au nombre de quatre : *tuple*, *list*, *array* et *set*. Les méthodes définies sur ces types génériques sont le seul moyen de manipuler la structure de données d'un ADT, que ce soit au niveau du langage de requêtes ou dans le corps des méthodes.

Le partage d'objet

La notion de partage d'entité, mal gérée en relationnel pur, est introduite en s'inpirant du concept d'identité d'objet issu des systèmes orientés objet. Le partage d'entité en relationnel se fait en partageant des n-uplets au moyen

d'attributs clés à la charge de l'utilisateur. Nous introduisons une notion beaucoup plus puissante en permettant de partager des instances d'ADT. Ces instances partagées peuvent être des valeurs d'attribut ou des composants d'objets complexes. C'est le type d'une instance qui détermine si elle est partageable ou non.

Le langage

Le langage associé à ce modèle est une extension de SQL, appelée ESQL (pour Extended SQL) [37]. Il permet de définir des types abstraits et d'inclure des appels de méthodes dans les requêtes de manipulation.

I.2.2 Réalisation du support

Une fois le modèle étendu défini, nous nous intéressons à la mise en œuvre d'un système pour le supporter. Différents principes de conception se présentent alors : ils correspondent à une intégration plus ou moins forte des technologies relationnelles et objets. Le premier prototype réalisé est basé sur un couplage faible entre un système relationnel existant et le noyau de gestion d'objets que nous avons conçu. Cela permet de tirer partie des mécanismes existants dans le système relationnel et de réduire ainsi le temps de développement.. Nous dégageons finalement plusieurs thèmes nécessaires au support des objets dans un tel système, ils sont évoqués ci-dessous. Nous nous apercevons que les mécanismes à mettre en œuvre et les problèmes soulevés sont très proches de ceux qui existent déjà dans la plupart des systèmes à objets actuels.

La gestion des nouveaux types de données

L'introduction d'un ensemble non figé de types de données dans un système relationnel nécessite la mise en œuvre d'un gestionnaire de définitions pour ces types. Les ADT définis par un utilisateur font partie du schéma de la base de données. Nous les gérons au même titre que les relations dans le catalogue du SGBD. La gestion de ces types nécessite quelques mécanismes particuliers pour traiter des problèmes connus des systèmes à objets comme l'héritage et les dépendances entre types.

Le stockage des objets

Les objets complexes que constituent désormais certaines valeurs d'attribut nécessitent des techniques de stockage adaptées. Nous n'avons pas voulu nous contenter d'un format séquentiel pour les stocker sous forme de chaînes de caractères dans les relations. Nous avons opté pour une approche que l'on peut qualifier de couplage faible et nous stockons les instances d'ADT dans un gestionnaire d'objets à part. Ce dernier gère des objets structurés par les constructeurs du langage, ainsi que la sémantique de partage. Nous avons étudié de nombreux gestionnaires d'objets qui mettent en œuvre des

techniques semblables. Les problèmes de gestion de la persistance, de groupement d'objets, de méthodes d'accès ont été étudiés.

L'exécution de méthodes dans un environnement relationnel

Les gestion des méthodes associées aux ADT présente plusieurs aspects. Il faut tout d'abord définir le langage avec lequel elles sont écrites. Nous avons choisi une approche consistant à étendre un langage existant. Notre mise en œuvre laisse envisager la possibilité d'avoir plusieurs langages d'écriture de méthodes. L'accès aux objets par les méthodes se fait au moyen des fonctions de la bibliothèque du gestionnaire d'objets. Nous avons ensuite introduit un mécanisme d'identification des méthodes permettant au noyau relationnel de les désigner. Un éditeur de liens dynamique est utilisé afin de pouvoir utiliser immédiatement une méthode nouvellement définie ou modifiée. Nous avons enfin étudié les problèmes de résolution tardive.

L'optimisation

Il est nécessaire d'adapter et d'enrichir les techniques d'optimisation relationnelles pour prendre en compte les requêtes qui font intervenir des ADT. Certaines de ces optimisations doivent s'appuyer sur des structures de stockages adéquates (méthodes d'accès, regroupement). Ces aspects ont aussi fait l'objet d'une étude.

I.3 Les résultats

Ce travail a été réalisé dans le cadre du projet ESPRIT EDS. Deux points nous ont particulièrement concerné dans ce projet : la définition du langage relationnel étendu ESQL et la réalisation d'un prototype centralisé permettant de valider ce langage. Ce premier prototype fut réalisé à partir du noyau de SGBD relationnel SABRINA et du gestionnaire d'objets GEODE. Il a permis de valider le compilateur du langage ESQL, son optimiseur logique, et en particulier les modules assurant le support d'ADT. Il est le résultat d'un travail d'équipe.

La part spécifique de mon travail dans ce projet, qui constitue les résultats de ce travail de thèse, comprend les aspects suivants :

- Le premier est une contribution à la définition du support d'ADT dans ESQL et dans son modèle associé. Un effort particulier a été consacré au problème du partage d'objet. Ceci a été réalisé en collaboration avec l'équipe de Georges Gardarin (Infosys), plus particulièrement chargé de la définition de ESQL.
- Le résultat suivant est la définition des mécanismes systèmes permettant de supporter l'extension objet d'un SGBD relationnel. Les besoins en gestion de types, en stockage d'objets et en gestion de méthodes ont été clairement spécifiés.

- Le troisième résultat est la concrétisation du précédent par la réalisation des modules du support d'ADT dans le prototype centralisé. Cette réalisation comprend l'écriture d'un gestionnaire de types, le développement d'un gestionnaire d'instances sur le gestionnaire d'objets GEODE, le compilateur de méthodes, l'implantation du mécanisme d'édition de liens dynamique et d'appel de méthodes. La dernière tâche consistant à coupler ces modules au noyau relationnel SABRINA a été faite en collaboration avec nos partenaires d'Infosys.

Le premier point est présenté dans le chapitre III du plan décrit ci-dessous (en I.4), tandis que les deux suivants constituent les chapitres IV, V, VI et VII.

I.4 Le plan

On peut distinguer trois parties principales dans ce document. La première est un état de l'art présenté dans le chapitre II. La deuxième est l'étude du modèle et du langage, elle constitue le chapitre III. Enfin les chapitres IV, V, VI et VII sont consacrés au support système de l'intégration des objets dans un SGBD relationnel.

Chapitre II

Cet état de l'art présente en premier lieu des travaux basés sur une extension du modèle relationnel. Cela permet de situer et de justifier notre approche dans ce domaine. Il aborde ensuite un certain nombre de systèmes, afin d'étudier les mécanismes dont nous avons besoin pour supporter les extensions objets. En effet, les mécanismes de gestion de types, de stockage et d'exécution se retrouvent dans de nombreux systèmes à base d'objets (SGBD et environnement langage). L'intérêt des différentes techniques de mise en œuvre est évalué par rapport à notre cas. Nous pouvons par exemple conforter nos choix concernant la structure de stockage pour le catalogue de types, l'architecture et les techniques de stockage du gestionnaire d'objets, le langage de développement des méthodes, etc..

Chapitre III

Ce chapitre est consacré au modèle et au langage. Il commence par la description d'une expérience complémentaire, réalisée lors de cette thèse, qui consiste à introduire des facilités de manipulations ensemblistes et associatives dans le langage à objets Guide. La réalisation de cette extension du langage Guide permet de mettre en évidence les faiblesses du système d'objets supportant le langage. Elle montre aussi qu'il est possible d'intégrer de façon homogène les notions de manipulation ensembliste dans un langage à objets. Il est intéressant de comparer ce travail à celui réalisé dans EDS, qui consiste, lui, à introduire le concept d'objet dans un langage fait pour les

manipulations ensemblistes et associatives (SQL). Les extensions au modèle relationnel et le langage ESQL sont présentés en détail dans la deuxième partie du chapitre. Les principes d'intégration de la notion d'objet sont précisemment définis. Une étude est réalisée sur les différents moyens d'introduire la notion de partage et leurs conséquences sur la sémantique des opérations relationnelles.

Chapitre IV

L'architecture générale du SGBD réalisé dans le cadre du projet EDS, ainsi que celle du prototype intégrant les ADT sont présentées dans ce chapitre. Les choix de conception architecturaux pour l'introduction du support d'objets sont exposés. Les modules relationnels impliqués dans le support des ADT sont précisés, ainsi que les trois nouveaux composants devant s'intégrer à l'architecture générale : le gestionnaire de types, le gestionnaire d'instances et le gestionnaire de méthodes.

Chapitre V

Ce chapitre présente la mise en œuvre d'un gestionnaire de types pour le système relationnel étendu. Il aborde des problèmes tels que les structures de stockage du catalogue, la gestion des dépendances entre types et l'héritage.

Chapitre VI

Les choix concernant l'architecture et les niveaux de fonctions du gestionnaire de stockage d'objets sont faits au début de ce chapitre. Nous justifions le choix d'un gestionnaire d'objets structurés spécialisé comme base de développement. Le gestionnaire d'instances centralisé, développé sur GEODE, est décrit. Différents choix de conception sont analysés comme l'adoption d'un stockage décomposé des objets complexes ou le fait de stocker de la même façon les objets (partageables) et les valeurs. Le cas du deuxième prototype sur machine parallèle est ensuite étudié. Le chapitre se termine sur l'étude des techniques d'optimisation d'accès. Une politique de regroupement d'objets est donnée, ainsi que quelques techniques d'optimisation de requêtes.

Chapitre VII

Ce chapitre concerne la prise en compte du comportement des données dans l'extension objet que nous proposons. Il commence par définir le langage qui est fourni à l'utilisateur pour écrire ses méthodes. La façon dont les méthodes accèdent aux objets qu'elles manipulent est ensuite précisée. Enfin les procédures de compilation, d'édition de liens et d'exécution sont décrites.

Chapitre VIII

La conclusion analyse les résultats obtenus et évoque un certain nombre de perspectives.

Chapitre II

Types, objets, et persistance dans les systèmes

Ce chapitre présente un état de l'art en deux niveaux de détail. La première partie (section II.2) présente des travaux ayant comme base le modèle relationnel. Ces derniers sont directement comparables à notre approche qui constitue une extension de système de gestion de bases de données (SGBD) relationnel. Une grande partie du travail réalisé concerne en fait les techniques de mise en œuvre utilisées pour supporter l'extension objet d'un SGBD relationnel. La deuxième partie de ce chapitre est donc consacrée à l'étude de différents mécanismes présents dans de nombreux systèmes (qui ne sont pas tous des SGBD relationnels étendus), et dont nous pouvons nous inspirer.

L'introduction de cet état de l'art situe les travaux qui nous intéressent dans le contexte actuel où l'approche objet couvre désormais les domaines langage, système, et base de données. La section II.2 présente quelques travaux ayant comme base le modèle relationnel. La suite du chapitre est consacrée aux mécanismes qui présentent un intérêt pour notre implantation, à savoir la gestion de types, le stockage d'objets, l'optimisation et la gestion de méthodes. La section II.3 présente et situe rapidement les systèmes qui sont abordés. Les sections II.4, II.5, II.6 et II.7 donnent pour chaque mécanisme une étude détaillée de l'approche suivie dans chacun des travaux présentés.

II.1 Introduction

Le domaine d'application de ce travail de recherche réunit un certain nombre de systèmes informatiques, jusque là toujours considérés indépendamment les uns des autres. Les domaines d'application de plus en plus larges et la nécessité d'aboutir à des réalisations efficaces ont contribué au rapprochement de ces différents systèmes. Les trois acteurs essentiels de cette évolution sont les systèmes d'exploitation, les systèmes de gestion de base de données et les langages de programmation. L'orientation objet a grandement contribué à ce rapprochement, en facilitant la mise en commun des concepts et mécanismes.

Les SGBDs ont rapidement intégré des concepts et mécanismes propres aux systèmes d'exploitation et aux langages. En effet, la mise en œuvre d'un SGBD implique la réalisation de mécanismes de base pour la gestion de disques, de

processus, d'autorisation d'accès, etc., qui sont en général fournis par les systèmes d'exploitation [28]. Toutefois, ces services sont rarement adaptés ou adaptables aux besoins des SGBD, qui redéfinissent alors leurs propres mécanismes, et court-circuitent ceux du système. L'effort est désormais produit lors de la conception de nouveaux systèmes d'exploitation, pour fournir des mécanismes utilisables tels quels pour la mise en œuvre de SGBD. En attendant, de nouveaux types de système ont pris la relève, il s'agit des SGBD extensibles. Ces derniers se présentent sous forme de "super-systèmes d'exploitation", fournissant les outils nécessaires à la construction de tout type de SGBD. Ils sont adaptables et peuvent intégrer facilement de nouveaux mécanismes.

Enfin, les langages de type déclaratif n'étant pas suffisants pour exprimer les traitements complexes des applications, les SGBD ont des besoins au niveau langage de manipulation de données. L'introduction d'un modèle et d'un langage à objets a donné naissance aux SGBD orientés objets qui ont repris la plupart des mécanismes de base des systèmes orientés objets (langages et leur environnement). Par ailleurs, l'introduction de la persistance dans les langages les a rapprochés un peu plus des concepts de manipulation considérés dans les bases de données. La convergence langage-SGBD a donné naissance aux langages de programmation de bases de données (LPBD), encore du domaine de la recherche.

Nous nous intéressons dans un premier temps aux systèmes et modèles qui, comme le nôtre, sont issus du relationnel. L'étude de différentes approches est présentée dans la section II.2. Cela comprend aussi bien l'élaboration de nouveaux modèles, que l'extension plus pragmatique de systèmes existants.

Nous abordons ensuite d'autres types de système pour les mécanismes particuliers qu'ils mettent en œuvre et qui présentent un intérêt pour notre travail. Les différents systèmes étudiés, qui couvrent l'ensemble des domaines cités ci-dessus, ne sont pas présentés de façon complète. Seules sont décrites les parties qui permettent d'étayer le travail qui sera présenté dans la suite. Ainsi seront étudiés différents mécanismes de gestion de types, de stockage d'objet, d'optimisation et enfin de gestion de méthodes. Une présentation générale des systèmes abordés est faite dans la section II.3.

Bien que ces systèmes soient à priori différents, aussi bien dans les services qu'ils doivent assurer, que dans leur niveau de mise en œuvre, ils utilisent un certain nombre de mécanismes assez semblables, que nous allons étudier afin de nous en inspirer pour la réalisation de notre prototype.

- Que ce soit pour les langages ou pour les SGBD, l'utilisateur a la possibilité de définir une fois pour toute la structure des données qu'il va manipuler. Pour les langages, ces définitions de types se trouvent au niveau du source même des programmes, pour les SGBD, elles se trouvent dans la base de données elle-même, gérées par un module spécifique, appelé gestionnaire de catalogue. L'extensibilité des systèmes de types, à savoir la possibilité pour

l'utilisateur de définir de nouveaux types de donnée, ainsi que la persistance de ceux-ci (c'est-à-dire que la durée de vie d'un type n'est plus liée à celle du programme qui l'a défini) ont contribué au développement de modules de gestion de types spécifiques. L'approche à objets a mis en évidence la ressemblance de ces gestionnaires de types qui gèrent désormais des entités fort semblables : un type comprend la description d'une structure de données, ainsi qu'un ensemble de fonctions de manipulation qui traduisent un certain comportement. Nous allons étudier les différents mécanismes de gestion de types présents dans les systèmes abordés dans ce chapitre.

- Nous appelons *instance* d'un type une donnée conforme au modèle défini par le type (un exemplaire). L'ensemble des instances constitue les données de l'application. Une instance est en fait un ensemble de données éventuellement structuré que nous appelons *objet*. La gestion de la persistance des données est assurée par un module que nous appelons *gestionnaire d'objets*. Ce module est également étudié dans les systèmes présentés.
- Pour finir, le modèle à objets a introduit la notion de comportement associé aux objets eux-mêmes. Cela veut dire que l'on associe à une structure de données un ensemble d'opérations appelées *méthodes*. Nous nous intéresserons plus particulièrement au langage utilisé pour écrire ces méthodes, ainsi qu'aux mécanismes d'exécution mis en œuvre.

II.2 Le modèle relationnel à l'origine de nombreux travaux

Nous assistons actuellement à un rapprochement entre les bases de données et les langages de programmation. La complexité des applications de base de données ainsi que la volonté de fournir des structures de données persistantes dans les langages de programmation en sont les raisons. C'est ainsi que l'on introduit la possibilité de décrire certains aspects comportementaux dans les bases de données, et certaines caractéristiques issues des bases de données dans les langages. Alors que jusqu'à maintenant un SGBD ne gérait que des données purement descriptives, il est désormais possible d'y trouver des informations relatives au comportement des données sous forme d'un certain nombre de procédures. On tente par ailleurs d'introduire au niveau d'un langage de programmation les notions de persistance et d'accès associatif. Un point fondamental de convergence apparaît dans la notion de **type** des langages et celle de **schéma** des bases de données. On trouve dans les SGBD relationnels la notion de *relation* et dans les SGBDOO la notion de *classe*, qui représentent à la fois une définition de structure de données et un ensemble d'entités conformes à cette définition, mais qui sont à rapprocher de la notion de type des langages. Ceci conduit à une évolution des modèles de données des SGBD qui donne lieu à de nombreuses actions de recherche ([61] [1] [49] [51] [12]).

Ces travaux se sont largement inspirés des concepts que l'on trouve dans les langages à objets, des types abstraits, et des modèles d'objets complexes. Ils intègrent notamment les notions de comportement associé aux données, d'objets construits, d'encapsulation, d'héritage et d'identité d'objet qui sont introduites au niveau du schéma d'une base de données.

De nombreuses approches ont comme origine le modèle relationnel, un standard de fait, dont les bases algébriques assurent une bonne capacité d'optimisation. On reproche cependant à ce dernier une trop faible puissance de modélisation face à la complexité des nouvelles applications. Il existe plusieurs façons de concevoir une extension ou même un nouveau modèle à partir du relationnel :

- Une des premières approches consiste à supporter un modèle sémantique sur un système relationnel.
- Schek ([49] [51]) voit une évolution du modèle relationnel vers un modèle d'objets complexes qui passe par le relationnel "imbriqué" ou NF2 (Non First Normal Form).
- D'autres ont une approche plus pragmatique qui consiste à concentrer l'extension apportée au modèle relationnel dans la notion de domaine pour y inclure des valeurs plus complexes.

Nous décrivons ces trois approches (modèle sémantique, objets complexes et extension de domaine) dans les trois sections suivantes.

II.2.1 Support d'un modèle sémantique

GEM [53] est une extension d'INGRES qui apporte au modèle relationnel certains concepts issus des modèles sémantiques comme la notion d'entité avec identificateur et la généralisation.

Une entité GEM est assimilable à une relation, mis à part le fait qu'il est possible d'avoir des *attributs multi-valués,* ainsi que des *attributs références.* Ces derniers ont pour valeur des occurrences d'entité (une occurrence est l'équivalent d'un n-uplet relationnel).

```
DEPARTEMENT (Nom: c, Etage: i2, Bureaux: {i2})
    key(Nom);
EMPLOYE (Nom: c, Dept: DEPARTEMENT) key(Nom);
```

L'attribut *Bureaux* est multi-valué, il aura pour valeur un ensemble d'entiers. La valeur de l'attribut *Dept* d'un employé est une occurrence de l'entité DEPARTEMENT. Il faut noter que *Dept* représente bien une référence sur une occurrence de l'ensemble des départements, et qu'il est ainsi possible de partager des occurrences. Si l'on veut connaître le nom du département de l'employé Lopez, on écrira simplement :

```
retrieve (EMPLOYE.Dept.Nom)
where EMPLOYE.Nom = "Lopez";
```

GEM permet aussi de gérer le concept de généralisation d'entité. On peut par exemple définir l'entité EMPLOYE de la façon suivante :

```
EMPLOYE (Nom: c, Dept: DEPARTEMENT,[EMPMARIE
    (Conjoint: i4)])key(Nom);
```

Cette exemple définit en fait une sous-entité EMPMARIE, qui possède en plus l'attribut *Conjoint*.

L'implantation de GEM est faite sur un système relationnel. Un schéma GEM est traduit en un schéma relationnel pur dans lequel sont générés des attributs et relations supplémentaires pour gérer les identificateurs d'occurrences d'entité, ainsi que les attributs multi-valués.

Ce modèle permet de partager des occurrences d'entité entre d'autres occurrences. La notion de comportement associé aux entités est absente et les objets complexes ne sont pas construits de façon homogène (le n-uplet est géré au niveau de l'entité, l'ensemble au niveau de l'attribut, tandis que l'ensemble des occurrences d'une entité est géré implicitement). La génération d'un schéma GEM en relationnel pur pose des problèmes de performance, car il se décompose en de nombreuses relations et les accès nécessitent de nombreuses opérations de jointure.

II.2.2 Du relationnel à l'objet complexe

Cette approche vise une évolution du modèle relationnel vers un modèle d'objets complexes. Un tel modèle est obtenu en levant les limitations de combinaison des constructeurs *ensemble* et *n-uplet* ([49]).

- Il n'existe qu'un seul constructeur dans le modèle relationnel, la *relation* ; celle-ci est en fait une combinaison de deux autres, puisqu'elle représente un *ensemble* de *n-uplets*. L'ensemble des combinaisons possibles de ces constructeurs se réduit à cette unique forme.
- Dans un modèle de *relations imbriquées* (NF2 : Non First Normal Form [50]), il est possible de combiner ces constructeurs de façon à ce qu'une *relation* puisse être le composant d'un *n-uplet*.
- Finalement une libre combinaison de ces constructeurs conduit aux modèles d'objets complexes ([59] [2]).

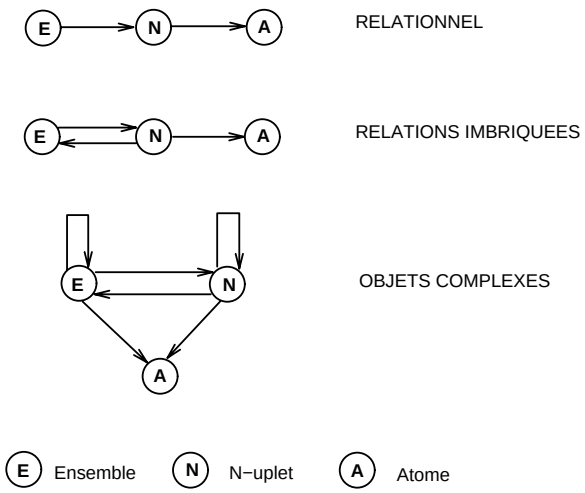

Fig. 2.1 : Du relationnel à l'objet complexe

La figure Fig. 2.1 montre les combinaisons possibles des constructeurs *n-uplet* et *ensemble* pour les différents modèles évoqués. A partir d'un constructeur donné, la flèche indique quels peuvent être les composants. Les atomes représentent les types de base. Une combinaison se termine toujours sur un *atome*. Dans le modèle de *relations imbriquées*, elle commence toujours par un *ensemble.*

Dans un modèle d'objets complexes, les objets sont obtenus par application itérative de constructeurs de *n-uplets* et d'*ensembles* à des objets atomiques. Les langages de manipulation sont des algèbres, des calculs ou des langages basés sur la programmation logique. Les modèles obtenus sont finalement assez éloignés du relationnel et ne permettent de traduire que la sémantique structurelle des données, et non pas les aspects dynamiques (comportement). Nous nous intéresserons en fait à des travaux beaucoup plus proches du relationnel, aboutissant si possible à des systèmes compatibles avec le relationnel pur.

II.2.3 Extension du modèle

Nous considérons ici deux SGBD relationnels étendus en généralisant la notion de domaine : il s'agit de POSTGRES et SABRINA-LISP. Nous nous contentons de noter qu'il est envisagé d'introduire des types de données externes (sous forme de types abstraits pour définir de nouveaux domaines) et des objets complexes dans le SGBD relationnel extensible STARBURST ([35]), mais que ce ne sont que des perspectives (une première étude est faite dans [61]).

SABRINA-LISP ([36], [40]) permet de définir de nouveaux types d'attribut grâce à son concept de type abstrait (ADT). Les instances de ces types ainsi que les fonctions LISP qui les manipulent sont stockées sous forme de chaînes de caractères dans les relations. Elles sont traitées par un interpréteur LISP intégré au SGBD. Plus de détails sont donnés dans les sections II.3.1 et II.7.3. Ce système ne fournit pas la notion de partage d'objet (identité d'objet), ni la notion de constructeur au niveau

du modèle. Les seules constructions possibles sont celles du langage et sont cachées au système qui ne peut mettre en œuvre des techniques de stockage d'objets complexes adaptées.

POSTGRES ([48], [56]) propose une extension du modèle relationnel en y ajoutant les concepts suivants :

- la notion de type abstrait de donnée (ADT),
- des données de type procédure,
- des règles,
- la notion d'héritage multiple entre relations ; il est possible de définir une sous-relation qui hérite des définitions d'attributs de ses relations "mères".

Nous allons décrire rapidement la notion d'ADT de POSTGRES. Il existe un certain nombre de types atomiques prédéfinis (*int2*, *float*, *bool*, *char*,...) utilisés dans les définitions d'attribut. Ils possèdent des opérateurs arithmétiques et de comparaison standards. L'idée est de permettre à l'utilisateur d'étendre cet ensemble de types en lui donnant la possibilité de définir de nouveaux types atomiques (les ADT), avec leurs opérateurs associés. Un ADT est défini dans POSTGRES par un nom, la longueur de la représentation interne d'un objet de ce type en octets, des procédures de conversion entre le format interne et le format externe, et une valeur par défaut. Le format interne est celui sous lequel est stocké un objet du type dans la base, le format externe est la représentation qui permet l'affichage et la saisie. La commande de création d'un ADT *boîte* représentant un rectangle est la suivante :

```
define type boîte is (InternalLength = 16,
InputProc = CharToBoîte,
OutPutProc = BoîteToChar,
Default = "")
```

La représentation externe d'une boîte peut être par exemple une chaîne de caractères donnant les coordonnées des coins inférieur gauche et supérieur droit de la boîte (*"10,20:30,40"*). La procédure *CharToBoîte* transforme alors une telle chaîne en une représentation interne qui peut être quatre valeurs de quatre octets chacune.

Il est possible de définir des opérateurs associés à ces ADT. L'utilisateur définit pour cela des *procédures* qui permettent d'implanter : les conversions de représentations (interne/externe), les opérateurs, et finalement les fonctions qu'il veut simplement pouvoir exécuter lors des requêtes. On définit une procédure par son nom, le type de ses arguments, le type de retour, le langage dans lequel elle est écrite, et finalement le fichier qui contient le code source.

```
define procedure Surface (Boîte) returns int4 is
(language = "C", filename = "surf.c")
```

En dehors des types atomiques (prédéfinis ou ADT), POSTGRES permet de définir des types construits et des types procédures. Il n'existe qu'un seul constructeur pour définir des types construits, il s'agit du tableau. Les valeurs d'attributs de type procédure sont des requêtes POSTQUEL paramétrées ou non, et

sont donc assimilables à des relations (l'ensemble des n-uplets retournés par la requête).

On remarque qu'il apparaît deux niveaux : celui des relations, et celui des ADT. L'héritage est défini pour le premier mais pas pour le deuxième. Il n'est pas possible de partager des valeurs d'ADT. La définition même des ADT est assez lourde, puisqu'il faut introduire au niveau du schéma des caractéristiques d'implantation comme la taille des représentations internes. Il n'est pas possible de définir des ADT de taille variable (puisque la taille est fixée à la définition). Le support d'objets complexes est assez pauvre et non homogène, puisqu'il faut recourir à la notion de type procédure pour représenter autre chose que des tableaux.

II.2.4 Orientation du projet

Un impératif du projet dans lequel ce travail est réalisé est d'aboutir à un système compatible avec le relationnel. Nous avons pour cela adopté une approche par extension de domaine.

Nous voulons aussi introduire la notion d'objet complexe, ainsi que la notion de comportement. Ceci est fait en adoptant de façon intégrée les notions de type abstrait et de constructeurs, ce qui n'est pas le cas dans POSTGRES où ADT et constructeurs sont traités indépendamment. Pour cette raison les constructeurs sont introduits sous la forme de types abstraits particuliers, dits génériques. Les fonctions d'accès aux structures de données construites se présentent alors comme des méthodes associées aux types abstraits génériques. Nous voulons que les objets complexes puissent être construits de façon homogène, contrairement à POSTGRES qui propose deux facilités de construction d'objets, les tableaux et les types procédures. Nous fournissons donc un seul ensemble de constructeurs qui représentent la seule façon de définir des objets complexes. Nous ne voulons pas non plus que la structure des objets soit imposée par le langage d'implantation des fonctions associées comme c'est le cas dans SABRINA-LISP. C'est pourquoi l'ensemble des constructeurs que nous fournissons définit en fait un format unique de données qui est commun aux différents langages d'implantation des méthodes.

Finalement nous prenons aussi en compte la notion d'héritage au niveau des ADT (et non des relations comme dans POSTGRES) et de partage d'objet. Cette dernière caractéristique est présente dans GEM, mais n'autorise dans ce dernier que le partage de n-uplet. Nous autorisons le partage de tout objet complexe.

II.3 Un tour d'horizon

La suite de cet état de l'art présente un SGBD relationnel étendu (SABRINA), un SGBD extensible (EXODUS), des gestionnaires de stockage d'objets de bas niveau (GEODE, Cricket), des SGBD orientés objet (O_2, ITHACA) et enfin un

environnement système à objets (GUIDE. Nous donnons dans cette section une présentation générale de chacun de ces systèmes.

II.3.1 SABRINA-LISP

SABRINA est un SGBD relationnel étendu développé en partie à l'INRIA, et commercialisé par la société INFOSYS. Le traitement d'objets complexes a été introduit en étendant la notion de domaine de valeur des attributs, traditionnellement limitée à des types de base ([40], [36]).

Le domaine de valeurs d'un attribut peut désormais être défini par un type abstrait (ADT). Un ADT est décrit en LISP, au moyen d'une fonction de domaine, qui permet de déterminer si un objet LISP appartient ou non au domaine défini par le type, et par des fonctions jouant le rôle d'opérateurs. Il est facile de définir en LISP des structures de données complexes (liste, ensemble, etc..). Les instances de ces ADTs sont manipulées par l'interpréteur LISP (qui devient un module du SGBD), appelé par la machine algébrique lorsqu'une fonction doit être appliquée. Les instances, ainsi que les fonctions, sont stockées dans les tables relationnelles sous forme de chaînes de caractères (les fonctions sont dans les tables qui constituent le catalogue).

Le langage de définition et de manipulation fourni est un SQL étendu, appelé *SQL-objet.* Voici par exemple la définition du domaine des formes rectangulaires, RECTANGLE, représentées par deux valeurs numériques :

```
(DD #:T:RECTANGLE (X)
  (and (listp X)
       (numberp (car X))
       (numberp (car (cdr X)))
       (null (cdr (cdr X)))))
```

Une valeur de type RECTANGLE doit donc être une liste de deux nombres, la longueur, et la largeur. On peut alors définir un opérateur calculant la surface d'un objet de type RECTANGLE :

```
(DE #:(#:RECTANGLE):SURFACE (X)
   (* (car X) (car (cdr X))))
```

Si R est une relation dont l'attribut COTE est de type RECTANGLE, alors la requête suivante recherche les formes de surface supérieure à 100 :

```
SELECT *
FROM R
WHERE SURFACE (COTE) > 100
```

II.3.2 EXODUS

L'approche d'EXODUS ([19] [20]) n'est pas de fournir un SGBD apte à supporter tous les types d'application, mais plutôt de fournir un générateur de SGBD, qui permettra de générer un SGBD par type d'application. Le système se

présente donc sous la forme d'un ensemble de composants de base, immuables, autour desquels il sera possible de construire un SGBD, à l'aide d'outils appropriés. Il apparaît une nouvelle notion, celle d'"implanteur" de SGBD, qui sera la personne chargée de développer le SGBD voulu à partir d'un outil comme EXODUS.

Les composants fixes sont le gestionnaire de stockage d'objets, qui assure la persistance des objets, et le gestionnaire de types. EXODUS fournit en plus un certain nombre d'outils, comme le langage de programmation E (extension persistante de C++), pour développer de nouveaux composants, ainsi que différentes librairies contenant par exemple des fonctions de gestion d'index (b-arbres,"hash-code", etc.), facilitant l'écriture de nouvelles méthodes d'accès, et un générateur d'optimiseurs de requêtes.

II.3.3 GEODE

GEODE est un gestionnaire de stockage d'objets complexes développé par la société INFOSYS et l'INRIA [14]. Il sert de système de base au concepteur de SGBD. Il fournit un support de persistance pour tout objet manipulé dans n'importe quel modèle de données (relationnel, orienté objet, déductif). Ces objets peuvent être simples ou complexes, volumineux ou compacts. Les objets sont désignés par un identifieur unique. Des facilités de regroupement d'objets sont offertes, ainsi que des mécanismes de reprise après panne et de contrôle de la concurrence d'accès (ces derniers ne figurent pas dans la dernière version et sont en cours d'implantation). La gestion des types est laissée à la charge de l'application.

La conception du système est basée sur le principe d'extensibilité d'une part, et sur l'hypothèse grande mémoire d'autre part. Cela veut dire que le concepteur de SGBD peut intervenir à tout niveau de GEODE pour développer de nouveaux mécanismes, et que les fonctions existantes prennent en compte le fait que la base de données doit généralement "tenir" en mémoire centrale (sur les machines à venir...).

On distingue deux parties principales dans GEODE : la mémoire virtuelle et le support des constructeurs. La première fournit les fonctions assurant la persistance, la création, et la suppression de chaînes d'octets dans la mémoire virtuelle. Celle-ci est découpée en pages, regroupées dans des segments. Les chaînes d'octets créées dans des segments sont adressées à partir d'identificateurs logiques, gérés par le système au moyen de tables. Le niveau des constructeurs fournit des structures de données et des fonctions pour manipuler des objets construits tels que les n-uplets, les listes, les ensembles ou les tableaux.

II.3.4 CRICKET

Cricket [54] est un système de stockage de données servant de base à la conception d'environnements et de langages de programmation persistants. Sa particularité réside dans la mise en œuvre d'une mémoire virtuelle persistante, qui représente un

unique niveau de stockage et de manipulation d'objets. Il est réalisé en utilisant les primitives de gestion de mémoire du noyau MACH. Le système n'a aucune connaissance de la sémantique ou de la structure des objets gérés, il se contente d'assurer la persistance de la zone mémoire qu'il gère. Un contrôle des accès concurrents est assuré au niveau des pages. Ce système d'assez bas niveau ne contrôle pas le format de représentation des objets manipulés par l'application ou le SGBD.

II.3.5 O2

O2 est un SGBD orienté objet développé par le Groupement d'Intérêt Public ALTAIR. Il présente les caractéristiques d'un système de gestion de bases de données, mais propose un modèle de données et un langage orienté objet [42].

Une architecture distribuée de type client/serveur est proposée. Le serveur assure le stockage des données tandis que les clients représentent les différents utilisateurs de la base de données (stations de travail).

Le langage de programmation offert par le système est composé d'un langage de définition de données, O2, pour définir la hiérarchie de classes, et d'un langage existant pour programmer le code des méthodes. L'approche est multi-langage, et C, Basic et Lisp sont actuellement disponibles (CO2, BasicO2, LispO2).

O2 fournit les notions de valeur et d'objet. Une valeur est l'instance d'un type, tandis qu'un objet est l'instance d'une classe. Un objet est composé d'une valeur associée à un identificateur d'objet (OID) et se trouve donc partageable. L'unique moyen de manipuler un objet est l'ensemble des méthodes de sa classe (encapsulation). Une classe se définit en fait comme un ensemble d'objets de comportement identique, elle comprend une structure de données, définie par un type, et un ensemble de méthodes. Les types sont construits à partir de types atomiques (string, float, integer,...), d'autres types et de trois constructeurs, *tuple*, *set* et *list*.

Les objets peuvent être créés dynamiquement lors de l'exécution d'un programme, par l'opération *new*. Les valeurs et les objets peuvent persister : il faut pour cela qu'ils soient nommés ou qu'ils soient composants d'entités persistantes. O2 fournit un mécanisme d'héritage basé sur la notion de sous-type telle qu'elle est définie par Cardelli [17]. Enfin, un mécanisme de "résolution" tardive permet de déterminer quel est le code de la méthode à appeler à l'exécution, plutôt qu'à la compilation.

II.3.6 ITHACA

Un des objectifs principaux du projet ESPRIT ITHACA (No 2121) est le développement d'un noyau supportant la programmation à objets et la persistance. Ce noyau comprend d'une part un langage de programmation (CooL) et d'autre part un SGBD orienté objet (NooDLE). CooL est un langage de programmation qui

supporte la persistance des données et intègre les concepts de l'orientation objet. Le noyau NooDLE présente deux interfaces : une première interface fonctionnelle permet de supporter le langage CooL ; une deuxième interface, appelée NooQL, donne un accès de type SQL étendu.

Le noyau NooDLE est implanté sur un SGBD relationnel (DDB/4). Il possède un modèle de données interne, NO2, qui intègre la définition de types de données, ainsi que leurs extensions (conteneur de l'ensemble des instances). Des types construits peuvent être créés à partir de constructeurs *tuple*, *set* et *tensor* (tableau à plusieurs dimensions) appliqués sur des types de base ou sur des références à des types.

II.3.7 GUIDE

GUIDE est un projet mené conjointement depuis 1986 par le Centre de Recherche Bull et le Laboratoire de Génie Informatique de Grenoble. Il fait aussi partie du projet ESPRIT COMANDOS (Construction and Management of Distributed Open Systems). GUIDE est une plateforme orientée objet de développement d'applications distribuées.

L'adoption d'un modèle objet a facilité l'intégration des concepts et méthodes propres aux langages de programmation et aux systèmes d'exploitation. Cela permet en outre de tirer partie des propriétés de structuration et d'encapsulation de données, d'héritage, de réutilisation, et peut permettre la coopération de systèmes hétérogènes en utilisant des réalisations différentes d'une même interface objet.

GUIDE fournit un langage orienté objet de haut niveau pour le développement d'applications distribuées. Il possède un système de stockage distribué des objets. Le mécanisme d'exécution est basé sur la notion de *domaine*, une machine virtuelle multi-activités à laquelle les objets sont liés dynamiquement et qui peut s'étendre sur plusieurs nœuds du système. Un *domaine* est composé de plusieurs activités concurrentes travaillant sur des objets. La communication entre activités, ou entre *domaines* se fait à travers les objets. La distribution reste transparente à l'utilisateur.

II.3.8 Classification des systèmes présentés

Afin de mieux comprendre la description de ces systèmes, nous allons les situer dans le contexte décrit ci-après. Nous allons considérer trois niveaux :

- Le premier et le plus élevé est le niveau langage ou modèle. Il représente l'implantation d'un langage ou d'un modèle de données. C'est ici qu'intervient un gestionnaire de types, afin de gérer la représentation d'une application dans le modèle du système.
- Le deuxième niveau concerne la gestion d'objets structurés, ainsi que la gestion des méthodes. On trouve ici des techniques de stockage d'objets complexes et des mécanismes d'exécution de méthode.

- Le dernier niveau représente la couche basse d'un système à objets persistants. Il assure le stockage au plus bas niveau.

La situation des systèmes présentés par rapport à ces trois niveaux est montrée sur la figure Fig. 2.2.

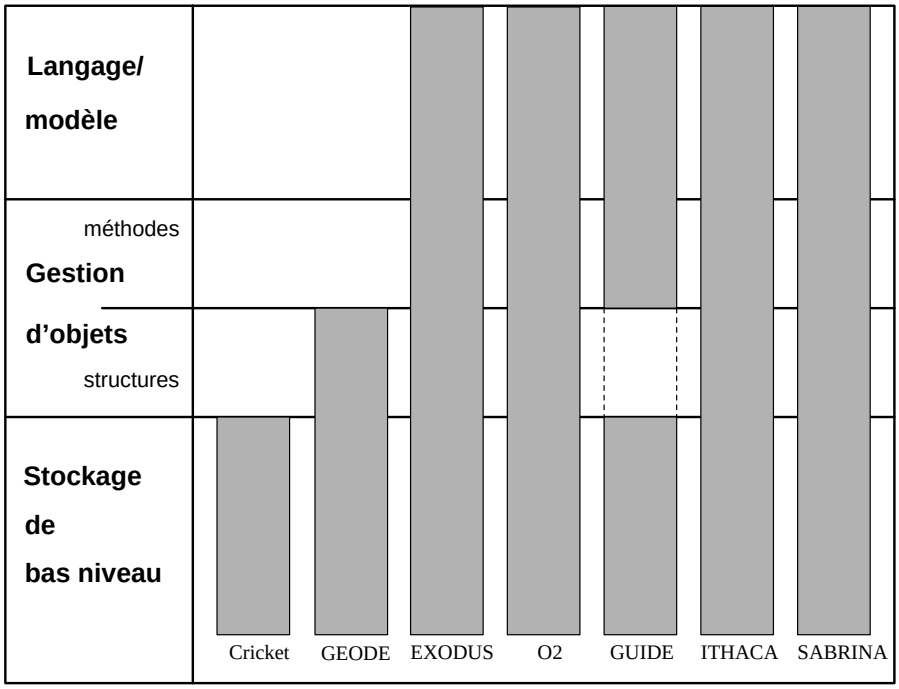

Fig. 2.2 : Classification des systèmes

Le système GUIDE n'offre pas de structures particulières pour le stockage d'objets complexes. Pour EXODUS, on peut considérer que le langage *Persistent E* lui permet de couvrir tout le spectre de la figure, bien qu'il ne se présente pas comme un système complet.

II.4 Gestion de types

La notion de type apparaît en premier lieu dans les langages. Elle a été enrichie dans les langages de programmation orientés objets, notamment avec les notions de types abstraits et d'héritage. Une étude sur un certain nombre de théories des types et la façon de les adapter pour représenter les concepts présents dans les langages orientés objets est faite dans [25]. Ces mêmes types apparaissent naturellement dans les SGBD qui maintiennent les données manipulées par des langages de programmation. Toute application définit un ensemble de structures de données sur lesquelles les programmes vont travailler. En première approximation, la définition de la structure de données d'une entité manipulée par l'application constitue son type. Dans une approche "type abstrait", un type est défini par l'ensemble des opérations permettant de manipuler une instance ; la définition de la structure de

données et le code des opérations constituent alors une implantation particulière du type, qui est cachée à l'utilisateur. Dans un SGBD relationnel les structures de données visibles par l'utilisateur sont les tables définies dans le schéma.

L'idée générale est de gérer la définition structurelle et parfois comportementale des données d'une application. Certains systèmes isolent cette gestion dans un module spécifique. Cependant il arrive souvent que le gestionnaire de types ne soit pas défini comme un module à part entière et que ses fonctions soient mises en œuvre de manière ad hoc dans différents composants du système.

Nous allons tenter de décrire la gestion de types d'un certain nombre de systèmes, afin d'en retirer les spécificités et les mécanismes communs.

II.4.1 Gestion de types pour un SGBD relationnel

Les types de données manipulés dans un système relationnel se limitent aux n-uplets, aux relations et aux types de base tels que les types numériques ou les chaînes de caractères. L'ensemble des types est relativement statique ; les relations constituent la seule la partie dynamique et sont gérées dans le catalogue relationnel. La définition d'une relation détermine à la fois le type des n-uplets qu'elle doit contenir et le nom de l'ensemble des instances de ce type. Ces définitions sont stockées dans le catalogue qui est consulté lors des phases de contrôle de type.

Il existe deux alternatives pour construire un catalogue relationnel : soit il est conçu comme une métabase, soit il est réalisé de manière ad hoc. Dans le premier cas le catalogue est stocké sous forme de relations ; on lui assure ainsi à moindre frais la même fiabilité qu'aux autres données de la base. Il est cependant soumis à des accès très fréquents et se trouve sujet à de nombreux accès concurrents, le plus souvent en consultation. Il peut rapidement devenir un goulot d'étranglement du système. C'est pourquoi il est parfois construit de manière ad hoc, afin d'accélérer les accès et d'optimiser le contrôle de la concurrence auquel il est soumis. Le gestionnaire de types qui a été développé dans ce travail de thèse est une extension d'un catalogue relationnel ; sa mise en œuvre est un compromis entre une approche spécifique et une métabase.

II.4.2 Gestion de types et de classes dans GUIDE

Un type GUIDE définit l'interface de manipulation de ses instances. Il comprend les signatures de méthodes et des variables d'état. Une classe est une réalisation particulière d'un type et contient le code des méthodes. On distingue la classe source, qui contient le code GUIDE, de la classe objet, qui contient le code exécutable et qui est chargée dans le système comme n'importe quel autre objet. Chaque type et chaque classe sont actuellement stockés dans un fichier UNIX. Le compilateur utilise ces fichiers pour effectuer ses contrôles de types. Cette solution n'est pas satisfaisante, car l'accès aux définitions est difficile. C'est un exemple de

système où la gestion de types n'a pas été conçue comme un module spécifique, mais se trouve intégrée dans le compilateur. Un gestionnaire de types est en cours d'étude.

II.4.3 Gestion de classes O2

Un type O2 définit une structure de donnée, tandis qu'une classe se compose d'un type associé à un ensemble de méthodes. Le gestionnaire de types et classes dans O2 constitue une métabase dans laquelle chaque définition de classe est représentée par un objet. Les liens inter-classes (héritage, composition) sont représentés par des références entre objets. C'est l'exemple même du système qui utilise ses propres mécanismes de gestion de données pour stocker les définitions de type.

II.4.4 Le gestionnaire de types d'EXODUS

Le langage de programmation E [52], utilisé pour construire les composants d'un SGBD, est une extension persistante de C++. Les types définis en utilisant ce langage sont catalogués dans un module appelé gestionnaire de types. Ce dernier maintient la hiérarchie des classes (C++) et le catalogue des fichiers dans lesquels sont stockées les instances de ces classes. Une classe définit soit un type de base, soit un type construit. L'héritage est multiple. Il est possible de définir de nouveaux types de base en E. Les types construits utilisent des constructeurs de tableaux, d'ensembles ou d'enregistrements. Le gestionnaire de types gère ces définitions, il maintient en particulier le code source E, les représentations intermédiaires, les modules objets des opérations définies sur les types abstraits, ainsi que des requêtes précompilées. Il gère les relations de dépendance entre ces différents éléments, qui guident notamment les procédures de recompilations automatiques.

Ce gestionnaire fait apparaître de nouvelles fonctionnalités. La description du comportement des données dans la définition des types, sous forme d'opérations, nécessite de gérer les modules sources et objets de ces opérations. Le code source des opérations fait partie de la définition d'un type. Le module objet doit être regénéré à chaque fois que le source est modifié. Nous verrons que dans notre approche, nous laissons à l'utilisateur le choix de déclencher ou non la recompilation d'une fonction.

II.4.5 Gestion de types dans NooDLE

NooDLE [26] est le sous système de gestion d'objets de l'environnement ITHACA. Il supporte le modèle de données NO2 qui est une extension du modèle de données O2 [42]. Il utilise à la fois un SGBD relationnel, pour le stockage des informations nécessaires à la gestion d'objets, et le système de fichiers du système d'exploitation hôte, pour stocker le contenu des objets. Ce gestionnaire maintient un *tas persistant* pour le langage de programmation Cool. Nous nous intéresserons plus

particulièrement au gestionnaire de stockage qui est composé des modules suivants : le gestionnaire de types, centre d'intérêt de cette section, gère les types définis par l'utilisateur et supportés par le modèle NO2 ; le gestionnaire de références gère les références et les chemins d'accès aux objets ; le gestionnaire de fichiers d'objets fournit une interface pour stocker les objets dans le système de fichiers du système d'exploitation ; finalement un gestionnaire de transactions s'occupe des aspects reprise et accès concurrents. Les trois derniers modules seront abordés dans la section II.5.

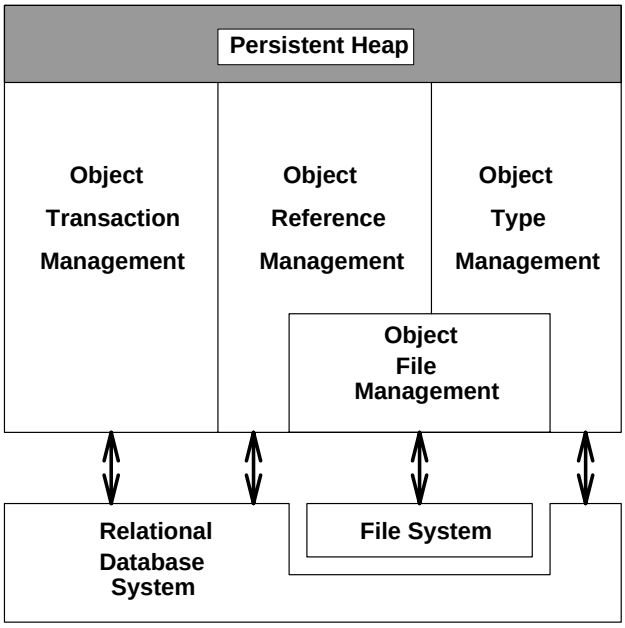

Fig. 2.3 : Architecture de NooDLE

La description originale d'un type NO2, qui peut être très complexe, est stockée dans un fichier. Le SGBD relationnel se contente de maintenir la liste des types existants dans une table, avec comme informations associées à chaque type, un index de l'enregistrement (dans le fichier) contenant la définition du type, un compteur d'utilisation qui indique le nombre de types qui référencent celui-là, ainsi qu'un compteur d'extensions indiquant le nombre de collections d'objets de ce type. Les compteurs de références et d'extensions permettent d'interdire la suppression ou la modification d'un type utilisé par ailleurs ; un mécanisme similaire sera utilisé dans notre mise en œuvre. Les définitions de type sont maintenues dans un fichier, sous forme de structures de données adéquates (ici des enregistrements Pascal). On distingue les types atomiques (integer, fixed, float, string, bool) des types construits au moyen des constructeurs *tuple*, *set*, et *tensor*. Ces derniers sont stockés de façon décomposée, c'est-à-dire que chaque composant d'un type construit est simplement référencé et constitue un autre type, dont le nom a pu être généré implicitement, s'il correspond à l'utilisation interne d'un autre constructeur dans la définition du type

construit. Finalement, le SGBD gère une table, avec une entrée par application ; cette entrée permet d'accéder à la liste des types utilisés par l'application. Cette dernière facilité est utilisée pour précharger les définitions de types d'une application dans les tampons du SGBD.

Plusieurs points nous semblent intéressants et discutables dans cette réalisation.

- Le préchargement des types par application ne nous semble intéressant que si des accès au catalogue sont nécessaires lors de l'exécution d'une application, ce qui n'est pas le cas dans une approche compilée comme la nôtre.
- La génération implicite de types nommés correspondant aux composants construits d'un type est intéressante, car elle permet la manipulation de valeurs structurées dont le type n'a pas été explicitement défini. Elle présente cependant l'inconvénient de surcharger le catalogue en nombre de définitions et rend les accès plus complexes, puisqu'il faut reconstituer une définition à partir de celles des composants. Nous ne l'adopterons pas.
- La technique des compteurs d'utilisation est intéressante, bien qu'elle ne permette pas de retrouver les types qui utilisent un type donné. Nous utiliserons cependant une solution similaire.
- Finalement nous retrouvons ici une technique que nous avons employée : elle consiste à stocker chaque définition de type complexe de manière ad hoc et à maintenir dans une table relationnelle la liste des types existants avec une référence sur leur définition et des informations de gestion comme le compteur d'utilisations.

II.4.6 Conclusion

L'extensibilité des systèmes abordés contribue à la dynamicité de l'ensemble des types de données qui doit être prise en compte au niveau du gestionnaire de définitions. L'introduction des modèles d'objets complexes et d'autres concepts liés à l'approche objet complique les définitions à gérer et introduit un certain nombre de relations entre types (héritage, composition). Les algorithmes de manipulation de définitions sont plus complexes et nécessitent des structures de données adaptées, très souvent construites de manière ad hoc. On s'aperçoit que de nombreux gestionnaires de types sont mis en œuvre suivant des techniques tout à fait spécifiques. L'autre solution consiste à gérer le catalogue comme une métabase, c'est-à-dire à le stocker de la même manière (sous le même format) que les autres données de la base. Dans certains systèmes orientés objet, le code des méthodes est considéré comme faisant partie de la définition des types ; il doit alors être géré par le catalogue qui peut se trouver responsable de la maintenance des codes sources et objets. Il doit alors mettre en œuvre des procédures de recompilation qui décident de la recompilation des modules en fonction des modifications apportées aux définitions. Enfin la gestion de types reste liée à un modèle de données, mais présente presque toujours les mêmes fonctions.

II.5 Stockage d'objets

Nous présentons ici un certain nombre de gestionnaires de stockage afin d'étudier les différentes techniques de mise en œuvre, ainsi que les niveaux de fonction assurés. Les systèmes de gestion de bases de données extensibles ou orientés objet, ainsi que les langages de programmation persistants se basent sur un module appelé *Gestionnaire d'Objets* pour assurer le stockage et la fiabilité de leurs données. On distingue trois façons de réaliser un tel module : il peut être construit sur un système de gestion de fichiers adapté (PICK, WISS) ou sur un SGBD, ou alors il peut être conçu de toutes pièces. Les objets gérés dans ces systèmes sont en général des chaînes d'octets sans aucune sémantique associée. De plus en plus, on essaie de paramétrer les mécanismes pour prendre en compte la sémantique d'un modèle ou d'une application au plus bas niveau, dans un souci de performance. On retrouve dans ces systèmes un ensemble de caractéristiques communes, ce qui laisse penser qu'un tel module pourrait être utilisé dans plusieurs projets, plutôt que d'être reconstruit à chaque fois. Voici une liste des fonctions essentielles supportées par un gestionnaire d'objets :

- Gestion du disque : cette fonction concerne essentiellement l'allocation de l'espace disque.
- Gestion de la concurrence : les SGBD sont multi-utilisateurs, la cohérence des données doit être assurée en contrôlant les accès conflictuels.
- Gestion de la reprise : un arrêt prématuré du système, dû à une panne matérielle ou logicielle, ne doit pas mettre en cause l'état cohérent d'une base de données. Ces mécanismes ont pour objectif de faire redémarrer le système avec un état stable de la base (le plus récent possible).
- Gestion de groupes d'objets ("clusters") : afin de limiter le nombre d'accès au disque, les objets fréquemment consultés ensembles sont stockés dans les mêmes pages. Il doit donc être possible de donner au gestionnaire d'objets des directives de placement.
- Gestion de tampons : il s'agit de gérer les zones de données du SGBD en mémoire principale. Des politiques de chargement/déchargement des pages de données entre le disque et les tampons sont définies.
- Gestion des structures de stockage : suivant le type de données à traiter, on peut utiliser des structures de stockage sur disque plus ou moins complexes. Cela va de la gestion d'enregistrements à la gestion d'objets structurés, sans oublier les objets de grande taille.
- Gestion de méthodes d'accès : l'accès aux éléments d'une collection d'objets peut être optimisé par l'utilisation de structures et fonctions annexes. Il peut s'agir d'index ou de fonctions de hachage qui évitent de parcourir toute une collection pour retrouver un élément particulier.

- Gestion de versions : il est parfois utile de conserver les différentes étapes de l'évolution d'une entité. Il peut être intéressant d'assurer dès le plus bas niveau la gestion des versions d'un objet.
- Gestion d'identificateurs d'objets : un identificateur d'objet, ou OID (Object IDentifier), est un moyen de désigner de façon unique un objet dans le système. Il assure à un objet une existence indépendante de sa valeur. Il permet notamment le partage d'objets. Un objet peut être mis à jour sans changer d'identité. L'OID peut être physique ou logique. Dans le premier cas, il contient l'adresse physique de l'objet qu'il désigne, dans le deuxième cas, il est nécessaire de le décoder pour déterminer cette adresse.

Les aspects transactionnels n'ayant pu être abordés dans le travail présenté par la suite, la gestion de la concurrence et de la reprise ne seront que succintement évoqués dans cette section.

Nous préférons préciser dès maintenant la problématique des identificateurs d'objets logiques ou physiques, afin de mieux comprendre les choix effectués dans les systèmes présentés. Nous donnons ci-dessous les caractéristiques, avantages et inconvénients des deux approches.

Identificateurs logiques

Ils peuvent être de simples numéros attribués séquentiellement (estampilles) à la création des objets. Leur inconvénient majeur réside dans la nécessité de maintenir des structures de données importantes pour assurer la correspondance entre l'identificateur et l'adresse de stockage d'un objet (en général une table des objets). Ils présentent cependant de nombreux avantages : les objets peuvent être facilement déplacés en mémoire secondaire (puisque leur adresse physique ne figure qu'à un seul endroit, dans la table des objets) ; ces OID peuvent être à la base d'une gestion de copies d'objet, de techniques de gestion de versions ou d'objets ombres (en faisant correspondre plusieurs adresses physiques à un OID).

Identificateurs physiques

Ils sont composés de l'adresse de stockage de l'objet qu'ils désignent. L'avantage essentiel est la suppression d'un niveau d'indirection dans la phase d'accès à un objet, puisqu'il fournissent directement son adresse. L'inconvénient majeur concerne les déplacements d'objet, puisqu'ils signifient "modification de l'identificateur" ; or ce dernier peut se trouver dans de nombreuses chaînes d'accès à l'objet. Le problème de la récupération de l'espace libéré par la destruction d'un objet se pose aussi, puisqu'il nécessite de réutiliser l'OID de l'objet détruit.

II.5.1 Le gestionnaire d'EXODUS : Object Storage Manager

Le gestionnaire de stockage est la seule partie fixe du système extensible EXODUS. Il se doit donc d'être aussi *générique* que possible, car tout type de SGBD doit pouvoir être construit au dessus (objet, multimédia, relationnel,...).

Identificateur d'objet

Le choix d'un OID physique plutôt que logique permet d'éviter d'avoir à maintenir un immense index d'identificateurs. Un OID est un numéro de volume assorti d'un numéro de page et d'un déplacement dans la page. Il contient aussi un identifieur unique permettant d'empêcher la réutilisation d'un même OID. Un fichier est représenté par une structure d'index similaire aux arbres B+ : la clé de l'index est en fait le numéro de page ; un tel index permet de regrouper l'ensemble des pages d'un fichier et facilite le parcours séquentiel des objets dans l'ordre physique.

Notion de paquets d'objets

Une caractéristique essentielle que l'on retrouve souvent est la capacité de regrouper les objets dans des ensembles logiques et physiques que peuvent être les collections, les fichiers ou les classes. Les objets sont ici stockés dans des fichiers. Les objets d'un fichier peuvent être parcourus séquentiellement et sont stockés de façon très localisée sur disque.

Clusters

Il est non seulement possible de grouper les objets dans des paquets, mais à l'intérieur même de ceux-ci il est possible de gérer le positionnement des objets : la primitive de création d'objet accepte en paramètre des directives de placement (près de l'objet d'identificateur ID, seul sur une page, ...).

Stockage de petits et de grands objets

Les petits objets sont stockés sur une page, ceux de taille supérieure sont stockés sous forme d'arbre B+. L'identificateur d'un objet de grande taille pointe sur une entête qui constitue la racine d'un arbre B+. La clé d'un tel index est en fait le positionnement (en octets) de la partie de l'objet recherchée. Des algorithmes permettent alors d'insérer, d'effacer, et de concaténer des chaînes d'octets à un objet.

Contrôle concurrent, reprise

Un algorithme de verrouillage à deux phases sur des chaînes d'octets ou des objets entiers, ainsi qu'un protocole spécial pour les arbres B+ sont utilisables. Des journaux avant et après sont gérés pour les petits objets, ainsi qu'un mécanisme de "pages ombres" pour les objets volumineux.

Adressage en mémoire principale

Les OIDs présentés ci-dessus permettent d'adresser les objets en mémoire secondaire. Lorsqu'à partir de ces identificateurs, les objets ont été chargés

en mémoire (dans le *buffer pool* du gestionnaire d'objets), il devient possible de les adresser au moyen d'un descripteur qui contient un pointeur sur une portion d'objet qui réside dans le *buffer pool*. Il y a donc deux niveaux d'indirection dans cet adressage, le premier correspond à l'OID, et le deuxième à ce descripteur souvent appelé poignée (*handle*) qui permet à l'objet de bouger dans le *buffer pool* sans invalider ses références.

Interface

L'interface du gestionnaire présente des primitives pour créer, détruire un paquet d'objets, et pour le parcourir (*scan* : obtenir l'OID de l'objet suivant). Des fonctions permettent de créer ou détruire un objet dans un paquet, de lire ou d'écrire un morceau d'objet (renvoi d'un pointeur sur une chaîne d'octets). Il existe aussi des primitives permettant de gérer des transactions : *begin*, *commit*, *abort*.

EXODUS fournit un gestionnaire de stockage de bas niveau assez complet, qui peut être une bonne base pour assurer la persistance dans un langage ou un SGBD, avec notamment la notion de transaction. Il n'offre cependant pas le support d'objets structurés qui peut sûrement être implanté au dessus. Il resterait à évaluer les performances de ce système, ainsi que la facilité avec laquelle les mécanismes fournis sont adaptables.

II.5.2 GEODE : un serveur d'objets

GEODE est un serveur d'objets propre à supporter plusieurs modèles de données (relationnel, objet, ...). Il a été conçu pour un environnement grande mémoire (quelques giga-octets de mémoire principale). Son but est d'assurer le stockage efficace d'objets complexes par une implantation adéquate des constructeurs, ainsi que la fiabilité par des mécanismes flexibles. On distingue cinq couches dans GEODE :

- Au plus haut niveau, une partie initialement prévue mais non réalisée dans le prototype de l'INRIA/INFOSYS concerne la *gestion des types* de données : c'est-à-dire des types de base et des constructeurs de type (*n-uplet*, *vecteur*, *ensemble*,...), mais aussi des types abstraits (ADT, hiérarchie d'héritage, bibliothèque pour les méthodes...). Nous avons construit une telle couche au dessus de GEODE à l'occasion de ce travail de thèse. Elle est en fait très dépendante du modèle et du langage du système que l'on désire mettre en œuvre.
- Le niveau *constructeurs* fournit une boîte à outil permettant de construire et manipuler des objets complexes. Ces constructeurs sont des structures de données génériques telles que le *n-uplet*, la *liste*, le *tableau* ou l'*ensemble* auxquelles sont associées un certain nombre d'opérations. Ces dernières constituent l'interface de ce niveau.

- Le niveau *support des objets complexes* fournit les structures de données servant de base au stockage : structure contiguë de taille inférieure ou supérieure à une page, structure virtuellement contiguë dont l'accès se fait au moyen d'un arbre B. C'est à ce niveau que se fait la répartition des données dans les pages. Cette couche encapsule la mise en œuvre physique des constructeurs, ce qui assure une meilleure extensibilité au système.
- Le niveau *gestion de segments* implante une mémoire virtuelle dans laquelle les objets sont des séquences d'octets non typées, associées à des identificateurs (OID). Un segment rassemble un certain nombre d'objets et permet d'y accéder séquentiellement. C'est en fait un moyen de regrouper physiquement des objets qui ont un lien "logique" entre eux. Le choix de regroupement des objets est évidemment laissé à l'utilisateur de GEODE, qui peut, dès les interfaces de plus haut niveau (constructeur) donner des directives de regroupement. Ces dernières se présentent sous la forme de paramètres des fonctions de l'interface permettant d'indiquer dans quel segment un objet doit être créé.

  Un OID est un numéro de segment suivi d'un numéro d'objet dans le segment. Chaque segment dispose en mémoire d'une table des pages (TPAG) et d'une table des objets (TOBJ). La première maintient l'ensemble des pages allouées au segment. La deuxième maintient la correspondance entre OID et adresse virtuelle (AMV = No segment, No page dans la TPAG du segment, déplacement dans la page). La gestion de cette mémoire virtuelle est schématisée dans la figure Fig. 2.5.

  L'interface offerte par cette couche est du style création/suppression d'un objet physique, insertion/suppression d'une chaîne d'octets dans un objet, attribution d'un OID à un objet, recherche d'un objet par son OID. Un verrouillage à deux phases est possible au niveau des objets, les verrous sont maintenus dans les TOBJ. La gestion des versions s'effectue aussi au niveau des TOBJ, l'AMV est alors remplacée par un pointeur sur une table comportant autant d'entrées que de versions, chaque entrée comporte une AMV et un verrou. On s'aperçoit que les OID sont logiques et donc indépendants des déplacements d'objets et des versions (un seul OID pour toutes les versions). La gestion de transactions est tout à fait classique. On trouve des requêtes telles que *verrouiller, déverrouiller, abandonner, valider, journaliser...* mais la journalisation de tous les objets est très coûteuse d'où la nécessité de pouvoir prendre en compte la sémantique des applications au niveau le plus bas (paramétrage des mécanismes).
- La gestion du disque est assurée par une tâche de report asynchrone, chargée d'écrire des états successifs de la base assortis de leurs journaux.

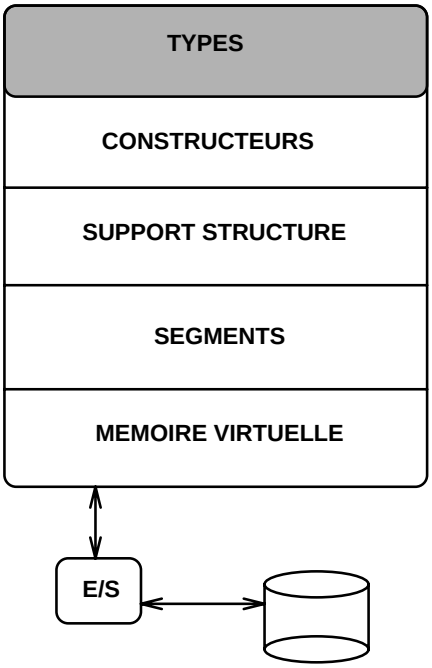

Fig. 2.4 : Architecture de GEODE

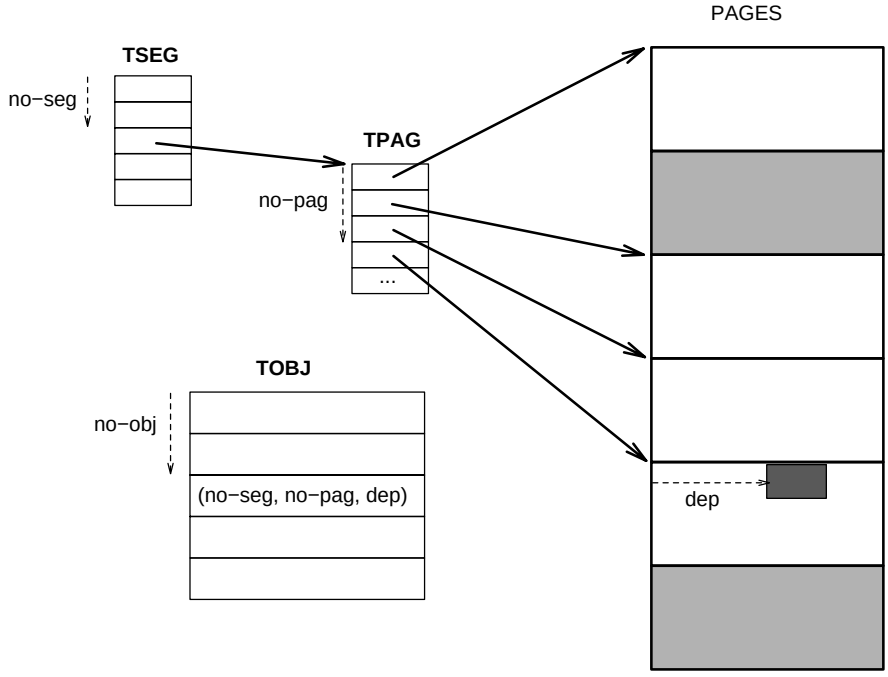

Fig. 2.5 : Mémoire virtuelle de GEODE

La figure Fig. 2.5 montre l'organisation de la mémoire virtuelle GEODE. Un OID est composé d'un numéro de segment (*no-seg*) et d'un numéro d'objet (*no-obj*)

dans le segment. Ce dernier permet d'obtenir l'AMV de l'objet (numéro de segment *no-seg*, numéro de page *no-pag* dans le segment et déplacement *dep* dans la page) dans la table des objets du segment TOBJ. Le numéro de segment permet d'accéder à la table TPAG des pages du segment. Le numéro de page sert d'entrée dans cette table pour accéder à l'adresse de la page. Finalement le déplacement permet d'accéder à l'objet dans la page.

GEODE offre un ensemble de fonctions intéressant. Certaines sont d'assez haut niveau comme la gestion d'objets complexes et la gestion de versions. Trouver de telles fonctions au niveau du gestionnaire d'objets même est appréciable car elles ont pu être implantées de façon efficace. Il est avantageux du point de vue des performances et du temps de développement de s'appuyer sur un module spécialisé dans la gestion d'objets structurés. Un autre aspect intéressant est l'approche "base de données en mémoire" qui consiste à gérer les données dans une mémoire virtuelle en supposant peu fréquents les défauts de page, du fait de la taille importante de la mémoire principale. Nous avons utilisé GEODE pour la mise en œuvre de notre prototype, mais n'avons cependant jamais pu bénéficier d'une version fournissant le contrôle de la concurrence.

II.5.3 Le gestionnaire d'objets d'O2

Ce gestionnaire[1] est le module chargé d'assurer la persistance et la gestion de transactions pour les objets complexes [60]. Il est construit sur WISS (Wisconsin Storage System) qui fournit déjà un bon nombre de services : gestion du disque, du contrôle de la concurrence, des tampons... WISS gère des fichiers séquentiels d'enregistrements. L'unité de stockage est la page. La persistance des objets est gérée grâce à un compteur de références associé à chaque objet. Tant qu'un objet est référencé par d'autres objets, il persiste, autrement il est détruit (un compteur de référence est maintenu pour chaque objet).

Un identificateur d'objet est ici un identificateur d'enregistrement WISS (RID pour "Record IDentifier"), soit **[***ident. de volume, ident. de page, déplacement***]**. C'est un OID physique qui représente donc l'adresse de stockage d'un objet. Si un objet bouge, un lien de poursuite est laissé à son ancienne place, afin de pouvoir le retrouver à partir de son OID, au moyen d'une indirection supplémentaire. Ce lien de poursuite est en fait la nouvelle adresse de stockage de l'objet ; si ce dernier est à nouveau déplacé, on ne crée pas un nouveau lien de poursuite, mais on modifie le premier en lui donnant la nouvelle adresse de stockage : il n'y a donc jamais plus d'une indirection.

Les objets sont accessibles en mémoire à partir d'une table de hachage sur OID qui permet de retrouver l'adresse de chargement en mémoire virtuelle de l'objet (AMV : adresse en mémoire virtuelle). Les objets qui viennent d'être créés (et notamment les

1 Version V1 de O2

objets temporaires) sont identifiés par leur AMV. Un OID (RID) ne leur sera assigné qu'à la fin de la transaction pour les raisons expliquées maintenant.

Il n'est pas raisonnable de considérer tous les objets créés au cours d'une transaction comme persistants, aussi a-t-il été décidé de ne pas leur assigner de RID dès leur création. En effet, de nombreux objets temporaires ne sont jamais promus persistants ; on évite ainsi de leur créer des enregistrements sur disque. A partir de leur création, ils sont alors référencés par des AMV, aussi bien dans les autres objets que dans les variables des programmes. La question se pose alors de savoir s'il est intéressant de leur assigner un RID au moment où ils sont promus persistants. L'intérêt d'assigner un RID à un objet est que l'on peut libérer l'espace qu'il occupe en mémoire virtuelle en le renvoyant sur disque et de disposer au plus tôt de sa référence persistante. Or, lorsqu'un objet créé est promu persistant, même si un RID lui est assigné, il n'est plus question de l'enlever de son emplacement en mémoire virtuelle, car il a pu être référencé auparavant au moyen de son AMV (pour la même raison, disposer de son RID présente peu d'intérêt). Il est donc préférable d'attendre la fin de la transaction, où toutes les variables référençant l'objet promu ont disparues pour lui assigner un RID ; il est possible de retrouver tous les objets persistants dans lesquels l'AMV de l'objet avait été introduite, pour la remplacer par son RID. De plus, attendre la fin de la transaction permet d'avoir plus d'informations concernant la localisation de l'objet à créer (voir plus bas les techniques de regroupement) et permet de créer l'enregistrement WISS dans la page adéquate.

Cette technique pose cependant quelques problèmes. Un objet identifié uniquement par une AMV ne peut être référencé que par cette AMV et ne peut donc être déplacé en mémoire. Cela veut dire aussi que les objets temporaires créés durant une transaction peuvent saturer la mémoire virtuelle (puisque l'espace ne peut être récupéré avant la fin de la transaction). De plus, lorsqu'un objet est promu persistant, il faut changer toutes les références qui existaient sur lui. Une partie de ces problèmes pourrait être évitée avec des OID logiques car l'assignation de l'OID pourrait se faire indépendamment de l'allocation de l'espace disque.

Représentation des objets

Le modèle de données d'O2 distingue les valeurs des objets : les valeurs peuvent être des *ensembles*, des *listes*, des *n-uplets* ou des *atomes*, sont manipulables par des opérateurs prédéfinis, et ne peuvent être partagées par d'autres valeurs ou objets ; les objets ont une identité et ne sont manipulables que par les méthodes définies par l'utilisateur, ils sont partageables grâce à leur identificateur. Les valeurs structurées du modèle sont toutefois gérées au niveau du gestionnaire d'objets de la même façon que les objets ; on leur associe notamment un identificateur.

Les objets complexes sont stockés de façon entièrement décomposée. Chaque composant est stocké dans un enregistrement et se trouve référencé dans l'objet englobant par un OID. Ceci facilite le traitement des composants dont

la taille augmente ou qui sont partagés entre plusieurs objets englobants. Un stockage linéaire (stockage des valeurs des composants à l'intérieur même des valeurs englobantes) nécessiterait dans ces derniers cas de pouvoir réorganiser le stockage d'un objet pour le faire grandir (ce qui est gênant lorsque les OIDs sont physiques), et de résoudre les conflits en cas de partage d'un composant entre plusieurs objets (éventuellement avec de la duplication).

Un n-uplet est stocké dans un enregistrement tenant sur une page, si la taille dépasse celle de la page, alors il est stocké au format LDI ("long data item", aussi géré par WISS). Les listes sont stockées sous forme d'arbres ordonnés. Les ensembles sont indexés par des arbres-B.

Les clusters

Les groupements d'objets dans O2 sont dirigés par des arbres de placement. Des algorithmes permettent de déterminer quels objets sont utilisés simultanément de façon fréquente [13]. A un arbre de placement correspond un fichier (WISS). Des facilités sont offertes par WISS pour placer les objets à l'intérieur même des fichiers : quand on insère un enregistrement dans un fichier, il est possible de spécifier un identificateur d'enregistrement auprès duquel on stockera le nouvel objet (dans la même page).

Contrôle de la concurrence

La concurrence est gérée par WISS qui pratique un verrouillage hiérarchique à deux phases sur les pages et les fichiers et qui propose une méthode particulière pour l'accès aux arbres B (méthode de Bayer). Une gestion différente sur les données du schéma est effectuée suivant que l'on se trouve en mode développement ou application.

Ce gestionnaire d'objets utilise des techniques de mise en œuvre intéressantes. Nous avons nous-mêmes choisi l'approche du stockage décomposé pour notre implantation de gestionnaire d'objets. Des techniques de regroupement prenant en compte la composition doivent alors être mises en œuvre pour ne pas trop pénaliser l'accès aux objets structurés. Nous distinguons aussi la notion d'objet partageable (avec identifieur) de la notion de valeur. La structure de données d'un objet ne différant pas de celle d'une valeur, nous avons aussi choisi de les stocker de manière identique. La sémantique de copie des valeurs est alors prise en compte au niveau des opérations du gestionnaire d'objets. Cette technique présente cependant l'inconvénient de générer de nombreuses copies qui devront être détruites par la suite. Nous proposons des solutions pour éviter la plupart de ces copies (en VI.2.5).

II.5.4 Gestion du stockage dans NooDLE

L'architecture de NooDLE a été décrite en II.4.5 et nous allons nous intéresser aux modules de gestion des références, de gestion des transactions et de gestion des fichiers d'objets. On rappelle que NooDLE utilise pour stocker les objets, à la fois un SGBD relationnel, et un gestionnaire de fichiers. Les références et chemins d'accès aux données, ainsi que les types sont gérés au niveau du SGBD relationnel, mais les instances et les définitions de types sont stockées dans des fichiers. Le gestionnaire d'objets est en fait constitué du gestionnaire de références et du gestionnaire de fichiers.

La gestion de références comprend la gestion des extensions (ou collections d'objets) et la gestion des références aux objets eux-mêmes. Cette gestion se fait au moyen d'un certain nombre de tables stockées dans le SGBD relationnel. Une première table tient lieu de catalogue. Chaque n-uplet représente une extension existante et contient des informations telles que le type des objets contenus, le nom du fichier dans lequel les objets sont stockés, le nombre d'éléments, le nom de la table qui décrit l'extension, et finalement un ensemble de chemins d'accès. La table qui décrit l'extension contient un n-uplet par objet de la collection ; ce n-uplet contient l'identificateur unique d'un objet, le compteur des références à cet objet, sa longueur et finalement son numéro d'enregistrement dans le fichier de l'extension. Enfin, les chemins d'accès sont des relations qui font correspondre des valeurs d'attribut à des identificateurs d'objets (index).

La gestion des transactions utilise les facilités offertes par le SGBD relationnel. Les verrous sur les objets sont traduits en verrous sur les n-uplets des relations décrivant les extensions. Le système supporte les transactions imbriquées ; les verrous ne sont relâchés qu'à la fin de la transaction englobante (transactions fermées) ; un ensemble de transactions imbriquées est en fait traduit en une seule transaction du SGBD relationnel. Un mécanisme est toutefois nécessaire pour la journalisation du contenu des objets ; en effet, leur contenu étant stocké dans des fichiers, le SGBD relationnel ne peut en assurer la journalisation ; le journal est stocké dans les fichiers.

Cette approche semble assez peu prometteuse. Si pour les index l'utilisation des tables relationnelles semble à priori intéressante (il vaudrait d'ailleurs mieux utiliser directement des index relationnels), cette même utilisation pour la correspondance avec les identificateurs semble assez pénalisante (une requête relationnelle pour décoder un identificateur). De plus, le stockage des objets dans des fichiers semble assez pauvre, il aurait été plus intéressant d'utiliser un gestionnaire d'objets. Toutefois cette approche ressemble à la nôtre car elle consiste à coupler un SGBD relationnel avec un gestionnaire de stockage d'objets (ici des fichiers). Cependant l'idée est ici beaucoup moins intéressante car il n'y a pas de données relationnelles à gérer et le SGBD n'est utilisé que pour assurer des services que tout gestionnaire d'objets doit fournir (gestion des identificateurs).

II.5.5 Une mémoire persistante d'objets : Cricket

Certains concepteurs de gestionnaires d'objets préconisent désormais une approche à un seul niveau de stockage. Tous les objets persistants et volatiles sont dans une même mémoire virtuelle qui se trouve mise en correspondance avec l'espace disque par un mécanisme de couplage [2]; le chargement effectif des zones mémoire accédées est réalisé par un paginateur. Cela a l'avantage d'offrir une vue uniforme des objets permanents et temporaires, et surtout de ne pas avoir à distinguer deux formats de données (format disque et format mémoire), ce qui élimine les problèmes de conversion entre ces formats. De plus, si les objets sont toujours couplés aux mêmes adresses virtuelles, celles-ci peuvent être utilisées comme références stables sur ces objets. Cette philosophie a été en partie suivie dans le gestionnaire d'objets GEODE, décrit en II.5.2. Une telle approche a été mise en œuvre pour le système de stockage de base de données Cricket [54].

La mise en œuvre de Cricket est basée sur le mécanisme d'*external pager* de MACH. MACH fournit la notion d'*objet mémoire* qui est une zone de données persistante gérée par un serveur de pages appelé *external pager*. Ce serveur peut être programmé par l'utilisateur du système MACH. Une tâche MACH peut associer une région de son espace d'adressage à un *objet mémoire*, l'*external pager* associé sera alors appelé chaque fois qu'une page de cette zone de mémoire aura à être écrite ou lue sur le disque. Un *objet mémoire* peut être couplé par plusieurs tâches, les données se trouvent alors physiquement partagées, c'est à l'*external pager* de gérer alors un éventuel contrôle de la concurrence. Le noyau du système MACH et l'*external pager* communiquent lors des défauts de page sur un *objet mémoire*.

Cricket présente une architecture client/serveur. Les applications clientes tournent dans des tâches séparées et communiquent avec le serveur Cricket au moyen d'une interface RPC. En fait la base de données est un *objet mémoire*, géré par un *external pager* qui n'est autre que le serveur Cricket. Lorsqu'une tâche client se connecte au serveur, l'*objet mémoire* représentant la base de données est couplé dans son espace d'adressage, elle peut donc accéder directement aux données persistantes, l'intervention du noyau MACH et de l'*external pager* Cricket est transparente. Le serveur Cricket s'occupe en outre de la gestion des transactions, du contrôle de la concurrence, de la reprise, des index, de l'allocation disque et des entrées/sorties.

La gestion de la concurrence se fait au moyen du mécanisme d'exception de MACH, qui permet à une exception levée par une tâche d'être traitée par la procédure de gestion d'exception d'une autre tâche. Cricket implante à l'aide de ce mécanisme un verrouillage à deux phases au niveau des pages. Lorsque le client exécute un *begin_transaction*, la section de son adresse virtuelle sur laquelle la base de données est couplée, est protégée en lecture et écriture. A chaque accès à cette

---

2 C'était aussi l'approche du SGBD Socrate, il y a vingt ans ...

zone, une exception est levée et se trouve traitée par la tâche qui représente le serveur Cricket. Ce dernier s'occupe de s'approprier les verrous nécessaires sur les pages concernées, se bloquant si nécessaire, puis laisse le client reprendre son exécution (il informe le noyau MACH que l'exception a été traitée avec succès, ce dernier reprend alors l'exécution de la tâche cliente suspendue).

La gestion de l'allocation mémoire sur disque est faite à partir de la notion d'extension, qui correspond à un nombre fixé de pages disques. Une fonction d'adressage dispersé (hash code) permet de retrouver l'extension correspondant à une adresse virtuelle. Des directives de placement peuvent être données lors de l'allocation d'*objets mémoires*. Il n'existe pas encore dans ce système de notion de collections d'objets (fichiers, segments,...), ni d'index (qui pourrait faire correspondre une adresse virtuelle à une clé définie par l'application).

Cette approche nous intéresse pour deux raisons : elle va dans le sens de l'hypothèse selon laquelle la base de donnée active tient en mémoire ; elle démontre la possibilité d'un couplage assez fort avec le système d'exploitation dont elle utilise les services. Les avantages et inconvénients seront discutés plus en détail dans le chapitre VI.

II.5.6 Conclusions

Un gestionnaire d'objets doit gérer la persistance et la fiabilité d'objets. Au plus bas niveau les objets sont des chaînes d'octets non structurées. Chaque objet a un idenficateur unique (OID) qui peut être soit physique (adresse sur disque), soit logique (invariant aux déplacements de l'objet sur disque, mais nécessitant une interprétation, souvent réalisée au moyen d'une table de correspondance). Du point de vue architecture, on distingue plusieurs approches : la première prend en compte deux niveaux de représentation des objets, l'un sur disque et l'autre en mémoire, des chargements et déchargements d'objets sont alors nécessaires ; dans la deuxième approche, il n'y a qu'une représentation des objets, dans une mémoire virtuelle, la persistance est assurée par le mécanisme de pagination (Cricket, GEODE).

Il doit être possible de regrouper des objets dans des *unités de stockage* : ces unités doivent être non seulement logiques, de façon à accéder séquentiellement aux objets, mais aussi physiques, afin de bénéficier d'une certaine localité de stockage qui permet de limiter les transferts de pages entre disque et mémoire. Les objets d'une collection, d'un ensemble, d'une classe, pourront ainsi être stockés ensembles dans un segment de GEODE, ou dans un fichier WISS ou EXODUS. Des directives de regroupement très fines doivent pouvoir être prises en compte (tel objet doit être stocké à côté de tel autre).

Le contrôle des accès concurrents doit être assuré. Les niveaux de granularité offerts pour le contrôle des accès concurrents sont l'objet, la page ou l'ensemble d'objets. Toutefois s'ils ne tiennent pas compte de la sémantique de l'application, ces mécanismes sont très coûteux (verrouillage à outrance, mauvaise granularité...). Ils

doivent donc être débrayables et surtout modifiables. Plusieurs mécanismes peuvent même être proposés. Pour la gestion de la reprise les mêmes problèmes se posent (on journalise trop). Les mécanismes doivent aussi être paramétrables.

La gestion de versions d'objets peut être effectuée dès ce niveau. Il faut aussi pouvoir prendre en compte des structures d'objets complexes, telles que le *n-uplet*, le *tableau*, l'*ensemble,* et donc fournir des structures de stockages adaptées (arbres B+, hachage, ...). Il est possible d'enrichir le niveau de fonctions d'un tel gestionnaire en assurant le support des primitives d'accès et de manipulation d'objets structurés : parcours, insertion, suppression dans les listes, les ensembles, accès aux champs d'un n-uplet, opérations ensemblistes... Ceci permet alors de gérer des méthodes d'accès aux objets (index).

II.6 Optimisation d'accès

Comme peu des systèmes étudiés présentent des techniques d'optimisation, nous avons préféré présenter cette section sous la forme d'une étude générale, plutôt que détailler l'apport de chaque système, comme cela a été fait précédemment. Il existe deux aspects dans l'optimisation des accès aux données. Le premier est une action d'organisation physique des données dans l'espace de stockage. On procède à des regroupements de données, mais aussi à des créations de structures de données annexes permettant d'accélérer les recherches (index). Le deuxième aspect constitue la phase d'optimisation par elle-même.

II.6.1 Méthodes d'accès et regroupements

Parallèlement à la notion d'index dans les SGBD relationnels, qui permettent de retrouver des n-uplets en fonction de la valeur de certains attributs, il est possible d'imaginer cette même notion d'index sur les collections d'objets d'un système orienté objet. Cependant, il faut prendre en compte le fait que les objets d'une collection n'ont pas une structure plate comme le n-uplet. Il faut donc envisager des index dont la clé d'accès peut être un attribut d'un composant d'objet de la collection. De ce fait, un tel attribut peut changer de valeur, sans que les objets de la collection indexée se trouvent directement modifiés. Il se peut enfin que l'on veuille indexer les objets, non pas sur la valeur d'un attribut, mais sur le résultat d'une méthode. On s'aperçoit que la gestion des index dans un tel contexte est compliquée. Une proposition pour gérer des index dans un SGBD orienté objet est présentée en [44].

Le regroupement physique des objets en mémoire et sur disque est un autre moyen d'accélérer les accès aux données, cette fois en limitant le nombre d'entrées/sorties, ou de chargements d'objet, par des techniques d'anticipation. Si les gestionnaires de stockage offrent la plupart du temps la possibilité de regrouper physiquement les objets (voir II.5), la grande difficulté reste de définir la politique de regroupement

de ces objets. Les systèmes orientés objet fournissent non seulement des accès séquentiels sur les ensembles d'objets, mais aussi des accès navigationnels à travers les références entre objets. Il ne suffit donc pas de regrouper les objets d'une collection comme les n-uplets d'une relation. Il faut aussi tenir compte des liens de composition et du comportement des méthodes. Les techniques de détermination des stratégies peuvent être très complexes. Une étude a été faite dans le cadre du SGBD orienté objet O2 [13]. Les stratégies de regroupement proposées pour le système O2 reposent sur la hiérarchie de composition de classes. A chaque classe est associé un arbre de placement qui indique la façon dont les composants d'un objet complexe (instance de cette classe) seront regroupés. Des stratégies pour définir ces arbres de placement ont été étudiées.

Il faut cependant remarquer que les regroupements d'objets peuvent aussi avoir des effets néfastes s'il portent sur des objets fréquemment traités par des transactions concurrentes. Sachant que l'on stocke en général plusieurs objets dans une page et qu'au plus bas niveau la granularité de verrouillage est la page, il peut se faire qu'une transaction soit bloquée sur l'accès à un objet dont la page a été verrouillée par une autre transaction qui traite en fait un autre objet du groupe. La conséquence est une réduction du parallélisme d'exécution des transactions et donc une perte de performance. Il faut non seulement être capable d'évaluer si un ensemble d'objets donne lieu à des accès consécutifs par une même transaction, mais il faut aussi s'assurer qu'il ne donne pas lieu à des accès concurrents (ce qui est encore plus délicat).

II.6.2 Optimisation de requêtes

On distingue généralement deux phases dans le processus d'optimisation : la première est appelée *optimisation logique* et consiste à reformuler la requête en fonction de règles liées au modèle et à la connaissance sémantique que l'on peut avoir de la base (comme les contraintes d'intégrité) ; la deuxième, appelée *optimisation physique*, consiste, elle, à élaborer un plan d'évaluation des requêtes en fonction du coût de chaque opération nécessaire et des chemins d'accès disponibles.

L'optimisation logique repose d'une part sur les propriétés **algébriques** du modèle, qui permettent de changer l'ordre d'exécution des opérations, et d'autre part sur des connaissances sémantiques de la base. Un optimiseur relationnel utilise les propriétés de commutativité et de distributivité de l'algèbre relationnelle. Une règle classique consiste à effectuer les opérations de sélection le plus tôt possible. Ce type d'optimisation est un des intérêts essentiels de la formalisation en tant qu'algèbre d'un modèle de manipulation de données. C'est pour cette raison que de nombreuses formalisations de modèles orientés objet ont vu le jour ([51][57]). Une étude sur l'optimisation de requêtes a été faite pour O2 ([22]) et a conduit à la réalisation d'un optimiseur pour le produit O2. La plupart des optimisations

possibles sont directement issues des techniques relationnelles qui ont été adaptées au modèle objet. On retrouve des techniques de factorisation des expressions constantes, d'ordonnancement des opérations de sélection et de prise en compte des index du système.

L'optimisation physique repose sur la connaissance de l'organisation physique et sur des informations statistiques concernant le contenu de la base. Des statistiques sur la cardinalité et la sélectivité des collections d'objets, mais aussi la présence d'index ou de regroupements, permettent d'évaluer le coût des opérations. Cela permet ensuite de les exécuter de la façon la plus appropriée.

II.7 Ecriture et gestion des méthodes

La gestion des méthodes dans un système à objets comprend plusieurs aspects. Nous traitons du langage d'écriture des méthodes, qui n'est pas toujours le langage du système. Nous abordons aussi la compilation des méthodes et le stockage des modules exécutables. Nous discutons enfin des problèmes d'édition de liens, dynamique ou statique, et d'exécution de méthode. Le dernier point abordé pose le problème des récupérations d'erreur dans les méthodes. En effet, ces dernières sont écrites par l'utilisateur, et soumettent donc le système aux aléas des erreurs de programmation.

II.7.1 Le cas de O_2

L'approche choisie dans O_2 pour décrire les traitements est multi-langage. Alors que les types de données (classes) sont décrits en O_2, le langage utilisé pour écrire les méthodes peut être C, BASIC, LISP. C'est en fait une combinaison du langage de définition et de manipulation de données O_2 et d'un langage de programmation qui est utilisé. On parlera de CO_2, BasicO_2, LispO_2. On trouvera une description complète du langage CO_2 dans [42].

Le compilateur génère dans le code qu'il produit des appels au gestionnaire d'objets pour les manipulations de valeurs O_2.

Le code des méthodes à exécuter est stocké dans le gestionnaire d'objets et n'est sélectionné qu'à l'exécution en fonction du type de l'objet sur lequel porte l'appel. Ce mécanisme de "résolution" tardive permet en premier lieu de résoudre les cas de polymorphisme que l'on ne peut pas traiter à la compilation. Il permet d'autre part d'éviter de recompiler toutes les méthodes qui utilisent une méthode donnée, lorsque celle-ci est modifiée. Un tel mécanisme est toutefois très coûteux à l'exécution. C'est pourquoi il a été prévu un mode dit d'"exécution" (par opposition au mode de "développement") dans lequel les applications qui tournent ont subi une compilation "en profondeur". Les appels au mécanisme de résolution tardive sont remplacés, chaque fois que cela est possible, par de simples appels de procédure. Cela nécessite de pouvoir déterminer à la compilation la méthode à appeler, ce qui est

parfois impossible. Finalement un mécanisme d'exception, basé sur la récupération des signaux, permet de parer aux erreurs introduites par le programmeur.

Cette approche présente l'intérêt d'être multi-langage, ce qui est l'un de nos objectifs. Nous utilisons aussi la technique consistant à étendre un langage de programmation existant. Le principe est d'introduire les données de la base au moyen du langage du système (ici O2), qui est intégré au langage de programmation. Les avantages d'un mécanisme de résolution tardive sont mis en valeur (polymorphisme et procédures de recompilation), mais la nécessité d'avoir deux configurations (développement et exécution) montre bien son coût élevé en performance.

II.7.2 Les méthodes dans GUIDE

Le système GUIDE est dans sa première version mono-langage. Le langage GUIDE est un langage à objets permettant d'écrire une application complète. Il est donc le langage d'écriture des méthodes qui constituent à elles seules le corps d'une application.

Le compilateur génère du code C qui contient pour chaque appel de méthode un appel à la primitive du noyau GUIDE "GuideCall". Le code C est ensuite compilé par le compilateur C traditionnel. C'est la fonction "GuideCall" du noyau qui se charge, en fonction de l'objet sur lequel la méthode est appliquée, de retrouver le code de celle-ci, d'opérer une éventuelle édition de lien dynamique (lorsque l'objet classe n'a pas encore été chargé dans le domaine), et de lancer l'exécution. Il faut noter que dans le cas de GUIDE, lorsqu'il est nécessaire de charger et lier une méthode, c'est la classe entière (l'ensemble des méthodes définies sur le type) qui est chargée. On remarque que le code des méthodes n'est jamais lié au système lui-même, mais est toujours lié dynamiquement au domaine de l'application qui s'exécute. Un mécanisme d'exception permet de prévoir des traitements de restauration après une erreur dans le déroulement d'un programme.

Nous retrouvons encore ici l'approche classique qui consiste à générer du code dans un langage cible (ici C) qui contient des appels au noyau du système. Il faut noter que les méthodes ne sont pas gérées individuellement, mais par classe.

II.7.3 Des méthodes en LISP pour SGBD relationnel étendu

L'utilisateur de SABRINA peut créer de nouveaux types de données. Un tel type est défini par un ensemble de méthodes écrites en LISP ou en C. Elles sont appelées à partir du langage SQL.

L'interpréteur LISP a été intégré à la machine algébrique du système relationnel. Les méthodes écrites en LISP sont stockées dans le système et sont interprétées à l'exécution. Cela permet de récupérer et de gérer les erreurs de programmation. Il est toutefois possible de lier dynamiquement et d'exécuter du code C. De telles

méthodes C s'exécutent plus rapidement, mais nécessitent toutefois un temps de chargement supplémentaire.

Ceci est un exemple de système où les méthodes sont écrites dans un langage de programmation existant. Ce langage est utilisé pour créer la structure de données d'un objet qui ne pourra être manipulée que par des méthodes écrites dans ce langage. Par exemple, une instance d'un ADT défini par une fonction LISP ne pourra être manipulée que par des méthodes écrites en LISP, mais pas par des méthodes écrites en C. Cette approche multi-langage instaure plusieurs formats de données (un par langage) incompatibles entre eux. L'approche interprétée est intéressante pour la récupération d'erreur, mais peut se révéler peu performante.

II.7.4 Conclusion

La technique la plus classique de définition d'un langage de programmation de méthodes consiste à étendre un langage de programmation existant afin de permettre la manipulation des entités du modèle à supporter. Les données du modèle sont décrites et manipulées à partir d'instructions spécifiques qui sont traitées par un précompilateur. Le résultat de la précompilation est un code dans le langage de programmation d'origine contenant des appels au système. C'est cette technique que nous utiliserons dans notre propre implantation du langage d'écriture de méthode. Une technique plus simple est utilisée lorsque le système n'impose pas de modèle de données particulier ; le langage utilisé n'a pas besoin d'être étendu, ce sont ses propres opérateurs de structuration qui sont utilisés pour décrire les objets. Il existe un problème de sécurité dans le cas où les méthodes utilisateurs sont liées au système dont elle peuvent occasionner la défaillance.

II.8 Synthèse

Le début de cet état de l'art nous a permis de faire le point sur les extensions de SGBD relationnels et de préciser les aspects sur lesquels nous comptons fournir un apport :

- L'introduction des objets complexes, du comportement et de l'héritage doit se faire de façon homogène, à un même niveau d'abstraction.
- Nous voulons introduire la notion de partage à travers l'identité d'objet.
- Nous désirons enfin que les objets aient un format de données commun, non imposé par le langage utilisé pour décrire leur comportement.

Cet état de l'art nous a ensuite permis de décrire partiellement les différents types de système qui ont inspiré le travail présenté dans la suite. Il a notamment mis en évidence les différentes techniques de réalisation de ces systèmes, en insistant sur les points qui nous intéressent particulièrement : la prise en compte d'un modèle de base à travers un gestionnaire de types, les techniques de gestion de la persistance des

données mises en œuvre dans les systèmes de stockage, l'optimisation des accès et enfin la gestion des méthodes pour les systèmes associant les données à leur comportement. Il est intéressant de noter que, bien que les objectifs de ces systèmes soient différents, ils présentent un certain nombre de points communs.

- Les fonctions principales d'un gestionnaire de types sont presque toujours les mêmes, il suffit de les adapter aux modèles de données traités qui sont souvent très proches à cause de la tendance objet des systèmes étudiés. Le choix entre une métabase et des structures de données ad hoc pour le stockage des définitions de types est un point important. Nous avons pris pour notre projet une solution intermédiaire en stockant des structures de données spécifiques sous forme de chaînes d'octets dans la base.
- Au niveau de fonction près, les gestionnaires de stockage d'objets ont des objectifs très similaires, il existe cependant un ensemble de techniques de mise en œuvre parmi lesquelles il faut faire un choix. Il est intéressant de noter les différences de niveau de fonctions entre les systèmes abordés, qui vont du gestionnaire d'enregistrements (objet = chaîne d'octets) au gestionnaire d'objets typés (comme celui d'O2), en passant par les gestionnaires d'objets structurés (GEODE). Il existe ensuite des choix d'implantation : architecture à un ou deux niveaux, OID physiques ou logiques, représentation décomposée des objets complexes, techniques d'optimisation d'accès (regroupements, index), etc.. La mise en œuvre de tels gestionnaires d'objets est un travail très important ; certains systèmes fournissent en fait un niveau de base de gestion de stockage, assez générique, qui permet de développer par dessus un SGBD. C'est le cas d'EXODUS et GEODE. Nous utiliserons en fait le deuxième pour développer notre propre gestionnaire d'objets.
- En ce qui concerne la gestion des méthodes nous retiendrons d'une part la technique consistant à étendre un langage existant, et d'autre part les mécanismes d'exécution basés sur une liaison tardive. Le compilateur d'un tel langage génère un programme dans le langage d'origine avec des appels au gestionnaire d'objets pour la manipulation des données de la base. Une telle approche permet d'être multi-langage.

Chapitre III

Modèle et langage pour la gestion d'objets dans les SGBD

III.1 Introduction

Ce chapitre montre comment il est possible d'intégrer des concepts de manipulation ensembliste et associative avec la notion d'objet. Deux approches abordant le problème de façons entièrement opposées sont présentées. La première prend comme point de départ le langage de programmation orienté objet GUIDE et y introduit les notions de manipulation ensembliste et d'accès associatifs. La deuxième part du langage SQL et tente de lui intégrer la notion d'objet. Nous abordons ici les implications sur le système GUIDE afin qu'il puisse supporter efficacement ce type de manipulation. Le support système du modèle relationnel étendu est développé, lui, dans les chapitres suivants.

III.2 Support BD pour le système à objets réparti GUIDE

III.2.1 Introduction

Cette section présente le travail effectué sur GUIDE afin d'introduire la notion de manipulation ensembliste dans le langage. Le système GUIDE est vu à la fois comme un environnement de programmation orienté objet et comme un système réparti assurant le stockage et l'exécution d'objets. L'objectif consiste à intégrer des fonctions BD au système, c'est-à-dire à le rendre apte au développement d'applications de type BD. Ces dernières gèrent de grandes quantités de données persistantes qui nécessitent des facilités de recherche et de manipulation. L'approche a essentiellement conduit à permettre les traitements ensemblistes et les accès associatifs au niveau langage et à accélérer les accès aux groupes d'objets au niveau système. Il a fallu d'une part étendre le modèle et le langage et d'autre part adapter le support système. Ces deux aspects sont décrits séparément dans les sections III.2.2 et III.2.3

III.2.2 Aspects modèle et langage dans GUIDE

Le modèle d'objet GUIDE n'a pas été modifié. L'essentiel a été introduit au niveau du langage en utilisant les types génériques. Un type générique prédéfini a été ajouté pour gérer des ensembles d'objets, il s'agit du type *collection (*[29]*)*. Une extension du langage permet d'exprimer de façon déclarative des critères de sélection sur les collections. Cette approche est limitée à des sélections mono-collections. Nous n'introduisons aucun concept particulier. Une collection est un objet. Tout élément d'une collection est un objet.

Un bref aperçu du langage Guide est donné dans la section III.2.2.1. La section III.2.2.2 décrit l'approche choisie pour introduire les manipulations ensemblistes dans le langage et expose toutes les décisions prises. La section III.2.2.2 décrit précisement le type générique collection. Le principe de réalisation du mécanisme de recherche par contenu est présenté dans la section III.2.2.5

III.2.2.1 Le langage GUIDE

Nous présentons ici de façon succincte le langage GUIDE. Seuls les éléments nécessaires à la compréhension de l'extension du langage proposée sont détaillés. GUIDE est un langage à objets pour la programmation d'applications réparties [47]. Un programme GUIDE est composé de types et de classes. Le type d'un objet définit son comportement par la spécification des méthodes (signatures) qui lui sont applicables. Il comprend aussi un certain nombre de variables d'état accessibles directement, appelées *variables visibles*. Une classe décrit une réalisation possible d'un type (il peut y en avoir plusieurs pour un même type) ; elle comprend des déclarations de variables d'état, le code des méthodes et des clauses de contrôle. Tous les objets manipulés sont persistants.

On différencie les types de base des types construits. Les variables permettant de désigner des instances de types construits sont des références.

```
// variable de type de base élémentaire
i : Integer;
// variable de type de base structuré
table : Array [12] OF Char;
// variable de type construit
p : REF Personne;
```

La création d'un objet permanent se fait en appelant la méthode *New* sur la classe de l'objet. Un exemple de type et de classe est donné ci-dessous :

```
TYPE Personne IS
   nom : String[80];
   age : Integer;
END Personne.

TYPE Acteur SUBTYPE OF Personne IS
   salaire : Integer;
   METHOD  compare_salaire  (IN  REF  Acteur ;  OUT
```

```
        integer) : REF Acteur;
        // Cette méthode compare le salaire de l'acteur
        // sur lequel elle s'applique avec celui de
        // l'acteur passé en paramètre.
        // Elle retourne l'acteur le mieux rémunéré,
        // ainsi que son salaire en paramètre de retour
    END Acteur.

    CLASS Acteur IMPLEMENTS Acteur IS
        METHOD compare_salaire (IN act: REF Acteur ; OUT
        sal : integer) : REF Acteur;
        BEGIN
            IF act.salaire > SELF.salaire
            THEN
                sal := act.salaire;
                RETURN act;
            ELSE
                sal := SELF.salaire;
                RETURN SELF;
        END compare_salaire;
    END Acteur.
```

SELF est une variable qui désigne l'objet courant (en cours d'exécution). On précise dans la signature d'une méthode quels sont les arguments au moyen du mot IN et quels sont les paramètres de retour par le mot OUT. *Acteur* est un sous-type du type *Personne*. La relation de sous-typage permet d'exprimer la spécialisation de comportement et elle est simple (un type ne peut avoir qu'un seul super-type). Un sous-type hérite de toutes les définitions faites dans son super-type, il peut les surcharger et les compléter. Il est également possible de définir une sous-classe d'une classe.

Le langage contient des instructions d'affectation, d'appel de méthode, des instructions conditionnelles, itératives, de parallèlisme, etc., que nous ne détaillons pas ici. Il existe également des instructions pour traiter les exceptions. Le contrôle de type est basé sur une règle de conformité (non détaillée ici) permettant des contrôles plus lâches des types de références ; avec une référence de type T, il est possible de désigner des objets de type T, ainsi que des objets dont le type est conforme à T (entre autres un sous-type de T).

Le langage GUIDE offre un mécanisme de généricité non-contrainte (le paramètre formel de généricité n'est pas typé) et un mécanisme de généricité contrainte (le paramètre formel est typé et le type du paramètre effectif doit être conforme à celui du paramètre formel). Nous utilisons, dans notre approche, la généricité non-contrainte qui permet de définir un type ou une classe avec un paramètre formel dont le type n'est pas précisé.

III.2.2.2 Principe d'intégration des manipulations ensemblistes

Nous fournissons un support minimal pour les manipulations ensemblistes et associatives, et cela sans modifier le modèle d'objet. Ce support se traduit par des facilités pour manipuler des ensembles de références (les collections) et la possibilité d'émettre des requêtes simples sur ces ensembles (requêtes mono-collections). Voici les caractéristiques de notre approche :

- Tout objet GUIDE peut appartenir à une ou plusieurs collections (i.e. les objets collectionnés n'ont pas de statut spécial). L'accès individuel est donc conservé pour tout objet appartenant à une collection.
- Une collection est un objet, c'est un ensemble de références. Des accès associatifs (par le contenu) sont possibles sur une collection. Des opérations ensemblistes sont réalisables entre collections. Les collections sont construites à l'aide d'un type et d'une classe générique qui sont détaillés plus loin.
- Une collection est définie sur un *type* (et non pas une *classe*) selon le modèle de généricité de GUIDE. Ce sont les variables d'état visibles définies au niveau d'un type qui sont utilisées lors des accès associatifs. Les références stockées dans une collection désignent donc à priori des objets d'un même type (voir cependant le point suivant).
- On peut trouver dans une collection définie sur un type T des objets dont le type est un sous-type de T. Ceci est possible, car on manipule dans une collection des références de type T, on peut alors naturellement manipuler des références de type conforme à T.
- Lors des accès associatifs, les critères de sélection portent sur les variables d'état visibles du type d'instanciation de la collection. Ces variables visibles peuvent être aussi bien des entiers, caractères ou chaînes que des références (objets composants). Le résultat d'une sélection est toujours une sous-collection, c'est-à-dire un sous-ensemble de l'ensemble des références qui constitue la collection.
- La collection rassemblant toutes les instances d'une classe ou d'un type n'est pas gérée automatiquement. Toutefois le programmeur peut toujours la gérer au niveau de la classe dont il veut grouper les instances. Il peut ajouter dans la méthode *init* de cette classe l'ordre d'insertion de *SELF* dans la collection. La méthode *init* s'exécutant lors de la création d'une instance, celle-ci sera automatiquement insérée dans la collection.

La collection a été définie en suivant ces principes comme un type générique du langage GUIDE, appelé *Collection*. Une méthode particulière a été définie afin d'exprimer des accès par le contenu, il s'agit de la méthode *select* ; cette dernière retourne une nouvelle collection contenant les éléments vérifiant un critère de sélection. Des possibilitées d'indexation des collections ont été envisagées. Nous illustrons maintenant l'utilisation des collections par un extrait de programme.

```
...
a1, a2: REF Acteur;
c1: REF Collection of [Acteur];
c2: REF Collection of [Acteur];
c3,c4:REF Collection of [Acteur];
cpt: REF Compteur;
...
a1 := Acteur.New;
a1.nom := "De Niro";
a1.salaire := 100000;
c1 := Collection.New;
c1.insert (a1);
...
c3 := c2.select ([salaire] > 20000);
c4 := c3.union (c2);
a2 := c4.get_first (cpt);
WHILE a2 # NIL DO
    output.WriteString (a2.nom);
    a2 := c4.get_next (cpt);
END;
...
```

Cet exemple illustre la création d'un objet de classe `Acteur` désigné par la variable `a1` et la création d'une collection d'acteurs, désignée par `c1`. L'acteur `a1` est ensuite inséré dans la collection. Une sélection est ensuite effectuée sur la collection `c2`. Le résultat, `c3`, contient les acteurs dont le salaire dépasse 20000 francs. La collection `c4` est obtenue en effectuant l'union de `c3` avec `c2`. La liste des noms des acteurs de la collection `c4` est ensuite affichée. Plus de détails sur l'interface et l'utilisation de la classe collection sont donnés dans la section suivante.

III.2.2.3 Le type générique Collection

Nous présentons ici l'interface GUIDE du type générique *Collection*. Le mot-clé SIGNAL, dans la signature d'une méthode, sert à définir les exceptions qu'elle peut lever.

```
TYPE CONSTRUCTOR Collection OF [T] IS
METHOD init;
METHOD insert (IN elt: T);
  SIGNALS already_there, duplicate_key;
METHOD remove (IN elt: T);
  SIGNALS not_here;
METHOD is_in (IN elt: T): Boolean;
METHOD size: Integer;
METHOD  select (IN p: Predicat): REF Collection OF
[T];
METHOD select_one (IN p: Predicat): T;
METHOD create_index (IN attr: Attr);
METHOD create_unique_index (IN attr: Attr);
```

```
  SIGNALS duplicate_key;
METHOD get_first (IN cpt: REF Compteur): T;
METHOD get_next (IN cpt: REF Compteur): T;
METHOD union (IN collection: REF Collection OF [T]):
REF Collection OF [T];
METHOD diff (IN collection: REF Collection OF [T]):
REF Collection OF [T];
METHOD intersect (IN collection:
REF Collection OF [T]): REF Collection OF [T];
END Collection.
```

- **init** est la méthode d'initialisation à vide de la structure d'un objet "Collection".
- **insert** permet d'ajouter un élément à la collection ; on désigne ce dernier par une référence sur un objet de type T ou d'un sous-type de T. Une collection est vue comme un ensemble d'objets, c'est-à-dire qu'on ne peut pas trouver deux fois le même objet dans une collection, et qu'il n'y a pas d'ordre. Un premier cas d'exception est donc prévu si l'on tente d'ajouter un objet déjà présent (*already_there*) ; c'est par son identité (référence) et non par son contenu, que l'on sait si un objet est ou n'est pas dans la collection. Le deuxième cas d'exception (*duplicate_key*) n'est utilisé que s'il existe un index "unique" sur la collection (voir méthode "*create_unique_index*").
- **remove** : il s'agit ici d'ôter un élément de la collection. Un traitement d'exception est possible pour prévenir que l'objet à retirer n'est déjà plus dans la collection.
- **is_in** teste si un objet dont la référence est passée en paramètre est dans la collection ou non.
- **size** renvoie le nombre d'éléments de la collection.
- **select** : cette méthode renvoie une sous-collection, constituée des objets de la collection qui satisfont le prédicat passé en paramètre. La façon d'exprimer le critère de sélection est détaillée dans la section suivante.
- **select_one** : cette méthode est identique à la précédente mis à part le fait qu'elle ne renvoie qu'un seul élément. A priori, si plusieurs objets satisfont le prédicat, celui qui est renvoyé est choisi arbitrairement.
- **create_index** : cette méthode (non implémentée, voir section III.2.3) permet de créer un index associé à la collection. Il s'agit d'une structure de données, qui, à partir d'une valeur de variable visible définie dans le type T, permet de retrouver les identificateurs des objets de la collection pour lesquels cette variable a cette valeur. Le paramètre passé est le nom de la variable visible à partir de laquelle l'index va être construit. Cette variable visible est appelée *clé* de l'index. Une exception peut être levée en cas de tentative de création d'un index existant déjà (sur la même variable d'état visible).

- **create_unique_index** : (méthode non implémentée) un index "unique" est un index spécial. Deux objets de la collection ne peuvent pas avoir la même valeur pour la variable visible à partir de laquelle est construit un index "unique". La création d'un index peut avoir lieu à tout moment, notamment lorsque la collection contient déjà des objets. Si parmi ceux-ci on trouve déjà plusieurs fois la même valeur pour une variable visible, il sera impossible de créer un index unique à partir de celle-ci et une exception "*duplicate_key*" sera levée lors d'une telle tentative. Lorsqu'un tel index est créé, il est alors impossible d'ajouter un élément à la collection si celui-ci viole la contrainte d'unicité de la clé (on rappelle que nous appelons *clé* la variable visible sur laquelle est construit un index). On prévoit donc l'exception "*duplicate_key*" pour la méthode *insert*. Le but des index est d'accélérer les accès aux objets d'une collection. Ils peuvent permettre d'assurer une contrainte d'intégrité telle que l'unicité d'une "clé". La façon d'implanter les index dans le système n'a pas été complètement déterminée.
- **get_first** permet d'initialiser le parcours séquentiel d'une collection et renvoie un premier élément. L'ordre de parcours n'a aucune sémantique, il est lié à la mise en œuvre de la collection. L'utilisation de cette méthode ainsi que *get_next* nécessite la création d'un objet de type *Compteur*, qui, passé en paramètre, joue le rôle de curseur. Ce curseur propre à l'appelant permet à plusieurs objets de parcourir la collection sans interférer. Il y a un objet de type *Compteur* par appelant, qui mémorise le rang du dernier objet lu. Il permet le parcours séquentiel[1] par plusieurs activités (l'activité est l'unité d'exécution de GUIDE, voir III.2.3.1), en parallèle, d'un objet collection.
- **get_next** : après l'appel de la méthode *get_first*, les appels répétés de cette méthode renvoient successivement tous les éléments de la collection, jusqu'au dernier, l'appel suivant renvoie NIL.
- **union** : cette méthode renvoie l'union ensembliste de la collection sur laquelle elle s'applique avec la collection passée en paramètre.
- **diff** renvoie la différence ensembliste de la collection sur laquelle elle s'applique avec la collection passée en paramètre.
- **intersect** renvoie l'intersection ensembliste de la collection sur laquelle elle s'applique avec la collection passée en paramètre.

III.2.2.4 Expression de prédicat

Nous décrivons ici comment il est possible d'exprimer un critère de sélection sur une collection. Nous définissons un tel critère au moyen d'un prédicat. Toute expression GUIDE qui renvoie une valeur booléenne peut être considérée comme prédicat. Une telle expression utilise en général des valeurs obtenues à partir d'un

1 mais pas la mise à jour (insertion ou suppression d'objet élément).

objet collectionné. Ces valeurs sont récupérées au moyen de l'interface offerte par le type, ce sont donc des valeurs de variable visible ou des résultats de méthode.

<VAL> ::= <nom de variable visible>|
<appel de méthode renvoyant une valeur>

Un prédicat peut aussi utiliser toute expression évaluable dans le contexte de l'appel de la méthode. Une telle expression utilise toute variable ou méthode connue dans le contexte d'appel. Un prédicat est finalement une combinaison logique d'expressions de comparaison de valeurs.

<expression> ::=
<expression><operateur_logique><sous_expression> |
<sous_expression>
<sous_expression> ::= (<expression>) | <expression_atomique>
<expression_atomique> ::= <operande><operateur><operande> |
<bool>
<operateur_logique> ::= OR | AND
<operateur> ::= <= | < | >= | > | = | #
<operande> ::= <expression de valeur du contexte> | <VAL>|
<valeur constante>
<bool> ::= <nom de variable booléenne> | TRUE | FALSE

Dans un premier temps l'expression du prédicat est en fait limitée à

<expression> ::= [<nom de variable visible>] =
<expression de valeur du contexte>

III.2.2.5 Mise en œuvre

Le principe de la mise en œuvre repose sur une phase de précompilation qui remplace les expressions de prédicat par des méthodes d'évaluation de ces derniers. Ces méthodes seront appelées lors de l'opération de sélection proprement dite.

Durant la phase de précompilation, l'analyse syntaxique du prédicat est effectuée, ainsi que la génération du code d'évaluation du prédicat, sous forme d'une méthode de la classe appelante. Cette méthode est rajoutée sur la classe qui appelle la méthode de sélection afin d'avoir accès à son environnement (variables et méthodes de la classe appelante utilisées dans le prédicat). On crée une méthode d'évaluation par prédicat utilisé dans la classe qui appelle les méthodes *select.* Cette méthode reçoit en argument un objet de la collection traitée et renvoie une valeur booléenne, vrai ou faux, suivant que l'objet satisfait le critère de sélection ou non. A l'exécution, la méthode *select* de la classe collection, va faire appel à cette méthode (correspondant à son prédicat) pour chaque objet de la collection ; il faut donc qu'elle connaisse l'objet appelant, qui contient la méthode d'évaluation du prédicat, et le nom de cette méthode. Ces informations lui sont passées en paramètre par l'intermédiaire d'un objet généré de type *Predicat*, qui contient une référence sur l'objet appelant et le nom de la méthode. Un exemple du code généré par le précompilateur est donné ci-dessous.

Exemple :

```
Type Personne IS
    nom: String[80];
    age: Integer;
END Personne.
Type Predicat IS
    appelant: Top;
    methode_pred: String[80];
END Predicat.
Class Bd
IMPLEMENTS Bd IS
...
...
    c: REF Collection OF [Personne]
    ...
    ...
    METHOD truc;
        i: Integer;
        resultat: REF Collection OF [Personne];
        ...
        ...
        resultat := c.select ([nom]="Gustave" AND [age]
        >= i);
        ...
    END truc;
END Bd.
```

La phase de précompilation transforme la classe *Bd* de la façon suivante (ce qui est en gras est rajouté par le précompilateur) :

```
Class Bd
IMPLEMENTS Bd IS
...
    i_global : Integer;
    ...
    ...
    METHOD truc;
        i: Integer;
        p: Predicat;
        ...
        i_global := i;
        p.appelant := self;
        p.methode_pred := "Predicat1";
        // l'appel de la méthode select devient:
        resultat := c.select (p);
        ...
    END truc;
    METHOD Predicat1 (IN r: REF Personne): Boolean;
        BEGIN
        RETURN (r.nom = "Gustave" AND r.age >=
        i_global);
```

```
    END Predicat1;
End Bd.
```

La méthode *select* de la classe Collection est présentée ci-dessous :

```
Class Collection OF [T]
...
    METHOD select (IN p: Predicat): REF Collection OF
    [T];
    ...
    e: REF T;
    ...
    Pour tout élément e de la collection
        Appel de la méthode p.methode_pred (dans notre
        cas "Predicat1") sur l'objet p.appelant (ici
        Bd), avec e comme paramètre.
        Si le retour est TRUE, insérer e dans le
        résultat.
        ...
END Collection OF [T].
```

Deux possibilités furent considérées concernant la génération de la méthode d'évaluation d'un prédicat : elle pouvait être créée soit sur la classe qui appelle la méthode *select*, soit sur la classe *Collection* elle-même. Nous avons choisi la première solution pour les raisons suivantes :

- On peut trouver dans le prédicat des variables de l'environnement (définies dans la classe appelante), comme la variable *i* dans notre exemple. Ces dernières sont directement accessibles par une méthode de la classe appelante[2], alors qu'elles devraient être passées en paramètres à la méthode *select* si l'évaluation du prédicat devait se faire sur la classe *Collection*. Cela est délicat car on ne connaît pas au moment de la compilation de la méthode *select* le nombre et le type de ces paramètres.
- Le prédicat est propre à la classe qui utilise une collection et non à la collection elle-même. Or définir la méthode d'évaluation du prédicat sur la collection implique de recompiler cette dernière, ce qui n'est pas nécessaire à priori.

Nous avons vu que les prédicats peuvent contenir des variables de l'environnement. Deux solutions se présentent pour leur évaluation : soit passer ces variables en paramètre à la méthode d'évaluation, soit en faire des variables globales. Nous avons choisi la deuxième solution, rendue possible par le fait que l'évaluation du prédicat s'effectue sur la classe appelante (voir ci-dessus). La première solution est plus complexe car elle implique de générer une méthode d'évaluation de prédicat avec des paramètres supplémentaires pour les variables d'environnement ; la conséquence est que la méthode *select* de la classe *Collection* ne se contente plus du

2 à condition qu'elles soient globales à la classe et non locales à la méthode où se situe l'appel. Cependant, dans ce dernier cas, elles sont facilement rendues globales à la compilation, comme cela est fait pour *i.*

nom de la méthode d'évaluation du prédicat, mais il lui faut la signature complète et aussi les valeurs de ces paramètres.

Deux parties se distinguent en fait dans l'implantation des collections :

- La première est la classe générique *Collection.* Elle est écrite en GUIDE et est fournie dans une bibliothèque au même titre que d'autres constructeurs comme la liste ou le tableau. Cette classe permet la manipulation d'un ensemble de références. La seule particularité qu'elle présente sont les méthodes de sélection qui font usage d'un type particulier, le type *Predicat*.
- La deuxième est un mécanisme du compilateur qui transforme une expression de prédicat en un objet de type *Predicat*[3] et en une méthode d'évaluation de prédicat. Cette partie est intégrée au compilateur et ne nécessite aucune action particulière afin d'être utilisée. L'objet *Predicat* contient le nom de la méthode d'évaluation et une référence sur l'objet qui la contient (qui est en fait l'objet où l'expression de prédicat était utilisée).

III.2.3 Support système

Le système GUIDE est conçu pour des accès à des objets individuels. Ces derniers sont couplés en mémoire à la demande, un par un. Le stockage se fait de façon inadaptée à l'accès à des ensembles d'objets. Certains mécanismes du système ont dû être révisés pour prendre en compte les notions d'unités de regroupement, également appelées "*cluster*" *(*[30], [31], [32]*),* de stockage et d'accès multi-objets. L'optimisation d'accès a aussi été considérée (introduction d'*index*). Nous présentons rapidement le système GUIDE avant d'aborder les améliorations envisageables.

III.2.3.1 Le système GUIDE

GUIDE peut être vu comme une machine virtuelle distribuée multi-processeurs. Un *domaine* comprend un espace d'adressage contenant un certain nombre d'objets. L'unité d'exécution est l'*activité*. L'exécution d'une activité est une succession d'appels de méthodes sur des objets. Une activité peut créer d'autres activités, éventuellement parallèles. Toutes les activités d'un domaine s'exécutent en parallèle et partagent le contexte du domaine. La communication entre domaines ou activités se fait uniquement au moyen des objets partagés. Le système fournit un support de synchronisation pour l'exécution concurrente de méthodes sur un objet partagé.

Les objets sont stockés dans une mémoire secondaire distribuée appelée Mémoire Permanente d'Objets (MPO). La Mémoire Virtuelle d'Objets (MVO) est le support des objets en mémoire principale, elle est réalisée par l'ensemble des mémoires

---

3 *Predicat* n'est pas implanté comme un type, aucune instanciation n'est nécessaire. C'est en fait un RECORD défini sur le type Top, donc connu dans toute classe, et ce que l'on passe en paramètre à la méthode *select* n'est pas un objet, mais une valeur d'"enregistrement".

(physiques ou virtuelles) des machines. La MVO joue le rôle de cache de la MPO. Pour être accessible dans un domaine, un objet doit y être *lié* ; un mécanisme de chargement assure la présence de l'objet en MVO lors de l'accès.

L'opération de base du système est l'"invocation d'objet", à partir d'une référence, d'un nom de méthode et d'un ensemble de paramètres. Si l'objet n'est pas lié dans le domaine d'appel, c'est un *défaut d'objet*. L'objet est alors localisé en MPO à partir de sa référence, puis est couplé en MVO. Il y a en fait une opération de *chargement* de l'objet, s'il ne se trouve pas en MVO, puis une opération de *liaison* de l'objet dans le domaine (une entrée de la table des objets du domaine est créée pour l'objet en question). La méthode peut ensuite être exécutée comme un appel de procédure classique.

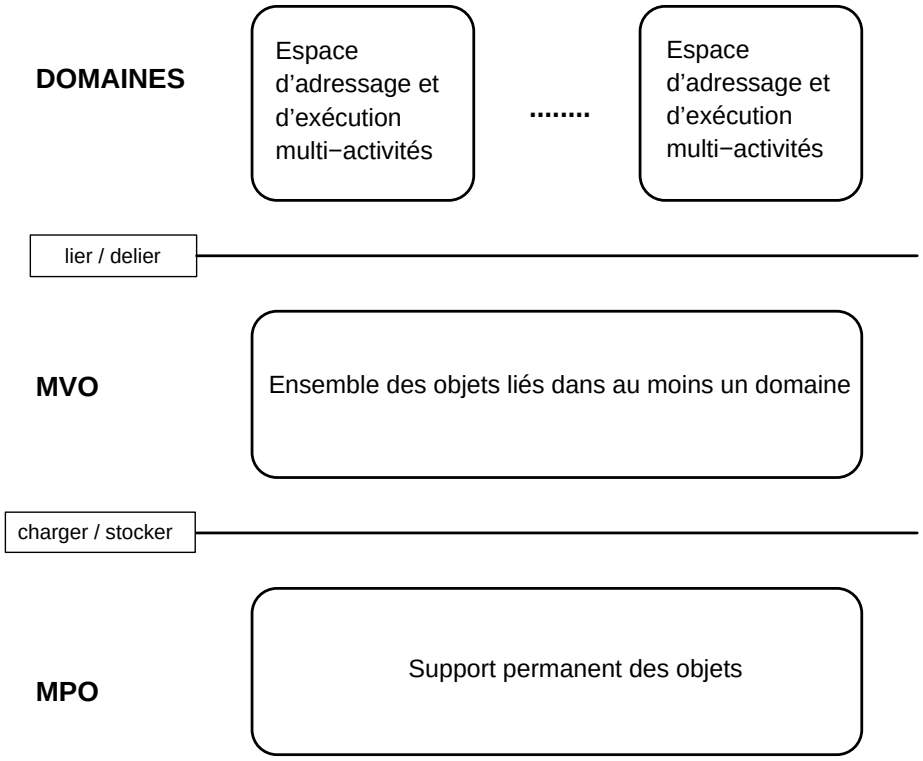

Fig. 3.1 : Architecture du système GUIDE

Les aspects modèle d'exécution et de gestion de mémoire du système GUIDE sont décrits en détails dans [6] et [34].

III.2.3.2 Adaptation du système

Les mesures effectuées sur les temps de réponse d'une petite application de test des collections permettent de déceler deux phénomènes : le premier concerne la politique de chargement individuel des objets en MVO, à savoir que les objets sont chargés un par un, à la demande. Ainsi, lors du premier parcours d'une collection dont les objets se trouvent en MPO, il y a chargement de chaque objet, liaison, accès à

l'objet, puis déliaison : le temps d'accès est alors très grand. Toutefois lors d'un deuxième parcours de la collection, si les objets ont pu rester en MVO (qui nous l'avons vu joue le rôle de cache), les temps sont plus acceptables ; mais si les objets ne peuvent être maintenus en mémoire faute de place, il faut de nouveau pratiquer ces accès individuels. Une solution à ce problème consiste à établir une politique de chargement multi-objet, ainsi que des stratégies de regroupement des objets à charger ensemble : c'est ce qui est fait dans l'étude des "*clusters*" ([30]). Actuellement les objets sont chargés de MPO en MVO individuellement, l'unité de transfert est en fait la page disque. La proposition faite dans [30] consiste à introduire la notion de *groupe* d'objets de façon facultative. Nous distinguons en fait les objets faisant partie d'un groupe, des autres pour lesquels le mécanisme d'accès reste inchangé. Le groupe est une unité de stockage et de transfert entre MPO et MVO contenant un certain nombre d'objets, il est lui-même considéré comme un objet. Lorsqu'un défaut d'objet intervient, si ce dernier fait partie d'un groupe, c'est tout le groupe qui est chargé en MVO. Tous les objets du groupe sont donc dans le cache, lors de leur accès il sera seulement nécessaire de les lier au domaine. La difficulté principale est en fait d'établir la politique de regroupement, nous avons préféré laisser au programmeur le soin de la définir en offrant la visibilité des groupes au niveau du langage.

Le problème de chargement mis à part, la recherche associative nécessite le parcours de toute la collection avec pour chaque référence au moins deux appels de méthodes, un pour obtenir une valeur de l'objet, un autre pour évaluer le prédicat. Ceci est dans notre approche inévitable. Une optimisation peut alors consister à gérer des index qui évitent le parcours séquentiel des collections. Nous pouvons considérer un index comme un objet gérant la correspondance entre des valeurs et des références. Ce dernier devra être utilisé par la classe *Collection* lors de l'exécution de la méthode *select*. La difficulté essentielle reste la mise à jour de ces index lors des modifications d'objets.

Nous constatons que la prise en compte de nouvelles fonctions au niveau du langage peut avoir des répercussions importantes sur le système sous-jacent. Il en sera de même dans l'approche présentée dans la section suivante, pour laquelle les aspects systèmes seront largement développés dans les chapitres suivants.

III.2.4 Conclusion

Ma contribution dans le travail sur GUIDE concerne l'ensemble de ce qui est décrit ci-dessus, c'est-à-dire :

- la définition des principes d'intégration des manipulations ensemblistes dans le modèle GUIDE ;
- la spécification et la réalisation du type générique *Collection* ;
- le mécanisme de traitement des prédicats en collaboration avec les personnes responsables du compilateur ;

- des propositions pour l'adaptation du système GUIDE aux traitements ensemblistes (notamment sur les "clusters").

Il est possible, par un apport restreint au niveau langage et système, de faciliter l'écriture d'applications manipulant des ensembles d'objets. Toutefois des modifications beaucoup plus conséquentes au niveau modèle et système seraient nécessaires pour arriver au niveau d'un SGBD orienté objet. Nous montrons ici qu'une approche consistant à étendre un langage à objets afin d'exprimer des manipulations ensemblistes est possible, mais qu'elle n'est pas suffisante pour l'écriture d'applications de base de données. Le système qui supporte le langage doit être adapté aux types de traitement que l'on trouve dans ces applications (accès séquentiel à un grand nombre d'objets). Nous allons dans la suite étudier l'opération inverse qui consiste à partir d'un SGBD et à proposer une extension pour y intégrer la notion d'objet.

III.3 Modèle et langage relationnels étendus

Nous proposons ici de partir du modèle de données relationnel utilisé dans les SGBD de la génération actuelle et de lui apporter quelques unes des caractéristiques des modèles à objets. Ceci se traduira au niveau langage par une extension de SQL, le langage le plus couramment associé aux SGBD relationnels. La notion de domaine du modèle relationnel est étendue au moyen des types abstraits (ADT : Abstract Data Type). Il est possible de définir le domaine de valeurs d'un attribut de relation par un type abstrait. La notion de constructeur en tant que type générique est aussi offerte afin de générer des objets complexes. Nous ne nous limitons pas à la gestion de valeurs structurées, mais offrons la notion d'identité d'objet. Les instances d'ADT ne sont manipulables qu'à travers leurs méthodes. L'identité d'objet, à savoir que tout objet peut être référencé au moyen de son identificateur, permet de partager des instances d'ADT entre n-uplets et entre objets composés. Ces nouveaux concepts posent des problèmes au niveau langage et sémantique. Ils ont en partie été étudiés dans [33], nous les reprenons dans la section III.3.2 Les aspects systèmes seront développés dans les chapitres suivants.

III.3.1 Définition du modèle et de ESQL

L'objectif visé dans le projet EDS[4] n'est pas de définir un nouveau modèle de données, mais simplement d'ajouter des caractéristiques au modèle relationnel, afin de satisfaire les besoins des nouvelles applications. Le modèle de base reste donc relationnel, afin d'assurer la compatibilité avec les applications existantes, mais aussi pour bénéficier de la simplicité, de l'aspect formel du modèle et des possibilités d'optimisation. Une des conséquences est que le concept d'objet n'est pas introduit

4 Projet ESPRIT dans lequel s'est déroulé ce travail.

de façon très homogène dans le modèle relationnel : deux catégories d'entités restent bien distinctes, les relations d'une part et les objets d'autre part. Les relations restent au premier plan, ce qui montre bien la priorité accordé au relationnel. D'autres approches proposent des extensions beaucoup plus intégrées du modèle relationnel (voir chapitre II), mais aboutissent généralement à des systèmes qui ne sont plus compatibles avec le relationnel. Deux aspects nouveaux ont été introduits, la déduction avec la notion de vue récursive et le concept d'objet avec les types abstraits. C'est à ce deuxième point que nous allons nous intéresser. Le langage permettant d'exprimer les concepts de ce modèle relationnel étendu est un langage compatible SQL, appelé ESQL (Extended SQL) [37], qui intègre les nouvelles notions d'objet et de déduction. Il a été développé par d'autres membres du projet EDS avec lesquels nous avons collaboré pour préciser l'extension objet. Les objectifs visés dans l'élaboration du modèle sont les suivants :

- enrichir l'ensemble des structures de données offertes par le modèle relationnel par l'introduction de la notion d'**objets complexes** ;
- introduire le concept de **partage** dans le modèle en utilisant l'**identité d'objet** ;
- introduire le **comportement** des entités au sein même des données, en utilisant les concepts d'**encapsulation** et d'**héritage** ;
- conserver un système **relationnel**.

Le premier objectif consiste à introduire la notion d'objet complexe dans le modèle de données relationnel. Les seuls objets construits du modèle sont les relations et les n-uplets. Nous avons voulu introduire d'autres constructeurs, sans pour cela bouleverser les bases du modèle. La solution est d'introduire les objets complexes en tant que valeurs des attributs de relation. Le domaine de valeurs de ces attributs était jusque là défini au moyen des types de base (numérique, chaîne, etc.), nous étendons cette notion de domaine en augmentant l'ensemble des types disponibles par des types complexes. L'ensemble des types disponibles n'est plus statique car nous offrons au programmeur la possibilité de définir de nouveaux types à tout moment. Le système devient extensible.

Le deuxième objectif introduit la notion de partage et d'identité d'objet. Un objet complexe, instance d'un type abstrait ESQL, pourra être partagé entre plusieurs n-uplets, et par conséquent entre relations. Il a été décidé d'associer cette caractéristique (objet = partageable, valeur = non partageable) au type. C'est donc le type d'un objet complexe qui détermine s'il peut ou non être partagé. Ce choix sera discuté plus en détail en III.3.2.

Le modèle de type introduit comprend les concepts d'encapsulation et d'héritage. Un type abstrait ESQL est constitué d'une structure de données, définie par une combinaison de constructeurs et de types de base et d'un ensemble de méthodes. Il contient aussi la caractéristique *objet/valeur* comme cela vient d'être précisé ci-dessus. Les constructeurs proposés sont des types abstraits génériques ESQL

(n-uplet, tableau, liste, ensemble)[5], disposant d'un ensemble de méthodes permettant de manipuler les structures de données qu'ils représentent.

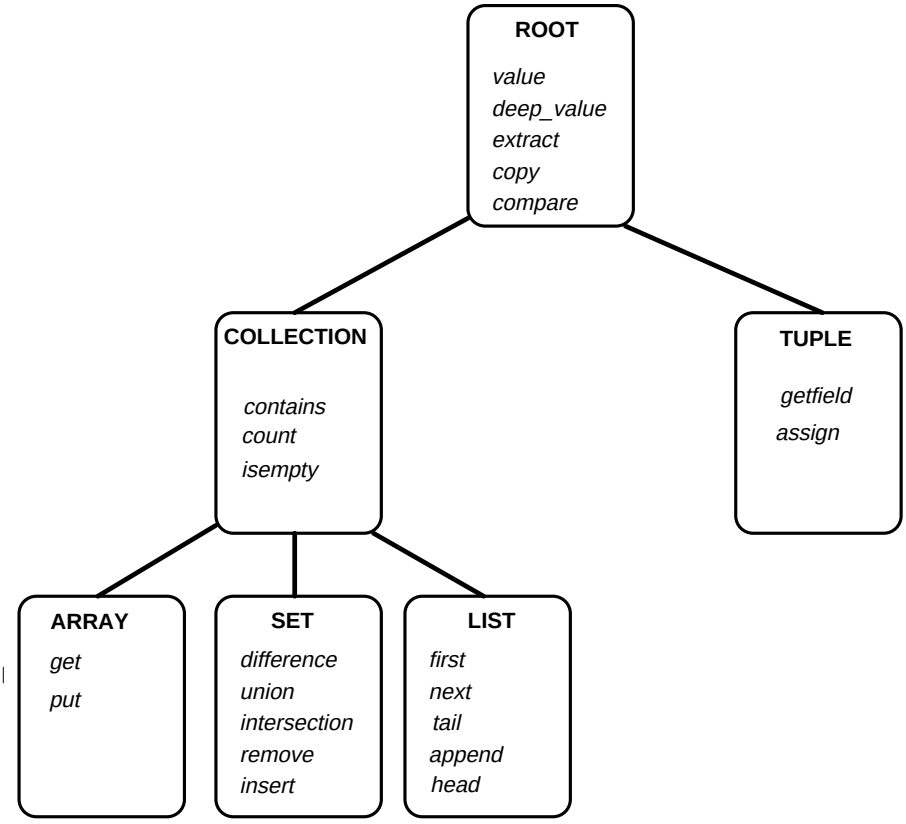

Fig. 3.2 : Les types génériques ESQL

Les méthodes définies sur un type abstrait ESQL sont écrites dans un langage de programmation classique étendu. L'approche est multi-langage, elle sera décrite en détails dans la section VII.1 du chapitre VII. Il existe des méthodes définies implicitement sur un type abstrait ESQL, il s'agit des méthodes héritées des supertypes, ainsi que des méthodes génériques du constructeur de premier niveau utilisé dans sa structure de données. L'héritage fourni est simple et ne permet la spécialisation de la structure de données dans un sous-type que pour les n-uplets, par ajout de champs. Quelques exemples de création et manipulation de types abstraits en ESQL sont données ci-dessous.

```
CREATE TYPE Personne AS OBJECT TUPLE OF (
    nom CHAR(80), prénom CHAR(25));
CREATE TYPE Acteur SUBTYPE OF Personne WITH (
    salaire NUMERIC(8));
CREATE TYPE Acteurs AS SET OF (Acteur);
```

5 respectivement TUPLE, ARRAY, LIST, SET dans la syntaxe anglaise d'ESQL.

Les instances du type *Personne* sont des n-uplets et sont partageables, celles du type *Acteur* possèdent le champ *salaire* en plus et héritent de la caractéristique *objet*. Le constructeur de premier niveau de ces deux types est le *tuple*, ils bénéficient donc des méthodes génériques de ce constructeur qui permettent de manipuler cette structure de données (par exemple la méthode *getfield* permet d'accéder à un champ du *n-uplet*). Le type *Acteur* hérite de toutes les méthodes définies sur le type *Personne*. Le type *Acteurs* définit un ensemble d'objets *Acteur*, ses instances peuvent être vues comme des ensembles de références ou identificateurs d'objets *Acteur*. Il est désormais possible de définir des méthodes sur ces nouveaux types de données. Ceci se fait à l'aide des ordres ESQL suivants :

```
CREATE FUNCTION augmenter_sal (Acteur, NUMERIC(2))
    RETURNS Acteur LANGUAGE C;
CREATE FUNCTION salaire_max (Acteurs)
    RETURNS NUMERIC(8) LANGUAGE C;
```

La première fonction est définie sur le type *Acteur*, elle augmente le salaire d'un acteur du pourcentage passé en paramètre et retourne l'acteur modifié. La deuxième fonction est définie sur le type *Acteurs* et renvoie le salaire le plus élevé d'un ensemble d'acteurs. Les deux fonctions sont écrites en C (voir VII.1). Le premier argument d'une méthode détermine toujours le type sur lequel elle est définie.

Il est désormais possible de définir des relations utilisant ces ADT parmi les types de leurs attributs. Considérons la table ACTEUR contenant l'ensemble de tous les acteurs et la table FILM décrivant des films et possédant un attribut *acteurs*, de type *Acteurs*, qui représente l'ensemble des acteurs d'un film.

```
CREATE TABLE FILM (numf NUMERIC(3), titre CHAR(50),
    acteurs Acteurs);
CREATE TABLE ACTEUR (acteur Acteur);
```

Les requêtes ESQL s'appliquant sur de telles tables peuvent naturellement manipuler les attributs de type ADT au moyen des méthodes définies sur ces derniers. Quelques exemples de telles requêtes sont donnés ci-dessous.

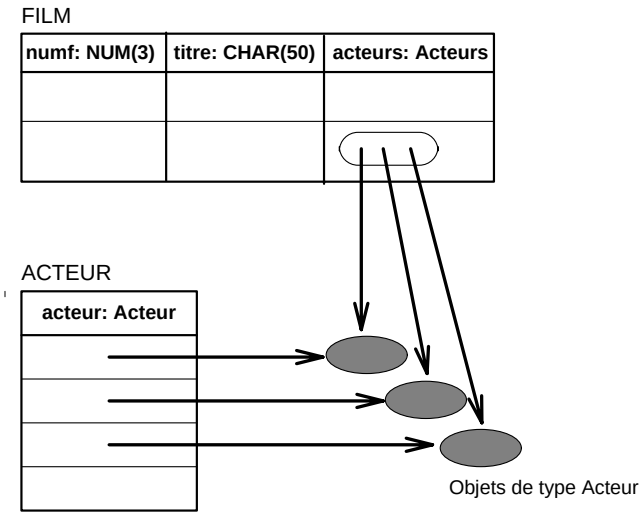

Fig. 3.3 : Un schéma ESQL

```
SELECT salaire_max (acteurs) FROM FILM
    WHERE numf = 3;
UPDATE ACTEUR SET acteur = augmenter_sal(acteur,10)
    WHERE getfield (acteur, nom) = 'Depardieu';
SELECT getfield (acteur,nom) FROM FILM, ACTEUR
    WHERE titre = 'Pretty Woman' AND
    getfield (acteur,salaire) = salaire_max(acteurs)
    AND contains (acteurs, acteur) = 1;
```

La première requête permet d'obtenir le salaire de l'acteur le plus payé qui joue dans le film numéro 3. La deuxième augmente de dix pour cent le salaire de Gérard Depardieu. Finalement la troisième requête indique le nom de l'acteur le plus payé jouant dans le film 'Pretty Woman'. On remarquera l'utilisation des méthodes génériques *getfield* définie sur *tuple*, donc sur *Acteur*, et *contains* définie sur *set* donc sur *Acteurs*.

III.3.2 Partage d'objet et modèle relationnel

L'introduction de la notion de partage d'objet dans le modèle relationnel ne se fait pas sans conséquence sur la sémantique des opérations relationnelles. Il existe en fait plusieurs façons d'apporter cette notion de partage dans le monde relationnel. En fonction de l'approche choisie, la granularité de partage, c'est-à-dire la plus petite entité du modèle susceptible d'être partagée, peut varier. Il convient aussi de reconsidérer la sémantique des opérations de mise à jour qui peut être grandement altérée par les effets de bord dus aux méthodes s'appliquant sur des objets partagés. L'ensemble de ces problèmes, par ailleurs décrit en [33], est résumé dans cette section.

III.3.2.1 Références et identité d'objet

La notion de partage est une propriété du monde réel : un père et une mère de famille partagent leurs enfants, les acteurs sont partagés entre les différents films dans lesquels ils ont tourné. Une telle propriété ne se modélise pas directement dans le monde relationnel, il est nécessaire de contourner le problème et de traduire le concept de partage en concept d'association au moyen d'un subterfuge appelé clé étrangère. Par exemple, pour modéliser les ensembles d'acteurs par film, il aurait fallu en relationnel pur avoir une table pour les films, avec une clé d'accès *numf*, une table pour les acteurs, avec une clé d'accès *numa*, et une table FILM-ACTEUR(numf, numa) établissant l'association entre les films et les acteurs.

```
FILM (numf, titre, année, ...)
ACTEUR (numa, nom, prénom, ...)
FILM-ACTEUR (numf, numa)
```

numf et *numa* sont considérés comme des clés étrangères dans la relation FILM-ACTEUR, car elles correspondent à des clés d'autres relations et doivent désigner des n-uplets existants dans ces relations. Un mécanisme de contrôle d'intégrité référentielle doit donc être fourni pour assurer que toute valeur d'attribut *numf* ou *numa* correspond bien à un film ou à un acteur existant. Le mécanisme de clé étrangère permet de référencer un n-uplet au moyen d'une de ses clés d'accès, à l'intérieur même d'un autre n-uplet. Il faut aussi noter qu'une sélection permettant d'obtenir tous les acteurs d'un film particulier nécessite des opérations de jointure explicites.

```
SELECT a.numa,a.nom
FROM ACTEUR a, FILM f, FILM-ACTEUR s
WHERE a.numa = s.numa AND f.numf = s.numf
    AND f.titre = 'Cyrano';
```

Nous pouvons élaborer une liste des problèmes associés à cette façon de partager les entités relationnelles :

- L'utilisateur doit gérer lui-même les identificateurs que constituent les clés d'accès des relations. Un n-uplet de la table FILM doit toujours posséder une valeur pour l'attribut *numf*, et celle-ci doit être unique. Ceci peut heureusement être contrôlé implicitement par le système relationnel en déclarant l'attribut *numf* comme clé primaire de la relation.
- Toute utilisation d'une clé primaire dans une autre relation doit donner lieu à un contrôle d'intégrité référentiel (mécanisme de clé étrangère).
- Il est nécessaire d'utiliser des jointures pour exprimer la plupart des requêtes. Ces dernières se trouvent plus compliquées qu'elles ne devraient l'être.
- L'utilisateur est souvent amené à créer des attributs "artificiels" pour jouer le rôle de clés afin d'assurer le partage. Ces attributs n'ont rien à voir avec les données manipulées et servent uniquement à établir les jointures.

- Dans le cas où un champ significatif pour l'application est utilisé comme clé, il est possible que ce dernier doive être modifié, ce qui devient une opération très lourde, car il faut modifier tous les n-uplets utilisant cette valeur comme clé étrangère.

Nous proposons une approche, tirant partie de l'introduction des objets complexes, pour exprimer de façon plus naturelle le partage d'objet. Nous proposons dans un premier temps de ramener la granularité de partage du n-uplet au niveau de l'attribut. Ceci est justifié par le fait qu'un attribut peut désormais représenter un objet complexe et présente donc un intérêt supplémentaire à être partagé. Il faudrait alors imaginer un type d'attribut référence permettant de désigner un autre attribut. Des problèmes se posent alors pour savoir si toute valeur d'attribut peut être partagée, s'il faut typer les références sur attribut, ce que devient une référence sur un attribut d'un n-uplet détruit, etc.. L'approche que nous proposons repose sur le principe d'identité d'objet tiré des systèmes orientés objet. L'identité d'objet est un moyen d'identifier un objet indépendamment de son état ; un identificateur est unique, associé de façon définitive à l'objet qu'il permet de désigner, et ne pourra jamais être réutilisé. Notre approche consiste à associer la caractéristique d'objet partageable à toute entité que l'on veut partager, au moyen de son type. Il existe donc des types appelés *types objets*, dont les instances sont partageables, et des types appelés *types valeurs*, dont les instances sont des valeurs complexes non partageables. Toute instance d'un type *objet* T sera conceptuellement stockée à l'extérieur des relations et se verra attribuer à sa création un identificateur unique (OID : Object IDentifier). C'est au moyen de cet OID que tout attribut de type T pourra référencer sa valeur. Plusieurs attributs contenant le même OID partageront donc le même objet. La durée de vie d'une valeur d'attribut non partageable est égale à celle du n-uplet auquel elle appartient. Une instance partageable, elle, ne pourra jamais être détruite directement. Elle ne disparaîtra que lorsqu'elle ne sera plus accessible à partir des racines de persistance que constituent les relations. Nous éliminons ainsi les possibilités de références invalides (pointant sur des objets détruits). Ceci constitue la solution présentée en III.3.1. On constate que de cette façon la granularité de partage ne se limite plus à l'attribut, puisque même les composants d'objets complexes, s'ils sont de type *objet*, pourront être partagés entre plusieurs objets complexes ou n-uplets. Les problèmes d'intégrité référentielle n'existent plus et la plupart des jointures sont désormais implicites.

III.3.2.2 Sémantique des mises à jour

A chaque type abstrait défini dans ESQL est associé un ensemble de méthodes ; la fonction de certaines d'entre elles est de modifier la valeur de l'instance sur laquelle elles sont appliquées. Or dans un système relationnel, les mises à jour de la base ne peuvent se faire qu'à travers les ordres SQL *UPDATE, DELETE* ou *INSERT*. La

sémantique d'un ordre *UPDATE* en SQL consiste à assigner une nouvelle valeur à l'attribut apparaissant dans la clause *SET*.

```
UPDATE R
SET ATT = <expr>
WHERE ...
```

Nous proposons de conserver cette restriction consistant à n'autoriser les mises à jour qu'à travers ces ordres SQL, en adaptant la sémantique du UPDATE à l'application de méthodes modifiant l'état des instances. Supposons que le type de l'attribut ATT de l'exemple précédent soit un type abstrait T :

- si T n'est pas un type *objet*, ses instances ne sont pas partageables, alors l'expression <expr> peut être soit une constante complexe, soit un appel de méthode retournant une valeur de type T. C'est un ordre de mise à jour habituel, où l'ancienne valeur de l'attribut ATT est remplacée par une nouvelle.
- si T est un type *objet*, alors <expr> peut être n'importe quel appel de méthode retournant l'identificateur d'un objet de type T. Une méthode qui modifie l'état de l'objet sur lequel elle s'applique doit obligatoirement retourner l'identificateur de ce dernier (signature M(T,...) -> T). Deux cas se présentent en fait : si <expr> retourne un OID différent de celui de ATT, l'ancien OID est remplacé par l'identificateur d'un autre objet, en aucun cas l'objet anciennement désigné par ATT ne doit être modifié lors de l'évaluation de <expr> ; dans le deuxième cas, si <expr> renvoie le même OID que ATT, alors il est probable qu'une mise à jour ait eu lieu sur l'objet désigné par cet identificateur. Ce dernier cas est illustré par l'exemple donné en III.3.1, que nous rappelons ici :

  ```
  UPDATE ACTEUR
  SET acteur = augmenter_sal(acteur,10)
  WHERE getfield (acteur, nom) = 'Depardieu';
  ```

  La méthode *augmenter_sal*, dont le code est donné en VII.1, met à jour l'acteur passé en paramètre (en augmentant son salaire), et retourne ce même acteur (son OID).

La sémantique de la mise à jour relationnelle est conservée, mais demande plus d'attention de la part du programmeur, qui doit connaître les effets des méthodes qu'il utilise.

III.3.2.3 Effets de bord

Il existe deux problèmes majeurs dus à l'utilisation des méthodes dans les requêtes ESQL : premièrement l'appel de méthode dans une requête de sélection ne devrait jamais affecter l'état de la base de données ; deuxièmement une méthode appelée dans un ordre de mise à jour ESQL ne devrait pouvoir modifier que les données concernées par cet ordre. En effet il est difficile de contrôler la portée d'une méthode qui, par le biais des identificateurs d'objets, peut modifier de nombreux

objets de la base, sans que l'utilisateur en soit forcément conscient. Si un tel contrôle devait se faire, c'est au niveau du code de la méthode qu'il serait mis en œuvre. Ce problème est présent dans tous les SGBD orientés objets. En ce qui concerne l'application de méthodes dans les ordres SELECT, nous proposons deux solutions pour empêcher que les données de la base soient modifiées à l'insu du programmeur :

- la première consiste simplement à interdire l'utilisation de méthodes de mise à jour dans les requêtes de sélection ou dans les clauses WHERE. La difficulté réside alors dans le repérage de ces méthodes. On peut mettre en œuvre une technique récursive pour détecter ces méthodes. Dans l'hypothèse où l'accès aux structures de données des objets ne peut se faire qu'à travers les méthodes génériques, y compris dans le code des méthodes (on verra que cette approche sera effectivement adoptée), la règle est alors la suivante : une méthode est considérée comme une méthode de mise à jour si elle utilise une autre méthode de mise à jour ou si elle utilise une méthode générique de mise à jour.
- la seconde solution est plus compliquée à mettre en œuvre, elle consiste à simuler le comportement d'une méthode de mise à jour utilisée dans un ordre de sélection, sans pour cela affecter l'état de la base de données. On peut imaginer un mécanisme effectuant des copies (en profondeur) des objets passés en paramètre à une méthode dans un ordre SELECT ou dans un WHERE et substituant aux identificateurs d'origine ceux des copies. La méthode va donc modifier et retourner des copies d'objets temporaires. Cette solution permet au programmeur de tester facilement le comportement d'une méthode, sans risque pour la base de données.

Nous choisirons la première solution, beaucoup plus simple à mettre en œuvre. De plus la deuxième solution peut se révéler très coûteuse en copies d'objets.

III.3.3 Modélisation

L'inconvénient majeur introduit par le support d'objets dans le monde relationnel est l'absence de méthodologie de développement et de modélisation. En effet, s'il existe des méthodes de modélisation d'une application en relationnel pur, aucune technique n'existe encore pour modéliser en terme d'objets, et à fortiori encore moins pour une modélisation mixte, relationnelle et objet. Au contraire le nombre de questions se posant au concepteur ne fait que croître : que modéliser sous forme de table, que décrire en objets, comment utiliser les méthodes, etc.. L'expérience nous a montré que pour une même application, il est possible de définir des schémas fortement relationnels, avec très peu d'objets, ou des schémas très fortement orientés objets, dans lesquels les relations n'ont plus qu'un rôle de racine de persistance ou de conteneurs. Entre ces deux extrêmes, il est difficile de choisir un juste milieu et une méthodologie s'avère nécessaire.

III.4 Conclusion

Nous avons pu constater que l'intégration des modèles de données des SGBD avec un modèle à objets ne se fait pas sans problèmes. L'objectif d'une telle manœuvre est double : il s'agit d'une part d'aboutir à un modèle plus puissant et d'autre part de spécifier un langage qui permette d'écrire non seulement le schéma de données, mais aussi les programmes des applications. Nous tirons ici profit des langages à objets qui sont plus facilement adaptables que les langages classiques grâce à l'extensibilité offerte par la notion de "type défini par l'utilisateur".

Le premier point de divergence entre les langages offerts par les SGBD et les langages de programmation est leur mode opératoire. Les premiers ont une vocation interactive et sont déclaratifs. L'utilisateur caractérise les données qu'il veut obtenir et le système se charge d'appliquer les algorithmes adéquats. Ces langages sont des langages de requêtes et ne permettent pas l'écriture complète d'applications. Il est nécessaire de leur adjoindre un langage de programmation qui permet d'écrire tout ce qui ne concerne pas l'accès aux données. On l'appelle "langage hôte", il contient des appels au SGBD permettant d'exécuter les ordres du langage de requêtes. Nous avons introduit dans GUIDE l'aspect déclaratif au niveau de la méthode *select* définie sur la classe *Collection*. La complexité des requêtes exprimées est encore très limitée et il n'existe pas de langage interactif permettant de consulter les données. ESQL est lui un langage déclaratif qui peut intégrer au niveau du code des méthodes des traitements impératifs. Cela ne semble cependant pas suffisant pour écrire toute une application. Pour combler cette lacune, une expérience a été menée pour fournir au serveur ESQL une interface C++ [58] : il s'agissait d'offrir la possibilité d'écrire en C++ des programmes manipulant des données d'une base ESQL[6].

Le deuxième point de divergence concerne les types des langages de programmation qui, dans le cas des langages hôtes, ne sont pas ceux du SGBD. Dans nos deux approches, le langage traité est le seul permettant de manipuler les données persistantes, nous n'avons donc pas vraiment le problème de la confrontation de deux systèmes de types. Dans GUIDE les manipulations ensemblistes ont été complètement intégrées sous forme d'objets et de méthodes ; les ensembles d'objets manipulés constituent des objets comme les autres. Au niveau d'ESQL il faut encore distinguer les données purement relationnelles que sont les relations, des données que l'on peut trouver dans les domaines. Les premières se manipulent avec les opérateurs relationnels, tandis que les autres ont leurs opérations associées (méthodes sur les types abstraits, opérations classiques sur les types de base) ; l'intégration faite de ces deux types d'opération au niveau du langage ESQL semble cependant satisfaisante.

---

6 Cette approche est basée sur la correspondance entre types ESQL et classes C++.

L'extension apportée au langage GUIDE montre qu'il est possible d'intégrer facilement les traitements ensemblistes et notamment les accès associatifs à un langage à objets. Bien que l'expérience soit très limitée, elle est réalisée de façon la plus intégrée possible au modèle, sans ajout de nouveaux concepts. Il faudrait maintenant la compléter en ajoutant des moyens de sélection sur plusieurs collections et en autorisant des résultats autres que des sous-collections.

ESQL est une approche originale d'extension du modèle relationnel qui allie la gestion d'objets complexes à la notion d'identité d'objet. Il est intéressant de constater que des organismes de normalisation comme l'ISO s'intéressent à des extensions tout à fait similaires de SQL ([45]). Son inconvénient majeur vient du fait qu'il confronte deux modèles de données, le relationnel et l'objet, ce qui rend difficile la modélisation des applications. Nous allons étudier dans la suite la mise en œuvre d'un tel système relationnel étendu.

Chapitre IV

Architecture et définition du support système du SGBD relationnel étendu

IV.1 Introduction

Nous présentons dans ce chapitre les choix de conception concernant l'intégration du support d'objets dans l'architecture de notre système. L'objectif du projet EDS est la conception d'un SGBD relationnel étendu (objet et déduction) sur une machine parallèle. L'extension objet a des répercussions au niveau modèle, langage et système qui font l'objet de notre étude. Nous nous intéressons ici à la partie système. Dans la première phase du projet EDS, deux prototypes furent développés : le premier met en œuvre un SGBD purement relationnel prenant en compte le parallèlisme, le deuxième est un système centralisé, qui a pour objectif de mettre en œuvre et de valider les fonctionnalités ajoutées au modèle relationnel ainsi que le langage ESQL. C'est ce deuxième prototype qui a servi de base expérimentale au développement de la plupart des mécanismes à objets. Un troisième prototype, en cours de développement, permet de combiner les aspects parallélisme et extension objet.

L'architecture du serveur relationnel EDS est décrite dans ce chapitre, ainsi que la position dans cette architecture des différents composants du support d'objets. Ce dernier est traité à tous les niveaux du système. Au plus haut niveau, nous avons écrit un gestionnaire de types qui permet de stocker la définition des types abstraits (ADT) et de maintenir un certain nombre de relations entre eux (composition, héritage). Nous avons ensuite intégré dans les couches basses du système un module de gestion des instances d'ADT et un module de gestion des méthodes. Le premier assure le contrôle du stockage, de l'accès et de la manipulation des objets. Le deuxième s'occupe de la compilation, de l'édition de liens et de l'exécution des méthodes.

L'architecture générale du système EDS est présentée dans la section suivante. L'intégration d'un support système pour l'extension objet, ainsi que l'impact sur l'architecture figurent dans la section IV.3.

IV.2 Architecture générale

La machine parallèle sur laquelle doit être développé le système est composée d'un certain nombre de nœuds interconnectés. Deux architectures de machine sont en fait utilisées, elles conduiront à la réalisation de deux versions du système. La première est une machine à mémoire distribuée : sur celle-ci, chaque nœud dispose d'un processeur et d'une mémoire de grande taille. La deuxième version est développée sur une machine parallèle à mémoire partagée : chaque nœud dispose d'un processeur, mais partage une mémoire commune avec les autres. C'est cette deuxième version du système que nous utiliserons lorsque nous intègrerons les aspects objets dans le prototype parallèle appelé DBS3, pour "Database System on Shared Store" ([15]).

Le système est divisé en trois modules : le *Gestionnaire de requêtes* (RM, pour *Request Manager*), le *Gestionnaire de données* (DM, pour *Data Manager*) et le *Gestionnaire de sessions* (SM, pour *Session Manager*).

Le RM effectue la compilation et l'optimisation des requêtes ESQL, et génère un programme qui pourra être exécuté par le DM. Ce traitement est réalisé en plusieurs phases qui communiquent par un langage intermédiaire appelé LERA (Language for Extended Relational Algebra). Le programme généré est un module objet contenant des appels aux opérateurs relationnels étendus de la librairie *Lera RunTime Library* (Lera RTL) du DM.

Le DM est responsable du stockage des données de la base et de l'exécution des programmes de requêtes parallèles générés par le RM. Il fournit donc l'environnement d'exécution, les opérations LERA, les méthodes de stockage et d'accès aux relations, et le support de stockage fiable.

Le SM est responsable de l'initialisation d'une session avec le serveur relationnel. Il s'agit de générer une instance de RM pour la connecter avec une tâche utilisateur. Dans le prototype à mémoire partagée (DBS3), développé sur le noyau MACH (machine ENCORE), le module de RM créé reçoit les requêtes de la tâche utilisateur et génère un code C compilé. Ce code est lié aux librairies du DM et exécuté dans une tâche MACH, appelée *tâche d'exécution*. Cette tâche transmet directement les résultats à la tâche utilisateur. Un tel mode de fonctionnement est illustré dans la figure Fig. 4.1. Les requêtes de définition de données (LDD) sont juste interprétées par le RM qui appelle directement le gestionnaire de catalogue pour mettre à jour le catalogue.

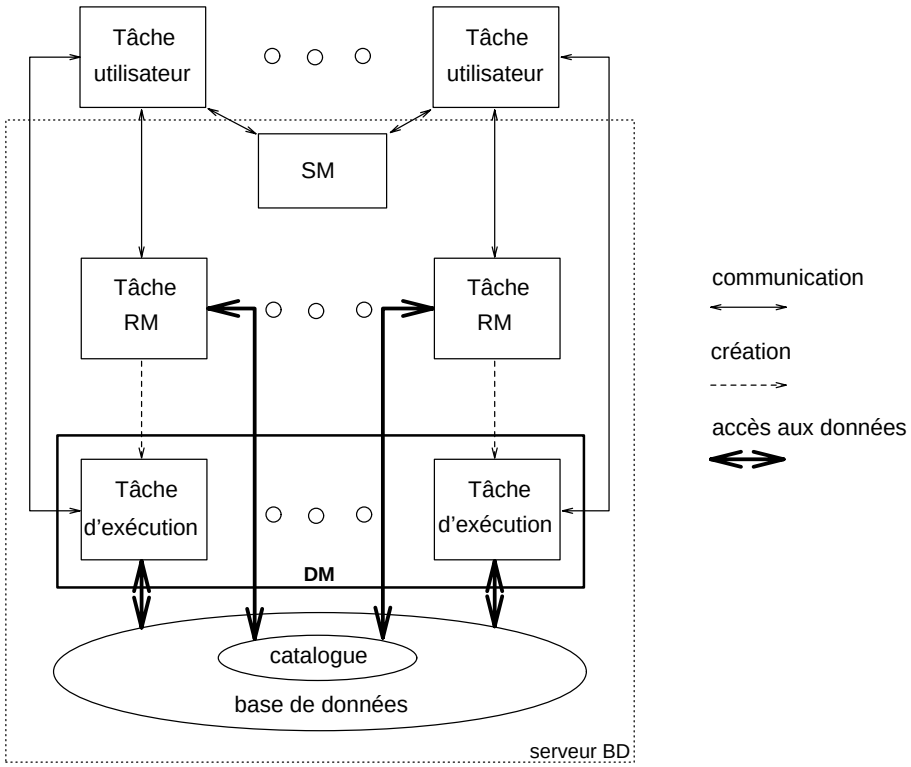

Fig. 4.1 : Architecture générale (DBS3)

Les modules nécessaires au support d'objets complexes sont présentés dans la section IV.3 et seront décrits en détail dans les chapitres suivants.

IV.2.1 Modèle d'exécution parallèle

Nous évoquons brièvement ici le modèle utilisé pour paralléliser l'exécution des requêtes relationnelles ([21]). Les relations sont distribuées sur les nœuds de la machine suivant un découpage horizontal (par ensemble de n-uplets). Cette distribution est basée sur une fonction (index, hachage, rang, ...) appliquée sur un ou plusieurs attributs. L'ensemble des nœuds qui contiennent des n-uplets d'une relation s'appelle le "home" de cette relation.

Chaque opération relationnelle est décomposée en plusieurs sous-opérations. Chaque sous-opération s'exécute sur un nœud du "home" de la relation. L'idée est de paralléliser au maximum l'exécution des opérations relationnelles. Une requête relationnelle est représentée par un graphe d'opérations dans lequel les arcs

représentent des flots de données (des n-uplets, des relations). Il existe trois façons de séquencer ces opérations afin d'assurer le maximum de parallélisme :

- l'exécution en "pipeline" : l'exécution des opérations relationnelles est ramenée au niveau du n-uplet, dès qu'une opération a traité un n-uplet, celui-ci est disponible pour l'opération suivante.
- l'exécution "séquentielle" exige la production complète du résultat (relation) intermédiaire avant que celui-ci ne puisse être utilisé par l'opération suivante. Ceci est par exemple nécessaire pour exécuter une opération de *différence*.
- l'exécution "parallèle" est possible lorsqu'il n'y a pas de dépendance entre deux opérations qui peuvent alors s'exécuter en parallèle.

Il existe enfin des mécanismes de contrôle permettant d'assurer une exécution parallèle globale correcte. Il s'agit notamment de pouvoir détecter la fin d'une opération locale qui s'exécute en "pipeline" ; cela se fait par envoi d'un message de "fin de transmission de données". Il existe aussi une opération de contrôle permettant de détecter la fin d'une opération distribuée ; cette opération collecte les "fins locales" des sous-opérations.

Supposons qu'une relation `R1` soit distribuée sur un ensemble de nœuds ("home") `h1`, et que `R2` soit distribuée sur `h2`. Considérons la requête suivante :

```
SELECT R2.a, R2.b
FROM R1, R2
WHERE R1.z = R2.a AND R1.x = 3;
```

Cette requête peut être décomposée en une opération de sélection `s1` sur `R1`, suivie d'une opération de jointure `j2` avec `R2`. L'exécution globale peut donc être constituée d'un ensemble de sous-opérations de sélection (avec le même critère que `s1` à savoir `x = 3`) sur les nœuds `h1`, qui s'exécutent en parallèle et qui transmettent en "pipeline" les n-uplets résultant aux sous-opérations de jointure s'exécutant sur les nœuds `h2`. Une telle exécution est illustrée par la figure Fig. 4.2 ; il faut noter que `h1` et `h2` ne sont pas forcément disjoints comme cela apparaît sur cette figure, ce qui conduit à réduire le parallélisme réel.

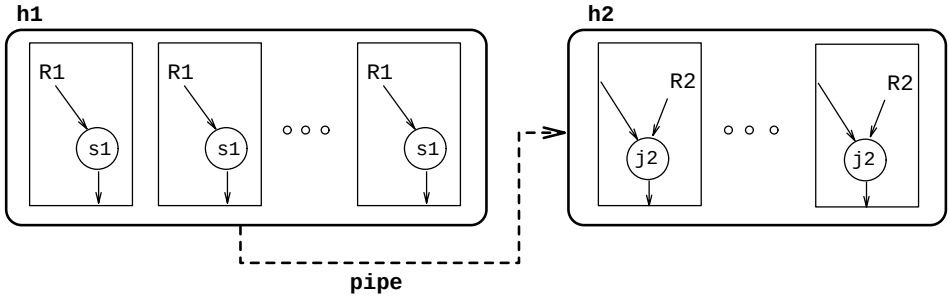

Fig. 4.2 : Exécution parallèle

IV.2.2 Gestionnaire de requêtes

Ce module compile et optimise les requêtes ESQL. Le langage intermédiaire LERA est utilisé pour la communication entre les différents composants qui sont décrits dans cette section. Le point d'entrée du RM est le *moniteur*, qui est chargé d'appeler successivement les différents composants. Le compilateur est composé de cinq modules responsables des phases suivantes : analyse et traduction, optimisation logique, optimisation physique, parallélisation et enfin génération de code.

L'*analyseur-traducteur* effectue l'analyse syntaxique et sémantique des requêtes LDD et LMD du langage ESQL et les transforme en programmes LERA. Les requêtes LDD ne passent que cette phase, elles sont ensuite directement interprétées par le *moniteur*, en interaction avec le catalogue.

L'*optimiseur logique* se charge de l'optimisation syntaxique et sémantique. La première est basée sur les propriétés des opérateurs de l'algèbre relationnelle et consiste à permuter les opérations. La deuxième utilise une certaine connaissance sémantique sur la base pour transformer les requêtes (une contrainte d'intégrité *age < 39* peut par exemple être utilisée pour annuler un critère de sélection de type *age > 45*).

L'*optimiseur physique* optimise les accès en utilisant les informations sur l'organisation physique des données. Les décisions à prendre concernent l'ordonnancement des opérations de jointure, le choix des méthodes d'accès et le choix entre les différents algorithmes possibles pour l'exécution des opérateurs LERA.

Le *paralléliseur* génère un programme LERA parallèle à partir du programme LERA optimisé. Il transforme en fait le programme en un ensemble d'opérations communicantes destinées à être exécutées sur différents nœuds de la machine.

Le *générateur de code* produit un module exécutable par le DM.

Le *catalogue* gère le schéma de la base, ainsi que des informations sur le stockage des données. Il est utilisé par l'analyseur sémantique ainsi que par les optimiseurs.

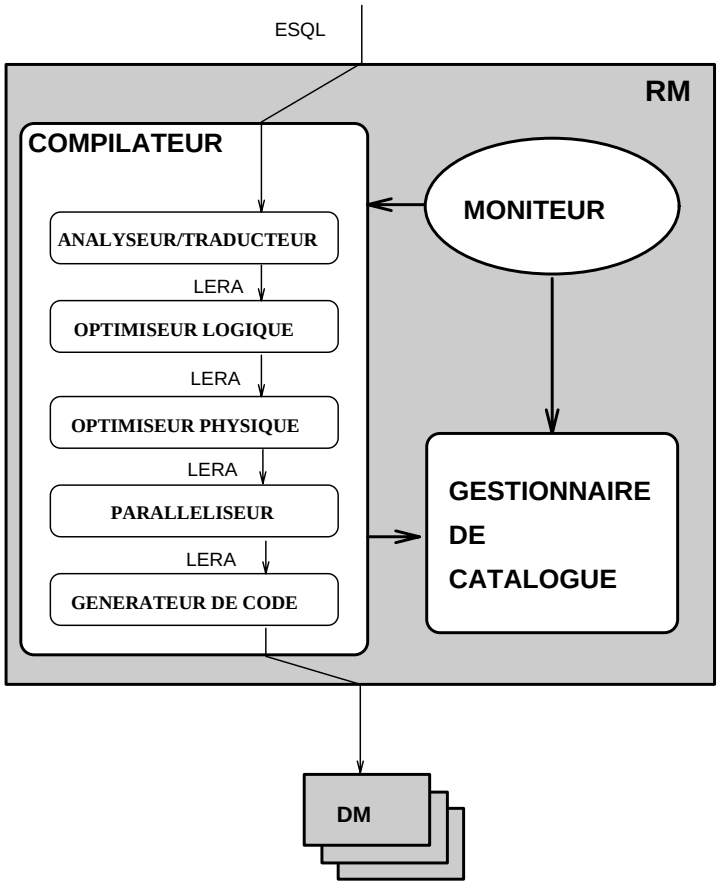

Fig. 4.3 : Gestionnaire de requêtes

IV.2.3 Gestionnaire de données

Le DM est responsable de l'exécution des requêtes et du stockage des données. A cet effet il dispose d'un certain nombre de modules chargés de la mise en œuvre des opérateurs relationnels, des mécanismes d'exécution parallèle, du stockage des objets et des méthodes d'accès aux relations.

- L'ensemble des opérateurs relationnels (LERA) constituent un module appelé LERA RTL (LERA Run Time Library).
- Le *Modèle d'Exécution Relationnel* (REM) fournit les mécanismes de distribution et de communication requis par l'exécution parallèle des opérateurs relationnels. Il fournit le support d'exécution de base, c'est-à-dire le concept de processus léger, les mécanismes de communication entre processus, etc..
- Le module *Méthodes d'Accès Relationnelles* (RAM) gère les méthodes d'accès associées à une relation. La méthode d'accès définie sur la clé primaire d'une relation définit la méthode de stockage de cette relation et

notamment la répartition des n-uplets sur les nœuds. Il est facile d'ajouter de nouvelles méthodes d'accès au système.

- Le *gestionnaire de pages* fournit la gestion de données de bas niveau, utilisée par les méthodes d'accès, et la gestion de transactions.

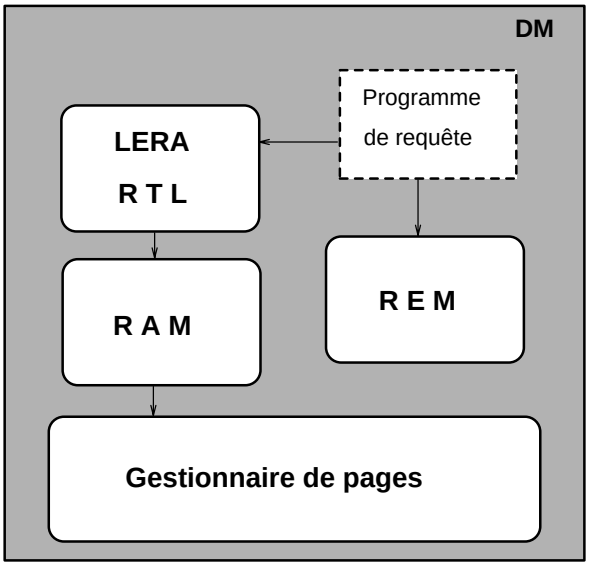

Fig. 4.4 : Gestionnaire de données

Les composants du DM constituent en fait un ensemble de librairies auxquelles sont liés les programmes générés par le RM. Le code résultant est exécuté dans une tâche MACH. On montre sur la figure Fig. 4.4 qu'un programme généré par le RM fait appel aux fonctions des modules LERA RTL (opérations relationnelles) et REM (fonctions de communication et de contrôle pour l'exécution parallèle). Les opérations relationnelles s'appuient sur les méthodes d'accès (RAM) qui, elles, utilisent le *gestionnaire de pages*.

IV.3 Support système

Nous appelons "support d'objets" l'ensemble des fonctions permettant de gérer la définition des types abstraits et de leurs instances dans le système. Cela comprend la gestion d'un catalogue des types créés, le stockage des instances de ces types, la gestion des méthodes et l'optimisation des manipulations sur ces données particulières. L'introduction de ces fonctionnalités est répartie sur différents niveaux du SGBD. Après avoir discuté les choix principaux de conception, nous présentons l'intégration des différents composants permettant de mettre en œuvre ces fonctions dans l'architecture du système.

IV.3.1 Choix de conception

Il existe différents niveaux d'intégration du "support d'objets" avec un système relationnel. En fonction du niveau où l'on décide de prendre en compte l'interaction entre les deux modules, ce couplage peut être fort ou faible. La façon de réaliser une telle intégration dépend aussi de l'orientation principale que l'on veut donner au système. Suivant que l'on désire un système principalement relationnel, avec quelques extensions objets, ou que l'on veuille un système où les capacités objets seront aussi importantes et autant utilisées que l'aspect relationnel, on adoptera une approche plus ou moins intégrée.

Nous avons abordé l'intégration objet-relationnel de façon progressive, afin de ne pas être confrontés à tous les problèmes à la fois. L'approche la plus simple pour dégager les problèmes liés à la confrontation des mondes objet et relationnel consiste à prendre un système relationnel et un système à objets, puis à tenter de les faire coopérer. C'est l'approche dite "couplage faible". Nous avons utilisé un SGBD relationnel existant (SABRINA), ce qui évite d'implanter toute la technologie relationnelle. Nous avons par ailleurs développé un gestionnaire d'objets pour les instances d'ADT ESQL. Il suffit ensuite de faire coopérer ces deux modules. Dans une approche "couplage fort", il est souhaitable d'unifier le stockage relationnel et le stockage des objets complexes. Cela peut être fait de deux façons : la première consiste à utiliser un système de stockage relationnel et à décomposer les objets complexes en relations ; la deuxième solution, elle, utilise un gestionnaire de stockage d'objets permettant de stocker à la fois les objets complexes et les relations, en tant qu'objets. La première solution, qui revient à construire une couche de gestion d'objets sur un système relationnel, est pénalisée du point de vue des performances. En effet, la décomposition des objets complexes en relations nécessite d'employer des requêtes relationnelles lourdes (beaucoup de jointures) pour accéder aux objets. La deuxième solution, elle, nécessite sûrement de reconsidérer la partie "accès aux relations" du noyau relationnel. Pour notre premier prototype nous avons adopté une approche coopérative ou couplage faible. Le gestionnaire de stockage d'objets est totalement externe au système relationnel et les manipulations d'objets sont séparées au maximum des opérations relationnelles. Le résultat est un système assez complexe, puisqu'il est en fait composé de deux systèmes, et pour lequel les mesures de performances sont assez peu significatives. Nous résumons ci-dessous les avantages d'un couplage faible :

- Il permet d'aborder progressivement, et dans un effort de développement minimal, les problèmes de l'intégration des mécanismes objets et relationnels.
- Le modèle de données lui-même (ESQL) se prête assez bien à un couplage faible car il ne fournit pas une intégration très forte des concepts objets et relationnels. On distingue assez facilement les manipulations du monde des

objets de celles du monde relationnel. A priori, le fait d'utiliser deux systèmes coopératifs ne semble pas nuire au mécanisme d'exécution global.

- Cela permet de récupérer la technologie relationnelle existante et assure le support d'applications purement relationnelles sans dégradation de performance par rapport à un système relationnel classique. Le couplage faible est finalement, à l'exemple d'INGRES, la solution industrielle au problème de l'extension du relationnel.

Cette première étape a apporté une première expérience pratique qui a permis de résoudre les problèmes de base et de mettre à jour de nouveaux problèmes liés à cette intégration. Nous pouvons notamment citer un certain nombre d'inconvénients du couplage faible :

- Le fait de coupler deux systèmes génère une certaine redondance des mécanismes (notamment pour les couches basses de la gestion du stockage). La taille du système résultant est importante.
- Un tel système est plus difficile à faire évoluer. Un système de taille plus réduite, pour lequel chaque composant a été développé de façon spécifique est plus simple à maintenir et à faire évoluer.
- Nous verrons que l'optimisation est beaucoup plus difficile dans une approche couplage faible, car les opérations (et leurs opérandes) ne sont pas toutes de même nature.

L'étape suivante consiste à réaliser une intégration un peu plus forte. Cela est fait dans le deuxième prototype pour lequel un noyau relationnel a été développé. Nous avons donc accès au module gérant le stockage, et pouvons l'utiliser pour stocker les objets. Nous conservons tout de même l'indépendance de la gestion d'objets et du stockage relationnel, mais utilisons pour les deux un gestionnaire de pages commun.

Un deuxième choix se présente alors concernant la prise en compte des fonctionnalités objet. Il s'agit cette fois de considérer ou non une approche serveur. Celle-ci consiste à placer les fonctions du support d'objets dans un serveur à part. Cette opportunité ne se présente donc que dans le cas d'un couplage faible. Ce serveur d'objets est appelé par le système relationnel pour les manipulations d'objets. Une telle approche est motivée par l'aspect récupération d'erreur dans les méthodes. Le code des méthodes utilisateurs peut mettre en défaut le système à tout moment (erreur de programmation dans la méthode du style division par zéro ou débordement de tableau). Il est alors intéressant de voir dérouler ce code dans un processus indépendant de celui du SGBD, qui pourrait alors récupérer l'erreur (et relancer le serveur). Une autre approche pour résoudre ce genre de problème consiste à utiliser un interpréteur de méthodes, plutôt que de les compiler. Nous avons cependant abandonné l'approche serveur pour des raisons de performance. En effet les coûts de communication pourraient rendre les temps d'exécution des requêtes très critiques (en général une invocation de méthode a lieu pour chaque

n-uplet sélectionné). Il est envisageable d'élaborer un mécanisme d'exception pour récupérer certaines erreurs ; mais il est toujours possible que des données systèmes soient écrasées, ce qui entraîne un arrêt inexorable du SGBD ; il faut alors compter sur les mécanismes de reprise pour assurer la cohérence de la base.

L'intégration des mécanismes d'exécution est assez simple, puisque le concept de base du modèle d'exécution objet est l'appel de méthode. Il suffit alors de permettre au système d'exécution relationnel d'effectuer ces appels de méthodes.

En ce qui concerne l'optimisation, le problème est plus délicat. Pour l'organisation physique des données (regroupements, index), le problème dépend entre autres de l'intégration réalisée au niveau du stockage. Pour l'optimisation elle-même, nous verrons que la plupart des techniques sont issues du relationnel, ce qui ne fait qu'accentuer le problème de la redondance des mécanismes dans le cas d'un couplage faible. L'optimisation des requêtes avec ADT ne sera prise en compte que dans le deuxième prototype.

Compte tenu des deux choix effectués, couplage faible sans serveur d'objets, nous pouvons maintenant définir les différents composants du support d'objets, leurs fonctions et leur interaction avec le système relationnel. Ces composants sont le gestionnaire de types, qui gère les définitions des types abstraits et fait partie du gestionnaire du catalogue, le gestionnaire d'instances et le gestionnaire de méthodes qui font partie du *gestionnaire de données* (DM). Ils sont décrits dans les trois chapitres qui suivent. Leur intégration dans l'architecture est présentée dans la section suivante.

IV.3.2 Intégration des composants du support objets

Nous allons étudier dans cette section la façon dont le support d'objets intervient dans les différents modules de l'architecture définie précédemment. L'architecture résultante de notre premier prototype est ensuite décrite.

IV.3.2.1 Principes généraux

L'accroissement des capacités de modélisation dû à l'introduction des ADT dans le modèle relationnel a une première répercussion au niveau du catalogue. Celui-ci a donc été conçu pour maintenir non seulement les définitions des structures relationnelles classiques, mais aussi les définitions inhérentes au modèle de types complexes ESQL. Le gestionnaire de types abstraits est un des modules du support d'objets qui a pu être développé de manière assez autonome, afin d'être intégré dans le catalogue et utilisé par d'autres composants du système. Il est décrit en détails dans le chapitre V.

L'analyseur prend en compte les composantes syntaxiques et sémantiques des définitions de types et des manipulations d'objets. C'est un travail fortement lié à la définition du langage (ESQL). Il faut tout de même noter que l'analyseur effectue des contrôles sémantiques au niveau des manipulations d'objet et qu'il utilise pour

cela les services offerts par le gestionnaire de types. D'autre part, sa connaissance sémantique des opérations en cours et des types des valeurs manipulées peut être utilisée pour insérer des appels aux fonctions de gestion d'objets au moment même de la traduction en LERA.

Par exemple dans l'ordre ESQL suivant, où ATT1 et ATT2 désignent des attributs de type ADT *valeurs* (non partageables), l'analyseur a pour rôle de générer l'appel à la fonction de copie de valeur, afin que celle référencée par ATT2 ne se trouve pas partagée. Une telle opération est nécessaire car nos valeurs sont gérées par références (voir chapitre VI), c'est-à-dire que les attributs ATT1 et ATT2 contiennent en fait des références sur des valeurs (soient `v1` et `v2` ces valeurs). Le fait d'assigner directement par cet ordre ATT2 à ATT1 conduirait à obtenir deux fois une référence sur `v2`, ce qui reviendrait à partager cette valeur entre ATT1 et ATT2 ; ceci est évidemment contraire à la sémantique d'une valeur. Le fait de générer cette copie permet donc à ATT1 et ATT2 de désigner deux valeurs égales, mais distinctes (non partagées).

```
UPDATE R
SET ATT1 = ATT2
WHERE ...
```

L'ordre LERA généré correspondra à :

```
UPDATE R
SET ATT1 = AdtCpy (ATT2)
WHERE ...
```

L'optimiseur logique ne prend pas encore en compte les manipulations d'objet. Son action pourrait consister à appliquer des optimisations sémantiques, comme la simplification des combinaisons d'appels de méthodes, à condition d'avoir les moyens de connaître la sémantique de ces dernières. Nous disposons d'un optimiseur logique à base de règles de réécriture. La prise en compte des ADT pourrait donc se faire simplement par l'ajout de nouvelles règles (on profite ici de l'aspect extensible de l'optimiseur). Toute la difficulté réside dans la définition de ces nouvelles règles, car seul l'utilisateur qui définit les ADT connait la sémantique des méthodes associées. Une étude est actuellement en cours sur la définition d'une interface permettant de décrire facilement ces règles. L'optimiseur physique pourrait lui tenir compte de données comme les temps d'exécution de méthode, le placement des objets sur les nœuds et les éventuels chemins d'accès définis sur les collections d'objets.

Paralléliseur et générateur de code sont peu influencés par le support d'objets.

En ce qui concerne le DM, le support d'objets est pris en compte à trois niveaux. Il s'agit en premier lieu d'ajouter à l'ensemble des opérateurs relationnels une primitive d'appel de méthode et de savoir la combiner avec les opérations de filtrage de relation (sélection sur critère). Il faut ensuite un mécanisme d'exécution de méthodes. Enfin il faut gérer le stockage et l'accès aux objets complexes. Le dernier

point comprend la gestion de la fiabilité des données et des accès concurrents qui doit être coordonnée avec les mécanismes existants du noyau relationnel.

IV.3.2.2 Architecture du premier prototype

Le premier prototype a été réalisé en utilisant le SGBD relationnel SABRINA comme noyau relationnel de base et le système de gestion d'objets GEODE comme support de stockage d'objets complexes. Il est basé sur une approche interprétée : le *gestionnaire de requêtes* transforme une requête ESQL en code LERA qui est interprété par une machine algébrique. De plus ce prototype est centralisé, le parallélisme n'entre pas en compte. L'architecture de ce premier prototype est décrite ci-dessous.

Le *gestionnaire de requêtes* est limité à l'analyseur/traducteur ESQL et à l'optimiseur logique. Il génère des programmes LERA directement envoyés au *gestionnaire de données*. Il utilise évidemment le gestionnaire de catalogue auquel a été intégré le gestionnaire de types. Nous incluons aussi dans ce RM le compilateur de méthodes. Il est activé lorsque le RM reçoit un ordre de compilation de méthode (qui fait partie du LDD) ; il génère le code objet de la méthode ; le RM envoie alors au DM un ordre LERA lui indiquant qu'il peut intégrer (entre autre faire l'édition de lien) la nouvelle méthode.

Le *gestionnaire de données* est chargé d'exécuter les programmes LERA générés par le *gestionnaire de requêtes*. Il est composé d'un interprète de programmes LERA et d'une librairie d'opérateurs algébriques capables d'exécuter des opérations relationnelles. Il comprend aussi un module pour la gestion d'objets, qui est constitué d'un ensemble de fonctions de stockage et de manipulation d'objets, et d'un gestionnaire de méthodes. Ce dernier contient un mécanisme d'exécution de méthode et un éditeur de liens dynamique. Les deux derniers composants du *gestionnaire de données* assurent le stockage des données. Il s'agit du gestionnaire de stockage du SGBDR SABRINA d'une part et du gestionnaire d'objets GEODE d'autre part. La figure Fig. 4.5 illustre cette architecture et fait ressortir en grisé les composants propres au support d'objets.

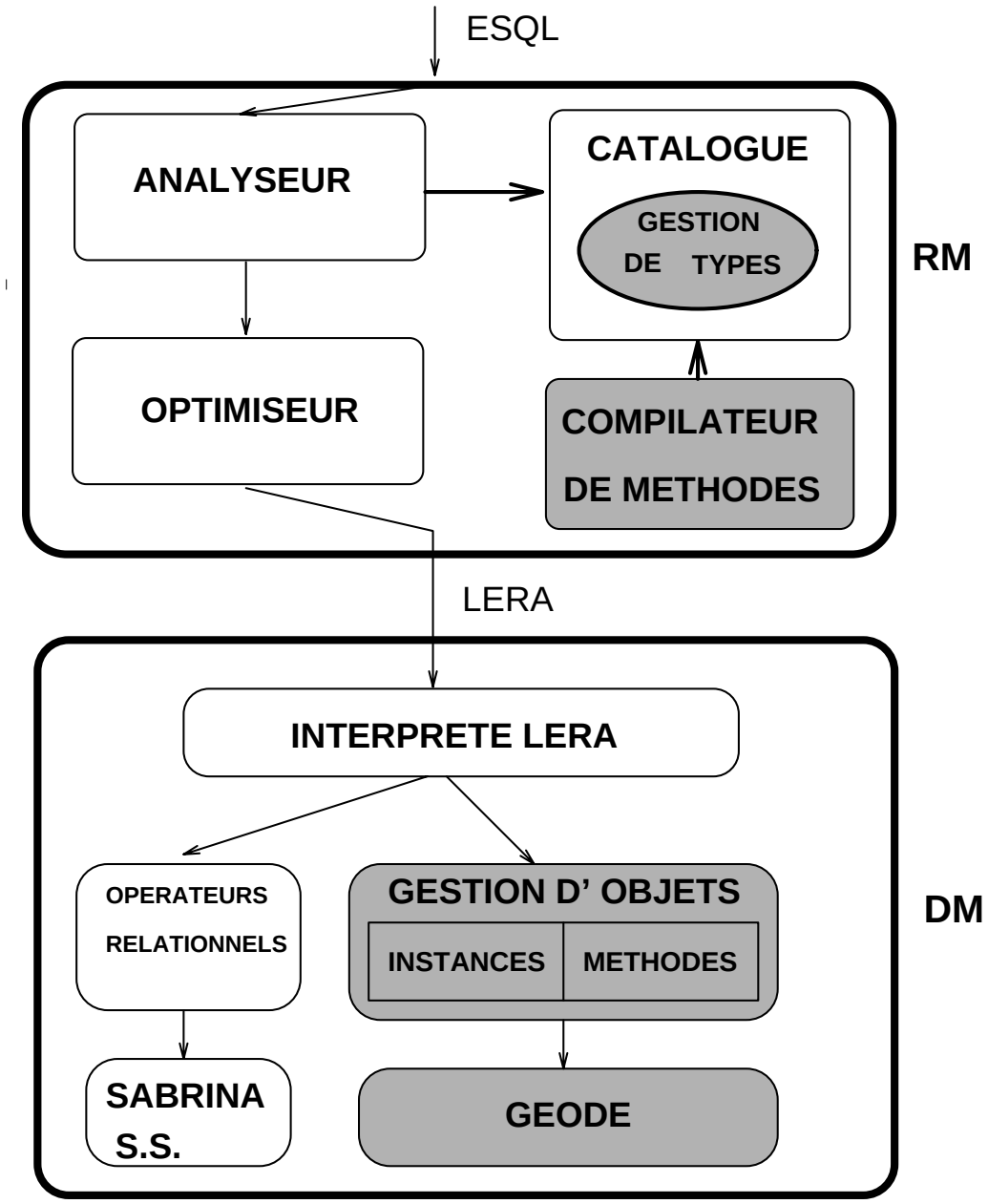

Fig. 4.5 : Architecture du prototype centralisé

IV.3.2.3 Architecture du prototype parallèle

L'architecture proposée pour le DM du prototype parallèle est décrite dans la figure Fig. 4.6. Le RM comprend cette fois l'ensemble des composants décrits en IV.2.2, avec le *gestionnaire de types* et un *compilateur de méthodes* en plus. La difficulté consiste à introduire le support d'objets dans le DM parallèle. Notre solution permet de conserver l'essentiel du code développé pour le premier prototype et constitue un couplage faible. Il s'agit de remplacer la gestion de

mémoire virtuelle de GEODE par le *gestionnaire de pages* du DM. On peut ainsi conserver notre gestionnaire d'objets et le coupler avec les autres modules du DM.

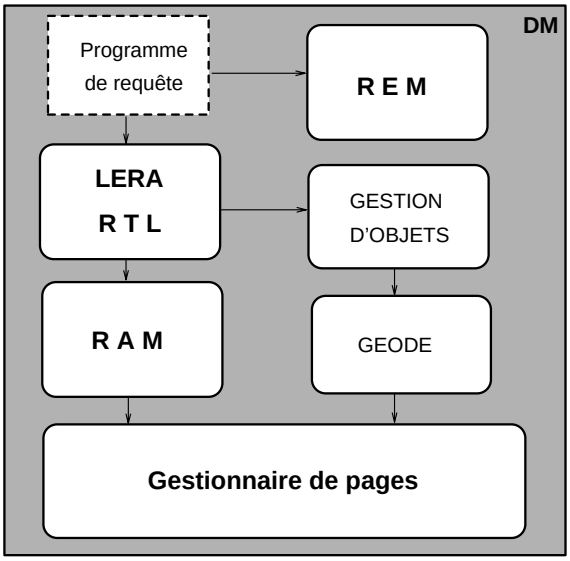

Fig. 4.6 : Le DM de DBS3 étendu

Ceci est la proposition résultant de nos discussions avec les concepteurs de DBS3 ; aucune implantation n'a été faite. Plus de détails sont donnés en VI.3. Une telle solution réduit cependant la redondance de code occasionnée par le couplage faible en mettant en commun la gestion de la mémoire persistante.

IV.4 Conclusion

Ce chapitre a montré qu'il existe plusieurs façons d'intégrer les mécanismes relationnels avec les mécanismes objets. Pour être efficacement pris en compte, le support d'objets doit être considéré à tous les niveaux du système, ce qui conduit à une intégration forte. Isoler tous les mécanismes objets dans un serveur serait inefficace, voire irréalisable.

La première étape consiste à réaliser un couplage faible qui permet de mettre en place les trois composants de base intervenant dans la réalisation d'un tel système. Ces trois composants sont le gestionnaire de types, le gestionnaire d'objets et le gestionnaire de méthodes. La réalisation de ces trois modules, leur couplage avec un SGBD relationnel existant et l'utilisation de l'analyseur/traducteur ESQL permettent d'aboutir à un premier prototype qui devient une base concrète d'investigation. Nous pouvons nous intéresser par la suite aux possibilités d'optimisation, à l'intérêt que peut avoir un stockage plus intégré, ainsi qu'aux mécanismes d'exécution dans le prototype parallèle.

Chapitre V

Gestionnaire de types

Le schéma d'une base de données ESQL est constitué de deux sortes d'entités : les types abstraits et les relations. La gestion des types abstraits s'apparente à ce que l'on trouve dans les systèmes orientés objets, tandis que la gestion des relations se fait au niveau du catalogue relationnel. Nous décrivons plus particulièrement ici le gestionnaire d'ADT qui a été intégré au catalogue. Nous n'avons pas cherché à unifier la gestion de types et avons laissé à la relation son statut particulier de type et d'instance (la relation désigne à la fois le type des n-uplets qu'elle contient et le conteneur de ces n-uplets, nous aurions pu la considérer comme un ADT construit par la composition des constructeurs *set* et *tuple*).

V.1 Définition

Le gestionnaire gère les définitions des types abstraits supportés par ESQL. Cela comprend aussi bien les types génériques (ou constructeurs), que les types définis par l'utilisateur. Les interactions ont lieu essentiellement avec l'analyseur ESQL pour les phases de contrôle de type. Les compilateurs de méthode l'utilisent aussi, lorsque les types ESQL sont importés dans les méthodes (voir chapitre VII). Dans la suite nous appelons *catalogue de types* l'ensemble des données constituant les définitions et *gestionnaire de types* le module chargé de la gestion de ces définitions.

Les fonctions de base du gestionnaire de types sont les suivantes :

- Stockage des définitions de types abstraits. Le gestionnaire de types stocke la définition structurelle d'un type abstrait (en terme de constructeurs ESQL), ainsi que la signature des méthodes. Le code des méthodes est actuellement stocké en dehors du catalogue de types ; il est cependant prévu de gérer ce code comme le reste des définitions car il peut être considéré comme faisant partie du schéma.
- Accès aux définitions de types abstraits. Une interface fonctionnelle permet d'accéder aux définitions de façon plus ou moins fine.
- Contrôle de type. Le gestionnaire de types fournit des fonctions de contrôle de conformité entre types et valeurs. Il effectue donc lui-même ces contrôles plus ou moins complexes et évite ainsi aux modules l'utilisant d'avoir à connaître ses structures internes.

- Héritage. Le gestionnaire de types maintient la hiérarchie d'héritage simple.
- Interdépendance de types. Un type abstrait peut être utilisé dans la définition d'un ou plusieurs autres types abstraits. Un tel type ne peut donc être supprimé. Le gestionnaire de types est responsable du maintien de cette cohérence.

V.2 Structures de données

Le choix de la structure de données interne du catalogue de types est important du point de vue de la vitesse d'accès d'une part et de la fiabilité d'autre part. On comprend facilement que la définition des types fasse partie du schéma d'une base de données et qu'elle subisse par conséquent des accès fréquents qui peuvent rapidement constituer un goulot d'étranglement pour le système. Il est donc important que les accès au catalogue soient le plus performant possible. D'autre part, ces données doivent être cohérentes et fiables, d'où l'intérêt de les faire gérer par le SGBD lui-même. Ces contraintes opposent les deux solutions décrites ci-après.

- La première consiste à gérer un catalogue de types complètement ad hoc, où les définitions sont représentées par des structures arborescentes beaucoup plus appropriées aux accès efficaces. Les définitions sont stockées en un seul morceau et ne sont pas éclatées sur plusieurs relations. Les algorithmes complexes de contrôle de type peuvent travailler beaucoup plus rapidement sur ces structures et surtout il n'est pas nécessaire d'opérer de coûteuses conversions de format pour les appliquer. En outre, un mécanisme de verrouillage adapté pourra être développé.
- La deuxième solution consiste, elle, à considérer le catalogue de types comme une **métabase** . Les définitions sont alors stockées dans la base elle-même, subissent les contrôles d'accès concurrents, les actions de journalisation, et sont donc fiables. Deux possibilités se présentent alors : soit le catalogue est décrit au moyen d'une structure relationnelle pure, soit il est décrit par des relations et des ADT. La première possibilité présente plusieurs inconvénients : les structures relationnelles ne sont pas adaptées au stockage de telles définitions, et les manipulations sont rendues lourdes par la décomposition des définitions dans les relations (de nombreuses jointures sont nécessaires). Nous verrons plus loin pourquoi nous n'avons pas utilisé les ADT pour représenter ces définitions.

La solution adoptée est en fait une combinaison pragmatique des deux précédentes. Elle combine les avantages de la métabase et des structures de données arborescentes. Les informations de base, souvent consultées, sont stockées dans une relation, tandis que la définition complète des types se trouve elle dans une représentation arborescente stockée sous forme de chaîne dans la relation. L'arbre

de définition d'un type, dérivé directement de l'arbre d'analyse syntaxique, comprend deux sous-arbres : le premier décrit l'ensemble des signatures des méthodes, tandis que le second représente la structure de données. Dans cette structure, chaque noeud correspond à un constructeur, et les feuilles à des types de base ou à des références à d'autres types définis par l'utilisateur. La relation contient l'identificateur et le nom du type, son éventuel supertype, son compteur d'utilisation (voir la section V.4) et l'arbre de définition, .

Table des types

ID_TYPE	NOM_TYPE	SUP_TYPE	CUB	DEFINITION
10	Personne		0	
11	Acteur	10	2	
12	Acteurs		1	

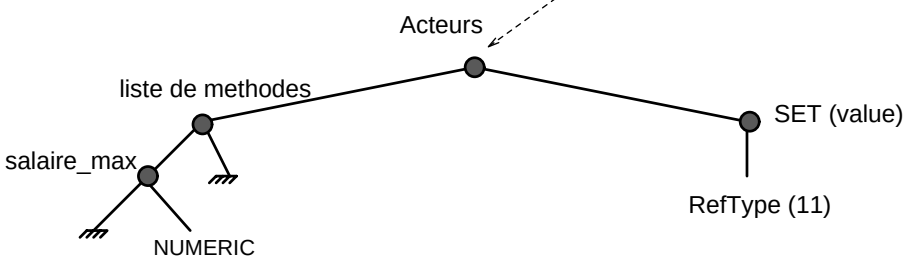

Fig. 5.1 : Représentation des ADTs

La figure Fig. 5.1 montre la structure d'un tel catalogue de types, pour une partie du schéma défini comme exemple dans le chapitre III (section III.3.1) et dont les définitions sont rappelées ci-dessous.

```
CREATE TYPE Personne AS OBJECT TUPLE OF (
    nom CHAR(80), prénom CHAR(25));
CREATE TYPE Acteur SUBTYPE OF Personne WITH (
    salaire NUMERIC(8));
CREATE TYPE Acteurs AS SET OF (Acteur);
```

Les compteurs d'utilisation (CUB : Counter Used By) indiquent le nombre de fois qu'un type a été utilisé dans d'autres définitions ; les valeurs données sur cette figure sont expliquées en V.4. *Acteur* est un sous-type de *Personne.* L'arbre de définition du type *Acteurs* indique en partie droite qu'il s'agit d'un ensemble d'acteurs (nœud SET), non partageable (*value*), et en partie gauche qu'il possède une méthode, *salaire_max*, dont la signature est la suivante :

```
salaire_max (Acteurs) -> NUMERIC
```

Cette méthode a été définie par l'ordre ESQL

```
CREATE FUNCTION salaire_max (Acteurs)
    RETURNS NUMERIC(8) LANGUAGE C;
```

On constate qu'il n'y a pas de paramètre autre que le type sur lequel est définie la méthode (feuille de gauche du nœud de la méthode à NIL).

Cette représentation arborescente des types abstraits aurait pu être faite au moyen des objets complexes ESQL eux-mêmes. Il aurait pour cela fallu définir un type abstrait pour décrire un arbre de définition de type. Nous aurions alors une véritable métabase ESQL. Cependant plusieurs raisons nous ont fait abandonner cette solution. Nous avons choisi de récupérer la structure arborescente (représentant la définition d'un type) issue directement de l'analyse syntaxique et sémantique. Cette structure est manipulable telle quelle grâce à l'utilisation dans tous les modules du projet EDS d'un gestionnaire d'arborescences commun, EMIR [43]. Ce dernier fournit un modèle de représentation des arbres, ainsi qu'un ensemble de fonctions pour les manipuler. La mise en œuvre est ainsi plus rapide. De plus, cela évite lors de la création des types abstraits, de traduire la structure EMIR issue du compilateur dans une structure ESQL. Finalement le catalogue se trouve stocké dans la base de données. Fournir une véritable métabase n'est pas primordial pour nous, car l'accès au catalogue est réalisé à travers une interface fonctionnelle et un accès à travers ESQL ne nous intéresse pas.

V.3 Stockage et accès

L'insertion d'une nouvelle information dans le catalogue peut prendre deux formes : création d'un nouveau type ou ajout d'une nouvelle méthode. Elle se déroule en deux phases : la première est une phase de contrôle de la nouvelle définition qui vérifie sa validité et complète l'arbre de description en ajoutant notamment les identificateurs des types utilisés ; la deuxième phase est la création proprement dite de l'entité dans le catalogue. L'interface du gestionnaire de types comprend donc les quatre fonctions suivantes :

1. **Adt_check_type()** vérifie que le type à créer n'existe pas déjà, que son éventuel supertype et tous les types utilisés existent. L'arbre de définition du type (arbre EMIR issu de l'analyseur syntaxique et sémantique) est complété avec les identificateurs des types utilisés.
2. **Adt_insert_type()** insère la définition du type dans le catalogue. Les compteurs CUB sont aussi mis à jour (voir section V.4).
3. **Adt_check_method()** vérifie que tous les types utilisés dans la signature de la méthode existent et insère leurs identificateurs dans l'arbre.
4. **Adt_insert_method()** insère l'arbre de la signature de la méthode dans l'arbre de définition du type sur lequel elle est définie. Un identificateur unique de méthode est généré et stocké dans le catalogue ; il sera utilisé par le

mécanisme d'exécution de méthodes qui est présenté dans le chapitre VII. Les fonctions de contrôle (*check*, 1 et 3) sont appelées lors de l'analyse sémantique des ordres LDD d'ESQL, tandis que les fonctions de création (*insert*, 2 et 4) sont appelées lors de l'exécution des ordres LDD.

Un certain nombre de fonctions d'interrogation du gestionnaire de types sont disponibles, quelques exemples sont donnés ci-dessous :

- **Adt_exist_type()** confirme l'existence d'un type donné.
- **Adt_is_a_method()** indique si une méthode est définie sur un type donné et retourne sa signature. C'est une fonction assez complexe qui parcourt la hiérarchie des types utilisateurs, ainsi que celle des types génériques afin de retrouver une méthode éventuellement héritée.
- **Adt_is_object()** indique si un type est *object* ou *value*.
- **Adt_is_a_column()** vérifie qu'un attribut est bien défini sur un type dont la structure de données est un n-uplet.
- **Adt_is_a_subtype()** indique si deux types sont liés par une relation de sous-typage.
- **Adt_method_returns()** donne le type retourné par une méthode.

Il existe enfin des fonctions de suppression de définition. Comme pour la création, deux phases sont nécessaires, une première pour vérifier la validité de la suppression, la deuxième pour la suppression même. La phase de vérification, effectuée lors de l'analyse sémantique vérifie que l'entité détruite ne fait partie d'aucune autre définition du catalogue (voir section V.4). Si un type ne fait plus partie d'aucune autre définition (de type ou de relation), cela veut dire qu'il ne peut plus exister d'instances de ce type et que l'on peut le détruire. On retrouve donc quatre fonctions : **Adt_check_drop()** et **Adt_remove_type()**, pour les types et **Adt_check_dm()** et **Adt_remove_method()** pour les méthodes.

V.4 Interdépendance de types

Un type T1 dépend d'un type T2, si T1 est utilisé dans la définition de T2, soit dans la structure de données, soit dans une signature de méthode.

```
CREATE TYPE T2 AS TUPLE OF (a1 CHAR(20), a2 T1)
```

Le type T1 ne doit pas être détruit ou modifié[1] tant qu'il dépend d'un autre type. La gestion d'une telle cohérence est faite au moyen d'un compteur de références (appelé aussi compteur d'utilisation), indiquant pour chaque type, le nombre de types dont il est dépendant. Ce compteur est stocké dans le catalogue de types (il est appelé CUB dans V.2), il est consulté et mis à jour à chaque création, suppression ou modification de type.

1 voir cependant la section V.7 pour les modifications

Dans la figure Fig. 5.1, le compteur indique qu'*Acteur* a été utilisé deux fois (une fois dans la table des acteurs et une fois dans le type *Acteurs*) et que le type *Acteurs* n'a été utilisé qu'une fois (dans la table des films). Par contre le compteur de *Personne* est à zéro, bien que celui-ci soit le supertype d'*Acteur*. En fait nous ne prenons pas en compte au niveau du compteur de références les relations d'héritage qui sont déjà traitées par ailleurs (un attribut de la table des types est dédié à cela, SUP_TYPE sur la figure Fig. 5.1).

Un type ne pourra être détruit que si son compteur de références est à zéro. Comme les relations d'héritage sont gérées à part, il faut aussi vérifier qu'il n'est pas utilisé comme supertype. Nous pourrions facilement prendre en compte les dépendances liées à l'héritage au niveau du compteur de références. Cela simplifierait les contrôles lors des destructions de type, et ferait en fait de la propriété "CUB = 0" une condition nécessaire et suffisante pour détruire un type[2]. En effet, si le compteur d'un type T passe à zéro, cela veut dire :

1. que T n'est plus utilisé dans une définition de colonne de relation, et donc qu'il n'existe aucune instance de T dans les tables relationnelles ;
2. que T n'est plus utilisé dans la définition d'un autre type, et donc qu'il n'existe aucune instance de T qui soit un composant d'un autre objet complexe.

Il n'existe donc plus aucune instance de T, qui peut être détruit dès que son compteur passe à zéro. On peut alors engager un processus récursif de décrémentation des compteurs des types utilisés dans T (entraînant leur éventuelle destruction).

Nous ne sommes cependant pas favorables à un processus de suppression de types qui n'informe pas l'utilisateur. Celui-ci dispose d'ordres ESQL pour créer et détruire des types de façon explicite (*CREATE TYPE* et *DROP TYPE*). L'utilisateur connaît tous les types qu'il a créés dans un schéma, et même si l'un d'eux n'est plus utilisé à un instant donné, il peut toujours s'en servir plus tard pour créer de nouveaux types ou de nouvelles tables. C'est pourquoi nous n'avons pas mis en œuvre une telle utilisation des compteurs.

Il est important de noter que dans notre cas, un type peut être dépendant d'une table relationnelle, lorsqu'il est utilisé dans la définition d'une de ses colonnes. On s'aperçoit qu'il existe une étroite corrélation entre le catalogue de types et le catalogue relationnel. Les définitions des tables constituent un ensemble de types spéciaux et la création ou suppression d'une table nécessite de mettre à jour les compteurs de références des types utilisés dans les définitions de colonne. Le fait d'inclure le catalogue de types dans le catalogue relationnel a grandement facilité la mise en œuvre de la gestion de cette cohérence entre les deux catalogues. Il reste maintenant à développer des mécanismes particuliers pour supporter les définitions de type récursives simples et croisées. En effet, si l'on considère maintenant que le

2 à condition qu'il n'y ait pas de types qui se référencent mutuellement (voir plus loin le problème des cycles).

type T1 a pu être défini comme ci-dessous, T1 et T2, même s'ils ne sont plus utilisés par ailleurs, maintiendront leur compteurs à un, par utilisation mutuelle :

```
CREATE TYPE T1 AS SET OF (T2);
```

Le traitement des compteurs de références se fait au niveau des fonctions de création et de suppression du gestionnaire de types :

1. Création d'un type. Dans la fonction **Adt_insert_type(T)**, on effectue l'opération suivante : *pour chaque type Ti utilisé dans la définition de T, incrémenter le compteur CUB de Ti*.
2. Suppression d'un type. On vérifie dans la fonction **Adt_check_drop(T)** que le compteur CUB de T est à zéro et que T n'est pas un supertype, sinon on signale que le type ne peut être détruit. Dans la fonction **Adt_remove_type(T)** , on effectue l'opération suivante : *pour chaque type Ti utilisé dans la définition de T, décrémenter le compteur CUB de Ti*.
3. Création de méthode. Dans la fonction **Adt_insert_method(T,M)** , on effectue l'opération suivante : *pour chaque type Ti utilisé dans la définition de la méthode M, qui n'est pas déjà dans la définition de T, incrémenter le compteur CUB de Ti.*
4. Suppression de méthode. Dans la fonction **Adt_remove_method(T,M)** , on effectue l'opération suivante : *pour chaque type Ti utilisé dans la définition de M et qui ne figure plus dans la définition de T après suppression de M, décrémenter le compteur CUB de Ti.*

A titre d'exemple nous allons examiner les actions du gestionnaire de types lors de la création du type *Acteurs.*

```
CREATE TYPE Acteurs AS SET OF (Acteur);
```

Lors de la phase d'analyse sémantique ESQL, la fonction **Adt_check_type()** est appelée avec comme paramètre l'arbre de définition de ce type *Acteurs.* Cette fonction vérifie l'existence du type *Acteur* utilisé, récupère son identificateur (11), et l'ajoute dans l'arbre. La fonction **Adt_insert_type()** appliquée sur ce même arbre, insère la définition dans le catalogue, et incrémente le compteur CUB du type *Acteur*, qui passe de 1 à 2 (voir figure Fig. 5.1).

V.5 Contrôle de type

Les fonctionnalités du gestionnaire de types pour effectuer ou faciliter les opérations de contrôle de type sont décrites brièvement ici. Nous distinguons deux sortes de contrôles effectués par les compilateurs qui s'appuient alors sur les fonctions du gestionnaire. La première consiste à vérifier qu'une constante complexe est conforme à un type. La deuxième vérifie qu'un appel de méthode est bien conforme à sa signature.

La fonction **Adt_is_conform(T,C)** permet de vérifier qu'une constante complexe C, décrite dans un format standard, est conforme à un type T. Ce format

standard est pour l'instant la représentation arborescente (EMIR) issue de l'analyse syntaxique d'une constante ; il est prévu au niveau du projet EDS de définir des formats d'échange de valeurs complexes qui pourraient être utilisés à cet effet. Cette fonction est réalisée dans le gestionnaire par un parcours récursif et parallèle de l'arbre de définition de la structure de données du type et de la constante. Fournir une telle fonction au niveau de l'interface du gestionnaire de types évite aux compilateurs l'utilisant de connaître ses structures de données internes, ils n'ont qu'à connaître le format standard de représentation d'une valeur complexe. Cette fonction est par exemple appelé par l'analyseur sémantique ESQL lorsqu'il traite la requête suivante :

```
UPDATE FILM
SET acteurs = set (tuple ('Newman', 'Paul', 3000),
    tuple ('Cooper', 'Gary', 3500));
WHERE ...
```

La table FILM est celle définie en III.3.1, l'attribut *acteurs* est de type *Acteurs*. On note ici le format de représentation d'une constante complexe en ESQL. La fonction **Adt_is_conform()** est donc appelée avec en paramètres le type *Acteurs* et cette constante complexe.

Nous avons aussi développé un mécanisme d'inférence de type qui permet de déterminer la méthode à appeler en fonction du contexte d'invocation. Il est mis en œuvre dans la fonction **Adt_is_a_method()** . Cette dernière est utilisée lors du contrôle de validité d'un appel de méthode. Elle permet à partir d'un identificateur de type et d'un nom de méthode, de retrouver la signature de cette méthode et son identificateur, à condition qu'elle soit définie sur le type. Cette fonction sera décrite plus en détails dans la section V.6.

Ces fonctions ne s'appuient sur aucune règle de conformité autre que celle permettant de dire qu'une valeur *est* de tel type. En effet, les tables relationnelles étant strictement typées, il est impossible de trouver dans une colonne des attributs de type différent de celui de la colonne.

V.6 Héritage

Pour des raisons de simplicité de réalisation, seul l'héritage simple est supporté dans ESQL. De plus, seule la structure de données *n-uplet* peut être spécialisée par ajout d'attribut dans une définition de sous-type. Il est prévu de supporter ultérieurement l'héritage tel qu'il est décrit par Cardelli [17]. Nous nous intéressons plus particulièrement à l'héritage du comportement.

Deux solutions se présentent pour gérer l'héritage des méthodes et de la structure de données du super-type. Ces deux solutions concernent la façon de stocker les définitions et d'y accéder. Les définitions des structures de données et les signatures de méthode se trouvent dans les arbres de définition de types, qui eux-mêmes se

trouvent dans une arborescence qui matérialise les liens d'héritage. Lorsqu'un type T2, sous-type de T1 est créé, il est possible de recopier les parties de structure de données et les méthodes héritées de T1, dans la définition de T2. On peut préférer la technique qui consiste, lors de l'accès à la définition de T2, à explorer sa hiérarchie ascendante et à reconstituer ainsi sa structure de données et ses méthodes. Cette solution par reconstruction a l'inconvénient de fournir des temps d'accès plus élevés, mais présente l'avantage d'être complètement insensible aux modifications de types et d'éviter la duplication d'informations. En effet, dans la solution avec expansion, il est nécessaire lors de la modification d'un type, de mettre à jour toute sa hiérarchie descendante.

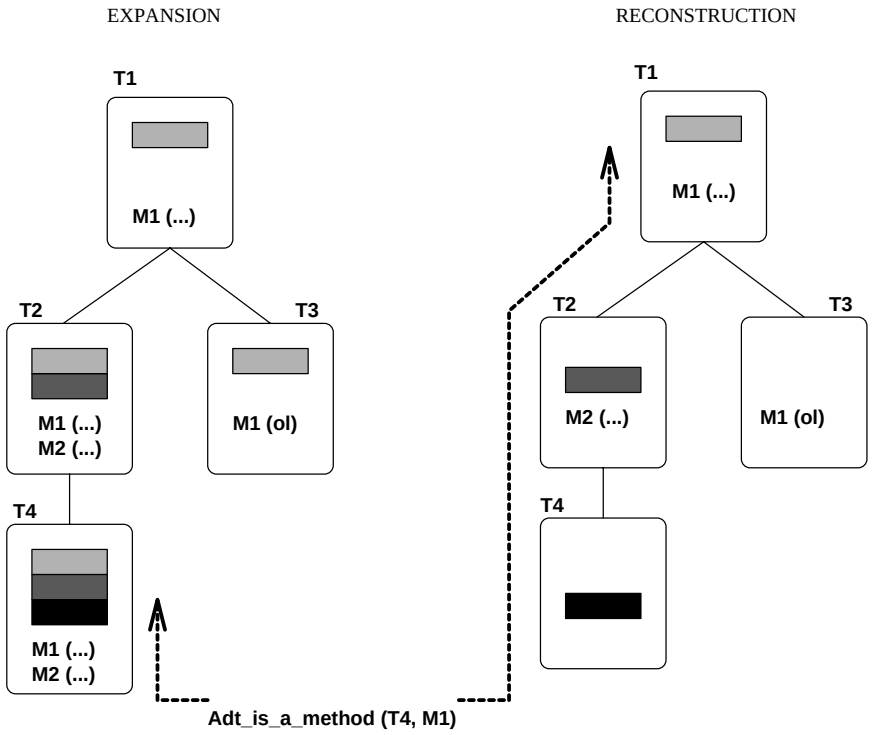

Fig. 5.2 : Gestion de l'héritage

La figure Fig. 5.2 illustre les deux mécanismes d'héritage cités. Dans la première approche, avec expansion, les structures de données (représentées par les rectangles de couleur) héritées sont dupliquées, ainsi que les signatures de méthodes. L'accès à une définition de type est immédiate, comme le montre l'accès à la signature de la méthode M1 de T4, qui est en fait héritée de T2, puis de T1. Dans l'approche par reconstruction, l'accès à cette méthode nécessite de remonter la hiérarchie des types, jusqu'à trouver celui dans lequel la méthode est définie. Cependant aucune information n'est dupliquée. Nous supposons que la méthode M1 a été redéfinie dans T3.

Nous avons adopté la deuxième solution (reconstruction), plus appropriée au mécanisme d'héritage des méthodes génériques décrit ci-dessous.

Trois sortes de méthodes sont en fait attachées à un type :

- les méthodes définies sur le type même,
- les méthodes héritées de la hiérarchie de super-types,
- les méthodes génériques du constructeur utilisé au premier niveau de la structure de données du type.

Les premières sont trouvées sur la définition du type même et les deuxièmes dans la hiérarchie ascendante. Les dernières sont enfin extraites de la hiérarchie des types génériques (constructeurs), eux aussi stockés dans le catalogue des types. Les méthodes génériques sont gérées comme les autres méthodes. Seuls les paramètres génériques ont une représentation spéciale, qui permet de les repérer et de spécifier le type qui les instancie lors de l'extraction de la signature pour le compte d'un type particulier. Supposons que le type T1 soit défini de la façon suivante :

```
CREATE TYPE T1 AS SET OF (T2);
```

La fonction **Adt_is_a_method()** appelée sur le type T1 avec le nom de méthode *insert*, qui est la méthode d'insertion d'un élément dans un ensemble, définie sur le type générique SET, retournera la signature suivante :

```
insert (T1, T2) -> T1
```

La signature de la méthode générique est spécialisée avant d'être retournée. Elle est en fait stockée dans le catalogue sous la forme :

```
insert (SET, <param. de généricité>) -> SET
```

Ce mécanisme de recherche de méthode est mis en œuvre dans la fonction **Adt_is_a_method()** .

V.7 Modification de schéma

Nous appelons ici schéma l'ensemble des types définis pour une application. La possibilité de modifier la définition de types existants engendre des problèmes très délicats à résoudre. Cela concerne notamment les instances existantes d'un type que l'on modifie, et les programmes utilisant les anciennes définitions qui peuvent ne plus être valides sur les nouvelles. Un système relationnel comme ORACLE permet d'ajouter ou modifier (agrandir) une colonne d'une relation. Ce type de modification n'influe pas sur les n-uplets déjà stockés dans la base[3], ni sur les requêtes déjà définies sur une relation. Les modifications que l'on peut apporter à un schéma de données orienté objet sont beaucoup plus nombreuses et complexes.

Des études ont été faites à ce sujet (pour ORION[62], O_2 [9], etc.) sur la base d'un modèle de données permettant de définir une hiérarchie de classes (l'équivalent de nos ADT). Nous donnons rapidement un aperçu de ces travaux. La

3 ORACLE ne stocke que les champs renseignés d'un n-uplet.

relation d'héritage joue un rôle important dans ces résultats. La première étape consiste à établir la liste des mises à jour que l'on peut effectuer. On peut les classer en trois catégories : changements de la structure de données (ajout/suppression d'un attribut, renommage, etc.), modifications au niveau des méthodes (ajout/suppression, changement de signature et/ou de code, etc.) et finalement changements dans le graphe d'héritage (ajout/suppression d'une classe, etc.). Les propriétés que doit vérifier un schéma pour être cohérent sont établies. On détermine ensuite, pour chaque type de mise à jour, les vérifications à faire pour tester sa validité. Par exemple, lors de la suppression d'un attribut, il faut vérifier que celui-ci n'est pas utilisé par ailleurs dans une méthode. Les vérifications à effectuer sont nombreuses et les cas possibles d'invalidation sont complexes et parfois difficiles à déceler. Cela nécessite notamment des informations supplémentaires sur le schéma de données qui ne sont pas toujours fournies par le catalogue (informations du style "qui utilise quoi"). Lorsqu'une mise à jour d'une classe a lieu, il faut éventuellement envisager une action sur les instances existantes de cette classe (actualisation immédiate de toutes les instances de la classe, ou différée au fur et à mesure de leur utilisation).

Nous avons traité de manière simple et restreinte les modifications d'ADT d'ESQL. En fait nous autorisons l'ajout et la suppression d'éléments de définition, mais nous ne garantissons pas la cohérence des programmes existant lors des suppressions. Nous ne contrôlons pas non plus les conséquences indirectes d'un ajout liées à l'héritage (voir plus loin).

Il est possible d'ajouter une nouvelle méthode sur une définition de type (ordre ESQL *ALTER TYPE T ADD FUNCTION*...). Cette opération n'altère en rien les programmes déjà existants qui manipulent le type concerné. Il est aussi possible de supprimer une méthode (ordre ESQL *ALTER TYPE T DROP FUNCTION*...). Dans ce dernier cas, l'exécution d'un ordre ESQL ou d'une méthode utilisant la méthode supprimée génèrera une erreur système. Les fonctions du catalogue assurant l'ajout et la suppression de méthode ont été décrites dans les sections précédentes, il s'agit de **Adt_check_method()** , **Adt_insert_method()** , **Adt_check_dm()** et **Adt_remove_method()** . La modification du corps d'une méthode n'a pas été traité, des procédures de recompilation peuvent être nécessaires suivant le mode de compilation et d'exécution choisi (voir chapitre VII).

Concernant les modifications de structure de données d'un type, seuls l'ajout et la suppression de colonne sur un type *n-uplet* (construit avec l'ADT générique *tuple*) sont autorisés. De même que pour les suppressions de méthodes, nous n'assurons pas la validité des programmes et requêtes existant en cas de suppression de colonne. Deux fonctions du catalogue sont disponibles à cet effet :

- **Adt_add_column()** ajoute une colonne sur un type n-uplet. Les compteurs d'utilisation sont mis à jour de la façon suivante : *pour chaque type Ti*

utilisé dans la définition de la colonne, qui n'est pas déjà dans la définition de T, incrémenter le compteur CUB de Ti.

- **Adt_drop_column()** supprime une colonne d'un type n-uplet. Les compteurs d'utilisation sont mis à jour de la façon suivante : *pour chaque type Ti utilisé dans la définition de la colonne et qui ne figure plus dans la définition de T après suppression de de la colonne, décrémenter le compteur CUB de Ti.*

Il est finalement possible de créer (*CREATE TYPE*) ou supprimer (*DROP TYPE*) un type, ce qui peut modifier le graphe d'héritage. Nous n'autorisons pas la suppression d'un type utilisé par ailleurs, comme cela a été vu en V.4. Ces opérations peuvent cependant générer des incohérences dues aux règles d'héritage. Nous abordons ce problème ci-dessous.

Les mises à jour de schéma peuvent avoir des conséquences indirectes par le biais des relations d'héritage. Nous illustrons ce phénomène en prenant l'exemple de la suppression et de l'ajout de méthode. Supposons qu'un type T définisse deux méthodes `m1` et `m2` et que T' soit un sous-type de T redéfinissant m2.

Partout où `m1` est utilisée sur un objet de type T', c'est la méthode `m1` définie sur T et héritée par T' qui est effectivement utilisée. Si l'on ajoute une méthode de même nom `m1` sur T', celle-ci doit être une redéfinition de celle de T et avoir une signature conforme (une telle règle de conformité est définie pour GUIDE et O2, elle reste à définir pour ESQL). C'est donc cette méthode définie sur T' qui doit désormais être utilisée lorsqu'on applique `m1` sur un objet de type T'. Cela peut avoir des conséquences sur le code existant, là où l'utilisateur pensait utiliser la méthode héritée de T : ses effets peuvent être différents, il faut éventuellement recompiler certains modules si l'on a adopté une approche de compilation avec liaison statique (voir chapitre VII).

Supposons maintenant la méthode `m2` soit supprimée : partout où elle était utilisée sur un objet de type T', c'est celle de T, héritée, qui sera désormais exécutée. Cela peut générer des incohérences qui peuvent aboutir à des erreurs d'exécution.

Nous ne gérons pas ce genre de problème, qui nécessite d'analyser le code des méthodes et de conserver des informations de dépendance complexes entre les éléments d'un schéma.

V.8 Conclusion

Nous avons décrit dans ce chapitre la mise en œuvre d'un gestionnaire de types pour notre prototype de SGBD relationnel étendu. Bien que de nombreux choix aient été pris pour des raisons de rapidité de prototypage, nous avons dégagé un certain nombre de possibilités de réalisation pour lesquelles des critères de choix ont été établis.

- La structure de représentation des données joue un rôle important au niveau des performances. Le choix doit ici s'effectuer entre une représentation spécifique et une métabase. La deuxième solution assure à peu de frais de développement la fiabilité des données, mais il faut cependant s'assurer que le mécanisme de contrôle d'accès concurrents offert par le système est adaptable à la spécificité du type d'accès au catalogue.
- Le niveau de l'interface offerte peut être plus ou moins élevé suivant l'utilisation souhaitée du gestionnaire. S'il s'agit d'une métabase, les structures de données sont plus facilement manipulables et l'on peut se contenter d'une interface de bas niveau, en laissant le soin au développeur du module utilisant le gestionnaire d'écrire les traitements de ses données. Dans le cas où les structures de données sont spécifiques, et où le gestionnaire est destiné à être utilisé par plusieurs modules différents (comme c'est le cas ici avec nos différents compilateurs), il est préférable d'offrir une interface de plus haut niveau. Les traitements sont alors internes au gestionnaire, ce qui évite de les dupliquer ; les modules clients n'ont pas à connaître les structures de données manipulées par le gestionnaire.
- Le problème de l'interdépendance de types a été résolu au moyen de "compteurs d'utilisation". La solution implémentée est la plus simple possible, il est par exemple impossible de déterminer quels sont les types qui utilisent un type donné. Cela aurait pu être fait en maintenant en plus ou à la place du compteur, le graphe de dépendances. Cela aurait de plus permis de résoudre plus aisément le problème des définitions récursives qui peuvent empêcher à jamais le retour à zéro d'un compteur.
- Le modèle d'héritage traité est assez simple. Il resterait à considérer les problèmes engendrés par l'héritage multiple, ainsi que l'évolution des types. Le choix entre une gestion des définitions par expansion ou par reconstruction dépend justement de la capacité d'évolution du schéma. Pour un schéma évoluant peu, une gestion par expansion peut s'avérer suffisante et se montrera plus efficace en accès.
- La modification de type n'a pas été traitée en détail. Il est seulement possible d'ajouter et supprimer des méthodes ainsi que de modifier la structure de données d'un type n-uplet par ajout ou suppression de colonne.

Il est évident qu'un tel gestionnaire de types est très dépendant du modèle de données auquel il s'applique, mais il faut cependant noter que les techniques d'implantation sont très proches de celles utilisées dans d'autres systèmes.

Chapitre VI

Stockage d'objets

Comme cela a été décidé dans le chapitre IV, la persistance des instances de types abstraits est assurée par un gestionnaire de stockage d'objets. Nous décrivons dans ce chapitre la mise en œuvre d'un tel module.

VI.1 Le problème du stockage dans sa généralité

Cette section se propose de définir les hypothèses et choix de base qui nous permettront de spécifier le gestionnaire de stockage d'objets du SGBD relationnel étendu. Nous nous appuyons sur l'étude faite sur des gestionnaires existants, afin d'une part d'en tirer les concepts clés qui peuvent guider nos choix et d'autre part d'évaluer la possibilité d'utiliser un gestionnaire existant.

On trouve actuellement un certain nombre de produits et prototypes appelés communément *Gestionnaires d'Objets*. Ils diffèrent en premier lieu par leur niveau de fonctionnalité et ensuite par leurs techniques de mise en œuvre. Ces deux aspects sont développés dans les sections VI.1.1 et VI.1.2. Les choix concernant ces deux points sont précisés dans la section VI.1.3. Pour finir nous abordons les implications des hypothèses "grande mémoire" et "parallélisme".

VI.1.1 Fonctions d'un gestionnaire de stockage

Certains gestionnaires d'objets, comme celui d'O2, fournissent non seulement la fonction de stockage, mais aussi le support d'exécution pour le modèle supporté et les langages associés. Un tel gestionnaire fournit la gestion de l'espace de travail, les mécanismes d'exécution et de contrôle de type, la gestion des transactions, de la concurrence et de la reprise, le support des objets complexes, des index, et finalement la gestion des disques.

Il intègre en fait une grande partie de la sémantique du modèle de données supporté et constitue pratiquement un noyau de système complet. Nous nous intéresserons en fait à des gestionnaires d'objets n'assurant que le niveau stockage, sachant que le module de gestion d'objets ne constitue pour nous qu'une partie du noyau et que la partie exécution est traitée ailleurs.

Il est important pour notre cas de bien déterminer les fonctions que le module de gestion d'objets doit assurer, sachant que nous traitons à part la gestion de types et la

gestion de méthodes. Les fonctions généralement fournies par un gestionnaire de stockage d'objets sont les suivantes :

1. Gestion de types
2. Gestion d'objets complexes
3. Identité d'objet
4. Méthodes d'accès
5. Contrôle des accès concurrents
6. Reprise après panne
7. Gestion de l'allocation sur disque
8. Gestion de tampons ("buffers")
9. Gestion de versions

Le niveau de fonction et la généricité d'un gestionnaire d'objets dépendent en fait de la part de sémantique du système supporté qui est intégrée. Certains gestionnaires se contentent de gérer la persistance d'objets qui ne sont pour eux que des chaînes d'octets sans aucune sémantique associée. Sur de tels gestionnaires, les fonctions 1 et 2 sont absentes, tandis que les fonctions 4 et 9 ne peuvent apparaître que sous des formes très primitives. Par contre, il est possible de les utiliser pour supporter différents types de système, tout en sachant qu'il sera difficile d'obtenir une grande efficacité. Les fonctions de recherche et d'indexation sont difficilement implantables sur des tels gestionnaires d'objets, car ils n'offrent aucune visibilité sur le contenu des objets. Les gestionnaires qui proposent la gestion de types sont dédiés au système qu'ils supportent et ne présentent alors aucune généricité. Entre les deux, nous trouvons des gestionnaires comme GEODE qui gèrent des objets complexes. Cela veut dire que les objets stockés possèdent une structure, obtenue par combinaison de constructeurs, et qu'ils sont manipulables par le biais des opérateurs associés à ces constructeurs. Tout en restant des systèmes génériques, il supportent la sémantique structurelle des objets et permettent de les gérer plus efficacement. C'est vers de tels gestionnaires que nous allons nous orienter, car ils s'accordent tout à fait au besoin de notre prototype, qui nécessite le stockage d'objets structurés.

VI.1.2 Architecture d'un gestionnaire de stockage

Indépendamment du niveau de fonctions du gestionnaire de stockage, l'architecture utilisée pour assurer la persistance est un choix de conception important. Les architectures possibles sont présentées ici, afin de comprendre parfaitement les choix effectués pour les prototypes, présentés dans la section suivante. On distingue deux techniques de mise en œuvre d'un gestionnaire de stockage d'objets. La première se caractérise par une gestion explicite de la persistance, au moyen de fonctions de chargement et de déchargement d'objets, la seconde consiste à implanter une mémoire virtuelle persistante. On trouve dans la

littérature ces architectures sous le nom de "disk based approach" et "two levels store" pour la première et "one level store" pour la seconde.

VI.1.2.1 Stockage à deux niveaux

Dans cette première approche, le contrôle de l'allocation sur disque, du placement des objets et de la gestion des tampons reste possible. Cependant il faut utiliser des opérations explicites pour charger un objet en mémoire (dans un tampon), le stocker sur disque, le créer, le détruire, etc.. Une fois en mémoire, l'objet ne se trouve pas forcément au format du langage qui le manipule et les éventuelles références qu'il contient sont des pointeurs en mémoire secondaire. Les opérations de "déréférençage" des pointeurs persistants et de conversion de format peuvent être coûteuses.

Une technique utilisée consiste à introduire une indirection en référençant les objets à travers des descripteurs, accessibles à partir des pointeurs persistants (par une fonction de hachage), qui contiendront l'adresse mémoire d'un objet si celui-ci a été chargé. Si l'on veut optimiser les accès à travers les objets (navigation), il faut alors mettre en œuvre des techniques de "pointer swizzling", consistant à remplacer dès que possible les pointeurs persistants, par des adresses désignant l'endroit de chargement des objets référencés. C'est ce qui est fait pour le pointeur OID1 de la figure Fig. 6.1. Cette méthode est difficile à utiliser, car elle nécessite de garder la trace de toutes les références sur un objet, afin de les transformer à nouveau en pointeurs persistants au cas où l'objet est renvoyé sur disque.

Le gestionnaire de stockage du prototype O2 est à deux niveaux. Une fois en mémoire, les objets contiennent toujours des OID qui, nous l'avons vu, sont des pointeurs sur disque (identificateur d'enregistrement WISS). Les accès navigationnels se font alors à travers un table des objets chargés en mémoire : une fonction de hachage appliquée sur un OID donne une entrée dans cette table qui contient l'adresse de chargement de l'objet. Le système EXODUS, décrit dans le chapitre II, fournit un autre exemple de gestionnaire de stockage gérant une mémoire à deux niveaux ; la difficulté occasionnée par la présence de deux types de références, pointeurs persistants et pointeurs en mémoire, lors de son utilisation pour implanter le langage persistant E, est décrite en [52].

La figure Fig. 6.1 illustre une mémoire de stockage à deux niveaux. Un objet contenant des pointeurs persistants (adresses disque) sur trois composants a été chargé en mémoire avec deux de ses composants. Le pointeur persistant OID1 a été remplacé par l'adresse de chargement en mémoire de l'objet qu'il désigne (technique de "pointer swizzling").

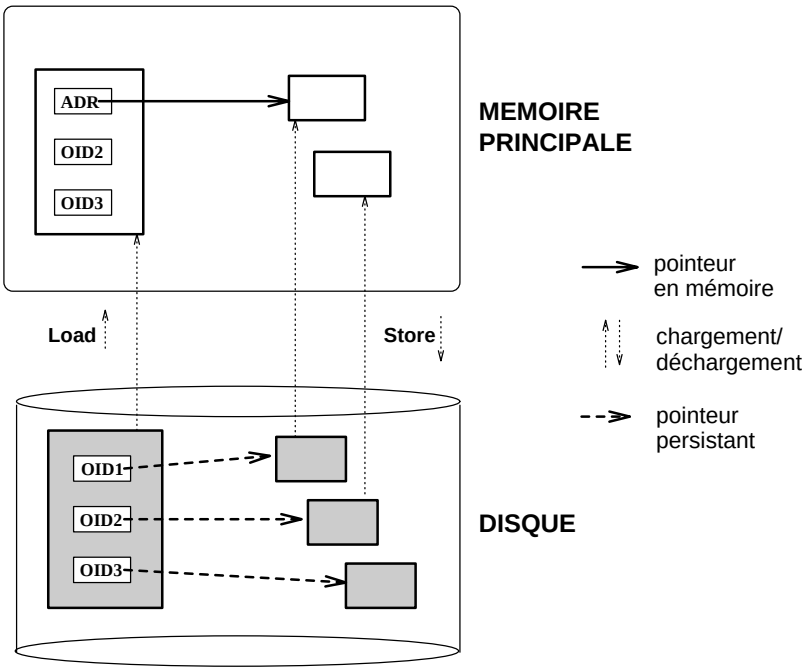

Fig. 6.1 : Stockage à deux niveaux

VI.1.2.2 Stockage à un niveau

La deuxième solution repose sur un mécanisme de gestion de mémoire virtuelle persistante. Les seules opérations consistent à coupler la base de données sur une zone de mémoire virtuelle, le mécanisme de pagination se charge par la suite d'amener les objets en mémoire (ce peut être le mécanisme de pagination du système d'exploitation ou un mécanisme ad hoc). Dans une telle approche, les formats des objets et les pointeurs sur disque peuvent être identiques aux formats et pointeurs en mémoire, si la base est toujours couplée à la même adresse virtuelle. Dans ce cas toute référence peut être une adresse virtuelle, que ce soit sur disque ou en mémoire. Les avantages d'une telle approche sont les suivants :

- La navigation à travers les objets est grandement facilitée par le fait que toute référence est une adresse virtuelle. Il n'est plus nécessaire de décoder un pointeur persistant pour obtenir une adresse de chargement en mémoire virtuelle.
- Il n'existe plus qu'un seul format de représentation des objets qui est le format en mémoire virtuelle, donc directement manipulable par les programmes compilés. Il n'est plus nécessaire d'avoir recours à de coûteuses opérations de conversion de format.
- Les copies d'objets entre les tampons de pages et les tampons d'objets sont éliminées, car les objets sont directement accessibles dans les tampons de pages (il n'y a plus de tampons d'objets).

- La persistance est assurée de façon transparente par le gestionnaire de mémoire virtuelle.

Il existe cependant un certain nombre d'inconvénients dans une telle approche :

- La taille de la base de données est limitée à celle de la mémoire virtuelle qui avec les processeurs actuels peut se révéler insuffisante (4 Giga-octets pour un processeur à 32 bits semble cependant suffisant pour beaucoup d'applications). L'arrivée des processeurs 64 bits devrait éliminer cet inconvénient.
- Le deuxième inconvénient réside dans le fait que la persistance est gérée à un très bas niveau (au niveau du paginateur), ce qui rend difficile le contrôle de l'allocation disque, la gestion des tampons, ainsi que le support des mécanismes de reprise. Ceci est dû au fait qu'il n'existe en général pas d'opération explicite d'écriture disque, ni de moyen de fixer une page dans les tampons. Il faut cependant noter que les systèmes récents comme MACH autorisent le contrôle du mécanisme de pagination et qu'il est donc désormais possible d'adapter le système à de tels besoins.

Les inconvénients les plus importants sont en fait liés à la gestion de la mémoire virtuelle :

- On ne peut pas considérer la mémoire comme un simple tas persistant. En effet, ignorer les frontières de page peut entraîner un accroissement des défauts de page (et donc des entrées/sorties), mais aussi la fragmentation de la mémoire principale. La gestion de l'allocation de mémoire est donc plus complexe qu'il ne semble.
- De même que pour les OID physiques sur disque, l'utilisation des adresses en mémoire virtuelle comme référence sur les objets pose des problèmes dans le cas où un objet doit changer de place (par exemple car sa taille augmente). On utilise ici aussi des techniques de liens de poursuite.
- La réutilisation de la place rendue disponible par les destructions d'objets est délicate.
- La gestion des objets temporaires doit se faire dans une zone non persistante de la mémoire virtuelle. Il se pause alors le problème du déplacement des objets temporaires promus persistants.
- Il faut pouvoir mettre en œuvre des techniques d'indexation et de regroupement pour optimiser le stockage des objets volumineux et des ensembles d'objets.

Un exemple de gestionnaire mémoire à un niveau, développé sur MACH, est le système Cricket [54], présenté en II.5.5. Une autre expérience menée dans le cadre du projet Bubba, prenant en compte les notions d'indexation et de "cluster", est décrite dans [23]. Le SGBD réseau SOCRATE gère lui aussi ses données dans une mémoire virtuelle persistante.

La figure Fig. 6.2 illustre un schéma de stockage à un niveau. Les OID sont désormais des adresses en mémoire virtuelle.

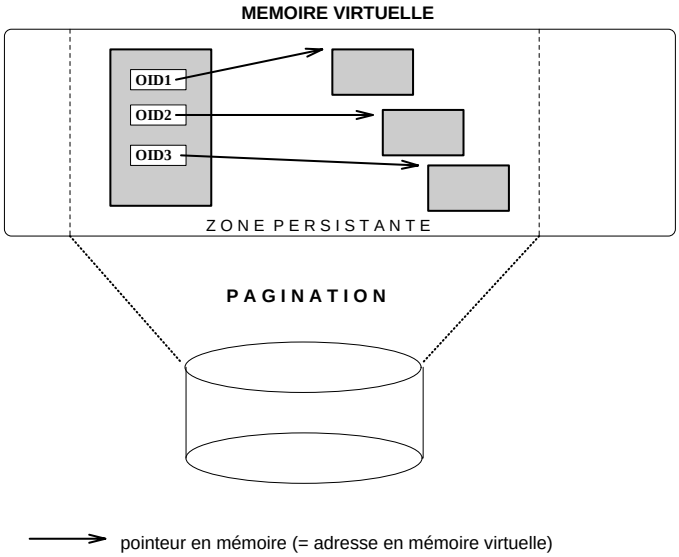

Fig. 6.2 : Stockage à un niveau

VI.1.3 Choix de conception

Une des hypothèses de travail du projet EDS concerne la taille des mémoires. Celle-ci est supposée suffisante pour que la base de données active (l'ensemble des données manipulées par toutes les transactions à un instant donné) tienne en mémoire. Nous émettons alors l'hypothèse que l'ensemble de tous les objets peut tenir en mémoire virtuelle. Ceci est une des raisons qui nous conduit à nous intéresser à des gestionnaires d'objets à un seul niveau. Nous verrons que certains gestionnaires de stockage, à priori à un niveau, résolvent le problème de la taille limitée de la mémoire virtuelle et celui du contrôle des accès disque en gérant eux-mêmes leur propre mémoire virtuelle (c'est le cas de GEODE). Cependant une telle simulation conduit à une gestion logicielle de la pagination qui introduit à nouveau le principal problème des approches à deux niveaux, à savoir le décodage des références. En effet, une adresse de la mémoire virtuelle d'un tel gestionnaire doit être décodée, au même titre qu'un OID dans un gestionnaire à deux niveaux, pour obtenir une adresse dans la mémoire virtuelle de la machine. On conserve cependant l'avantage de la transparence des opérations de chargement et de déchargement des objets au niveau de l'interface du gestionnaire d'objets.

Concernant le niveau de fonction du gestionnaire d'objets, il semble qu'un système gérant lui-même ses techniques de stockage d'objets complexes est beaucoup plus efficace et beaucoup plus simple à utiliser, qu'un système de bas niveau, assurant le stockage d'objets non structurés. Il faut en effet, dans ce dernier

cas, implanter au dessus du gestionnaire d'objets des mécanismes de gestion de listes, d'arbres, etc., qui seront plus ou moins efficaces suivant que l'on peut adapter ou non le système sous-jacent. Le coût de développement d'un tel gestionnaire d'objets est important, d'où l'intérêt que l'on peut avoir à utiliser un système existant. GEODE est un gestionnaire d'objets développé sur une approche simulant un seul niveau de stockage et présentant le niveau de fonction voulu, nous avons donc décidé de l'adopter comme base de développement de notre module de gestion d'objets.

Le dernier point concerne l'interaction des méthodes avec le gestionnaire d'objets. Les objets sont manipulés par les méthodes associées aux types dont ils sont des instances. Nous avons adopté une approche multi-langage pour l'écriture des méthodes associées aux ADTs ; c'est-à-dire que le programmeur d'ADT peut choisir différents langages d'implantation. Cela pose le problème de la multiplicité des formats de représentation possible pour les instances d'ADT (car il est possible d'utiliser les structures offertes par le langage de programmation des méthodes). La structure de données d'un ADT n'est pas spécifique à un langage de programmation particulier, mais suit les règles de construction dictées par ESQL (et ses constructeurs *tuple*, *list*, *array*, *set*). Le choix du format de représentation des objets pour les méthodes est discuté en détails en VII.2.1 lorsque nous abordons la mise en œuvre des méthodes. L'idée est de fournir une structure de données commune qui soit par la même occasion le format du gestionnaire de stockage. Les méthodes utilisent alors l'interface du gestionnaire d'objets pour manipuler les instances d'ADT, le format interne n'est pas directement connu, mais c'est celui qui est manipulé. L'adoption d'un gestionnaire d'objets à un seul niveau est alors intéressante, car il est possible de programmer et générer le code des méthodes sans se préoccuper du chargement et du déchargement des objets en mémoire. Notre approche nous évite aussi de convertir les objets dans un autre format lors de leur manipulation par une méthode. Une méthode écrite dans n'importe lequel des langages autorisés, peut manipuler les objets en mémoire, dans le format du gestionnaire d'objets, de la façon décrite en VII.2.1. La persistance est assurée par le mécanisme de pagination.

VI.1.4 Grande mémoire et parallélisme

L'hypothèse selon laquelle la base de données active, c'est-à-dire l'ensemble des données pertinentes pour un programme d'application, tient en mémoire principale conforte le choix d'un gestionnaire d'objets à un niveau de stockage. On peut en effet dans le cas d'un couplage faible envisager que la totalité des objets tient en mémoire virtuelle. Ceci nous autorise à employer pour ces derniers un véritable gestionnaire à un niveau (les objets seront toujours couplés aux mêmes adresses virtuelles), tandis que l'on utilise pour les relations un gestionnaire de stockage classique.

L'influence du parallélisme sur le gestionnaire d'objets se traduit par des problèmes de distribution de données sur les nœuds de la machine. Il sont abordés en VI.3.2.

VI.2 Gestion d'objets pour le prototype centralisé

Le gestionnaire d'objets développé pour le premier prototype du système relationnel étendu est présenté dans cette section. Le stockage, l'accès et la manipulation des instances de types abstraits constituent ce que l'on désignera par *gestion d'objets*. Le module de gestion d'objets se compose essentiellement de deux couches. La couche basse est le gestionnaire d'objets GEODE, présentant une interface d'assez haut niveau. La couche haute est une librairie de fonctions de manipulation et de stockage d'objets du niveau du modèle de données ESQL. L'interface présentée par la couche haute contient entre autres l'ensemble des fonctions génériques associées aux constructeurs ESQL (voir III.3.1). Les fonctions constituant cette interface sont appelées par l'interpréteur relationnel, lorsque celui-ci est confronté à des manipulations ou créations d'objets complexes.

VI.2.1 Le gestionnaire d'objets

Nous avons choisi d'utiliser le gestionnaire d'objets GEODE [14], décrit dans le chapitre II, pour réaliser notre premier prototype. Les raisons essentielles étaient d'une part la facilité procurée par la présence d'une interface "constructeur" d'assez haut niveau, nous permettant de réaliser rapidement l'ensemble des fonctions génériques d'ESQL, et d'autre part l'approche "gestion d'objets en mémoire".

GEODE présente la particularité de gérer sa propre mémoire virtuelle. Ceci est dû au fait que les mémoires virtuelles offertes par les systèmes d'exploitation ne conviennent pas aux besoins spécifiques d'un gestionnaire d'objets : elles sont de taille insuffisante et ne permettent pas de contrôler l'allocation de l'espace. La couche de base de GEODE met en œuvre une mémoire virtuelle persistante sur laquelle s'appuie l'ensemble du système. La conséquence est que tout accès à un objet nécessite de décoder son adresse en mémoire virtuelle (AMV = No segment, No page, No d'objet dans la page), afin de la transformer en pointeur C (adresse en mémoire principale, en fait mémoire virtuelle UNIX). Ceci est coûteux par rapport à une véritable approche "one level store", où le mécanisme de décodage d'adresse virtuelle est câblée dans le système.

GEODE présente deux interfaces. La première, dite "segment", permet de gérer des objets de type chaîne d'octets, dans des espaces logiques appelés segments. Ces segments permettent de regrouper des objets fréquemment accédés ensemble. Nous utilisons notamment cette interface lors de la mise au point du prototype pour lister l'état des segments (la liste des objets qu'ils contiennent). La deuxième interface,

dite "constructeur", permet, elle, de gérer des objets construits à l'aide de quatre constructeurs : *n-uplet*, *tableau*, *ensemble*, *arbre*. Aucune gestion de types n'est cependant assurée, elle est entièrement à la charge de l'utilisateur. Cela signifie qu'aucune information, que ce soit dans les identifiants d'objets ou dans un catalogue quelconque ne permet de connaître la structure d'un objet complexe. C'est à la charge de l'utilisateur de GEODE de le savoir et d'appliquer dessus l'ensemble des procédures de manipulation fournies. Cette interface donne toutefois la visibilité des segments, ce qui permet de contrôler le regroupement des objets. C'est au dessus de cette interface que nous avons implanté la gestion des constructeurs ESQL.

VI.2.2 Choix de conception

Comme il a été dit en III.3, ESQL supporte deux sortes d'objet complexe : les objets et les valeurs. D'un point de vue purement logique, les valeurs sont stockées à l'intérieur même des tables relationnelles et leur durée de vie est égale à celle du n-uplet auquel elles appartiennent. Les objets sont stockés à l'extérieur des tables et sont seulement référencés par les n-uplets. Contrairement aux valeurs, ils sont donc partageables entre plusieurs n-uplets, voire entre tables, et leur durée de vie est supérieure ou égale à celle des n-uplets qui les référencent. En fait un objet disparaît lorsqu'il n'est plus référencé par un autre objet ou n-uplet. D'un point de vue physique, implanter les valeurs sous un format "plat", pour les stocker directement dans les tables relationnelles présente plusieurs inconvénients : les méthodes génériques auraient à connaître deux formats de stockage des instances de type abstrait, le format "plat" et le format du gestionnaire d'objets, ce qui nécessiterait en fait d'avoir deux jeux de méthodes génériques, un pour chaque format ; d'autre part l'accès aux valeurs complexes volumineuses est moins efficace sur un format séquentiel plat que dans un format de stockage adapté comme il peut l'être dans un gestionnaire d'objets comme GEODE. C'est pourquoi nous avons décidé de stocker valeurs et objets sous un même format, dans le gestionnaire d'objets, et de ne stocker dans les tables relationnelles que des références sur ces objets. C'est seulement au niveau de certaines opérations de manipulation et de l'analyseur ESQL que la sémantique des valeurs est prise en compte. Par exemple une opération d'insertion d'élément dans une valeur *ensemble* va retourner un nouvel ensemble, égal à l'ancien augmenté de cet élément, tandis que la même opération sur un objet *ensemble* va retourner ce même objet modifié.

A partir de maintenant, nous appelerons **objet** aussi bien les valeurs que les objets, en précisant explicitement lorsque cela sera nécessaire s'ils sont ou non partageables. Les objets complexes sont donc stockés à l'extérieur des tables relationnelles. Le même type de choix reste à faire en ce qui concerne les objets composés eux-mêmes, c'est-à-dire les objets dont l'état est constitué d'autres objets, qui peuvent être ou non partageables. Les objets composants pourraient être stockés à

l'intérieur même de l'objet englobant. Cela pose cependant des problèmes dans le cas ou un composant est partagé ou dans le cas où la taille d'un composant augmente. Nous avons opté pour la solution d'un stockage entièrement décomposé, où tous les composants sont stockés par référence, y compris les composants atomiques, tel que cela est montré dans la figure Fig. 6.3. Cette figure décrit comment un objet complexe ESQL est représenté de façon décomposée en une hiérarchie de sept objets (un objet par constructeur et un par composant). Une optimisation réalisée dans une nouvelle version de ce gestionnaire d'objets permet de stocker les composants atomiques (chaînes et numériques) à l'intérieur même des objets qu'ils composent, ceci afin de limiter le nombre d'objets et surtout d'accélérer l'accès aux objets composés.

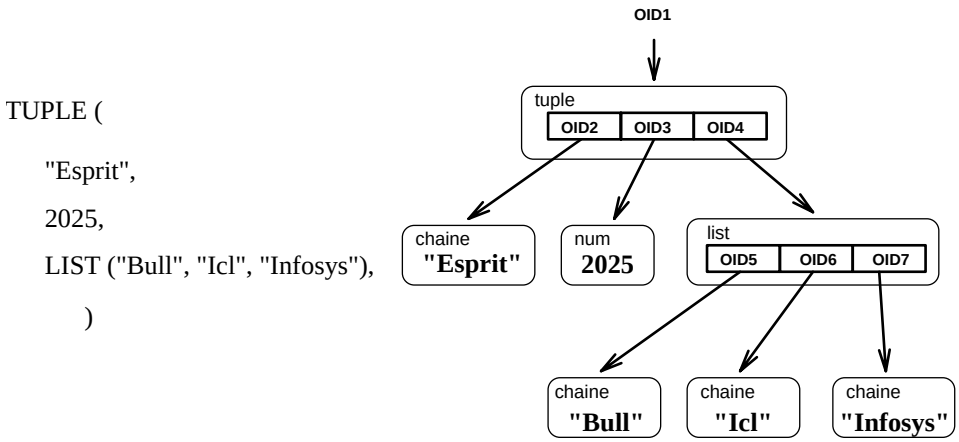

Fig. 6.3 : Stockage décomposé

Nous remarquerons que des choix identiques concernant le stockage des valeurs par rapport aux objets, ainsi que l'aspect décomposition ont été faits dans des systèmes tels qu'O2 [60].

VI.2.3 Fonctionnalités et mise en œuvre

La mise en œuvre d'une telle solution a consisté à développer au dessus de GEODE une couche de gestion de types génériques ESQL. Il a fallu pour cela enrichir la notion d'identificateur d'objet fournie (OID). En effet, la notion d'OID de GEODE permet simplement de localiser et d'identifier de façon unique un objet à l'intérieur d'un segment. Nous avons enrichi cette notion d'OID en y ajoutant la notion de **valeur/objet**, indiquant si un objet est partageable ou non, et la notion de constructeur. Cette dernière indication permet de déterminer le constructeur ESQL utilisé au premier niveau de la structure de l'objet (**tuple**, **set**, **list**, **array** ou simplement **chaîne** ou **numérique** si l'objet est atomique). Par exemple, pour les

instances des types définis ci-dessous, ces constructeurs seront respectivement *tuple* pour T1 et *set* pour T2.

```
CREATE TYPE T1 AS TUPLE OF (a1 CHAR(23),
                            a2 SET OF (NUMERIC(10))
                            );
CREATE TYPE T2 AS OBJECT SET OF (LIST OF (CHAR(2)));
```

Comme nous avons choisi un stockage décomposé (voir section précédente), le gestionnaire d'objets n'a pas besoin de connaître la structure d'un objet complexe lors d'un accès à cet objet, puisqu'il la détermine de façon récursive, en parcourant l'objet : à chaque niveau, l'OID lui indique la structure à venir, et donc les opérations à mettre en œuvre pour la parcourir.

Nous avons aussi envisagé d'indiquer dans l'OID l'identificateur du type abstrait de l'objet stocké. Cette possibilité n'est toutefois pas utilisée dans un but d'optimisation. En effet nous supposons que les contrôles de type sont complets à la compilation et qu'il n'est pas nécessaire de connaitre le type des objets manipulés à l'exécution. L'accès au catalogue de types à l'exécution serait coûteux, aussi avons nous stocké au niveau des OID le minimum d'informations nécessaire aux procédures de manipulation des objets complexes. Il s'avère que ces informations se limitent à l'identificateur GEODE, le constructeur et l'indication valeur/objet. Nos OIDs sont donc constitués de la façon suivante :

```
OID {
identificateur segment Geode;
identificateur objet Geode (dans le segment);
Constructeur(TUPLE,LIST,ARRAY,SET,STRING,NUM);
ValOrObj (VALUE, OBJECT);
    }
```

Le module construit au dessus de GEODE se caractérise par une bibliothèque de fonctions de manipulation d'objets complexes pouvant se répartir en trois catégories : des fonctions de stockage, des fonctions d'extraction et exportation vers des formats externes, et enfin des fonctions de manipulation directe, comprenant entre autres les méthodes génériques des constructeurs.

Il existe actuellement pour le stockage une fonction *Store* permettant de stocker une constante complexe. Cette dernière lui est passée sous forme arborescente (l'arbre résultant de l'analyse sémantique ESQL). La fonction travaille de manière récursive avec des fonctions *Maketuple* , *Makeset* , *Makelist* , *Makearray* qui créent respectivement des objets structurés *n-uplet*, *ensemble*, *liste* et *tableau* .

Les fonctions d'extraction permettent d'extraire un objet complexe du gestionnaire d'objets et de le restituer dans un format externe quelconque, afin par exemple d'être affiché ou utilisé dans un langage de programmation ou d'interface. La seule fonction d'extraction existant actuellement (*Extract*) permet de récupérer

le format de présentation ESQL d'une valeur complexe sous forme de chaîne de caractères directement affichable.

Les fonctions de manipulation des objets complexes sont constituées des méthodes génériques et d'un ensemble de fonctions utilitaires dont quelques exemples sont donnés ci-dessous.

- Il existe des fonctions de comparaison de deux objets : la première permet de déterminer si deux objets sont identiques (égalité des identificateurs si ce sont des *objects*, égalité des valeurs si ce sont des *values*) ; la deuxième est une fonction de comparaison profonde qui compare toutes les valeurs composant les objets comparés, que ceux-ci soient partageables ou non.
- Une fonction de copie permet de dupliquer un objet.
- La fonction VALUE permet de dupliquer le contenu d'un objet partageable et d'en faire un objet non partageable (*value*).

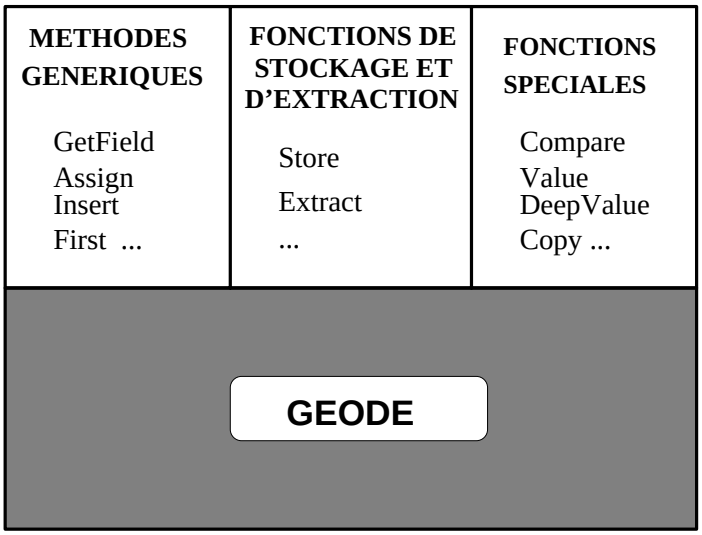

Fig. 6.4 : Gestion d'objets sur GEODE

VI.2.4 Contrôle des accès concurrents et reprise

Aucun mécanisme particulier de contrôle de la concurrence et de reprise après panne n'a été étudié lors de ce travail, d'une part par manque de temps et d'autre part car il semblait naturel de s'appuyer sur les mécanismes offerts par les systèmes sous-jacents. De plus la version de GEODE utilisée ne fournissait pas encore cette fonction, ce qui a retardé le lancement de cette étude. Il est cependant important de noter que dans le cas du couplage faible, nous serons confrontés à deux mécanismes de gestion de la concurrence et des reprises : celui du système relationnel, Sabrina, et celui du gestionnaire d'objets, GEODE. Il sera essentiel de faire collaborer ces deux systèmes, afin d'instaurer un mécanisme global cohérent.

VI.2.5 Persistance

Tout objet, instance d'ADT, est créé dans l'espace offert par GEODE et se trouve par conséquent persistant. Nous n'avons pas dans notre système la notion d'espace temporaire, bien que des objets temporaires puissent être créés. Nous utilisons dans ce cas le mécanisme de *segment temporaire* offert par GEODE. Cette notion permet de gérer de la même façon que les autres un objet qui disparaîtra en fin de session. Tout objet créé sans être rattaché à une racine persistante est stocké dans un segment temporaire. Au moment de le rattacher à une racine persistante, la fonction responsable de cette opération (par exemple la fonction qui insère l'objet dans un tableau) est chargée de copier cet objet dans le segment persistant adéquat (voir VI.4.1).

VI.2.5.1 Ramasse-miettes et compteurs de références

La définition de la persistance dans un système relationnel étendu comme le nôtre est dictée par le fait que tout accès à la base se fait via ESQL, et donc à travers les tables relationnelles. Il en résulte que tout objet accessible à partir d'une relation est persistant. Au niveau du gestionnaire d'objets, les racines de persistance sont donc les identificateurs qui se trouvent dans des relations. Il peut arriver que des objets partagés ne soient plus accessibles à partir des relations, ils doivent alors être détruits. De plus les fonctions de manipulation d'objet sur GEODE sont amenées à créer de nombreuses copies temporaires qui doivent aussi être détruites.

Considérons l'exemple décrit dans la figure Fig. 6.5 où les tables relationnelles R1 et R2 contiennent des racines de persistance pour les objets O1, O2, O3, O4 et O5. Nous supposons que l'objet O1 a une structure de n-uplet, dans lequel le champ *a* contient une référence sur l'objet O2. L'objet O2 contient lui des références sur O4 et O5. De même l'objet O3 contient une référence sur l'objet O5. Il existe plusieurs façons de modifier le graphe de références entre les objets, dont les trois suivantes[1] :

1. La première action consiste à modifier une référence à l'intérieur même d'un objet. Si l'on suppose que O1 est partageable, l'effet de la fonction générique ASSIGN est une modification de l'objet lui-même.
   UPDATE R1
   SET ATT2 = ASSIGN (ATT2, a, <autre chose que O2 ...>)
   WHERE ATT1 = 123;
2. La deuxième action est une modification de référence au niveau d'un attribut d'une relation.
   UPDATE R1
   SET ATT3 = <autre chose que O2 ...>
   WHERE ATT1 = 124;

1 Les flèches pointillées de la figure Fig. 6.5 correspondent aux références qui vont disparaître, suite à ces actions. Elles portent d'ailleurs le numéro de l'action qui va les détruire.

3. Finalement la troisième action est la suppression d'un n-uplet d'une relation contenant des références sur des objets.
   DELETE FROM R2
   WHERE ATT1 = 321;

Lorsque les actions (1) , (2) et (3) auront été accomplies, les objets O2 et O4 ne seront plus considérés comme persistants et pourront être détruits.

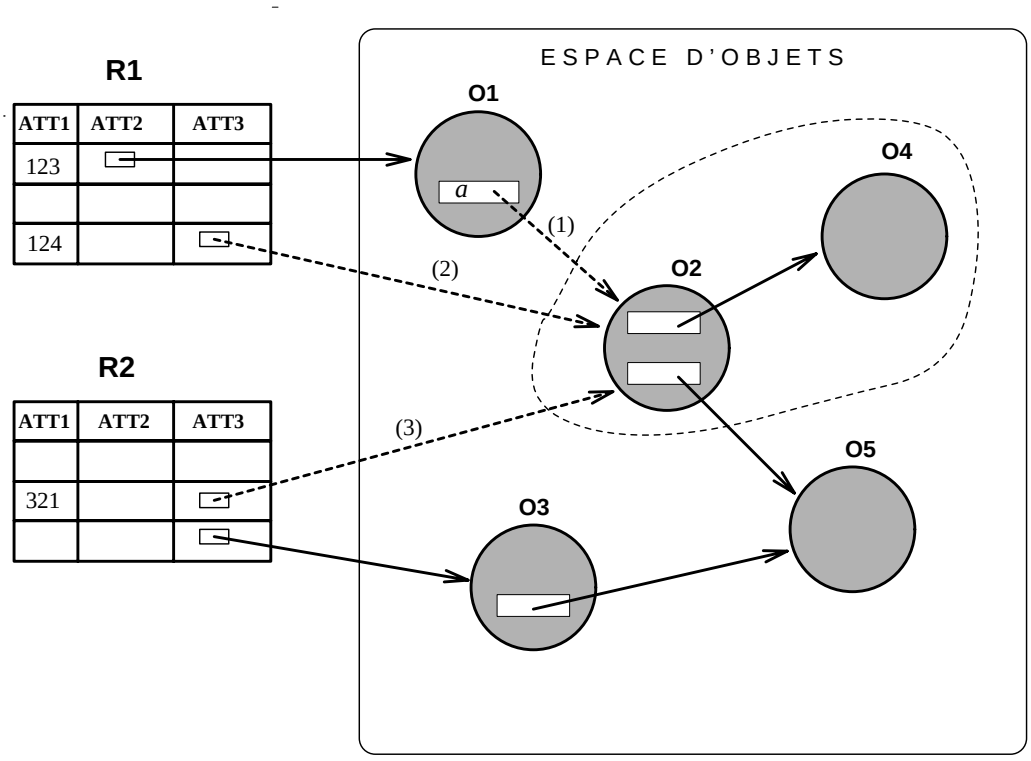

Fig. 6.5 : Gestion de la persistance

Deux techniques sont envisagées pour détruire les objets comme O2 et O4.
La première est un ramasse-miettes classique procédant par un algorithme de parcours et marquage. Il nécessite de connaître toutes les références à des objets présentes dans les relations et consiste à marquer tous les objets qu'elles permettent d'atteindre. Il faut ensuite accéder à tous les objets stockés (par un parcours séquentiel des objets d'un segment GEODE, pour chaque segment de la base d'objets) et détruire tous ceux qui ne sont pas marqués. Un tel mécanisme ne peut être mis en œuvre que périodiquement, lorsque le système n'est pas utilisé, car autrement il y aurait un risque de détruire des objets sur lesquels une référence était en cours d'affectation, et qui se trouvaient temporairement inaccessibles. Nous verrons que le parcours-marquage est la seule des techniques envisagées qui permet l'élimination

complète de tous les objets inaccessibles. Une étude détaillée des algorithmes de ramasse-miettes est disponible dans [46].

La seconde technique de ramasse-miettes met en œuvre pour chaque objet un compteur de références indiquant le nombre d'objets persistants ou de relations le référençant. Voici par exemple la valeur de ces compteurs sur les objets de la figure Fig. 6.5 entre les actions (1), (2) et (3) :

Compteur sur	avant (1)	après (1)	après (2)	après (3)
O1	1	1	1	1
O2	3	2	1	0
O3	1	1	1	1
O4	1	1	1	0
O5	2	2	2	1

Les mises à jour des compteurs se font au cours d'une application, au fur et à mesure des opérations sur les objets. A la fin d'une transaction, les objets dont le compteur est à zéro peuvent être détruits. La fonction de mise à jour des compteurs s'applique récursivement sur les composants d'un objet lorsque le compteur de celui-ci devient nul. A titre d'exemple, on constate que lors des actions (1), (2) et (3), le compteur de O2 est décrémenté. Lorsqu'avec l'action (3), il devient nul, les compteurs des composants O4 et O5 sont aussi décrémentés. Il reste en fait à savoir si les compteurs de O4 et O5 sont effectivement décrémentés lorsque celui de O2 passe à zéro, ou si l'on doit attendre la destruction effective de O2. Ce problème est discuté ci-dessous.

La deuxième technique pose le problème de savoir quand un compteur de référence doit être mis à jour. La première phase consiste à repérer qu'il doit être modifié ; cela intervient lors d'une affectation de référence, lors d'une destruction d'objet ou de n-uplet, ... Il faut ensuite savoir si la modification doit intervenir immédiatement ou s'il faut attendre la fin de la transaction. En effet, au cours de l'exécution d'une méthode, une référence peut être désaffectée, conservée dans une variable temporaire, puis réaffectée avant la fin de la méthode. Il peut alors se faire qu'un compteur passe à zéro, d'où décrémentation récursive des compteurs des composants, puis repasse à un, ce qui implique d'incrémenter les compteurs des composants. Deux solutions se présentent alors :

- la première consiste à incrémenter le compteur de références d'un objet lorsque celui-ci est affecté à une variable temporaire et à le décrémenter lorsque cette variable disparaît du contexte d'exécution ou se trouve réaffectée ;
- la deuxième consiste à ne déclencher la décrémentation récursive des compteurs (lorsque l'un d'eux passe à zéro) qu'à la fin de la transaction (où l'on est sûr que plus aucune variable temporaire n'existera).

L'implantation de la deuxième solution peut se limiter au gestionnaire d'objets alors que la première nécessite une intervention compliquée au niveau des compilateurs des langages utilisables (notamment les langages d'écriture des méthodes) afin de repérer et traiter ce genre d'affectation. Nous verrons que notre mise en œuvre décrite ci-dessous consiste justement à concentrer la gestion des compteurs au niveau des fonctions du gestionnaire d'objets et du compilateur ESQL. Il nous semble donc plus simple d'utiliser la deuxième solution.

Dans notre mise en œuvre se pose aussi le problème engendré par le fait que les références se trouvent réparties entre l'espace de stockage relationnel et le gestionnaire d'objets : le domaine d'application du mécanisme ne peut donc pas se limiter aux fonctions du gestionnaire d'objets, il doit s'appliquer au niveau d'ESQL.

La solution que nous avons adoptée repose sur une gestion des compteurs implantée à deux niveaux :

- Le premier aspect concerne les références inter-objets et se trouve implanté dans les fonctions de mises à jour des objets complexes (les méthodes génériques). Par exemple, la méthode qui permet d'ajouter un élément dans un ensemble se charge d'incrémenter le compteur de l'élément inséré.
- Le deuxième concerne les références contenues dans les tables relationnelles et est réalisé au niveau du compilateur ESQL. Par exemple, pour un ordre ESQL d'affectation d'un objet à un attribut de relation, un appel au gestionnaire d'objets est généré pour incrémenter le compteur de cet objet.

Pour finir il existe encore deux inconvénients majeurs à la technique des compteurs de références : les objets contenus dans des cycles (A référence B et B référence A) ne seront jamais détruits, le mécanisme ne se suffit donc pas à lui même ; la mise à jour des compteurs de référence est coûteuse en verrouillage et journalisation (cela revient à mettre à jour des objets pour lesquels aucun accès n'aurait dû avoir lieu).

Nous avons tout de même choisi d'implanter une solution utilisant les compteurs de références, qui permet de contrôler en continu une éventuelle prolifération d'objets inaccessibles. Cette solution devra cependant être complétée par la suite avec la mise en œuvre d'un ramasse-miettes par parcours-marquage.

VI.2.5.2 Gestion particulière des valeurs

La gestion de la persistance est très coûteuse, aussi nous sommes nous intéressés aux façons d'optimiser ces mécanismes. Il semble possible de limiter le nombre d'objets non persistants "laissés" à la charge de ces mécanismes. Il est notamment inutile, à priori, de gérer un compteur de références sur une valeur (objet non partageable) et il est bon de la détruire dès qu'elle n'est plus utilisée. Les manipulations d'objets génèrent un certain nombre d'objets temporaires qui peuvent être créés dans un espace réservé aux objets temporaires ; GEODE fournit

la notion de segment temporaire qui est utilisée à cet effet. Nous retrouvons ici des problèmes classiques liés à la gestion implicite de la persistance.

Nous avons déjà spécifié une optimisation de notre gestion d'objets pour limiter le nombre de copies de *valeurs* (objets non partageables). Rappelons que toutes nos fonctions travaillent sur des références (ou OID), y compris pour les valeurs. Jusqu'à maintenant, toute fonction d'accès aux objets qui retournait une valeur renvoyait en fait un OID d'une copie de cette valeur, afin qu'il n'y ait pas de risque, lors d'une utilisation ultérieure de ce résultat, de partager cette valeur. Supposons que l'objet O1, identifié par OID1, soit un tableau de valeurs. Le type T1 de O1 est défini ci-dessous :

```
CREATE TYPE T1 AS OBJECT ARRAY OF (T);
```

sachant que T est un type *valeur*. La fonction d'accès *get* (méthode générique du type ARRAY) permet d'accéder à la troisième valeur de ce tableau de la façon suivante :

```
OID2 = get (OID1, 3);
```

Comme les éléments du tableau sont des valeurs, OID2 identifie une copie du troisième élément. Cette valeur peut ensuite être utilisée pour mettre à jour un autre objet (ou valeur). Considérons une valeur n-uplet, identifiée par OID3, dont le type est défini comme suit :

```
CREATE TYPE T2 AS VALUE TUPLE OF (a1 numeric, a2 T);
```

On peut assigner la valeur désignée par OID2 à l'attribut *a2* du n-uplet OID3 de la façon suivante :

```
OID4 = assign (OID3, a2, OID2);
```

On remarque que le n-uplet OID3 est une valeur, qu'il ne peut donc être mis à jour directement, et donc que OID4 désigne une copie de ce n-uplet mis à jour. Quoiqu'il en soit, si OID2 ne désignait pas une copie du troisième élément de O1, ce dernier se trouverait partagé entre O1 et la valeur désignée par OID4, ce qui est contraire à la sémantique d'une valeur. Il s'avère donc nécessaire que chaque fonction de consultation retournant une valeur génère une copie de la valeur retrouvée. Le nombre de valeurs temporaires ainsi générées est considérable, car peu sont utilisées par la suite (il s'agit en général de simples consultations).

L'optimisation que nous proposons consiste à n'effectuer la copie d'une valeur qu'au moment où celle-ci est modifiée, ou utilisée pour mettre à jour une autre entité. Ainsi OID2 désigne véritablement l'élément du tableau O1, aucune copie n'est générée lors de la consultation (par *get* ici). Par contre la fonction de mise à jour *assign* est chargée de faire la copie de toute valeur qu'elle utilise. Elle fera donc la copie de la valeur désignée par OID2, afin d'assigner cette copie à l'attribut *a2* du n-uplet. Comme les structures de données des objets ne sont manipulables qu'à travers les fonctions génériques, il nous est facile de déterminer toutes celles qui font des mises à jour (*assign* pour le n-uplet, *insert* pour l'ensemble, etc.) et d'intégrer dans leur code la copie des paramètres *valeurs*. Aucune des fonctions de

consultation ne fait désormais de copie de valeur. Il faut cependant générer cette copie au niveau d'un ordre ESQL de mise à jour qui utilise une fonction de consultation :

```
UPDATE ...
SET ATT1 = get (ATT2, 3)
WHERE ...
```

La copie n'étant plus générée par la fonction de consultation, il faut générer l'appel de la fonction de copie au niveau de l'analyse ESQL.

Cette politique de copie de valeur nous a permis une deuxième optimisation basée sur le **partage de valeur** et la **copie sur mise à jour**. Lorsqu'une valeur est insérée en tant que composant dans un objet complexe (fonctions de mise à jour déjà citées : *insert*, *assign*, ...), on ne réalise la copie que si cet objet complexe se trouve dans un segment différent (pour des raisons de regroupement, voir section VI.4.1). Dans les autres cas, on laisse la valeur être partagée entre les objets (ou valeurs) dont elle est un composant. Cela n'est pas gênant tant que la valeur n'est pas modifiée. Or, elle ne peut l'être que par une fonction de mise à jour, qui, on l'a vu plus haut, est chargée d'effectuer la copie des valeurs qu'elle modifie. Il n'y a donc pas de risque qu'une modification de valeur partagée soit visible de plus d'un objet qui la partage. Ainsi, dans l'exemple précédent, la fonction *assign* ne fera pas de copie de la valeur désignée par OID2 (sauf si le segment contenant le n-uplet OID3 est différent du segment contenant la valeur OID2), et l'assignera directement à l'attribut *a2* du n-uplet. Par contre OID4 désigne bien une copie du n-uplet initial (OID3) car il s'agit d'une modification de valeur. Nous illustrons dans la figure Fig. 6.6 un cas de copie sur mise à jour d'une valeur partagée, en poursuivant le même exemple.

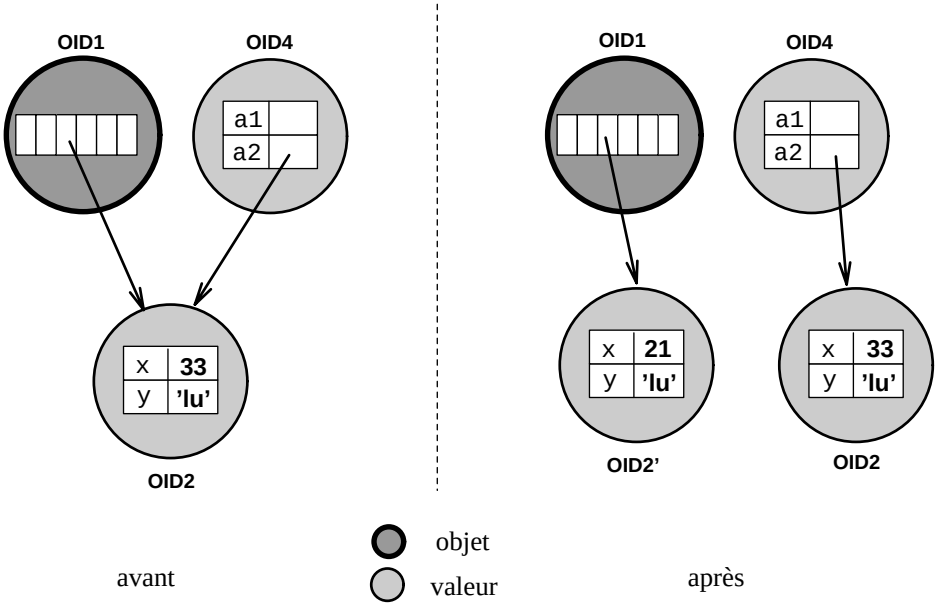

Fig. 6.6 : Mise à jour de l'attribut a2 de OID4

Nous supposons que la valeur désignée par OID2 a une structure de n-uplet composée de deux champs `x` et `y`, dont les types sont respectivement numérique et chaîne de caractères. Nous avons vu que cette valeur est partagée entre le tableau désigné par OID1 et le n-uplet désigné par OID4. Supposons maintenant que l'on veuille modifier le troisième élément du tableau par l'attribution au champ `x` de cet élément du nombre 21.

```
put (OID1, 3, assign (get (OID1,3), x, 21));
```

La fonction *assign* génère une copie de la valeur (que nous désignons par OID2'), qu'elle modifie et retourne. La fonction *put*, qui modifie le tableau en affectant le troisième élément, ne génère pas de copie puisqu'elle s'applique sur un objet (OID1).

VI.3 Cas du prototype parallèle à mémoire partagée

Parallèlement au prototype centralisé décrit en VI.2, une autre équipe du projet EDS (Bull - Les Clayes) a développé un prototype de SGBD relationnel sur une machine parallèle à mémoire partagée. Leurs objectifs concernaient essentiellement la mise en œuvre d'un modèle d'exécution relationnel parallèle. Ils ont développé un noyau de SGBD relationnel parallèle. Ce dernier n'intègre cependant aucune fonction de gestion d'ADT. L'objectif présent consiste donc à intégrer nos résultats dans ce prototype. Nous disposons cette fois d'un noyau relationnel maîtrisable, qui repose, pour le stockage, sur un gestionnaire de pages clairement spécifié. L'idée est donc d'abandonner SABRINA, et d'intégrer notre module de gestion d'objets dans ce nouveau système. Le niveau de fonction offert par GEODE nous convenant parfaitement, il a été décidé de porter GEODE sur le gestionnaire de pages qui est développé sur la machine parallèle. Ce gestionnaire de pages sera donc utilisé pour stocker les objets aussi bien que les tables relationnelles.

VI.3.1 Le gestionnaire de pages

Ce système est développé sur le noyau MACH, disponible sur la machine parallèle ENCORE. MACH met en œuvre une mémoire virtuelle paginée. Cette mémoire virtuelle est gérée entre la mémoire physique et un espace de "swap" sur disque. Un mécanisme matériel se charge de faire correspondre une adresse virtuelle à une adresse en mémoire physique. Lorsque la mémoire physique est saturée et que l'adresse virtuelle traitée correspond à une page non présente en mémoire, ce mécanisme doit renvoyer une page sur l'espace de "swap" afin d'en charger une autre. Il utilise pour cela une technique classique LRU (on décharge la page la moins récemment utilisée). Avec un processeur 32 bits, la taille de cette mémoire virtuelle est d'environ 4 giga-octets.

Le noyau MACH offre des possibilités intéressantes pour développer un gestionnaire de stockage à un seul niveau. On pourrait envisager dans un premier

temps de gérer la base en mémoire virtuelle et d'assurer la persistance par le mécanisme de "swapping" MACH. La base étant toujours couplée à la même adresse virtuelle, les adresses virtuelles peuvent être utilisées comme références sur les objets. Une telle solution présente de sérieux inconvénients : le premier est que le système d'exploitation ne laisse aucun contrôle sur l'allocation disque, les entrées/sorties, ainsi que sur les stratégies de remplacement ; le deuxième est la limitation de la taille de la base de données à celle de la mémoire virtuelle. Si la seule solution pour effacer le deuxième inconvénient est d'attendre la venue de processeurs avec 64 bits d'adressage, le système MACH est d'ores et déjà prévu pour parer au premier. Il fournit en effet la notion de paginateur externe, utilisée par Cricket [54], et décrite à cette occasion en II.5.5. Il est en effet possible de programmer un module de pagination et de l'associer à une zone de mémoire virtuelle. C'est ce module qui sera alors mis en œuvre à chaque défaut de page dans cette zone. L'utilisateur du noyau MACH pourra en programmant ce paginateur externe définir sa propre gestion des entrées/sorties, son allocation du disque et forcer les écritures de page. On peut donc envisager de gérer la base de données en mémoire virtuelle, la persistance étant assurée par le paginateur externe. Un tel schéma est représenté dans la figure Fig. 6.7.

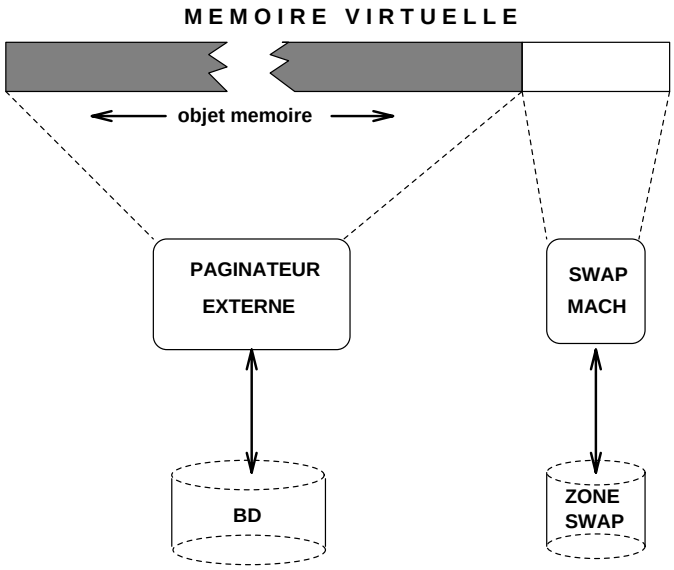

Fig. 6.7 : La BD complète en mémoire virtuelle

La limitation de la taille de la base de données a cependant conduit à abandonner cette approche, pour développer un *gestionnaire de cache* chargé de coupler les pages de la base de données en mémoire virtuelle, tout en laissant le contrôle du "swap" au système MACH. Les objets ne sont plus toujours couplés aux mêmes adresses, puisque la base de données ne tient pas en mémoire virtuelle, et que le gestionnaire de cache ne couple pas systématiquement les pages disques aux mêmes

adresses en mémoire virtuelle. Les adresses virtuelles ne peuvent plus tenir lieu de références stables et un décodage des adresses persistantes s'avère nécessaire.

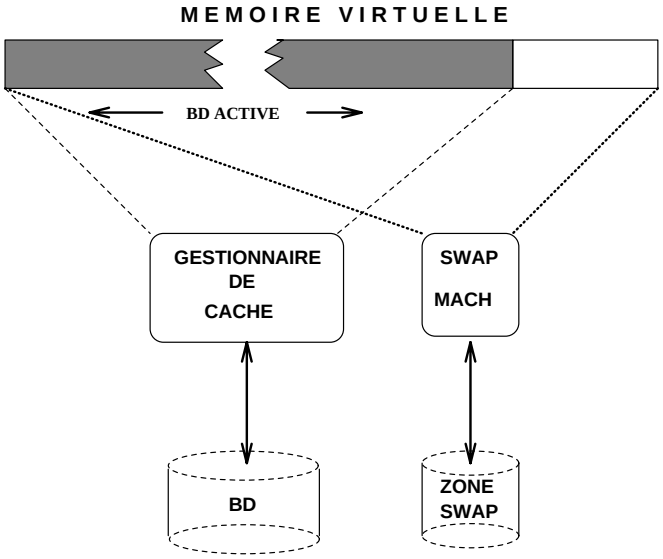

Fig. 6.8 : Gestionnaire de cache

Un tel gestionnaire de pages va être utilisé pour supporter le stockage des relations. Il assure la fiabilité des données (contrôle des accès concurrents et reprises après panne). Notre approche consiste à réutiliser la partie haute de notre gestionnaire d'objets en remplaçant la couche mémoire virtuelle de GEODE par ce gestionnaire de pages. Nous réduisons donc d'autant la redondance qui existait dans notre premier prototype.

VI.3.2 Exécution parallèle

Nous avons vu en IV.2.1 que le mécanisme d'exécution parallèle se base sur la décomposition des requêtes en un graphe de sous-opérations. Chaque sous-opération s'exécute sur un nœud de la machine. L'exécution est dirigée par les données : une sous-opération reçoit des n-uplets, les traite, et communique ses résultats à la sous-opération suivante. Les relations sont réparties horizontalement (par n-uplets) sur les nœuds. Les problèmes occasionnés par l'introduction des ADTs sont les suivants :

- le premier consiste à introduire la notion d'exécution de méthode parmi les opérations de base ;
- le deuxième concerne le placement des objets sur les nœuds de la machine.

Nous ne détaillerons pas le premier aspect, lié au modèle d'exécution. Nous supposons que le générateur de code inclut les appels de méthodes dans les programmes qu'il produit. Le deuxième problème concerne plus particulièrement le gestionnaire d'objets.

La génération d'un plan d'exécution parallèle prend en compte la répartition des n-uplets sur les différents noeuds. Il ne faut pas que le placement des objets sur les nœuds pénalise l'exécution d'un tel plan : lorsqu'une méthode doit être exécutée sur un nœud, il faut que l'objet concerné soit présent. Il n'est pas admissible d'avoir à exécuter une méthode sur un nœud distant car l'efficacité d'une exécution parallèle est en grande partie basée sur la localité des sous-opérations. La règle pour assurer cette localité consiste à ce que *tout objet se trouve toujours sur le même nœud que le n-uplet qui le réfère.* Cela veut aussi dire qu'un objet doit pouvoir être transmis d'un nœud à l'autre au même titre qu'un n-uplet[2]. Cette règle ne peut pas toujours être vérifiée à cause des conflits engendrés par le partage d'objets. Il se peut effectivement que deux n-uplets situés sur des nœuds différents partagent le même objet. Ce partage d'objet, que l'on peut qualifier de *spatial* (entre deux entités de l'espace de données[3]), engendre en fait un partage *temporel* (limité dans le temps, entre deux unités d'exécution). Plusieurs solutions sont envisageables :

- Les objets à l'origine de conflits pourraient être dupliqués afin d'être présents sur chaque noeud où ils sont référencés. Cela nécessite de gérer la cohérence des copies lors des mises à jour de tels objets.
- On peut malgré tout imaginer un mécanisme d'appel à distance. C'est-à-dire que chaque nœud joue alors le rôle de "serveur d'objets", dans le sens où l'on peut lui adresser des demandes d'exécution de méthode sur un objet.

Quelle que soit la solution adoptée, il faudra de toute façon gérer la distribution des objets : il faut être capable de localiser un objet et de le faire migrer. Ces problèmes restent actuellement ouverts pour notre projet. De nombreux travaux existent sur les systèmes à objets distribués, mais ne conviennent pas forcément aux contraintes particulières de notre modèle d'exécution parallèle.

VI.4 Optimisation d'accès

L'optimisation d'accès aux objets comporte deux phases complémentaires. La première consiste à instaurer une organisation physique des données permettant d'accélérer la recherche. Il s'agit non seulement de placer correctement ces données dans l'espace de stockage, mais aussi de gérer des structures de données annexes comme les index. La deuxième phase de l'optimisation consiste à générer des plans d'exécution tirant partie d'une part de cette organisation physique et d'autre part d'information sur la sémantique opérationnelle de certaines fonctions.

---

2 Car le modèle d'exécution est basé sur la transmission de n-uplets entre opérations.

3 C'est ce type de partage que nous avons traité jusqu'à présent.

VI.4.1 Regroupement

L'objectif du regroupement d'objets en mémoire secondaire est de limiter le nombre de chargements de pages en mémoire principale lors de l'exécution d'une requête. Dans un environnement grande mémoire cet intérêt pourrait se trouver limité, cependant nous avons constaté que dans les solutions adoptées, seule la base de données active pouvait tenir en mémoire virtuelle, ce qui implique que des accès disques seront toujours nécessaires. De plus, la taille de la mémoire centrale étant encore inférieure à la taille de la mémoire virtuelle, le "swapping" existera toujours et le regroupement dans les pages des objets accédés ensemble peut limiter ce phénomène. Nous utiliserons la notion de *segment* GEODE comme unité de regroupement. La difficulté principale réside dans la définition de la politique de regroupement des objets. Celle-ci se trouve en fait fortement corrélée avec la politique de placement des relations. Plusieurs critères de regroupement sont envisageables et peuvent même être combinés :

- il est possible de regrouper les objets par type ;
- les composants d'un objet complexe peuvent être stockés ensemble ;
- enfin il serait souhaitable de regrouper tous les objets accessibles à partir d'une relation.

Si le premier critère est important dans un système à objet classique, où la notion de classe, ou ensemble des instances d'un type, peut être la notion principale de collection d'objets, il n'en est pas de même dans notre système. En effet, les accès séquentiels à un ensemble d'objets sont issus du parcours des tables relationnelles ou du parcours d'un objet construit *liste*, *ensemble* ou *tableau*. En fait nous allons voir que les deux derniers critères nous suffisent.

Les deux notions de collection d'objets présentes dans notre système sont la relation, et les objets construits à l'aide des constructeurs multi-valués *liste*, *ensemble* ou *tableau*. Il semble donc important de regrouper l'ensemble des objets accessibles à partir d'une relation, sans distinction de types, car une opération de parcours de cette relation peut amener à accéder aux objets référencés dans les n-uplets. Il est aussi nécessaire de regrouper les composants d'un objet construit, ce qui permet de traiter le cas des *ensembles*, des *listes* et des *tableaux*. Nous allons donc appliquer une politique de regroupement basée sur les deux derniers critères cités ci-dessus.

Les objets partagés entre plusieurs relations engendrent des conflits. Ces derniers sont résolus par une politique "premier arrivé, premier servi", c'est-à-dire qu'un objet partagé est stocké dans le segment associé à la relation à partir de laquelle il a été créé. Nous n'envisageons pas de gérer de duplication.

La figure Fig. 6.9 illustre une telle stratégie de regroupement. Les flèches représentent les références aux objets, les différents types d'objet sont imagés par des formes différentes. On distingue deux segments associés à deux relations ; les

objets et leurs composants sont stockés dans le segment associé à la relation qui les réfère.

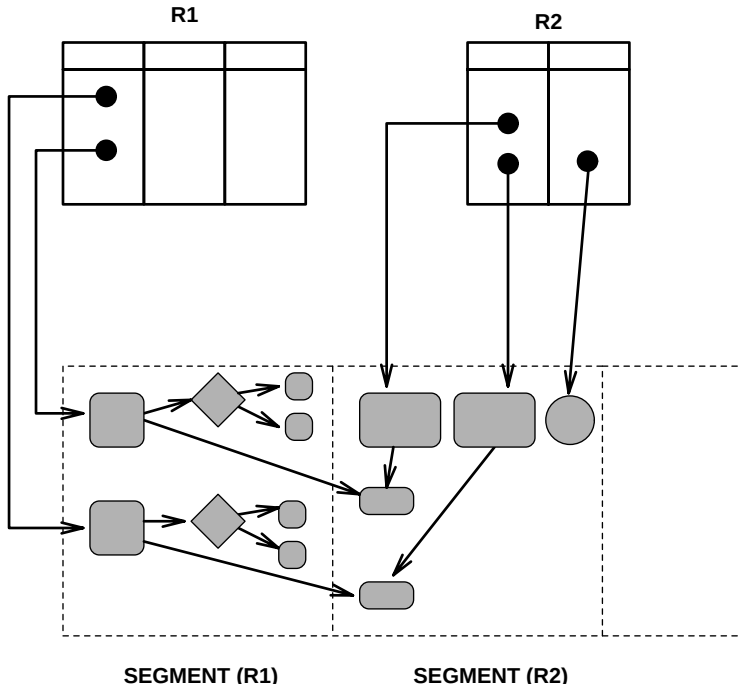

Fig. 6.9 : Groupement d'objets

Les directives de regroupement sont prises en compte dans les fonctions de stockage d'objets. Les fonctions de création d'objets construits (MakeTuple, MakeList, MakeSet, ...), ainsi que les fonctions d'ajout de nouveaux composants (insertion d'un élément dans un ensemble, une liste, un tableau, mise à jour du champ d'un n-uplet) créent les composants dans un même segment, celui de l'objet père. La fonction générale de création d'un objet, *Store*, a un paramètre permettant d'indiquer la relation (ou le segment associé) de création de l'objet. Lorsqu'un objet doit être créé à partir d'un ordre ESQL, il est très facile de déterminer la relation d'origine, lorsqu'il est créé dans une méthode, soit il est temporaire, auquel cas on le stocke dans un segment temporaire, soit c'est un composant d'un autre objet, auquel cas on le stocke dans le même segment que ce dernier.

```
INSERT INTO ACTOR
VALUES (tuple("Newman", "Paul", 1000), list(10,20));
```

Dans l'exemple précédent, les deux objets seront créés dans le segment associé à la relation ACTOR. Il en est de même pour l'objet *liste* de l'exemple suivant.

```
UPDATE ACTOR
SET FILMS = list (20, 30)
WHERE getfield(ACTOR, nom) = "De Niro";
```

VI.4.2 Optimisation de requêtes

Les requêtes qui manipulent des objets complexes sont celles qui contiennent des appels de méthode. Nous présentons dans cette section quelques solutions pratiques pour optimiser de telles requêtes. Aucune implantion n'a été faite, ceci constitue nos premières idées sur le problème de l'optimisation. Nous montrons ensuite comment le faible couplage des modèles de données (objet et relationnel) rend difficile la phase d'optimisation.

Les travaux effectués dans le domaine de l'optimisation pour les systèmes orientés objets montrent qu'il est en fait nécessaire de briser l'encapsulation ([38]). Effectivement, il n'est pas possible d'optimiser pleinement une requête contenant des appels de méthodes, si l'on ne connait pas le comportement de ces méthodes (à quels et à combien d'objets elles accèdent, si elles effectuent des mises à jour, etc.). Nous n'avons pas poussé nos travaux au point d'avoir la possibilité de révéler à l'optimiseur le comportement des méthodes, aussi nous contentons nous de proposer quelques optimisations où les méthodes sont vues comme des "boîtes noires".

Des recherches ont aussi été menées sur l'optimisation des accès particuliers qui sont effectués dans une base d'objets complexes ([41]). Ces accès empruntent des chemins qui traversent un ensemble d'objets liés par une relation de composition (sélections sur un critère portant sur un composant d'un composant ...). Ce genre d'optimisation est basée sur l'existence de regroupements d'objets et d'index. Ces accès à travers les objets restent visibles tant qu'ils sont exprimés au niveau de la requête (chez nous au moyen des méthodes génériques associées aux constructeurs), mais peuvent aussi être cachés dans le code des méthodes utilisateurs. Ces phases d'optimisation ne sont pas non plus abordées dans cette section.

Nous partons de l'hypothèse qu'un appel de méthode dans une requête est coûteux, car il implique des accès au gestionnaire d'objets, et que le code qui doit être déroulé (développé par l'utilisateur) peut être important. Les techniques d'optimisation que nous proposons consistent en fait à limiter le nombre d'appels de méthodes à effectuer pour évaluer une requête.

Le premier principe consiste à retarder le plus possible l'évaluation d'une méthode apparaissant dans une requête. Ainsi, lorsque dans une expression de sélection figurent des prédicats classiques et des prédicats contenant des appels de méthode, les premiers doivent être évalués en priorité, ce qui peut éviter d'avoir à évaluer les seconds. Par exemple dans la requête suivante on commencera à sélectionner les n-uplets en évaluant `c = 'Bull'`, puis `b = 3`, pour finir par exécuter la méthode *f* dans le cas où le premier est faux et le deuxième vrai (a,b et c sont des attributs de `R`).

```
SELECT ...
FROM R
WHERE  (f(a) = 25 AND b = 3) OR c = 'Bull';
```

Le deuxième principe consiste à limiter le nombre d'appels de méthodes sur un parcours de relation. Considérons les trois requêtes suivantes :

```
SELECT f(a), b
FROM R
WHERE b > 3;                [1]

SELECT b
FROM R
WHERE f(a) > 100;           [2]

SELECT ...
FROM R1, R2
WHERE f(R1.a) = g(R2.b);    [3]
```

L'évaluation de la requête [1] qui limite le nombre d'appels de méthodes consiste à effectuer tout d'abord la sélection (sur b > 3), puis à trier le résultat sur a, pour ensuite le parcourir en n'évaluant *f*(a) que pour les valeurs de a distinctes (les valeurs identiques se suivant, on les évite simplement). Une telle stratégie d'évaluation n'est intéressante que si le nombre de valeurs distinctes de a pour la relation R est faible par rapport au nombre de n-uplets de celle-ci. Pour évaluer la requête [2], on trie la relation R de la même manière, afin de limiter le nombre d'appels de *f* lors du parcours. Pour effectuer la jointure [3] on peut trier R1 sur a, trier R2 sur b, puis effectuer la jointure en boucles imbriquées en n'évaluant *f* et *g* que pour les valeurs distinctes de a et b. L'algorithme d'évaluation de [3] est succintement décrit ci-dessous.

```
R1' = trier R1 sur a;
R2' = trier R2 sur b;
vieux_a = NIL;
vieux_b = NIL;
Pour chaque n-uplet r1 de R1'
    {
    Si r1.a != vieux_a alors
        {
        fa = f(r1.a);
        vieux_a = r1.a;
        }
    Pour chaque n-uplet r2 de R2'
        {
        Si r2.b != vieux_b alors
            {
            gb = g(r2.b);
            vieux_b = r2.b;
            }
        Si fa = gb alors produire le n-uplet résultat à
        partir de r1 et r2.
        }
    }
```

Il est évident que le gain obtenu sur le temps d'exécution des méthodes (temps économisé) doit être supérieur au temps utilisé pour le tri des relations. Une telle évaluation nécessite de connaître les temps d'exécution de méthodes ainsi que les taux de duplication d'attributs (nombre de valeurs distinctes par rapport au nombre total de valeurs).

Remarque :

dans notre cas les valeurs d'attribut qui représentent des instances d'ADT sont des OID du gestionnaire d'objets. Deux *objets* identiques ont le même OID ce qui n'est pas le cas de deux *valeurs* (objets non partageables) égales. La technique décrite ci-dessus ne peut donc être efficace pour les applications de méthodes sur des attributs ADT *valeurs*, sauf si la technique de *partage de valeur* définie à la fin de la section VI.2.5 occasionne un taux de partage important (auquel cas on peut trouver un nombre d'OID identiques significatif pour un ensemble de valeurs).

Les techniques qui viennent d'être présentées permettent d'optimiser le nombre d'accès aux objets et non pas l'accès aux objets lui-même. Ce dernier est difficile à maîtriser à cause de la séparation entre les manipulations relationnelles et objet. Les accès aux objets se font de manière transparente au monde relationnel par l'intermédiaire des méthodes. Cette encapsulation ne permet pas à l'optimiseur d'avoir une vision de la structure des objets manipulés et du type de manipulation auxquels ils sont soumis. Considérons l'exemple suivant :

Supposons que deux relations `R1` et `R2` sont définies de la façon suivante :

```
CREATE TABLE R1 (a numeric, b SetOfT);
CREATE TABLE R2 (c T, d char(25));
```

où `T` est un ADT objet et *SetOfT* un ADT défini comme un ensemble d'éléments de type `T`.

```
CREATE TYPE T AS OBJECT ...;
CREATE TYPE SetOfT AS SET OF (T);
```

La figure Fig. 6.10 ci-dessous illustre un tel schéma de données :

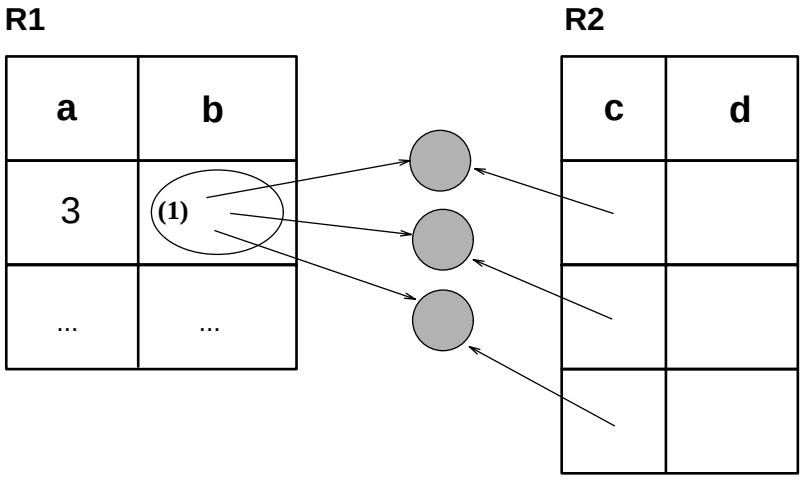

Fig. 6.10 : Schéma de l'exemple

Considérons maintenant la requête suivante :

```
SELECT R2.d
FROM R1, R2
WHERE contains (R1.b, R2.c) AND R1.a = 3;
```

La fonction *contains* indique si l'objet désigné par `R2.c` appartient à l'ensemble `R1.b`. Cette requête revient à faire une sorte de jointure entre l'ensemble (1) (voir figure Fig. 6.10) et la relation `R2`. On pourrait alors considérer deux stratégies d'évaluation :

1. Pour chaque n-uplet de `R2`, parcours de l'ensemble (1) (par la fonction *contains*) pour tester si `R2.c` est dedans ; si oui, on retient `R2.d`.
2. Pour chaque élément `t` de (1), parcours de la relation `R2` ; si `R2.c` = `t` on retient `R2.d`.

Des critères d'organisation physique (taille des relations, existence d'un index) permettent en général, lorsqu'on évalue une jointure par boucles imbriquées, de déterminer quelle doit être la relation interne. Le problème est semblable ici : il peut être intéressant de parcourir la relation `R2` dans la boucle interne (il peut par exemple exister un index sur `R2.c`, ce qui limite le parcours de `R2` à un simple accès à l'index) ; la stratégie 2 est alors avantageuse. Or il s'avère que cette stratégie n'utilise plus la fonction *contains*, et que l'on est incapable, au niveau de l'évaluation relationnelle, de parcourir l'ensemble (1). Ceci est un exemple typique qui permet de constater que l'optimiseur et l'évaluateur relationnels n'ont pas la visibilité des objets complexes qui représentent des collections d'objets. Ils n'ont donc pas la possibilité d'optimiser les accès à de telles collections. Ce problème disparaît lorsque le modèle ne présente qu'un seul niveau d'abstraction des collections d'objets, comme dans O2 ([22]), où les techniques d'optimisation relationnelles peuvent être beaucoup plus facilement étendues à la manipulation des objets complexes.

VI.5 Conclusion

Cette étude nous a permis d'acquérir une bonne expertise sur la façon de concevoir un gestionnaire de stockage d'objets. Nous avons pris une approche descendante (par raffinements) consistant à choisir en premier lieu une architecture et un niveau de fonctions. Nous avons ensuite étudié des techniques de stockage particulières, puis nous les avons améliorées par quelques optimisations. Les bases de l'implantation du prototype parallèle ont été fixées. La fin de cette étude fut consacrée au problème de l'optimisation.

Il existe deux types d'architecture pour un gestionnaire de stockage d'objets : la première, dite "à un niveau", implante une mémoire persistante d'objets ; la deuxième, "à deux niveaux", offre une gestion explicite de la persistance pour chaque objet, qui doit être chargé en mémoire et renvoyé sur disque par des opérations spécifiques. Les niveaux de fonction vont de la gestion de la persistance de "chaînes d'octets" à la gestion d'objets typés (comprenant les mécanismes d'exécution de méthodes). Nous avons choisi d'utiliser comme base de développement de notre prototype le gestionnaire d'objets GEODE qui offre une architecture "à un niveau" (la pagination est cependant logicielle). Nous avons construit au dessus un gestionnaire de stockage d'objets dont le niveau de fonctions assure la gestion d'objets structurés (les objets complexes ESQL).

Nous avons pu par la suite aborder des techniques de stockage particulières, que nous avons adaptées à notre cas. Nous avons choisi de stocker de la même façon les objets et les valeurs (objets non partageables). Nous avons adopté un stockage par "décomposition". La persistance des objets est assurée au moyen d'un compteur de références. Des techniques de "partage de valeurs" et de "copie sur mise à jour" ont permis de limiter le nombre de copies de valeurs générées.

Concernant le prototype parallèle, les bases de l'architecture ont été définies. Il reste cependant un travail important à faire concernant la prise en compte des objets dans le modèle d'exécution et leur répartition sur les nœuds de la machine.

Finalement nous avons spécifié des techniques de regroupement d'objets. Nous appliquons deux stratégies qui sont mises en œuvre dans les fonctions du gestionnaire d'objets : regroupement par relation et regroupement composé/composants. La partie optimisation se termine par une étude des possibilités d'optimisation de requêtes contenant des appels de méthode. On met ici en évidence la difficulté d'optimiser un mécanisme d'exécution issu du couplage faible entre deux modèles de données.

Chapitre VII

Gestion de méthodes

Ce chapitre aborde deux aspects de la gestion de méthodes : il s'agit d'une part de l'élaboration d'un langage de programmation de méthodes pour ESQL, ainsi que du développement du compilateur correspondant, et d'autre part de la mise en œuvre d'un mécanisme d'exécution des méthodes pour le noyau relationnel étendu. Nous décrivons plus particulièrement dans ce chapitre la mise en œuvre effectuée pour le premier prototype centralisé.

La structure de données d'un type abstrait et les signatures des méthodes associées sont décrites au niveau de ESQL. Ce langage purement déclaratif, dont les opérateurs ne s'appliquent que sur des tables relationnelles, ne peut évidemment pas convenir pour écrire le corps de ces méthodes. Il convient donc d'imaginer un langage procédural permettant d'exprimer des traitements s'appliquant sur les structures de données des instances de types abstraits. Nous avons décidé d'utiliser un langage existant comme moyen d'expression de ces manipulations. Le principe consiste à étendre ce langage en y introduisant le concept d'ADT. Une phase de précompilation permet de traiter toutes les instructions de manipulation d'ADT apparaissant dans le corps d'une méthode. Nous verrons qu'une telle approche permet de proposer plusieurs langages au programmeur de méthodes.

Le mécanisme d'exécution des méthodes comprend deux aspects distincts. Le premier consiste à déterminer de quelle façon les méthodes vont accéder aux objets et de quelle manière elles vont manipuler leurs structures de données. Le deuxième aspect concerne l'appel de ces méthodes par le noyau d'exécution relationnel. Concernant le premier aspect, nous avons adopté une approche originale consistant à donner accès aux objets à travers la même interface qu'en ESQL, c'est-à-dire à travers les méthodes génériques et les méthodes précédemment définies. De plus, ces accès se font directement sur les données de la base. Pour le deuxième aspect, nous avons mis en place un mécanisme d'identification et de liaison des méthodes. Une méthode est mise en œuvre lors de l'évaluation d'une requête relationnelle et s'applique sur les valeurs d'attribut retournées par les opérateurs relationnels. Pour être appelée par la machine algébrique, une méthode doit être identifiée au niveau du système et le module objet résultant de sa compilation doit être lié ; c'est ce que nous assurons. L'identification est gérée au niveau du catalogue et un éditeur de liens dynamique permet de lier instantanément une méthode qui vient d'être compilée.

La première section décrit le langage de programmation de méthode proposé. La section suivante présente les choix de conception adoptés pour le mécanisme

d'exécution des méthodes. Ces choix ont des conséquences sur le compilateur du langage qui fait l'objet de la section VII.3. Les phases d'édition de liens et de lancement de méthode sont enfin décrites dans la section VII.4.

VII.1 Langage de programmation des méthodes

Le langage de programmation de méthode est un langage existant auquel a été ajoutée la notion de type abstrait. Nous appelons LP (Langage de Programmation) le langage existant que nous étendons, et LPM (Langage de Programmation de Méthode) le langage obtenu. Les instances de type abstrait sont visibles à travers des variables et manipulables uniquement par leurs méthodes associées. Nous avons en fait ajouté au langage LP la possibilité de définir des variables de type abstrait, dont les valeurs représentent des instances d'ADT, et sur lesquelles il est possible d'appliquer des méthodes. L'approche est similaire à celle adoptée pour ESQL, le système de types abstraits de ESQL est ajouté au langage LP, comme il avait été ajouté à SQL. Le programmeur dispose, pour manipuler la structure de données d'un objet, du jeu de méthodes génériques définies sur les constructeurs ESQL. Il a de plus la possibilité d'utiliser les autres méthodes déjà définies (par lui-même ou un autre utilisateur) sur le type concerné. Dans cette solution, la structure de données physique des objets n'est pas vraiment visible à l'utilisateur, il ne voit que la notion de constructeurs ESQL, et dispose pour les manipuler des méthodes génériques.

Une autre solution pour manipuler les instances de type abstrait à travers un langage LP consiste à représenter les objets par des structures du langage LP et à les manipuler par les opérateurs des constructeurs du langage LP. L'avantage est que le programmeur utilise directement le langage qu'il connaît parfaitement. Cependant cela donne deux vues différentes de la structure de données d'un type abstrait au programmeur : la vue ESQL et la vue LP. Il faut donc qu'il gère lui-même la correspondance entre ces deux structures. La structure d'un objet est en fait imposée par les constructeurs ESQL, nous pouvons la qualifier de "logique". De plus, le jeu de méthodes génériques est suffisant pour écrire toutes les manipulations de structures de données. Il est donc dommage de multiplier les vues de l'état d'un objet. Cette approche serait valable si la structure logique d'un ADT n'était pas visible au niveau ESQL. Dans ce cas, l'état d'un objet serait une véritable "boîte noire" et sa structure pourrait être définie dans le langage LP d'implantation des méthodes. Comme ce n'est pas le cas, nous n'adopterons pas cette deuxième solution. Nous verrons en VII.2.1 que notre première solution permet en outre une réalisation beaucoup plus simple et plus efficace.

Il est clair que l'encapsulation proposée dans ESQL n'est que partielle, puisque la structure logique d'un objet est visible et manipulable ; ces manipulations sont toutefois limitées aux méthodes génériques. Nous pourrions envisager d'interdire

l'utilisation de ces dernières au niveau ESQL pour les réserver à l'écriture des méthodes et renforcer ainsi la notion d'encapsulation offerte.

Nous avons donc mis en œuvre la première approche, en utilisant C comme langage de programmation LP. Les objets sont représentés par des variables spéciales, tandis que les types de base ESQL ont leur correspondance directe en C.

Une expérience annexe nous a montré que l'utilisation d'un langage de programmation orienté objet (LPOO) pour écrire les méthodes n'est pas forcément intéressant. Cela est dû à notre solution où tout l'aspect "objet" est amené dans l'extension du langage LP. Le seul apport du langage est constitué des structures de contrôle permettant d'écrire des algorithmes (ce que l'on ne peut faire avec ESQL). Utiliser les caractéristiques d'un LPOO nous obligerait à pratiquer une correspondance entre les types ESQL et ceux du langage (on retrouve les inconvénients de la deuxième solution abordée ci-dessus). Un des résultats du travail réalisé en [58] est d'avoir réalisé pour C++ les deux approches[1]. La première est analogue à ce que nous avons fait pour C ; les caractéristiques objets de C++ ne sont pas mises en valeur. La deuxième consiste à définir en C++ des classes équivalentes aux types ESQL et à manipuler les objets de la base en C++ après les avoir importés dans les classes. Ce deuxième type de solution ne nous satisfaisant pas, nous ne portons pas d'intérêt particulier aux LPOO dans ce chapitre.

Il semble assez simple d'étendre ainsi un langage de programmation LP afin d'obtenir un LPM. C'est une approche réaliste et facile à mettre en œuvre. Cela permet par ailleurs de proposer au programmeur un langage qui lui est familier, toujours mieux accepté qu'un nouveau langage. Afin d'illustrer notre solution, le code des méthodes présentées en III.3 est donné ci-dessous :

```
Acteur augmenter_sal (acteur, taux)
Acteur acteur;
int taux;
{
int sal,newsal;
sal = GetField (acteur,salaire);
newsal = ((sal/100)*taux)+sal; /* Nouveau salaire */
Assign(acteur, salaire, newsal);
return acteur;
}

int salaire_max (acteurs)
Acteurs acteurs;
{
Acteur p;
int sal;
int max = 0;
```

1 L'objectif de ce travail n'était pas de concevoir un langage d'écriture de méthodes, mais de fournir un langage de programmation d'applications utilisant le serveur ESQL pour gérer les données persistantes. L'aspect LPM en est un résultat annexe.

```
p = First (acteurs);
for ( ; p.tag != null; p = Next (acteurs))
    {
    sal = GetField (p, salaire);
    if (sal > max) max = sal;
    }
return max;
}
```

L'en-tête d'une méthode est similaire à celui d'une procédure C ; le compilateur l'identifie par le fait que le premier paramètre est de type ADT et en consultant le catalogue des types de la base pour s'assurer qu'elle est bien définie sur cet ADT. La méthode *augmenter_sal* augmente du pourcentage donné en paramètre le salaire d'un acteur. On notera l'utilisation du type abstrait *Acteur* dans le code de la méthode, parmi les types C, ainsi que l'utilisation de méthodes génériques *GetField* et *Assign* définies sur le type générique *tuple* (on rappelle que la structure de données d'un objet *Acteur* est un n-uplet). La méthode *salaire_max* renvoie le salaire maximum d'un ensemble d'acteurs.

```
Acteurs augmenter_special (acteurs, taux)
Acteurs acteurs;
int taux;
{
Acteur p;
int maxsal, sal;
maxsal = salaire_max (acteurs);
p = First (acteurs);
for ( ; p.tag != null; p = Next (acteurs))
    {
    sal = GetField (p, salaire);
    if (sal != maxsal)  augmenter_sal (p, taux);
    }
return acteurs;
}
```

La méthode *augmenter_special* augmente du pourcentage passé en paramètre les acteurs d'un ensemble d'acteurs sur lequel elle s'applique, sauf l'acteur le plus payé. Cette méthode illustre l'utilisation d'autres méthodes définies par l'utilisateur.

VII.2 Choix de conception

VII.2.1 Accès aux objets

Pour des raisons de performance et de simplicité nous avons décidé de considérer les méthodes prédéfinies (dont les génériques) et celles de l'utilisateur comme des routines du SGBD. Elles sont donc liées au SGBD et s'exécutent dans le même espace de travail. Les méthodes génériques et les fonctions systèmes travaillent

directement sur le gestionnaire d'objets ; elles utilisent le format de représentation interne des objets. Les méthodes utilisateurs agissent donc sur les données du gestionnaire d'objets, puisqu'elles utilisent ces mêmes méthodes génériques pour manipuler les objets. Nous décrivons ci-dessous la démarche qui nous a conduit à cette solution.

Deux alternatives se présentent en fait pour l'accès aux objets par les méthodes utilisateurs :

- la première consiste à charger les objets dans une structure de données du langage LP, afin de travailler dessus. Cette solution est obligatoire si l'on décide que le programmeur de méthodes voit directement la structure de données LP d'un objet.
- la deuxième solution consiste à manipuler les objets dans le format instauré par le gestionnaire d'objets lui-même.

La première solution présente l'inconvénient majeur d'avoir à charger les objets dans un format LP à chaque application de méthodes, puis, s'ils ont été modifiés, de les retraduire dans le format du gestionnaire d'objets pour les stocker à nouveau dans la base. De telles opérations de traduction de format à chaque appel de méthode peuvent être très coûteuses. Ceci est particulièrement vrai lorsque l'on traite des objets volumineux ; notamment lorsqu'un objet est composé de nombreux autres objets, il faut pour le charger dans des structures LP décider si tous les composants doivent aussi être extraits. De plus, travailler sur un format de données LP implique l'écriture d'un jeu de méthodes génériques par langage de programmation de méthodes LPM. En effet, si les méthodes génériques sont utilisées dans une méthode, elles doivent s'appliquer sur les données représentant l'objet en cours de traitement, dans ce cas un objet au format LP. Dans le cas où le gestionnaire d'objets est de bas niveau (il ne gère que des chaînes d'octets), les objets peuvent être stockés directement dans le format du langage LP en faisant une copie de la zone mémoire contenant la structure LP de l'objet (si cette structure est relogeable, c'est-à-dire ne contient pas de pointeurs mémoire). Nous avons cependant décidé d'adopter un gestionnaire d'objets de plus haut niveau pour des raisons d'efficacité de stockage d'objets complexes et volumineux.

Dans la seconde approche, les méthodes s'exécutent sur des objets au format du gestionnaire d'objets. Elles utilisent pour cela les méthodes génériques permettant de manipuler ces structures de données. Les paramètres de ces méthodes représentant des objets sont des identificateurs (OID) du gestionnaire d'objets. Nous avons choisi cette approche pour développer notre gestionnaire de méthodes. Cela permet de développer plusieurs langages de programmation de méthodes, inspirés de différents langages LP, mais travaillant sur un format de données commun, avec le même jeu de méthodes génériques. Ces méthodes travaillant sur le même format de données pourront éventuellement s'appeler mutuellement. Un autre avantage consiste en ce que les méthodes travaillent directement sur les objets dans

la base de données, en opérant directement sur le gestionnaire d'objets. Ce dernier avantage ne peut être obtenu dans la première solution que si les objets sont stockés dans le format LP, vu comme une chaîne d'octets par le gestionnaire d'objets, et si les opérateurs du langage peuvent s'appliquer sur les objets dans les tampons du gestionnaire d'objets.

VII.2.2 Identification des méthodes

Le mécanisme d'exécution des méthodes est basé sur l'identification unique de celles-ci dans le système. Lors de l'entrée d'une nouvelle méthode dans le catalogue, par l'ordre LDD ESQL de définition de sa signature, un identificateur unique lui est assigné et est stocké dans le catalogue. Cet identificateur est composé de deux parties : la première indique le type retourné par la méthode, la deuxième servira d'index dans les tables des méthodes du système lorsque la méthode y apparaîtra. Il existe une table par type retourné (numérique, caractère, booléen, float ou ADT), la première partie d'un identificateur permet donc de déterminer dans quelle table se trouve la méthode. Les méthodes génériques associées aux constructeurs possèdent elles aussi des identificateurs, assignés à l'initialisation du catalogue. Le polymorphisme de ces méthodes génériques, dû non pas à l'héritage, mais à la généricité, est résolu statiquement en utilisant ces identificateurs. En fonction du type du paramètre de généricité (numérique, caractère, booléen, float ou ADT), un identificateur différent est associé, correspondant à une fonction différente. Lors de la compilation d'une requête ESQL comprenant un appel de méthode, le catalogue est interrogé pour vérifier que l'appel est correct et pour récupérer l'identificateur de la méthode. C'est ce dernier qui sera utilisé pour l'appel, il permet en effet de déterminer le tableau et l'entrée dans le tableau des méthodes qui contient l'adresse de la fonction correspondant à la méthode.

VII.2.3 Exécution et sécurité

Le fait de considérer les méthodes comme des routines du SGBD pose des problèmes de sécurité. Ce sont en fait des programmes écrits par l'utilisateur qui ne sont pas forcément sans erreurs. Elles s'exécutent dans le même espace d'adressage et une erreur peut donc avoir de graves conséquences.

Une erreur du style "division par zéro" occasionne un arrêt du système. Mais ce dernier dispose de procédures de reprises qui lui permette de repartir dans un état cohérent. L'inconvénient majeur est alors l'indisponibilité occasionnée. Une solution consiste à mettre en œuvre un mécanisme d'exception pour récupérer ce genre d'événement sans perturber le SGBD.

Les cas plus graves sont les anomalies du style "débordement de tableau" et autres "violation de mémoire" dues à l'utilisation de pointeurs. Dans ce cas les zones de données du SGBD peuvent être écrasées et des données incohérentes écrites sur

disque. Dans ce cas l'état de la base de données est altéré et difficile à restaurer car l'on ne sait pas, à priori, quand ces données ont été écrasées. Nous avons envisagé deux solutions à ce problème :

- La première a été évoquée dans le chapitre IV (en IV.3.1), et consiste à faire traiter l'exécution des méthodes par un serveur. Il n'est alors plus possible de faire travailler les méthodes dans l'espace du gestionnaire d'objets. De lourds mécanismes de copie d'objets et de communication inter-processus sont nécessaires.
- La deuxième solution consiste à fournir à l'utilisateur un "langage sûr". Un tel langage rend impossible l'occurence d'erreurs du type évoqué ci-dessus. La notion de pointeur n'existe pas, les débordements de tableaux sont contrôlés, etc..

La première solution semble trop pénalisante. Bien qu'aucune solution n'ait été mise en œuvre, nous proposerions d'étudier la deuxième. Cette étude n'a cependant pas été menée lors de ce travail de thèse.

VII.3 Compilation

La compilation des méthodes écrites en LPM se déroule en deux phases. La première est une phase de précompilation qui génère du LP. La deuxième est la phase de compilation du LP. Les modules objets des méthodes sont ensuite stockés dans une bibliothèque contenant toutes les méthodes utilisateurs.

La mise en œuvre d'un tel LPM est assez simple, puisqu'elle se limite au développement du précompilateur. De plus, la partie du précompilateur qui s'occupe du traitement des types abstraits est assez peu dépendante du langage cible LP. En effet, les opérations de contrôle de types abstraits, de validité des appels de méthodes sont assez semblables d'un langage LP à l'autre, puisque c'est la même extension qui est apportée pour obtenir le LPM ; seules les règles de conversion entre les types ESQL (ADT et types de base) et ceux du LP sont à reconsidérer. Par ailleurs, le coût de développement d'un précompilateur n'est pas très important ; il est en général possible de récupérer un analyseur syntaxique existant et de l'adapter. Les problèmes les plus sérieux concernent l'édition de liens multi-langage et les conversions de paramètres entre ESQL et le langage. Il semble donc possible d'envisager rapidement un environnement d'écriture de méthodes multi-langage.

VII.3.1 Précompilation

Le précompilateur s'occupe de toutes les références aux instances de types abstraits et des appels de méthodes présents dans le LPM afin de générer du LP pur. C'est-à-dire qu'il transforme toutes les manipulations de types abstraits en expression du langage LP. Dans ce premier prototype, le langage LP utilisé est C.

Le module de précompilation a été réalisé à partir d'un analyseur syntaxique C assorti d'un traducteur. Le travail de précompilation est réalisé sur l'arbre syntaxique généré par l'analyseur syntaxique. Il a lieu lors de la phase de décompilation de cet arbre dont le résultat est de ce fait un source C précompilé. Ce module est appelé par le moniteur du *gestionnaire de requêtes* (RM) lorsqu'un ordre LDD de compilation de méthode est donné.

La première étape consiste à analyser l'en-tête de la méthode à compiler. Un appel au catalogue permet de vérifier sa signature et de récupérer son identificateur. Ce dernier permet de juxtaposer au nom de la méthode le préfixe M<id_méthode>_. *M<id_méthode>_<nom_methode>* est en effet le nom symbolique qui permettra d'identifier la procédure C associée à une méthode. Ceci est indispensable pour discerner les méthodes de même nom qu'il est possible de définir sur des types différents [2].

Une autre phase consiste à vérifier que tous les types qui ne sont pas des types du langage LP sont des types abstraits et existent bien dans le catalogue des types que nous avons décrit dans le chapitre V. Chacun de ces ADT est remplacé par le type REF, qui est lui un type du langage LP. Conformément au choix pris en VII.2.1, ce type REF est une structure prévue pour contenir un identificateur d'objet du gestionnaire d'objets.

Le traitement des appels de méthode consiste à vérifier que la méthode est bien définie pour le type de la variable sur laquelle elle est appliquée. On met en pratique dans cette phase le mécanisme d'inférence de type du catalogue qui permet de déterminer exactement quelle méthode est à appliquer (récupération d'un identificateur de méthode). On vérifie ensuite que l'appel de méthode est bien conforme à la signature, puis on le remplace par un appel de procédure du langage LP, la procédure à appeler étant définie par l'identificateur de la méthode. Ce dernier point consiste simplement à préfixer le nom de la méthode par M<id_méthode>_. La méthode *augmenter_special* décrite plus haut deviendra après la phase de précompilation la procédure C suivante :

```
REF M284_augmenter_special (acteurs, taux)
REF acteurs;
int taux;
{
REF p;
int maxsal, sal;
maxsal = M132_salaire_max(acteurs);
p = First (&acteurs);
for ( ; p.tag != null; p = Next (&acteurs))
    {
    sal = NUM_GetField(p, 3);
```

2 On peut remarquer que *M<id_méthode>* serait un identificateur symbolique suffisant. Il est cependant intéressant de conserver le nom des méthodes, notamment pour des raisons de "déboguage", lorsqu'il faut analyser les sources LP générés.

```
        if (sal != maxsal)  M274_augmenter_sal(p, taux);
        }
    return acteurs;
    }
```

Le précompilateur effectue aussi un traitement sur les méthodes génériques. Il s'agit en particulier de générer un appel à la procédure C correspondant à la méthode travaillant sur le bon type de paramètre de généricité. Par exemple, la procédure correspondant à la fonction *GetField* (méthode définie sur un n-uplet, et qui renvoie la valeur d'un champ) qui retourne une valeur numérique s'appelle *NUM_GetField*. Le précompilateur se charge aussi de remplacer le nom du champ d'un n-uplet par son rang.

VII.3.2 Compilation

Il ne reste plus qu'à compiler le résultat de la précompilation avec le compilateur standard du langage LP et à archiver le module généré dans la librairie des méthodes. Les méthodes sont identifiées dans la librairie par le nom symbolique généré par le précompilateur (*M<id_méthode>_<nom_méthode>*).

VII.4 Edition de liens

La définition du corps d'une méthode fait partie du schéma d'une base de données relationnelle étendue. Lorsqu'une table relationnelle vient d'être créée par le concepteur du schéma, celui-ci peut insérer des n-uplets pour tester son application ; lorsque le code d'une méthode vient d'être défini et compilé, ce même concepteur doit pouvoir mettre en œuvre la méthode afin de la tester. Par conséquent, une fois compilées, les méthodes sont liées au SGBD, afin de pouvoir être exécutées lors de l'interprétation d'une requête ESQL.

L'utilisateur dispose d'une commande ESQL pour lancer la compilation et l'édition de lien d'une méthode. L'édition de lien a lieu dynamiquement dans un premier temps, juste après la compilation de la méthode. Cela permet à l'utilisateur qui vient de la compiler de la tester immédiatement. Toutefois cette liaison a lieu en mémoire et sera perdue au prochain arrêt du système. Des dispositions sont donc prises pour que la méthode soit liée statiquement au système à la prochaine génération de ce dernier (construction du module exécutable du SGBD). Cela veut dire que la méthode fera partie des routines du SGBD de façon permanente. Cette génération est déclenchée lors du redémarrage du système lorsque des méthodes ont été ajoutées ou modifiées lors de l'utilisation précédente.

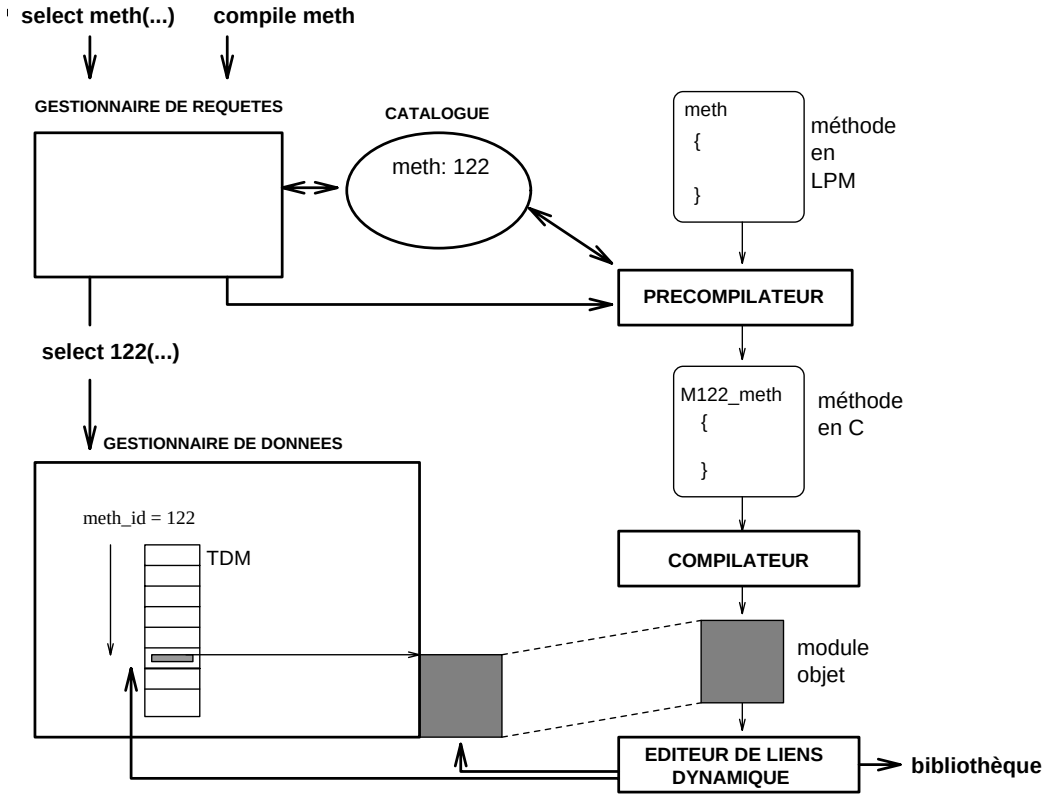

Fig. 7.1 : processus de compilation d'une méthode

Le mécanisme utilisé est décrit maintenant et se trouve schématisé dans la figure Fig. 7.1. La méthode est liée dynamiquement au *gestionnaire de données* d'une part et archivée dans une librairie d'autre part. La liaison dynamique consiste à charger le code de la méthode en mémoire et à inscrire l'adresse du point d'entrée de la méthode dans la table des méthodes[3] (`TDM`) du système, à l'entrée correspondant à l'identificateur de la méthode. Si la méthode est créée pour la première fois, une entrée dans la `TDM` est générée, à la fois dans la version de la `TDM` en mémoire principale, et dans celle en mémoire secondaire qui est utilisée lors de la génération du système. Nous appellerons `TDM_gen` cette deuxième `TDM` ; c'est la version permanente qui est utilisée pour créer la version initiale de la `TDM` dans le module exécutable du SGBD. L'entrée de la `TDM` en mémoire principale contient l'adresse du point d'entrée de la méthode ; celle de la `TDM_gen` en mémoire secondaire

3 Rappel de la section VII.2.2 : il y a en fait une TDM par type de retour de méthode.

contient le nom symbolique de la méthode (*M<id_meth>_<nom_meth>*). Le code de la méthode qui vient d'être compilée est inséré dans la librairie des méthodes (il remplace éventuellement une ancienne version de la même méthode). Cette librairie sera liée au *gestionnaire de données* à la prochaine génération de celui-ci. Les références à résoudre pour effectuer cette liaison des méthodes au système correspondent aux entrées de la `TDM_gen` utilisée pour la génération (la `TDM_gen`se présente en fait sous forme d'un tableau de fonctions qui est compilé et lié au SGBD). La méthode est donc liée statiquement lors de la génération du système.
Dans le code d'une méthode qui appelle une autre méthode, ainsi que dans la `TDM_gen`de génération, les références sont résolues par l'éditeur de liens au moyen du nom symbolique de la méthode M<id_meth>_<nom_meth>, alors que dans une requête ESQL, l'identificateur de celle-ci suffit à générer l'appel, car la liaison est déjà effectuée. Lorsque l'utilisateur recompile une méthode, déjà liée statiquement ou non, c'est toujours la dernière version, qu'il vient de compiler, qui est accessible à travers la `TDM`.

Nous n'avons pas de mécanisme de résolution tardive. Les relations étant strictement typées, la détermination du type des paramètres peut se faire à la compilation de la requête ESQL. La nécessité d'un tel mécanisme se fait sentir au niveau de l'exécution des méthodes mêmes et non pas des requêtes ESQL. En effet, si l'on peut à la compilation déterminer le type des paramètres d'une méthode appelée dans une requête, il n'est parfois pas possible de le faire pour une méthode appelée dans une autre méthode. Supposons que T2 soit un sous-type de T1, héritant des méthodes m1 et m2 définies sur T1. Supposons d'autre part que la méthode m2 soit redéfinie sur T2 (surcharge). Considérons maintenant la méthode m1 définie sur T1 de la façon suivante :

```
m1 (t,...)
T1 t;
{
...
a = m2 (t);
...
}
```

La méthode m2 est appelée dans le corps de m1. En fait, la variable *t* peut désigner non seulement des objets de type T1, mais aussi des objets de types qui héritent de T1 (par exemple T2). La méthode m2 à appeler doit être celle définie sur le type de l'objet effectif. Suivant que m1 est appelée sur une instance de T1 (sur lequel elle est définie) ou sur une instance de T2 (par lequel elle est héritée), la méthode m2 à appeler sera celle définie sur T1 ou celle surchargée par T2, et cela ne peut être su à la compilation de la méthode. Un mécanisme de résolution tardive permet de résoudre un tel problème, mais il ne constitue pas l'unique solution. Une deuxième possibilité consiste à recompiler et à faire l'édition de lien des méthodes

héritées au niveau de chaque sous-type. Ainsi lorsqu'une méthode héritée fait usage d'une autre méthode, qui, elle, a été redéfinie, c'est bien cette dernière qui sera liée ; en fait le code exécutable d'une méthode définie sur un type n'est pas utilisé pour les sous-types qui héritent cette méthode, il est généré pour chaque sous-type. On aurait dans notre cas deux modules exécutables correspondant à m1, un sur T1 et un sur T2 ; au premier est liée la version de m2 définie sur T1, à l'autre celle de m2 redéfinie sur T2. Cette solution nécessite des procédures de compilation lourdes et entraîne une duplication de code très importante qui devrait normalement se trouver réduite grâce au mécanisme d'héritage. Il existe finalement une solution au niveau du précompilateur qui peut dans un tel cas générer une instruction de *choix*, qui en fonction du type de *t*, fait appel à la méthode m2 correspondante. Dans le cas précédent, la précompilation de m1 donne le code suivant :

```
M231_m1 (t,...)
REF t;
{
...
switch (get_type (t))
  {
   case <id_T1>: a = M242_m2 (t);
   case <id_T2>: a = M252_m2 (t);
  }
...
}
```

On suppose que la méthode m2 définie sur T1 est identifiée par 242 (son nom symbolique est donc M242_m2), et que celle qui la surcharge sur T2 est identifiée par 252. Un tel mécanisme nécessite que le précompilateur détermine tous les sous-types du type d'une variable (en consultant le catalogue) afin de générer des instructions de *choix* pour les appels de méthodes sur ces variables (les choix possibles comprennent le type et tous ses sous-types). La contrainte la plus importante reste qu'il faut à l'exécution déterminer le type d'une variable. Nous introduisons pour cela la fonction **get_type**. Puisque tout objet est désigné par un OID, une solution consiste à introduire dans l'OID l'identificateur du type de l'objet désigné (cela a été prévu en VI.2.3 dans le chapitre VI). Aucune de ces solutions n'a encore été implantée, mais nous préférons cependant la dernière, qui est efficace au niveau exécution, et assez simple à réaliser.

VII.5 Conclusion

Nous avons montré une manière simple et efficace de développer un langage de programmation de méthodes pour ESQL. Elle présente l'avantage de travailler sur les objets dans le format de la base de données et laisse envisager le développement d'autres langages qui travailleront sur ce format commun. De même que ESQL a été développé à partir d'un langage relationnel existant, SQL, le langage LPM est une

extension d'un langage de programmation connu, ce qui facilite la phase d'apprentissage des programmeurs.

Cette approche est fortement dépendante du niveau du gestionnaire d'objets utilisé pour stocker les instances d'ADT. Le nôtre permet de stocker des objets structurés et de manipuler ces structures au niveau du gestionnaire lui-même. Ceci nous a permis d'adopter un format de données commun pour les langages de manipulation. De plus, ce format est intéressant puisque c'est celui du gestionnaire d'objets. Nous aurions pu adopter un gestionnaire d'objets de beaucoup plus bas niveau, pour lequel les objets ne sont que des chaînes d'octets sans aucune structuration, et où les seules opérations disponibles sont le chargement en mémoire et le déchargement d'un objet. Nous aurions alors pu décider de stocker les objets dans un format plat (séquentiel), interprétable au niveau du langage. Deux solutions se présentent alors :

- La première consiste à établir un format séquentiel standard, utilisable à partir de différents langages. Des méthodes génériques manipulant ce format ou des procédures de conversion dans un format LP sont nécessaires.
- La deuxième consiste à stocker directement les objets dans le format du langage LP, à condition que ce format corresponde à une zone de mémoire contiguë et relogeable. L'approche multi-langage serait alors beaucoup plus délicate, car à chaque langage correspondrait un format unique, et un objet ne pourrait alors être manipulé qu'à partir d'un seul langage.

Nous avons développé un mécanisme d'exécution de méthode adapté aux besoins de notre système. L'édition de liens dynamique proposée permet de tester facilement les nouvelles méthodes. Un mécanisme de résolution tardive pour l'appel de méthode au sein même du code des méthodes reste à mettre en œuvre. Plusieurs solutions ont été proposées. Il reste aussi à résoudre les problèmes de sécurité lors de l'exécution des méthodes au sein du SGBD.

Chapitre VIII

Conclusion

Ce travail de thèse se situe dans le domaine des Systèmes de Gestion de Bases de Données (SGBD) de nouvelle génération. Plusieurs directions ont été explorées pour aboutir aux successeurs des SGBD relationnels : extensions de systèmes relationnels, systèmes extensibles, SGBD orientés objets. Ce travail se situe dans la première catégorie. Nous avons voulu intégrer les concepts et la technologie **objet** dans un SGBD relationnel. Nous avons abordé les aspects modèle et langage pour ensuite nous intéresser aux problèmes de la mise en œuvre. Nous avons pour cela fait appel à des concepts et à des mécanismes que l'on retrouve dans les SGBD, les langages à objets, les systèmes d'exploitation et les SGBD orientés objets.

VIII.1 Résultats

VIII.1.1 Travail effectué

La première étape a consisté à définir un modèle intégrant les concepts relationnels et objets. Ceci fut fait en même temps que la définition du langage ESQL (Extended SQL). L'apport de mon travail fut ici de préciser la nature et la sémantique de l'extension objet. Les caractéristiques principales de cette extension sont : l'abstraction de données (introduction des types abstraits de données et encapsulation), l'héritage, les objets complexes et le partage d'objet. La solution adoptée repose sur la notion de *domaine* du modèle relationnel, jusque là limitée aux types de base du système. Nous avons étendu l'ensemble des *domaines* possibles pour un attribut avec la notion de type abstrait (ADT). Un ADT est un type défini par l'utilisateur comprenant une structure de données spécifiée à l'aide de constructeurs (n-uplet, ensemble, liste, tableau) et manipulable par un ensemble de méthodes. Les types définis peuvent être liés par une relation d'héritage. Certains confèrent à leurs instances la propriété d'être partageables. Ainsi un objet peut être partagé entre plusieurs n-uplets ou entre d'autres objets dont il est un composant. Cette dernière caractéristique est celle qui a la plus grande influence sur le modèle relationnel ; elle modifie la sémantique de certaines opérations relationnelles et augmente sensiblement la capacité de modélisation ; elle constitue un point fort de notre approche.

L'étude a ensuite porté sur la spécification et la réalisation des modules chargés d'assurer le support de cette extension objet. Ces modules furent intégrés dans un des prototypes du projet ESPRIT EDS. Il s'agit des trois composants suivants :

- Le **gestionnaire de types** est chargé de gérer les définitions d'ADT. Il s'occupe des relations d'héritage et fournit des fonctions de contrôle de types. Il est utilisé par le compilateur ESQL et par les compilateurs de méthodes.
- Le **gestionnaire d'objets** assure le stockage des instances d'ADT. Sa mise en œuvre fut précédée par une étude des systèmes de gestion d'objets existants. Cette étude a montré que les systèmes portant le nom de "gestionnaire d'objets" peuvent présenter différents niveaux de fonctions (stockage de chaînes d'octets, gestion d'objets structurés, gestion d'objets typés) et être construits sur des architectures différentes (mémoire à un ou deux niveaux de stockage). Notre module est construit autour du système GEODE. Il assure la persistance des instances d'ADT ainsi que la gestion des objets complexes (stockage et fonctions de manipulation). Le point fort de notre approche réside dans le fait que le seul format de manipulation des objets est celui du gestionnaire d'objets. Les fonctions de manipulation des structures de données ESQL (méthodes des constructeurs) constituent une partie de l'interface du gestionnaire d'objets. Différentes techniques de stockage d'objets complexes sont utilisées dans ce gestionnaire. Une politique de regroupement d'objets a été définie. La persistance des objets est assurée au moyen de compteurs de références. Nous avons réalisé des optimisations basées sur des techniques de *partage de valeurs* et de *copies sur mise à jour*.
- La **gestion des méthodes** comprend plusieurs aspects. Nous avons en premier lieu défini le langage qui doit être utilisé pour écrire les méthodes. Nous avons choisi d'étendre un langage existant (C dans un premier temps). Un compilateur a été développé pour ce langage (il s'agit en fait d'un précompilateur qui génère du code C pouvant être traité par le compilateur standard). Nous avons ensuite mis en œuvre un mécanisme d'exécution pour que le noyau relationnel utilisé dans le prototype puisse lancer l'exécution des méthodes. Un éditeur de liens dynamique a été intégré au prototype.

L'intégration de ces modules dans un prototype du système EDS a permis de valider le langage ESQL et a montré la faisabilité de l'intégration des techniques relationnelles et objets. L'intérêt d'un tel système est qu'il permet de conserver un SGBD compatible relationnel, ainsi qu'un langage déclaratif (Extended SQL).

La définition de quelques techniques d'optimisation a débuté mais aucune mise en œuvre n'a eu lieu. L'introduction de ces techniques touchera essentiellement les optimiseurs logique et physique du système, et vraisemblablement les mécanismes d'exécution relationnels.

Nous avons adopté une méthode de "couplage faible" entre le monde objet et relationnel, autant au niveau du modèle que du point de vue de la mise en œuvre. Les inconvénients majeurs sont la redondance des mécanismes et les problèmes d'optimisation.

Pour finir, la mise en œuvre de ce même système dans un environnement parallèle nous a permis de reconsidérer l'architecture et le niveau d'intégration de la technologie objet dans les mécanismes relationnels. La distribution des données sur les nœuds de la machine devra être prise en compte au niveau du gestionnaire d'objets. Le modèle d'exécution parallèle devra intégrer les contraintes liées aux objets.

VIII.1.2 Enseignement tiré

Nous avons constaté qu'il était possible d'intégrer dans le modèle relationnel les concepts d'un modèle à objets. La puissance apportée par l'identité d'objet permet de modéliser le partage d'objet de manière beaucoup plus simple et naturelle qu'avec un modèle relationnel pur (pas besoin de clé étrangère, granularité de partage réduite, ...). L'approche consistant à étendre les domaines permet de conserver un modèle (et par la suite un système) "compatible" relationnel. Cependant la faible intégration des concepts issus des deux modèles laisse apparaître pour l'utilisateur deux niveaux d'abstraction : les relations et les ADT. Cela peut poser des problèmes de modélisation (voir les perspectives à ce sujet en VIII.2.1).

La mise en œuvre du gestionnaire de types nous a permis d'aborder des problèmes assez classiques des systèmes à objets. Il s'agit notamment de la gestion de l'héritage et des modifications de schéma.

L'étude du stockage d'objets montre la nécessité d'offrir des techniques adaptées à la gestion d'objets complexes : structures de données spécialisées, identification d'objet, décomposition des objets, partage de valeurs, regroupement d'objets, etc.. Un tel gestionnaire d'objets offre en général une interface d'assez haut niveau permettant de gérer des objets structurés.

Notre solution concernant le langage d'écriture de méthodes et sa mise en œuvre a montré la faisabilité d'une approche multi-langage. Le format logique de représentation des données est commun (ce sont les constructeurs du modèle), tout comme le format physique, qui n'est autre que celui du gestionnaire d'objets. On montre ici un intérêt supplémentaire à employer un gestionnaire d'objets de haut niveau.

La prise en compte de l'optimisation a permis de mettre en évidence les lacunes du couplage faible, tant au niveau modèle que mise en œuvre.

Nous avons acquis une expertise sur la plupart des composants nécessaires à la mise en œuvre de tout système de gestion de données devant supporter un modèle à

objet. Nous avons en effet pu constater que le domaine d'application de la plupart des techniques employées ne se limite pas aux SGBD relationnels étendus.

VIII.2 Perspectives

Deux sortes de perspectives sont envisageables après un travail comme celui-ci. Les premières concernent l'extension objet du relationnel. Il s'agit d'aborder les points jusque là laissés de côté et de reconsidérer certains choix. L'autre classe de perpectives est beaucoup plus générale. Il s'agit cette fois de tirer profit de cette étude pour définir les éléments de base d'un système plus "universel" (langage, BD). Nous développons maintenant ces deux points.

VIII.2.1 Extensions du relationnel

Nous avons volontairement négligé les aspects transactionnels et concurrence d'accès : nous utilisons d'une part les mécanismes fournis par le noyau relationnel, et d'autre part ceux du gestionnaire d'objets GEODE. Il faut, dans le cas d'un couplage faible étudier la façon de faire coopérer ces deux mécanismes.

Il faudrait vraisemblablement envisager une approche plus intégrée entre les mécanismes relationnels et objets. Notre approche entraîne une redondance importante de composants logiciels et ne permet pas d'optimiser pleinement les requêtes qui traitent des objets. Cet inconvénient du couplage faible, révélé par l'étude de l'optimisation, remonte jusqu'au modèle lui-même, que nous allons aborder maintenant.

L'optimisation des requêtes ESQL est difficile du fait de la séparation entre les entités relationnelles (tables, n-uplets) et les entités du monde objet qui ne sont pas manipulées par les mêmes opérateurs. Ces deux niveaux d'abstraction peuvent aussi gêner le développeur d'application. Il peut être délicat de choisir entre une représentation relationnelle et une représentation objet pour une même entité. De même, le choix des traitements à implanter sous forme de méthode n'est pas évident. Une même application peut être développée de façon très "relationnelle" ou très "objet" avec ce genre de système. Deux perspectives de travail concernant ce problème semblent intéressantes : la première consiste à réviser le modèle de données pour aboutir à un résultat offrant un ensemble de concepts plus homogène ; la deuxième serait d'écrire une méthodologie de développement d'applications pour un tel modèle.

Finalement ESQL ne constitue pas un langage de programmation d'applications de bases de données complet. La deuxième version d'ESQL inclut des structures de contrôle (instructions d'itérations, instructions conditionnelles), mais la définition d'un tel langage n'est pas terminée et sa mise en œuvre sur un système comme EDS n'a pas encore été abordée.

VIII.2.2 Support de systèmes à objets

Les environnements de développement orientés objet offrent des capacités de traitement de type "Base de Données" (persistance, manipulations ensemblistes). Les SGBD relationnels étendus utilisent la notion d'objet. Et les SGBD orientés objet sont supposés fournir un environnement intégré (langage de programmation et gestion de BD) de développement d'applications. On peut dès lors se demander si ces systèmes ne vont pas converger vers un unique environnement de développement d'applications, intégrant tous les mécanismes de base et toutes les fonctions réunies. Nous avons pu dégager ici un ensemble de concepts et mécanismes de bases communs, nécessaires pour construire ces trois types de systèmes qui mêlent l'orientation objet à la gestion de bases de données. Si la construction d'un système "universel" (qui présente toutes les fonctions) n'est pas encore envisageable, il semble toutefois possible d'obtenir un système adaptable (ou extensible), qui peut, à moindre frais, être transformé pour s'adapter à un type d'application.

Les deux expériences acquises lors de ce travail, support BD pour un système à objets (GUIDE), et support objet pour un SGBD (EDS), laissent déjà apparaître certains mécanismes de base nécessaires à un système BD à objets. Le modèle à objets offre un ensemble de types dynamique qu'il faut supporter au moyen d'un **gestionnaire de types**. Les instances de ces types se présentent sous forme d'objets complexes pour lesquels il faut assurer la persistance et des accès efficaces : c'est le rôle d'un **gestionnaire de stockage d'objets**. Il faut aussi des mécanismes d'**exécution de méthodes** et d'**édition de liens dynamique**. Ces composants se trouveront nécessairement dans un système "universel" ou adaptable, et constituent donc un point de départ pour l'étude d'un environnement intégré de développement d'applications.

Bibliographie

[1] Serge Abiteboul, Peter Buneman, Claude Delobel, Richard Hull, Paris Kanellakis et Victor Vianu, ''New Hope on Data Models and Types: Report of an NSF-INRIA workshop'', *SIGMOD RECORD*, Vol. 19 (No 4), pp. 41-48 , décembre 1990.

[2] Serge Abiteboul et Stéphane Grumbach, ''Bases de données et objets structurés'', *Technique et Science Informatiques*, Vol. 6 (No 5), pp. 383-404 , 1987.

[3] Malcolm P. Atkinson et O. Peter Buneman, ''Types and Persistence in Database Programming Languages'', *ACM Computing Surveys*, Vol. 19 (No 2), pp. 105-190 , juin 1987.

[4] Timothy R. Ayers, Douglas K. Barry, John D. Dolejsi, Jeffrey R. Galarneau et Randal V. Zoeller, ''Development of ITASCA'', *Journal of Object-Oriented Programming*, Vol. 4 (No 4), pp. 46-49 , juillet/août 1991.

[5] Roland Balter, Jacques Bernadat, Dominique Decouchant, Andrzej Duda, André Freyssinet, Sacha Krakowiak, Marie Meysembourg, Patrick Le Dot, Hiep Nguyen Van, Eric Paire, Michel Riveill, Cécile Roisin, Xavier Rousset de Pina, Rodrigo Scioville et Gérard Vandôme, ''Architecture and Implementation of Guide, an Object-Oriented Distributed System'', *Computing Systems (Usenix)*, Vol. 4 (No 1), pp. 31-67, 1991.

[6] Roland Balter, Jacques Bernadat, Dominique Decouchant, Sacha Krakowiak, Michel Riveill et Xavier Rousset de Pina, *Modèle d'exécution du système Guide*, (Guide-R3), Bull-IMAG/Systèmes, 2, rue Vignate, 38610 Gières -France-, décembre 1987.

[7] Roland Balter, Sacha Krakowiak, Marie Meysembourg, Cécile Roisin, Xavier Rousset de Pina, Rodrigo Scioville et Gérard Vandôme, *Principes de conception du système d'exploitation réparti Guide*, (Guide-R1), Bull-IMAG/Systèmes , 2, rue Vignate, 38610 Gières - France -, avril 1987.

[8] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk et Nat Ballou, ''Data Model Issues for Object-Oriented Applications'', *ACM Transactions on Office Information Systems*, Vol. 5 (No 1), pp. 3-26, janvier 1987.

[9] Jay Banerjee et Won Kim, ''Semantics and Implementation of Schema Evolution in Object-Oriented Databases'', *Proceedings of ACM SIGMOD 1987*, édité par Umeshwar Dayal Irv Traiger, pp. 311-322 , San Francisco, mai 1987.

[10] Jean-François Barbé, François Exertier, Georges Gardarin et Samer Haj Houssain, ''*A Cooperative Approach to Extend Relational Technology with Object Oriented Capabilities* '', (DRPAG/OSI/91012), Direction des Etudes Europe - Direction Recherche & Programmes avancés Groupe, Bull-IMAG/Systèmes, 2, rue Vignate 38610 Gières - France -, avril 1991.

[11] D. S. Batory, ''Principles of Database Management System Extensibility'', *Database Engineering*, Vol. 10 (No 2), pp. 40-46 , juin 1987.

[12] Catriel Beeri, ''Formal Models for Object Oriented Databases'', *Proceedings DOOD89*, Department of Computer Science The Hebrew University Jerusalem 91904 Israel, décembre 1989.

[13] Véronique Benzaken, ''Un modèle d'évaluation de stratégies de regroupement dans un système de bases de données orienté objet'', *Sixièmes journées bases de données avancées BD3*, édité par INRIA, pp. 397-418 , Montpellier, septembre 1990.

[14] Marie Jo Bellosta-Tourtier, A. Bessède, Claude Darrieumerlou, Philippe Pucheral, Hermann Steffen et Jean Marc Thévenin, ''GEODE: concepts and facilities'', *Sixièmes jounées bases de données avancées BD3*, édité par INRIA, pp. 375-396 , Montpellier, septembre 1990.

[15] Björn Bergsten, Michel Couprié et Patrick Valduriez, ''DBS3, an Implementation of the EDS DBMS on a Shared-Memory Multiprocessor'', *PDIS à paraître*, 1991.

[16] Andrew Black, Norman Hutchinson, Eric Jul et Henry Levy, ''Object Structure in the Emerald System'', *SIGPLAN Notices (Proc. First ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon)*, Vol. 21(No 11), pp. 78-86 , novembre 1986.

[17] Luca Cardelli, ''A Semantics of Multiple Inheritance'', *Lecture Notes in Computer Science*, 1984.

[18] Luca Cardelli et Peter Wegner, ''On understanding types, data abstraction and polymorphism '', *ACM Computing Surveys*, Vol. 17 (No 4), pp. 471-522 , décembre 1985.

[19] Michael J. Carey et David J. Dewitt, ''An Overview of the EXODUS Project'', *Database Engineering - Bulletin of the Computer Society of the IEEE*, Vol. 10 (No 2), pp. 47-54 , juin 1987.

[20] Michael J. Carey , D. Frank , M. Muralikrishna , David Dewitt , G. Graefe , J.E. Richardson et Eugene J. Shekita, ''The Architecture of the EXODUS Extensible DBMS'', *Proceedings International Workshop on Object-Oriented Database Systems (Asilomar Conference Center Pacific Grove California)*, September 1986.

[21] Carla Chachaty, Pascale Borla-Salamet et Björn Bergsten, ''Parallel LERA, un langage pour l'exécution parallèle des requêtes relationnelles'', *Sixièmes journées Bases de Données Avancées BD3*, édité par INRIA, pp. 357-374 , Montpellier, septembre 1990.

[22] Sophie Cluet, *Langages et Optimisation de Requêtes pour Systèmes de Gestion de Base de Données Orientés-Objet*, Thèse de Docteur en Sciences, Université de Paris-Sud (Centre d'Orsay), juin 1991.

[23] George Copeland, Michael Franklin et Gerhard Weikum, ''Uniform Object Management'', *Lecture Notes in Computer Science (EDBT 90 Proceedings)*, (416), pp. 253-268 , 1990.

[24] O. Deux et al., *The Story of O2*, (Altaïr Technical Report 37-89), GIP Altaïr, BP 105, 78153 Le Chesnay Cedex, France, octobre 1989.

[25] Scott Danforth et Chris Tomlinson, ''Type Theories and Object-Oriented Programming '', *ACM Computing Surveys*, Vol. 20 (No 1), pp. 29-72, mars 1988.

[26] Andreas Elsholtz et Gerhard Müller, *Architecture of the NooDLE Database Interface* , (Work Package X4), Nixdorf Computer AG, Paderborn , janvier 1990.

[27] J. Eliot B Moss, ''Design of the Mneme Persistent Object Store'', *ACM Transactions on Information Systems*, Vol. 8 (No 2), pp. 103-139 , avril 1990.

[28] François Exertier, *Systèmes d'exploitation pour les systèmes de gestion de bases de données*, (Rapport de D.E.A.), USMG et INPG, Grenoble, 1987.

[29] François Exertier, *Collections dans GUIDE*, (Note interne No 57), Projet Guide, février 1990 .

[30] François Exertier et André Freyssinet, *Clusters*, (Note interne No 48), Projet Guide, août 1989.

[31] François Exertier et André Freyssinet, *Clusters*, (Working Paper Bull-Imag 0010), Esprit Project COMANDOS (2071), septembre 1989.

[32] François Exertier, *Clusters: Position Paper*, (Working Paper Bull-Imag 0022), Esprit Project COMANDOS (2071), mai 1990.

[33] François Exertier et Samer Haj Houssain, *Sharing Objects in Extended SQL*, (DRPAG/OSI/91010), Direction des Etudes Europe - Direction Recherche & Programmes avancés Groupe, Bull-IMAG/Systèmes, 2, rue Vignate 38610 Gières - France -, avril 1991.

[34] André Freyssinet, Rodrigo Scioville et Gérard Vandôme, *Gestion des objets persistants dans le système Guide*, (Guide-R4), Bull-IMAG/Systèmes, 2, rue Vignate, 38610 Gières - France -, décembre 1987.

[35] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce Lindsay, Hamid Pirahesh, Michael J. Carey et Eugene Shekita, ''Starburst Mid-Flight: As the Dust Clears'', *IEEE Transactions on knowledge and Data Engineering*, Vol. 2 (No 3), pp. 143-160 , mars 1990.

[36] Georges Gardarin, Jean Pierre Cheiney, Gerald Kiernan, Dominique Pastre et Hervé Stora, ''Managing Complex Objects in an Extensible Relational DBMS'', *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pp. 55-65 , août 1989.

[37] Georges Gardarin et Patrick Valduriez, ''ESQL: An Extended SQL with Object Oriented and Deductive Capabilities'', *1st DEXA Conference*, édité par Springer Verlag, Vienne, août 1990.

[38] Gœtz Graefe, David Maier, Scott Daniels et Tom Keller, ''A Software Architecture for Efficient Query Processing in Object-Oriented Database Systems with Encapsulated Behavior'', *Draft*, 1990.

[39] Sacha Krakowiak, Marie Meysembourg, Hiep Nguyen Van, Michel Riveill, Cécile Roisin et Xavier Rousset de Pina, ''Design and Implementation of an Object-Oriented Strongly Typed Language for Distributed Applications'', *Journal of Object-Oriented Programming*, Vol. 3 (No 3), pp. 11-22 , septembre/octobre 1990.

[40] Gérald Kiernan, *Intégration des types abstraits de données dans un SGBD relationnel déductif*, Thèse de doctorat, Université Pierre et Marie Curie - Paris VI, septembre 1989.

[41] R.S.G. Lanzelotte, Patrick Valduriez, Mikal Ziane et Jean-Pierre Cheiney, ''Optimization of Nonrecursive Queries in OODBs'', *Proceedings DOOD'91 à paraître dans Lecture Notes in Computer Science*, 1991.

[42] Christophe Lécluse et Philippe Richard, ''The O2 Database Programming Language'', *Proceedings of the 15th VLDB Conference*, Amsterdam, The Netherlands, August 1989.

[43] Christian Lenne, *EMIR : un outil de gestion de structures arborescentes attribuées*, BULL Direction des Etudes - Centre de Recherche Groupe - Division OSI, (Bull-IMAG/Systèmes, 2 rue Vignate, 38610 Gières -France -), septembre 1990.

[44] David Maier et Jacob Stein, ''Indexing in an Object-Oriented DBMS'', *IEEE*, pp. 171-182 , 1986.

[45] Jim Melton, Jonathan Bauer et Krishna Kulkarni, *Object ADTs (with improvements for Value ADTs)*, (Project Database Languages :ISO/IEC JTC1 SC21 WG3, Réf. X3H2-91-083rev2/ARL-029), International Organization for Standardization (ISO) - USA Expert Contribution, mai 1991.

[46] Hiep Nguyen Van, *Compilation et environnement d'exécution d'un langage à objets*, Thèse de Doctorat, Institut National Polytechnique de Grenoble, février 1991.

[47] Hiep Nguyen Van, Michel Riveill et Cécile Roisin, *Manuel du langage Guide (V1.5)*, (3-90), Bull-IMAG/Systèmes, 2, rue Vignate, 38610 Gières - France-, décembre 1990.

[48] Lawrence A Rowe et Michael R. Stonebraker, ''The POSTGRES Data Model'', *Readings in Object-Oriented Database Systems (edited by S.B. Zdonic and D. Mayer)*, pp. 461-473 , 1990.

[49] Hans-Jörg Schek et Marc H. Scholl, ''Evolution of Data Models'', *Lecture Notes in Computer Science "Database Systems of the 90s" International Symposium Berlin*, (No 466), pp. 135-151, novembre 1990 .

[50] Hans-Jörg Schek et Marc H. Scholl, ''The relational model with relation-valued attribute '', *Information Systems*, 11(2), pp. 137-147, juin 1986.

[51] Marc H. Scholl et Hans-Jörg Schek, ''A Relational Object Model'', décembre 1989.

[52] Dan Schuh, Michael Carey et David Dewitt, ''Persistence in E Revisited - Implementation Experience'', *Proc. of the 4th Intl. Workshop on Persistent Object Systems Design, Implementation and Use*, 1991.

[53] Shalom Tsur et Carlo Zaniolo, ''An Implementation of GEM - supporting a semantic data model on a relational back-end'', *SIGMOD'84*, 14(2), pp. 286-295 , juin 1984.

[54] Eugene Shekita et Michael Zwilling, ''Cricket: A Mapped, Persistent Object Store'', *Proc. of the 4th Intl. Workshop on Persistent Object Systems Design, Implementation and Use*, 1991.

[55] Eugène Shekita, *High-Performance Implementation Techniques For Next-Generation Database Systems*, thèse Ph.D., University of Wisconsin-Madison, 1990.

[56] M. Stonebraker, ‘‘Inclusion of new types in relational data base systems’’, *Proceedings of the 2nd Conference on Data Engineering, Los Angeles,* 1986.

[57] Dave D. Straube, *Queries and Query Processing in Object-Oriented Database Systems,* PhD Thesis, Université d’Alberta - Canada -, 1991.

[58] Nina Tayar, *Couplage fort entre un langage de programmation orienté objet et un SGBD relationnel étendu,* (Rapport de D.E.A.), INPG et Université Joseph Fourier, Grenoble, juin 1991.

[59] Patrick Valduriez, ‘‘Objets complexes dans les systèmes de bases de données relationnels’’, *Technique et Science Informatiques*, 6(5), pp. 405-418 , 1987.

[60] Fernando Vélez , Guy Bernard et V. Darnis, ‘‘The O2 Object Manager: an Overview’’, *Proceedings of the 15th VLDB Conference,* Amsterdam, The Netherlands, août 1989.

[61] P.F. Wilms , P.M. Schwarz , Hans-Jörg Schek et L.M. Haas, *Incorporating Data Types in an Extensible DataBase Architecture*, IBM, août 1988.

[62] Roberto Zicari, *A Framework for O2 Schema Updates*, (Altaïr 38-89), GIP Altaïr, Domaine de Voluceau BP 105 Rocquencourt 78153 Le Chesnay Cedex -France-, octobre 1989.

Chapitre I

Introduction

Chapitre II

Types, objets, et persistance dans les systèmes

Chapitre III

Modèle et langage pour la gestion d'objets dans les SGBD

Chapitre IV

Architecture et définition du support système du SGBD relationnel étendu

Chapitre V

Gestionnaire de types

Chapitre VI

Stockage d'objets

Chapitre VII

Gestion de méthodes

Chapitre VIII

Conclusion