LES BASES DE DONNEES ET LE PARADIGME DE L'ORIENTE OBJET

Support de cours

Réalisé par :
Dr. Atidel LAHOULOU

Département d'informatique
Faculté des sciences exactes et informatique
Université de Jijel

NOVEMBRE 2016

Préambule

Ce cours est destiné aux étudiants en 2ème année master informatique, spécialité Systèmes d'Information et Aide à la Décision (SIAD). Il leur est indispensable car il propose de nouvelles alternatives aux bases de données relationnelles précédemment abordées dans leur cursus.

L'intégration du paradigme orienté objet (OO) dans les bases de données (BD) constitue une phase importante et historique dans le développement des BD. Comme son titre l'indique, ce document est adressé aux étudiants ayant des connaissances préalables sur les bases de données relationnelles d'un côté et sur la programmation orientée objet de l'autre, indispensables pour une bonne assimilation du contenu du présent cours. En outre, quelques rappels sur les concepts prérequis sont injectés tout au long des chapitres chaque fois que cela s'avère nécessaire.

Ce document est constitué de deux parties qui représentent deux approches majeures dans l'incorporation de l'objet dans les BD : (1) les bases de données purement orientées objet et (2) les bases de données étendues relationnelles-objets.

A l'issue de la lecture de ce document, vous serez censé capable de répondre aux questions suivantes :

1. Pourquoi cherche-t-on des alternatives aux bases de données relationnelles ?
2. Pourquoi l'intégration de l'objet dans les BD est une solution très intéressante ?
3. Quelles sont les approches qui offrent la possibilité d'incorporer les concepts de l'objet dans les bases de données ?
4. Comment concevoir une base de données en explorant les deux approches ?
5. Comment interroger une base de données en explorant toujours les deux normes rivales ?

Plan du document

PARTIE I

LES BASES DE DONNEES ORIENTEES OBJET (BD-OO)

Chapitre 1

Pourquoi a-t-on besoin de nouvelles Technologies de bases de données ?

- ❖ Fonctionnalités des SGBD relationnels
- ❖ Applications avancées des bases de données
- ❖ Faiblesses des SGBD-R

1. Introduction

Devenues la plaque tournante de tout système d'information en entreprise, elles ont rapidement supplanté les systèmes de gestion de fichiers (SGF) de par leur simplicité, leur efficacité et leur facilité d'utilisation. Vous l'aurez certainement deviné, il s'agit bien des bases de données relationnelles. En effet, les SGF constituent un ensemble de programmes inflexibles et de données inaccessibles aux novices. Leur utilisation impose d'une part, à l'utilisateur de connaître l'organisation des fichiers qu'il utilise afin de pouvoir accéder aux informations dont il a besoin et, d'autre part, d'écrire des programmes pour pouvoir effectivement manipuler ces informations. Pour des applications nouvelles, l'utilisateur devra obligatoirement écrire de nouveaux programmes et il pourra être amené à créer de nouveaux fichiers qui contiendront peut-être des informations déjà présentes dans d'autres fichiers. De tels systèmes sont par conséquent rigides, contraignants et coûteux à mettre en œuvre.

La prise de décision, quant à elle, est une part importante de la vie d'une entreprise. Mais elle nécessite que l'on soit bien informé et donc que l'on ait des informations à jour et disponibles immédiatement. La technologie des bases de données rend les systèmes d'informations globaux, cohérents et directement accessibles. En tant qu'utilisateur du système d'information basé sur les bases de données, on n'a plus besoin ni d'écrire des programmes ni de demander à un programmeur de les écrire pour soit. On a des réponses immédiates aux questions que l'on se pose.

2. Rappels sur les fonctionnalités d'un SGBD idéal

Des objectifs principaux ont été fixés par les constructeurs des systèmes de gestion de bases de données (SGBD) dès l'origine de ceux-ci et ce, afin de résoudre les problèmes causés par la démarche classique basée sur les systèmes de gestion de fichiers tel qu'expliqué plus haut. Les fonctionnalités des SGBD-R visées sont les suivantes :

2.1 L'indépendance physique La façon dont les données sont définies doit être indépendante des structures de stockages utilisées.

2.2 L'indépendance logique Un même ensemble de données peut être vu différemment par des utilisateurs différents. Toutes ces visions personnelles des données doivent être intégrées dans une vision globale.

2.3 La manipulations des données par des non informaticiens Il faut pouvoir accéder aux données sans savoir programmer ce qui signifie l'utilisation de langages "quasi naturels".

2.4 L'efficacité des accès aux données Ces langages dits "de manipulation de données" doivent permettre d'obtenir des réponses aux interrogations en un temps raisonnable. Par conséquent, ces langages doivent être optimisés tout aussi comme le nombre d'accès disques et cela de façon complètement transparente pour l'utilisateur.

2.5 L'administration centralisée des données Des visions différentes des données (entre autres) se résolvent plus facilement si les données sont administrées de façon centralisée.

2.6 La non redondance des données Afin d'éviter les problèmes lors des mises à jour, chaque donnée ne doit être présente qu'une seule fois dans la base.

2.7 La cohérence des données Les données sont soumises à un certain nombre de contraintes d'intégrité qui définissent un état cohérent de la base. Elles doivent pouvoir être exprimées simplement et vérifiées automatiquement à chaque insertion, modification ou suppression des données.

2.8 La partageabilité des données Il s'agit de permettre à plusieurs utilisateurs d'accéder aux mêmes données au même moment. Si ce problème est simple à résoudre quand il s'agit uniquement d'interrogations et quand on est dans un contexte mono-utilisateur, cela n'est plus le cas quand il s'agit de modifications dans un contexte multiutilisateurs. Il s'agit alors de pouvoir :

- permettre à deux (ou plus) utilisateurs de modifier la même donnée "en même temps";
- assurer un résultat d'interrogation cohérent pour un utilisateur consultant une table pendant qu'un autre la modifie.

2.9 La sécurité des données Les données doivent pouvoir être protégées contre les accès non autorisés. Pour cela, il faut pouvoir associer à chaque utilisateur des droits d'accès aux données.

2.10 La résistance aux pannes Que se passe-t-il si une panne survient au milieu d'une modification ? Ou si certains fichiers contenant les données deviennent illisibles? Les pannes, bien qu'étant assez rares, se produisent quand même de temps en temps. Il faut pouvoir, lorsque l'une d'elles arrive, récupérer une base dans un état cohérent. Ainsi, après une panne intervenant au milieu d'une modification deux solutions sont possibles : soit récupérer les données dans l'état dans lequel elles étaient avant la modification, soit terminer l'opération interrompue.

3. Propriétés des SGBD relationnels

Malheureusement, les objectifs que nous venons d'évoquer ne sont pas toujours atteints. Néanmoins, les SGBD-R ont réussi à partager un certain nombre de propriétés très importantes et qui ont assuré, entre autres, leur fiabilité et leur longévité. Ces propriétés sont regroupées sous le terme ACID, l'acronyme de Atomicité, Cohérence, Isolation, Durabilité (*en Anglais : Atomicity, Consistency, Isolation, Durability*).

- **Atomicité :** La propriété d'atomicité assure qu'une transaction se fait au complet ou pas du tout : si une partie d'une transaction ne peut être faite, il faut annuler toute trace de la transaction et remettre les données dans l'état où elles étaient avant la transaction. L'atomicité doit être respectée dans toutes situations, comme une panne d'électricité, une défaillance de l'ordinateur, ou une panne d'un disque.
- **Cohérence** : La propriété de cohérence assure que chaque transaction amènera le système d'un état valide à un autre état valide. Tout changement à la base de données doit être valide selon toutes les règles définies, incluant mais non limitées aux contraintes d'intégrité, aux déclencheurs de base de données (en Anglais Triggers), et à toutes combinaisons d'évènements.
- **Isolation :** Toute transaction doit s'exécuter comme si elle était la seule sur le système. Aucune dépendance possible entre les transactions. La propriété d'isolation assure que l'exécution simultanée de transactions produit le même état que celui qui serait obtenu par l'exécution en série des transactions. Chaque transaction doit s'exécuter en isolation totale : si T1 et T2 s'exécutent simultanément, alors chacune doit demeurer indépendante de l'autre.
- **Durabilité** : La propriété de durabilité assure que lorsqu'une transaction a été confirmée, elle demeure enregistrée même à la suite d'une panne d'électricité, d'une panne de l'ordinateur ou d'un autre problème. Par exemple, dans une base de données relationnelle, lorsqu'un groupe d'énoncés SQL ont été exécutés, les résultats doivent être enregistrés de façon permanente, même dans le cas d'une panne immédiatement après l'exécution des énoncés.

4. Applications avancées des bases de données [1]

La dernière décennie a vu des changements de fond dans l'industrie informatique. Parmi les systèmes de bases de données, nous avons constaté l'acceptation généralisée des SGBD-R dans le cas des applications commerciales traditionnelles, telles que la gestion des commandes, la gestion d'inventaire, les applications bancaires et les réservations de billets de transport ferroviaire ou aérien. Cependant, les SGBD-R existants se sont montrés inappropriés pour les applications dont les besoins diffèrent considérablement des applications de bases de données commerciales traditionnelles. Ces applications sont, entre autres :

4.1 La conception assistée par ordinateur (CAO)

Une base de données de CAO emmagasine des données de conception relatives aux domaines de la mécanique, de l'électricité et de l'électronique, couvrant des éléments tels que les constructions de génie civil, des navires, des avions et des puces électroniques. Tous ces types de conceptions ont certaines propriétés communes:

- La conception se caractérise par un grand nombre de types, comportant chacun un nombre réduit d'instances. Les bases de données conventionnelles sont généralement à l'opposé. Par exemple, une base de données peut ne comporter qu'une petite dizaine de relations, alors que ses relations contiennent des milliers de tuples.
- Le résultat de la conception peut être gigantesque, comportant jusqu'à des millions de pièces, dont la plupart résultent de la conception de sous-systèmes interdépendants.
- La conception n'a rien de statique mais évolue avec le temps. Lorsqu'une modification se produit dans le design, ses implications se propagent tout au long des représentations qui font partie de la conception. La nature dynamique de la conception peut provoquer l'apparition d'actions qui n'ont pas été prévues au départ du projet.
- Souvent, les concepteurs ne se contentent pas d'une seule solution mais en envisagent plusieurs pour chaque composant, le système doit donc mémoriser

chacune des versions. Ceci implique une certaine forme de gestion des versions et de gestion de configurations.

- Les projets font intervenir souvent des centaines de personnes au stade de la conception, qui parfois travaillent en parallèle sur plusieurs versions d'un vaste design. Même ainsi, le produit final doit garder toute sa cohérence et nécessite une grande coordination. C'est ce que l'on désigne parfois d'ingénierie collaborative.

4.2 Fabrication assistée par ordinateur (FAO)

Une base de données de FAO emmagasine des données semblables à celles d'un système de CAO, en plus de données discrètes (des valeurs finies, sous forme de mesures physiques) de production, comme les caractéristiques des voitures dans leur chaîne de production, et de données de production continue. Dans une usine chimique, par exemple, une application affiche les informations concernant l'état du système, comme les températures dans une cuve de réaction, les débits des flux et les rendements des réactions. D'autres applications contrôlent les différents processus physiques tels que l'ouverture des vannes, l'augmentation de température des cuves de réaction et l'augmentation du flux des systèmes de refroidissement. Ces applications sont souvent disposées en une hiérarchie, dont l'application du sommet surveille la totalité de l'usine et les applications de bas niveau surveillent individuellement les différents processus.

Ces applications doivent être capables de réagir vite et d'ajuster les processus pour maintenir les performances à leur optimum dans des tolérances très faibles. Dans cet exemple, le système doit gérer de vastes volumes de données de nature hiérarchique et entretenir des associations complexes entre les données. Il doit également naviguer très rapidement parmi les données à consulter et répondre aux changements.

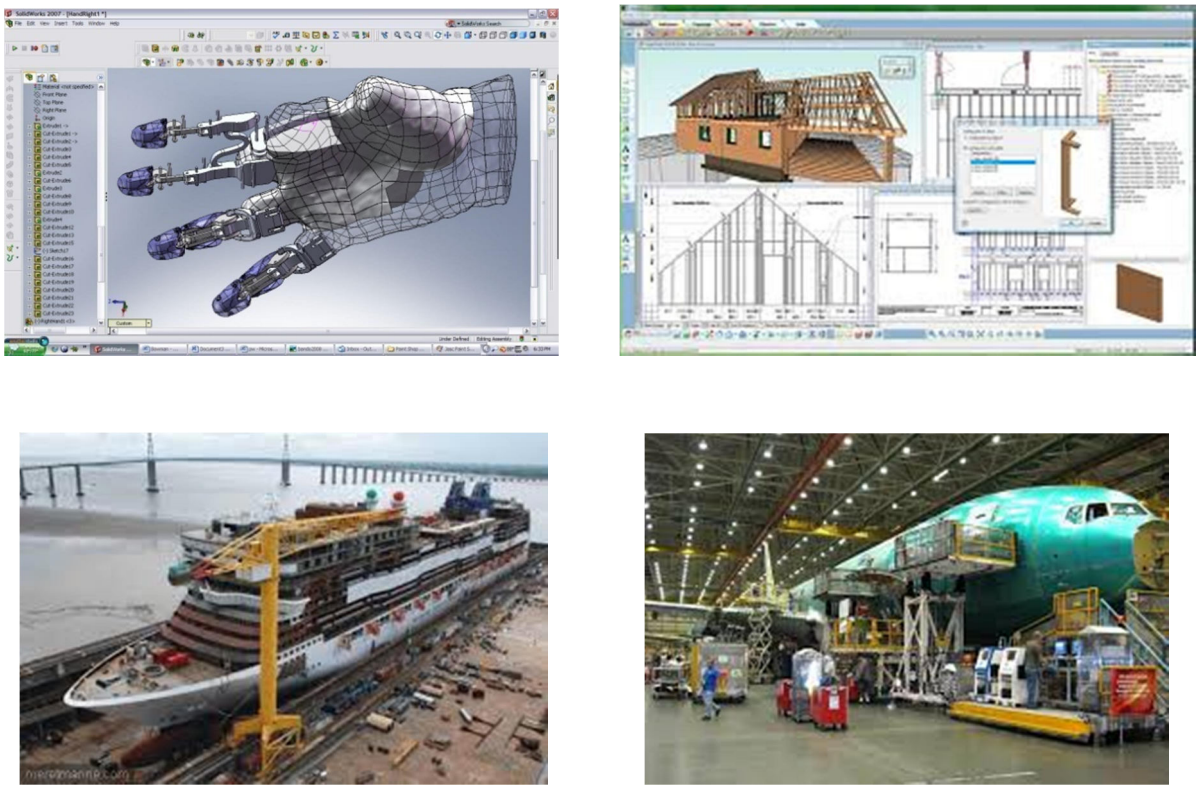

Fig 1. Exemples de systèmes de CAO (en haut) et de FAO (en bas)

4.3 Génie logiciel assisté par ordinateur (GLAO)

Une base de données GLAO stocke des informations collectées durant les phases du cycle de développement d'un logiciel :

- la planification,
- la collecte des exigences et leur analyse,
- la conception,
- l'implémentation,
- les tests,
- la maintenance et la documentation.

Comme dans le cas de la CAO, les designs peuvent atteindre une grande ampleur et l'ingénierie collaborative est la norme. Par exemple, les outils de gestion de configuration de logiciel facilitent le partage de la conception, du code et de la documentation du projet. Les outils de gestion de projet facilitent la coordination de diverses activités de gestion des projets, telles que la planification de tâches indépendantes et potentiellement complexes, l'évaluation des coûts et le suivi de la progression des projets. Des exemples concrets de

systèmes nécessitant le GLAO : les systèmes d'exploitation et les moteurs de recherche dont le développement requiert des années de travail et des équipes de centaines de personnes de plusieurs spécialités et qui doivent travailler en étroite collaboration.

4.4 Systèmes de bureautique et systèmes multimédias

La base de données des systèmes de bureautique et multimédias emmagasine toutes les données relatives au contrôle électronique des informations d'une activité commerciale : le courrier électronique, les documents, les factures et ainsi de suite. Pour offrir un meilleur soutien dans ce domaine, nous devons gérer une gamme de plus en plus étendue de types de données autres que des noms, des adresses, des dates et des prix. Les ordinateurs de bureau modernes gèrent des textes de forme libre, des photographies, des diagrammes, ainsi que des séquences audio et vidéo. Un document multimédia peut contenir du texte et des photos mais aussi des extraits de feuilles de tableur et des commentaires vocaux.

Les documents ont éventuellement une structure imposée, décrite à l'aide d'un langage de balisage tel que le SGML (*Standardized Generalized Markup Language*), le HTML (*Hypertext Markup Language*) ou le XML (*eXtended Markup Language*).

Les documents se partagent aujourd'hui parmi plusieurs utilisateurs grâce à des systèmes tels que le courrier électronique et les bulletins de nouvelles qui empruntent les technologies d'Internet. De telles applications demandent de stocker des données d'une structure plus riche que des tuples formés de nombres et de chaînes de caractères. Le besoin de saisir des notes manuscrites se fait également de plus en plus sentir, empruntant souvent les services des appareils électroniques portatifs. Bien qu'il soit quelquefois possible de transcrire en ASCII du texte manuscrit, grâce à des techniques de reconnaissance d'écriture, toutes les données de ce genre ne sont pas convertibles car, en plus du texte, les notes sont souvent agrémentées de petits schémas, de diagrammes, de symboles, etc.

Nous pourrions trouver parmi les exigences des utilisateurs le besoin de traiter des données multimédia :

- ✓ Des données de type image : Un client interroge une banque de données photographiques contenant les photos des propriétés à louer. La plupart du temps, les requêtes concernent la description textuelle des propriétés mais, la sélection opérée, des images apportent un léger « plus » qui devient de plus en plus indispensable dans ce genre d'application. Ainsi, les logements disposant souvent de

caractéristiques propres qui titillent l'envie des candidats locataires, ceux-ci apprécieront de voir une photo par exemple de la vue offerte par un salon, l'équipement de la cuisine ou le charme d'une terrasse en toiture de telle propriété ou de telle autre.

- ✓ Des données de type vidéo : Le client peut consulter une banque de séquences vidéo qui décrivent les logements à louer. Certaines requêtes empruntent une description textuelle pour identifier les vidéos des propriétés recherchées, mais dans d'autres cas, les clients apprécient de consulter les caractéristiques en vidéo des logements, comme la vue sur la mer ou sur les collines environnantes.
- ✓ Des données audio : Si les requêtes sont essentiellement textuelles, l'apport de commentaires vocaux aidera sans aucun doute le candidat locataire malvoyant et, d'une manière générale, quiconque souhaite juger du niveau de calme offert par un logement déterminé appréciera de disposer d'une séquence sonore reproduisant les bruits environnants de la propriété.
- ✓ Les données manuscrites : Un membre du personnel qui inspecte les logements appréciera de prendre quelques notes à la volée sur un assistant numérique personnel. Quelques temps après, il souhaitera interroger de telles données pour retrouver les notes qu'il a écrites à propos de l'odeur de moisissure constatée dans un appartement qu'il s'apprête à mettre en location.

4.5 Publication assistée par ordinateur (PAO)

L'industrie de l'édition électronique a subi des modifications fondamentales depuis une vingtaine d'années et est susceptible de subir de véritables révolutions. Il est aujourd'hui possible de mémoriser des livres, des périodiques, des articles sur support électronique et de les livrer par des réseaux ultrarapides directement chez les lecteurs. Comme les applications de bureautique, la PAO gère des documents multimédias constitués de texte, d'audio, d'images statiques ou animées et de vidéo. Dans certains cas, la quantité d'informations mises en ligne peut être gigantesque, de l'ordre du péta-octet (10^{15} octets soit un million de milliards), ce qui donne lieu aux plus vastes bases de données qu'un SGBD ait jamais eu à gérer.

4.6 Cartographie numérique et systèmes d'informations géographiques (SIG)

La cartographie numérique et toutes les applications fondées sur des données géographiques (notamment de positionnement par satellite) gèrent toutes sortes d'informations spatiales et temporelles, telles que celles qu'utilisent l'administration de l'occupation des sols et l'exploration sous-marine. La plupart des données de ces systèmes proviennent de relevés géométriques et de photographies par satellite. Par conséquent, les données sont également très volumineuses. Les requêtes se basent sur des caractéristiques d'identification, soit, par exemple, des formes, des couleurs ou des textures, et empruntent des techniques numériques de reconnaissance de motifs.

Ainsi, la NASA a lancé dans les années 1990 un programme baptisé EOS (*Earth Observing System*) et constitué d'une série de satellites, pour collecter des informations destinées aux scientifiques concernés par les tendances à long terme qui caractérisent l'atmosphère, les océans et les sols.

La taille gigantesque de cette base de données et la nécessité d'offrir de gros volumes de données à des milliers d'utilisateurs constituent des défis remarquables posés aux SGBD.

Fig 2. Exemples de SIG (à gauche) et de systèmes cartographiques (à droite)

4.7 Sites Web interactifs et dynamiques

L'exemple le plus trivial est celui d'un site Web proposant en ligne un catalogue de vente de vêtements. Le site Web conserve une série de préférences affichées par les visiteurs précédents et permet à un visiteur de :

- Parcourir une liste d'images des articles du catalogue et, d'un clic de souris, de voir une image agrandie d'un article particulier, avec une description et des détails supplémentaires.

- Rechercher les articles qui correspondent à une série de critères émis par le visiteur.
- Visualiser un article d'habillement (la couleur, la taille, le type de vêtement, etc.).
- Choisir des accessoires qui vont avec l'article, sélectionnables dans une liste.
- Accéder à des commentaires vocaux donnant des détails supplémentaires sur les articles.
- Consulter le total de la facture et les remises appropriées.
- Conclure l'achat et le régler par transaction sécurisée en ligne.

Fig 3. Exemple d'un site web interactif et dynamique

Les exigences de ce type d'application ne diffèrent pas vraiment de certaines des applications plus techniques citées plus haut : le besoin de gérer un contenu multimédia (du texte, de l'audio, de la vidéo, des images statiques et animées) et de modifier de manière interactive l'affichage en fonction des préférences et des choix de l'utilisateur. Dans cette situation, la base de données ne se limite pas à simplement présenter les informations au visiteur, mais s'engage activement dans la vente, en offrant dynamiquement tout les détails souhaités à l'éventuel acheteur.

4.8 Autres applications

D'autres applications complexes de bases de données existent. En voici quelques exemples :

1. Les applications scientifiques et médicales emmagasinent et traitent des données qui représentent des systèmes tels que les modèles moléculaires des composés chimiques de synthèse et le matériel génétique.
2. Les systèmes experts sont capables d'acquérir la connaissance et d'assimiler des règles de base des applications d'intelligence artificielle.

3. On ne compte plus aujourd'hui les applications qui exploitent des objets complexes reliés et des données procédurales.

5. Faiblesses des SGBD-R [1]

La logique des prédicats de premier ordre a conduit au développement de SQL, un langage déclaratif devenu la norme en matière de définition et de manipulation des bases de données relationnelles. Les SGBD-R possèdent en plus des propriétés ACID (expliquées en section 2), les forces du modèle relationnel qui se situe au niveau de sa simplicité, son adéquation au traitement de transactions en ligne et son apport à l'indépendance des données. Malheureusement, le modèle de données relationnel et, surtout, les SGBD-R présentent aussi des inconvénients parmi lesquels, on cite :

5.1 Piètre représentation des entités du monde réel

Le processus de normalisation conduit généralement à la création de relations qui ne correspondent pas aux entités du monde réel. La fragmentation des entités réelles en plusieurs relations, avec une représentation physique qui reflète cette structure, est peu efficace et entraîne l'obligation d'effectuer toute une série de jointures lors du traitement des requêtes. Or, nous savons que la jointure est une des opérations les plus coûteuses à effectuer.

5.2 Surcharge sémantique

Le modèle relationnel ne possède qu'une seule construction pour représenter les données et les associations entre les données : **la relation**. Par exemple, pour représenter une association de plusieurs à plusieurs (*:*) parmi les deux entités A et B, nous devons créer trois relations, une par entité et une supplémentaire pour représenter l'association. Aucun mécanisme n'existe en outre qui permette de distinguer les entités des associations, ni même de distinguer les différentes sortes d'associations qui existent entre les entités (les associations binaires, ternaires, n-aires, réflexives). Par exemple, une relation 1:* pourrait s'appeler Possède, Emploie, Administre et ainsi de suite. S'il était possible d'opérer de telles distinctions, nous pourrions intégrer la sémantique au sein des opérations. C'est la raison pour laquelle le modèle relationnel est qualifié de sémantiquement surchargé.

L'histoire a vu de nombreuses tentatives de contournement de ce problème, par la mise en place, entre 1987 et 1988 de modèles de données sémantiques, c'est-à-dire des modèles qui tentent de représenter la signification des données. Ceci dit, le modèle relationnel n'est pas totalement dépourvu de sémantique, puisqu'il dispose par exemple de domaines et de clés, ainsi que de dépendances fonctionnelles et de jointure.

5.3 Structure de données trop homogène

Le modèle relationnel part de l'hypothèse de l'homogénéité tant horizontale que verticale. L'homogénéité horizontale signifie que tous les tuples d'une relation sont composés des mêmes attributs. L'homogénéité verticale signifie que les valeurs d'une colonne déterminée d'une relation donnée doivent obligatoirement venir d'un seul et même domaine. En outre, l'intersection entre une ligne et une colonne est impérativement une valeur atomique. Or, cette structure est trop restrictive pour s'adapter à nombre d'objets du monde réel qui présentent une structure beaucoup plus complexe. Ceci a pour conséquence l'obligation de mettre en œuvre des jointures non naturelles qui sont très inefficaces. Pour étayer la défense du modèle relationnel, nous devons toutefois reconnaître que la symétrie de sa structure constitue précisément une de ses forces.

Nombre de SGBDR autorisent aujourd'hui le stockage d'objets binaires de grande taille (BLOB, *Binary Large Object*). Le BLOB est une valeur de donnée qui contient des informations binaires représentant une image, une vidéo ou une séquence audio numérisées, une procédure ou tout autre objet vaste et non structuré. Le SGBD n'a aucune connaissance de ce que signifie le contenu du BLOB, ni de sa structure interne, ce qui a pour effet d'interdire au SGBD d'effectuer la moindre requête ou opération sur des types de données à priori riches et structurés. Normalement, la base de données ne gère pas directement ce genre d'information mais retient simplement une référence à un fichier. L'usage des BLOB n'est pas une solution très élégante et le stockage réel dans des fichiers externes déjoue nombre des mécanismes de protection qu'offre le SGBD. Plus important, les BLOB ne peuvent contenir d'autres BLOB, de sorte qu'un type de donnée de ce genre ne peut renfermer d'autre objet composite. En outre, le BLOB ignore généralement les aspects comportementaux des objets. Par exemple, un SGBD relationnel peut enregistrer dans un BLOB d'une quelconque relation une image mais l'image ne peut y être qu'enregistrée ou

lue; il n'est pas possible de manipuler la structure interne de l'image, ni d'afficher ou manipuler des parties de cette image.

5.4 Nombre assez limité d'opérations

Le modèle relationnel ne dispose que d'un jeu figé d'opérations, relatives aux ensembles ou aux tuples et fournies par les spécifications techniques de SQL. SQL ne permet pas de spécifier de nouvelles opérations. En plus, le jeu d'instructions SQL est trop restreint pour modéliser le comportement de points, des lignes, des groupes de lignes, des polygones et des surfaces; elle nécessite également des opérations spécifiques pour calculer des distances, des intersections ou des contenus.

5.5 Dichotomie d'impédance

La dichotomie d'impédance veut dire un mélange de plusieurs paradigmes différents de la programmation quand il s'agit du langage utilisé pour exprimer les requêtes (SQL pour le relationnel) et le langage d'interfaçage qui permet aux utilisateurs d'interagir aisément avec la base de données. Ce mélange peut induire des incompatibilités au niveau de la programmation :

- SQL est un langage déclaratif qui gère des lignes de données, tandis qu'un langage de programmation de haut niveau tel que C est un langage procédural qui ne peut gérer à la fois qu'une seule ligne de données.
- SQL et les L3G (langages de programmation de 3ème génération) font appel à des modèles différents pour représenter les données. Par exemple, SQL propose les types prédéfinis Date et Interval, inexistants dans les langages de programmation traditionnels. De ce fait, le programme d'application se doit d'assurer la conversion parmi les deux représentations, impliquant autant d'efforts de programmation peu rentables. Une estimation établit que près de 30 % des efforts et du code de programmation sont gaspillés à gérer ce genre de conversion. Pire encore, l'utilisation de deux systèmes de types différents interdit de vérifier les types au sein de l'application dans sa globalité. En réponse à cette remarque, il a été avancé que la solution à ces problèmes ne passe pas par le remplacement des langages relationnels par des langages orientés objet et opérant au niveau des lignes, mais plutôt à introduire des fonctionnalités de gestion des ensembles dans les langages de

programmation. Néanmoins, le but fondamental des SGBD-OO est de fournir une intégration tout en douceur du modèle de données des SGBD et des langages de programmation hôtes.

5.6 Autres problèmes des SGBD-R

1. Les transactions dans la gestion commerciale ont une durée de vie plutôt réduite; les primitives et les protocoles de contrôle de la concurrence tels que le verrouillage en deux phases ne sont pas particulièrement adaptés aux transactions de longue durée, qui constituent la majorité de celles rencontrées dans les objets utilisés dans les conceptions complexes.
2. Les modifications des schémas présentent des difficultés. Les administrateurs de base de données doivent intervenir pour modifier les structures de base de données et, normalement, les programmes qui accèdent à ces structures doivent subir des modifications pour s'adapter aux nouvelles structures. Toutes ces procédures sont lourdes et complexes, même avec les technologies actuelles. Par conséquent, les entreprises se retrouvent enfermées dans leurs structures de bases de données existantes et, même si elles souhaitent et sont capables de changer la manière dont elles conduisent leurs affaires pour respecter de nouvelles exigences, elles n'ont pas la possibilité de concrétiser ces changements, tout simplement parce qu'elles n'ont ni le temps ni les moyens de modifier leurs systèmes d'information. Pour respecter l'exigence d'une plus grande souplesse, nous avons besoin d'un système qui prend en charge l'évolution du schéma de façon naturelle.

Il est important de noter que ces problèmes s'appliquent à la plupart des SGBD et non seulement aux systèmes relationnels.

6. Conclusion

Il existe deux propositions rivales de normes permettant l'intégration du paradigme de l'orienté objet dans les bases de données :

- **ODMG**, défini en 1993, est une spécification d'interfaces pour SGBDO "purs". ODMG rassemble, entre autres, un groupe de constructeurs de SGBDO qui se sont engagés à respecter ces spécifications dans leurs produits.
- **SQL3** est une norme pour les SGBD relationnels-objets, définie par le comité international de normalisation ISO. C'est une extension du SQL relationnel aux fonctionnalités orientées objet. Cette extension est compatible avec le SQL conventionnel, c'est-à-dire que les bases de données relationnelles traditionnelles peuvent être gérées par SQL3 et que toute requête SQL classique peut s'exécuter sur un des nouveaux systèmes offrant SQL3.

  SQL3 est offert par les dernières versions des SGBD relationnels des grands constructeurs comme IBM, Oracle ou Sybase.

Nous consacrons les deux parties de ce document à chacune de ces deux alternatives, respectivement.

Chapitre 2

Fonctionnalités des SGBD orientés objet (SGBD-OO)

- Evolution des SGBD
- Le paradigme de l'orienté objet
- Puissance des SGBD-OO
- Stratégies de développement des SGBD-OO

1. Introduction

Les bases de données orientées objets (BD-OO) cherchent à répondre à de nouveaux besoins et à de nouvelles technologies visuelles que les bases de données conventionnelles ont échoué à intégrer. Les BD-OO sont nées de la convergence de deux domaines, à savoir les bases de données, qui ont connu un essor considérable pendant de longues années et les langages de programmation orientés objets, tels que Eiffel, Smalltalk, C++, Java… dont l'objectif est d'accroître la productivité des développeurs en permettant de créer des logiciels structurés, extensibles, réutilisables et de maintenance aisée. Leurs principes essentiels

Dans ce chapitre, nous expliquons comment cette convergence a été négociée et quelles étaient les mutations technologiques qui ont permis de développer ces schémas.

2. Evolution des SGBD

La technologie des bases de données a connu une évolution plus ou moins rapide depuis les années 50. En voici un aperçu :

Années 50' – 60'

• Systèmes de Gestion de Fichiers (SGF) et méthodes d'accès : séquentiel, direct et séquentiel indexé.

• Systèmes IDS 1 et IMS 1 précurseurs des SGBD modernes

Années 62 - 63

• Apparition du concept de Base de Données

Années 65 - 70

• Conception des premiers SGBD (1ère Génération)

• Séparation de la description des données de la manipulation de celles-ci par les applications

• Modèle hiérarchique et modèle réseau :

- SGBD IMS2 d'IBM (modèle hiérarchique),
- SGBD IDS2 de General Electric (modèle réseau) qui a servi de modèle de base aux propositions du groupe CODASYL.

• Langages d'accès navigationnels : on ne pouvait pas interroger une base sans savoir où était l'information recherchée (donc on "naviguait" sans avoir à écrire de programmes).

Années 70 - 85

• 2ème Génération des SGBD organisés sur le modèle relationnel fondé sur la théorie des relations, elle-même formalisée par CODD en 1970

• Le modèle relationnel spécifie davantage de moyens d'accès aux données

• Structure physique cachée aux utilisateurs

• Développement des langages déclaratifs

• Architecture répartie client / serveur

• Systèmes commercialisés dans les années 1980 :

- MRDS de Honeywell diffusé par CII-HB,
- QBE (Query By Example),
- SQL/IDS d'IBM,
- INGRES de Relational Technology,
- ORACLE de Relational Software.

Années 85 - 90

• Les systèmes de gestion de bases de données relationnels dominent le marché.

• Formalisation des modèles sémantiques afin de surmonter la surcharge sémantique du modèle relationnel (voir explication à la section 5.2)

• Les modèles sémantiques offrent une meilleure représentation du réel

• Cependant, ils en sont restés aux outils de conception uniquement, pas de systèmes commercialisés

Année 1986

• Modèles de données plus riches : premiers SGBD orientés objet (3ème génération)

• Meilleure représentation du réel au niveau logique

• Réutilisation

• Examples : OBJECTSTORE, O2

Année 1993

• Première norme ODMG (Object Database Management Group) pour SGBD-OO

Année 1998

• Norme UML pour la conception d'applications OO

Année 1999

• Norme SQL3 pour SGBD relationnel-objet

Aujourd'hui

• Données plus variées (textes, sons, images, parole, ..),

- Bases de Données réparties,
- Bases de Données orientées objets,
- Bases de Connaissances et Systèmes Experts,
- Bases de données déductives,
- Génie Logiciel et SGBD,
- Bases de données mobiles,
- Bases de données NOSQL (Not Only SQL)
- Accès intelligent multimodal et naturel (langage naturel écrit, graphique, parole, etc.).

Résumé des principaux systèmes

SGBDs	BD intégrées	Sharewares
Oracle DB2 (IBM) Ingres Informix Sybase SQL Server (Microsoft) O2 Gemstone …….	Access Paradox FoxPro 4D Windev …….	MySQL MSQL Postgres InstantDB …….

3. Rappels sur le paradigme orienté objet

Les concepts de l'orienté objet (OO) ont été introduits par Alan Kay qui a écrit le premier langage de programmation OO baptisé Smalltalk dans les années 70. Les concepts autour desquels est axé le paradigme de l'orienté objet sont les suivants :

3.1 L'objet

Un objet est avant tout une structure de données chargée de gérer des données, de les classer, et de les stocker sous une certaine forme. Ce qui distingue l'objet des autres structures de données est qu'il regroupe les données et les traitements sur ces données. Un objet rassemble deux éléments :

- Les champs : aussi appelés attributs
- Les méthodes : elles servent d'interface entre les données et le programme. Elles peuvent êtres des fonctions ou des procédures destinées à traiter les données.

Les champs et les méthodes d'un objet sont ses membres. Un objet est donc un type servant à stocker des données dans des champs et à les gérer au travers des méthodes.

3.2 La classe

La classe est une définition abstraite d'objets similaires qui peuvent être informatiquement décrits par la même abstraction. Une classe est donc un type d'objet. L'objet est par conséquent une instance de la classe obtenu en affectant une valeur à au moins l'un des champs définis par cette classe.

3.3 Les fondements de l'orienté objet

3.3.1 L'encapsulation

Derrière ce terme se cache le concept même de l'objet ; réunir sous la même entité les données (les champs) et les moyens de les gérer (les méthodes). L'encapsulation introduit donc une nouvelle manière de gérer les données car il ne s'agit plus de déclarer des variables générales puis un ensemble de procédures et fonctions implémenter afin de les gérer de manière séparée. En effet, sous ce nouveau concept se cache également un autre élément à prendre en compte : pouvoir masquer aux yeux d'un programmeur extérieur certaines parties d'un objet et donc un ensemble des méthodes destinées à la gestion interne de l'objet, auxquelles le programmeur final n'aura pas à avoir accès.

L'encapsulation permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.

3.3.2 L'héritage

La notion d'héritage stipule qu'il y ait une classe dite "générique" et une ou souvent plusieurs classes dites "spécifiques". Diverses terminologies sont également employées telles que : classe mère / classes filles, super classe / sous-classes, classes

parent / classes enfants, classe ascendante / classes descendantes. Grâce au concept de l'héritage, les classes spécifiques bénéficient des caractéristiques propres de leur ancêtre, à savoir ses champs et méthodes. Cependant, les descendants conservent la possibilité de posséder leurs propres champs et méthodes et de se spécialiser ainsi.

On note qu'une fois qu'un champ ou une méthode sont définis, il ou elle le reste pour toutes les sous-classes quel que soit leur degré d'éloignement. Généralement, les niveaux d'héritage ne sont pas limités.

3.3.3 Le polymorphisme

Le terme polymorphisme se compose de deux mots : *poly* qui veut dire plusieurs et *morphisme* qui désigne morphologie ou encore forme. Ce concept traite de la capacité de l'objet à posséder plusieurs formes et se déploie sous deux catégories : le polymorphisme par surcharge et le polymorphisme par redéfinition.

La surcharge est la possibilité d'utiliser dans une même classe, le même nom pour déclarer des méthodes différentes mais qui ont la même sémantique. Nous pouvons voir le mécanisme de surcharge lorsque plusieurs constructeurs sont définis pour une classe (voir le concept de constructeur plus bas).

La redéfinition est la possibilité d'utiliser exactement la même signature pour définir une méthode dans une super classe et dans une sous classe. Le type de retour de la méthode doit être le même, mais la visibilité peut changer (voir le concept de visibilité plus loin).

3.4 Les constructeurs et destructeurs

3.4.1 Constructeurs

Les constructeurs sont des méthodes d'une classe ayant le même nom que celui de la classe servant à créer l'objet en mémoire. S'il peut exister en mémoire plusieurs instances d'un même type objet, seule une copie des méthodes est conservée, de sorte que chaque instance se réfère à la même zone mémoire en ce qui concerne les méthodes. Les champs sont en revanche distincts d'un objet à un autre. Les remarques suivantes sont à prendre en considération concernant les constructeurs :

- Un objet peut ne pas avoir de constructeur explicite
- Un objet peut avoir plusieurs constructeurs

3.4.2 Destructeurs

Le destructeur est une méthode servant de détruire l'instance de l'objet. La mémoire allouée est par conséquent libérée. Là encore, différentes remarques doivent être gardées à l'esprit :

- Un objet peut ne pas avoir de destructeur. Dans ce cas un mécanisme appelé « ramasse miette » peut être mis en place afin de décharger le programmeur de la gestion des objets.
- Un objet peut avoir plusieurs destructeurs.

3.5 La visibilité

3.5.1 Champs et méthodes publics

Les champs et méthodes d'un objet dits publics sont accessibles depuis tous les autres objets des autres classes et dans tous les modules (programme, unité, ..)

3.5.2 Champs et méthodes privés

La visibilité privée restreint la portée d'un champ ou d'une méthode au module où il ou elle est déclaré(e).

3.5.3 Champs et méthodes protégés

Ce mode de visibilité est à mi-chemin entre le mode public et le mode privé. En effet, il correspond à la visibilité privée excepté que tout champ ou méthode protégé(e) est accessible dans tous les objets des classes filles, quelque soit le module où ils se situent.

4. Puissance des SGBD-OO [1, 2, 3]

Les SGBD-OO doivent leur puissance à la convergence de deux technologies redoutables : les bases de données notamment relationnelles d'un côté et les langages de programmation orientés objet (LP-OO) de l'autre. Le Manifeste du système de base de données orientée objet (*Object-Oriented Database System Manifesto*) de 1989 suggérait une panoplie de caractéristiques qui concernent un SGBD-OO parfait, comme le montre la figure ci-dessous. Il stipule que :

- ✓ **Un SGBD-OO est mieux qu'un LP-OO** : En plus de ce qu'offre un LP-OO, un SGBD-OO doit être capable de :

  1) gérer des objets complexes : Le système doit permettre de construire des objets complexes en appliquant des constructeurs à des objets de base.

  2) gérer l'identité d'objet : Tout objet doit posséder une identité unique, indépendante des valeurs de ses attributs. Ce concept sera amplement abordé au chapitre 3.

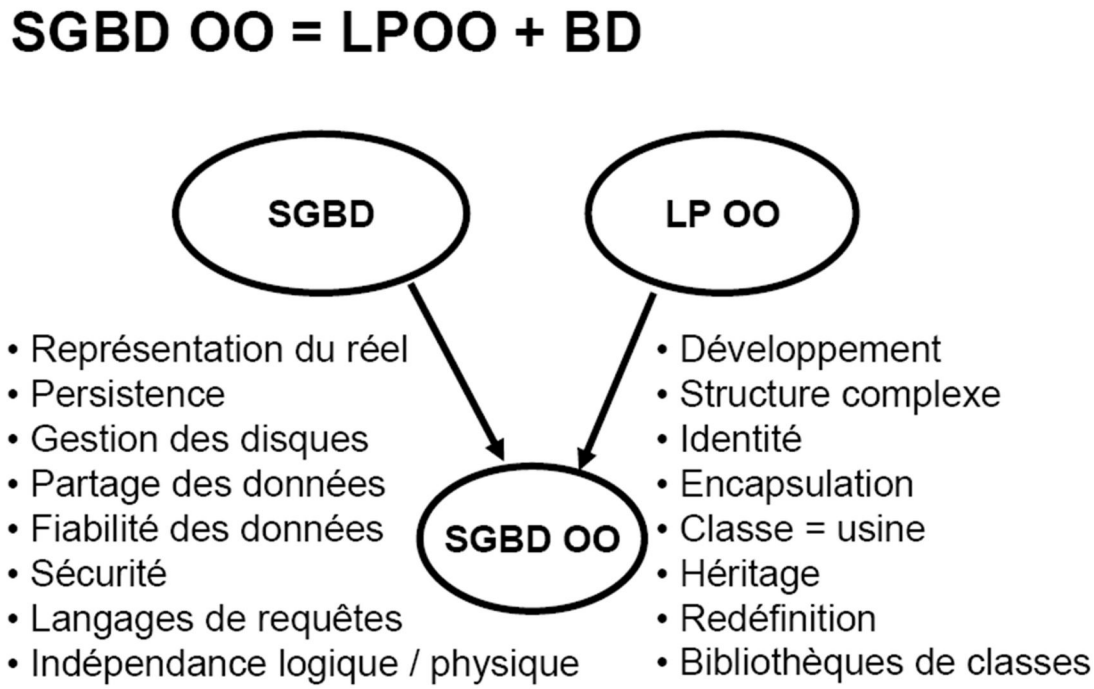

Fig. 4 : Convergences du domaine de BD et de celui du LP-OO

3) Il permet l'encapsulation : Dans un SGBDOO, l'encapsulation est proprement obtenue en garantissant que les programmeurs n'ont accès aux méthodes que par l'intermédiaire de l'interface et que les données et l'implémentation de ces méthodes sont masquées dans les objets.

4) Il gère les types ou les classes : Le manifeste n'exige le support que d'un de ces concepts. Le schéma de base de données dans un système orienté objet comporte un ensemble de classes ou un ensemble de types. Le système n'est toutefois pas obligé de conserver ou d'entretenir l'*extension* d'un type, c'est-à-dire l'ensemble des objets

d'un type donné, dans la base de données. Et, si l'extension est maintenue, le système n'est pas obligé de le rendre accessible à l'utilisateur.

5) Les types ou les classes doivent pouvoir hériter de leurs ancêtres : Un sous-type ou une sous-classe doit hériter des attributs et des méthodes, respectivement, de sa superclasse.

6) Il doit assurer les liens dynamiques : Les méthodes doivent s'appliquer à des objets de types différents (polymorphisme).

7) L'ensemble des types de données est extensible : L'utilisateur doit être en mesure de construire de nouveaux types à partir de l'ensemble des types prédéfinis par le système. En outre, il ne peut y avoir de différence d'usage entre les types prédéfinis du système et ceux définis par l'utilisateur.

- ✓ **un SGBD-OO est mieux qu'un SGBD-R** : dans la mesure où il permet :

  8) Le système permet la persistance des données : Comme dans un SGBD conventionnel, les données doivent continuer d'exister (donc persister) après l'achèvement de l'application qui les a créées.

  9) Le SGBD doit être capable de gérer de très vastes bases de données : Un SGBD conventionnel dispose de mécanismes qui lui permettent de gérer efficacement le stockage secondaire, sous la forme notamment d'index et de tampons. Un SGBDOO doit disposer de mécanismes semblables, invisibles pour l'utilisateur, ce qui signifie l'assurance d'une indépendance claire entre les niveaux logique et physique au sein du système.

  10) Le SGBD accepte des utilisateurs simultanés et concurrents : Un SGBDOO doit fournir des mécanismes de contrôle de concurrence semblables à ceux des systèmes conventionnels.

  11) Le SGBD est capable de récupérer à la suite de défaillances matérielles et (ou) logicielles. Un SGBDOO doit fournir des mécanismes de récupération semblables à ceux des systèmes conventionnels.

  12) Le SGBD fournit un moyen simple pour interroger les données : Un SGBDOO doit fournir un utilitaire de requêtes de haut niveau (c'est-à-dire raisonnablement déclaratif), efficace et indépendant de l'application. Le système ne doit pas

obligatoirement fournir un langage de requête, mais peut proposer à la place un navigateur graphique.

- ✓ Le manifeste propose également les caractéristiques **facultatives** suivantes :
  13) l'héritage multiple,
  14) la vérification de type et l'inférence de type,
  15) la distribution des données sur un réseau,
  16) la conception des transactions et la gestion des versions.

- ✓ Il est curieux de constater que le manifeste n'évoque aucune notion de sécurité ni d'intégrité et que la présence d'un langage de requête déclaratif n'est pas indispensable.

4. Stratégies de développement d'un SGBD-OO [1]

Le développement des SGBD-OO connaît plusieurs stratégies alternatives, que nous pouvons résumer comme suit :

* Étendre un langage de programmation orienté objet existant avec des fonctionnalités de base de données : cette approche ajoute les fonctionnalités de base de données à un langage de programmation orienté objet tel que Smalltalk, C++ ou Java. C'est l'approche empruntée par le produit GemStone, qui apporte des extensions à ces trois langages.

* Proposer des bibliothèques de SGBD orientées objet : cette approche aussi ajoute des fonctionnalités de base de données traditionnelles à un langage de programmation orienté objet existant. Cependant, au lieu d'étendre le langage même, des bibliothèques de classes sont fournies qui acceptent la persistance, l'agrégation, les types de données, les transactions, la simultanéité, la sécurité et ainsi de suite. C'est l'approche suivie par les produits Ontos, Versant et ObjectStore.

* Intégrer des constructions orientées objet du langage de base de données dans un langage hôte conventionnel : cette stratégie poursuit la même idée d'intégration d'un langage de base de données orienté objet dans un langage de programmation hôte. C'est l'approche suivie par O2, qui offre des extensions intégrées pour le langage C.

* Étendre le langage de base de données existant avec des caractéristiques orientées objet : du fait de l'acceptation presque unanime de SQL, des éditeurs de logiciels préfèrent l'étendre pour lui donner les constructions orientées objet. C'est l'approche préférée des éditeurs de SGBDR et de SGBDRO. L'édition de 1999 de la norme SQL, SQL:1999, accepte des caractéristiques orientées objet. En plus, l'Object Database Standard de l'Object Data Management Group (ODMG) spécifie une norme Object SQL, que nous étudions au chapitre 3 suivant. Les produits Ontos et Versant proposent une version d'Object SQL et nombre d'éditeurs de SGBDOO se mettent en conformité par rapport à la norme de l'ODMG.

* Développer un tout nouvel ensemble modèle de base de données et langage de données : c'est l'approche la plus radicale qui reprend les problèmes dès le début, développe un langage de base de données neuf et un SGBD avec des caractéristiques orientées objet. C'est l'approche suivie par SIM (Semantic Information Manager), qui se fonde sur le modèle de données sémantique et emploie un langage de définition de données (LDD) et un langage de manipulation de données (LMD) tout nouveaux.

5. Conclusion

Dans le chapitre suivant, nous abordons le standard ODMG qui constitue l'une des stratégies de développement des SGBD-OO les plus suivies. Nous passons en revue les grands principes d'élaboration des bases de données orientées objet. Nous présentons le langage de définition des données d'ODMG nommé ODL (*Object Definition Language*) ainsi que le langage de manipulation des données de cette norme appelé OQL (*Object Query Language*).

Chapitre 3

Le standard d'ODMG

- Modélisation d'une BD-OO
- ODL : le langage de définition de données
- OQL : le langage de manipulation de données

1. Introduction

La communauté des BD-OO a développé le standard ODMG (Object Data Management Group) afin d'établir une spécification commune d'interfaçage objet pour promouvoir la portabilité entre les produits commerciaux basés sur les BD-OO. Le modèle de données d'ODMG est basé sur celui d'OMG (Object Management Group), qui fournit des capacités d'interfaçage orienté objet assurant ainsi l'interopérabilité entre les applications. La norme finale d'ODMG a été publiée en 2001. Elle regroupe de nombreux constructeurs de SGBO-OO tels que Poet, Ardent, Objectivity, Versant ainsi que des chercheurs et des utilisateurs. Il est à noter que nombre de constructeurs n'ont pas suivi cette norme.

Le concepteur de base de données utilise ODL (*Object Definition Language*) de la norme ODMG pour spécifier les types d'objets qui seront nécessaires dans l'application. Une fois le schéma de la base est défini, OQL (*Object Query Language*) d'ODMG fournit un langage déclaratif pour passer des requêtes et interroger ainsi la base.

Tout au long de ce chapitre, nous utiliserons la syntaxe d'ODMG.

2. Modèle de données d'une BD-OO selon ODMG

2.1 Structures de données

La structure de données des modèles orientés objets est caractérisée par le fait que le concept principal autour duquel s'articule le processus de modélisation est **l'objet** ; il n'y a plus de notion de Table. Chaque objet est avant tout une structure complexe ayant une identité qui lui est propre et qui le distingue de tous les autres. Cette identité est permanente et immuable. On l'appelle l'OID de l'objet (*en Anglais "object identity*").

Il est également possible de créer d'autres structures appelées **collections**. Elles permettent de définir des attributs multivalués. Nous allons voir qu'il est possible de définir avec ODL des attributs complexes, des attributs multivalués, des attributs complexes et multivalués et des attributs complexes comportant des attributs complexes, auxquels peuvent être associées des fonctions permettant de les manipuler.

2.1.1 La classe, l'objet et le littéral

La classe est, comme expliqué précédemment, une définition abstraite d'un type d'objet. Elle est construite à travers la primitive "CLASS". Une classe peut comporter la désignation d'une clé parmi la liste des attributs de la classe, moyennant la clause "KEY". Une classe possède aussi une extension ou plus communément appelée « un récipient » qui lui à sert mémoriser les objets créés en tant qu'instances de cette classe. En d'autres termes une extension est une collection contenant une population d'une classe (la population étant les objets de la même classe). En ODMG, l'utilisateur est déchargé de la gestion des populations, mais lui laisse le loisir de nommer le récipient de données des classes. Si l'utilisateur ne le fait, c'est le SGBD qui s'en occupera. La clause "EXTENT" est facultative, elle sert à nommer l'extension de la classe, elle sera plus utile dans la formulation des requêtes en OQL abordé en section 4.

En outre, les primitives de modélisation du standard d'ODMG sont les objets et les littéraux (voir figure 15). Les objets possèdent une identité (comme expliqué en section 2.1.2) alors que les littéraux n'en ont pas. Aussi bien les objets que les littéraux sont caractérisés par leurs types (appelés encore domaines). La notion de littéral est essentiellement utilisée dans le langage d'interrogation des BD-OO, en l'occurrence OQL.

```
class Nom_de_la_classe ( extent nom_du_récipient     key la_ou_les_clés )
{ attribute type_attribut nom_attribut;
relationship nom_relation Nom_classe2 inverse nom_relation_inverse;
// signatures_des_méthodes;
};
```

Fig. 5 : Syntaxe de définition d'une classe

2.1.2 L'identité d'objet

L'objectif est de pouvoir identifier tout objet indépendamment de sa valeur (ce qui implique, entre autres, de pouvoir gérer des objets de même valeur mais distincts). Cet identifiant doit être :

- permanent qui existe pendant au moins toute la durée de vie de l'objet,
- fixe qui ne change pas durant la vie de l'objet et
- unique dans la base et dans le temps, c-à-d, que deux objets distincts de la même base, même s'ils n'existent pas en même temps n'auront jamais la même identité.

L'identité des objets (l'OID) est gérée par le SGBD-OO. Dès la création d'un objet, un identifiant système lui est attaché et sa valeur n'est ni affichable, ni imprimable, ni modifiable.

2.1.3 L'égalité d'objets

L'égalité de deux objets, o1 et o2, peut se mesurer à plusieurs niveaux:

a) l'identité d'objets, notée: **o1 == o2** qui signifie:

Est-ce le même objet (le même OID)?

b) l'égalité de valeur, notée: **o1 = o2** qui signifie:

Les objets o1 et o2 ont-ils la même valeur ? C'est-à-dire tous leurs attributs ont la même valeur, que ce soient des attributs-valeur ou des attributs-référence.

c) l'égalité de valeur après fermeture, notée: **o1 =f o2** qui signifie:

Les objets o1 et o2 ont-ils la même "valeur après fermeture" ? C'est-à-dire:

* les objets o1 et o2 ont des graphes de composition isomorphes (leurs attributs-référence constituent des graphes équivalents), et

* les attributs-valeur correspondants, à tous les niveaux, sont égaux.

2.1.4 Propriétés inhérente aux objets et à leur identité

Pour deux objets, o1 et o2, de la même classe, on a les propriétés suivantes:

1) Si o1 == o2 alors o1 = o2 (l'égalité des OID de deux objets implique qu'il s'agit d'un seul et même objet)
2) Si o1 = o2 alors o1 =f o2 (l'égalité de deux objets implique qu'ils sont également égaux après fermeture. La notion de fermeture sera réexpliquée après avoir abordé les liens de composition entre les objets)

N.B. L'identité d'objet est un concept central qui marque la différence des BD-OO par rapport aux BD-R. En relationnel, l'utilisateur doit inventer des identifiants (ou clés) et les gérer (donner des valeurs différentes aux clés des différents tuples), ce qui pose problème en cas de mise à jour des attributs de la clé.

Exercice

Afin d'étayer les propriétés des objets abordées plus haut, nous présentons cet exemple [2]. Soient les deux classes d'objets représentées ci-dessous. La classe1 possède un attribut A de domaine entier, et un attribut référence B qui contient l'OID d'un objet de la classe 2. La classe 2 possède un attribut C de domaine entier.

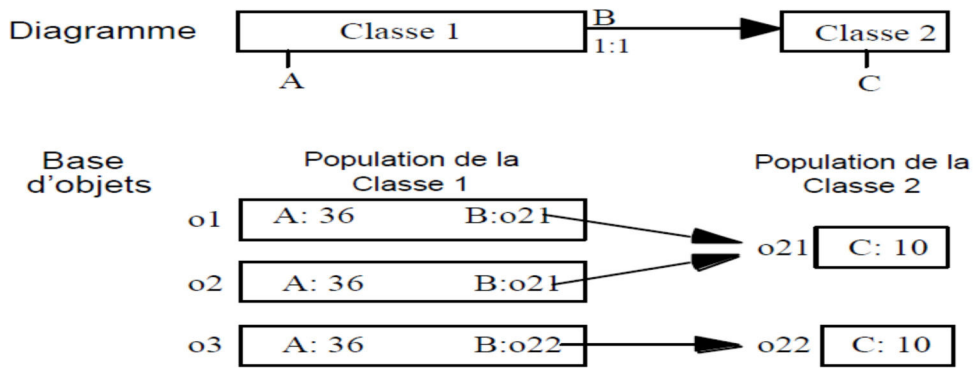

Fig. 6 : Exemple de données logiques de classes

Parmi les formules suivantes, désigner celles qui sont correctes et celles qui sont fausses :

(1)	o1.B == o2.B	(2)	o1 == o2	(3)	o1 = o2	(4)	o1 = o3
(5)	o1 =f o3	(6)	o21 = o22	(7)	o21 == o22		

Réponse :

(1)	vrai	(2)	faux	(3)	vrai	(4)	faux
(5)	vrai	(6)	vrai	(7)	faux		

2.1.5 La structure complexe

Les structures complexes sont utilisées par les concepteurs de BD-OO dans deux cas:

a) pour définir la structure des objets, c'est-à-dire celle de leur classe;

b) pour étendre l'ensemble des domaines (appelés aussi types) prédéfinis existants dans le SGBD-OO (tels que STRING, INT, DATE, MONEY…) en définissant des domaines spécifiques à leur application et en utilisant la clause "TYPEDEF".

Les structures complexes sont définies en employant le constructeur "STRUCT" qui crée une structure composée d'une suite d'attributs.

STRUCT (type_attr$_1$ nom_attr$_1$, … type_attr$_i$ nom_attr$_i$) ;

où " type_attr$_i$ " est le domaine de l'attribut de nom " nom_attr$_i$ "

Il est soit :

- un domaine prédéfini (STRING, FLOAT, INT, DATE...), ou
- un type défini par l'utilisateur, ou
- le nom d'une classe

Fig. 7 : Syntaxe de définition d'une structure complexe

2.1.6 Les objets structurés

Les objets structurés s'inspirent des types SQL2 utilisés pour la gestion du temps. L'intérêt de définir ces types comme des objets est de permettre de spécifier leur comportement sous forme d'interfaces. Des structures correspondantes sont aussi fournies dans les types de base proposés par l'ODMG. Voici les types d'objets structurés supportés :

– "DATE" représente un objet date par une structure (mois, jour, an) munie de toutes les opérations de manipulation classique des dates.

– "INTERVAL" représente un objet durée par une structure (jour, heure, minute, seconde) munie de toutes les opérations de manipulation nécessaires, telles que l'addition, la soustraction, le produit, la division, les tests d'égalité, etc.

– "TIME" représente les heures avec zones de temps, en différentiel par rapport au méridien de Greenwich ; l'unité est la milliseconde.

– "TIMESTAMP" encapsule à la fois une date et un temps. Un objet timestamp permet donc une référence temporelle absolue en millisecondes.

2.1.7 Les collections

Les constructeurs de collection permettent de définir des attributs multivalués. ODMG offre quatre constructeurs de collections :

"SET" crée un type ensemble d'éléments non ordonnés et non indexés

"LIST" crée une liste d'éléments ordonnés et indexés

"BAG" crée un type multi-ensemble (c-à-d un ensemble pouvant contenir des doublons) d'éléments non ordonnés et non indexés

"ARRAY" crée un tableau d'éléments ordonnés et non indexés

Remarques

- La différence entre les types LIST et ARRAY est que la collection LIST est de taille indéterminée tandis que la deuxième est de taille préalablement définie qui restera fixe.
- La différence entre les types SET et LIST et que la collection SET peut être vide contrairement au deuxième qui doit contenir au moins un élément.
- La différence en les types SET et BAG et que le multi-ensemble comme son nom l'indique peut contenir plusieurs occurrences du même élément, ce qui n'est pas permis dans les ensembles.

2.1.7 Exemples

```
CLASS Etudiant (extent lesEtudiants  key num)      // déclaration d'une classe d'objets
{                                                  // voir utilisation de EXTENT en section 2.2.1
attribute INT num ;                                // attribut monovalué de type prédéfini
attribute STRING nom ;                             // attribut monovalué de type prédéfini
attribute LIST <STRING>  prénoms ;                 // attribut multivalué de type liste
attribute DATE date-nais ;                         // attribut monovalué de type prédéfini
attribute adresse : STRUCT                         // attribut monovalué complexe
        (INT num ,
        STRING rue ,
        STRING ville ,
        STRING pays  ) ;
attribute cours-suivis : SET STRUCT                // attribut complexe et multivalué
        (STRING nom-cours ,
        FLOAT note ) ;
}
```

Fig. 8 : Exemple 1

```
TYPEDEF T_Adresse  STRUCT                  // déclaration d'un nouveau type utilisateur complexe
        (INT num ,
        STRING rue ,
        STRING ville ,
        STRING pays  ) ;
CLASS Etudiant (extent lesEtudiants  key num)   // voir utilisation de EXTENT en section 2.1.1
{
attribute INT num ;
attribute STRING nom ;
attribute LIST <STRING>  prénoms ;
attribute DATE date-nais ;
attribute T_Adresse  adresse ;                  // attribut complexe de type défini par l'utilisateur
attribute cours-suivis : SET STRUCT
        (STRING nom-cours ,
        FLOAT note ) ;
}
```

Fig. 9 : Exemple2

En fait, les figures 8 et 9 représentent deux versions d'un même exemple avec une petite différence entre elles, car dans le 2ème exemple, nous avons déclaré un nouveau type utilisateur que nous avons appelé T_Adresse que nous pouvons éventuellement réutiliser comme domaine dans plusieurs classes et même lors de la définition d'un autre nouveau type utilisateur.

2.2 Les liens entre objets [2]

Il existe deux types de liens entre les objets :

- les liens de composition: "tel objet est composé de tel(s) objet(s)". Ces liens sont décrits par des attributs particuliers, appelés attributs référence, qui ont pour domaine une classe d'objets;
- les liens de généralisation / spécialisation: "tel objet de la base décrit sous un autre point de vue la même entité du monde réel que tel autre objet de la base". Ce sont les liens **IS-A** des modèles sémantiques auxquels est associé un mécanisme d'héritage des propriétés, relations et méthodes de la classe générique par les classes spécialisées.

2.2.1 Les liens de composition

Un lien de composition relie une classe C1 à une classe C2 si les objets de C1 (appelés objets composés) sont composés d'objets C2 (appelés objets composants). C'est un lien orienté de cardinalité quelconque (0:1 ou 1:1 ou 1:N ou N:M ou 0:N…). Par exemple, les objets maison sont composés d'objets mûr, porte, fenêtre, plancher,... Il y a un lien de composition entre la classe maison et la classe mûr, un second entre la classe maison et la classe porte, un troisième entre la classe maison et la classe fenêtre, ..., etc. Les liens de composition sont représentés sur les diagrammes par des flèches fines.

Ces liens de composition sont implantés par deux façons :

- par des attributs-référence, ou
- par des liens de type association

Nous détaillons dans la suite de cette section les deux moyens de définir les liens de composition.

a) Les attributs référence

Ces liens de composition peuvent être implantés par des attributs-référence qui pointent sur les objets composants. Leur déclaration est faite à l'intérieur de la description de la classe composée par un attribut-référence dont le domaine est celui de la classe composante. Leur valeur est l'identité de l'objet référencé (OID). On peut avoir des objets partagés qui sont composants de plusieurs objets composés. Dans l'exemple suivant, la classe Cours est modélisée comme étant une classe composante de la classe Etudiant :

```
CLASS Etudiant{                          // classe composée
attribute INT num ;
attribute STRING nom ;
attribute LIST <STRING> prénoms;
attribute cours-suivis : SET STRUCT
        (Cours cours ;                   // attribut référence de 2è niveau
        note : FLOAT }
} ;
CLASS Cours{                             // classe composante
attribute STRING nomC ;
 ....
} ;
```

Fig.10 : Exemple de définition d'un attribut référence

(version non autorisée en ODMG)

Certains SGBD-OO, parmi les plus récents, se rapprochent du modèle entité association. La norme ODMG n'autorise les attributs référence qu'en attributs du premier niveau des classes. Les attributs composants d'attributs complexes ne peuvent pas être des attributs référence. Par conséquent, l'exemple de la figure 10 n'est pas correct car l'attribut référence "cours" est un attribut de 2ème niveau dans la classe composée "Etudiant". Pour résoudre cette situation en ODMG, il faudrait introduire une classe intermédiaire (qui représente une association du schéma entité association équivalent) de la façon suivante:

```
CLASS Etudiant{
attribute INT num ;
attribute STRING nom ;
attribute LIST <STRING> prénoms;
attribute SET <Cours-suivis> cours-suivis;
} ;
Class Cours-suivis {                    // classe intermédiaire
attribute att : Struct
        (Cours cours ;                  // attribut référence de 1er niveau
        note : FLOAT }
} ;
CLASS Cours{                            // classe composante
attribute STRING nomC ;
 ....
} ;
```

Fig. 11 : Exemple de définition d'un attribut référence (version corrigée)

D'autres contraintes :

A part le modèle de BD-OO de la norme ODMG, les autres modèles existants proposent différentes variantes du lien de composition et sont régies parfois par des contraintes d'intégrité qui peuvent lui être associées. Ces variantes sont essentiellement:

- Un objet composant peut être, ou non, **partagé** entre plusieurs objets de la même classe composée. Par exemple, dans une base de données décrivant des voitures en réparation dans un garage, Moteur est une classe d'objets composants non partagés de la classe Voiture. Par contre, dans une base de données décrivant les modèles de voitures, Moteur serait une classe d'objets composants partagés de la classe Voiture, puisque deux modèles de voitures peuvent avoir le même type de moteur.
- Une classe composante peut être, ou non, **dépendante** de sa (d'une de ses) classe composée. Un objet o1 composant est dépendant de son objet père o2 si l'existence de o1 dépend de celle de o2, c'est-à-dire si la destruction de o2

entraîne automatiquement celle de o1. Par exemple, dans la base de données décrivant les voitures en cours de réparation, si le garage ne récupère aucune pièce sur les voitures envoyées à la casse, alors Moteur serait une classe d'objets composants dépendants de Voiture. Si au contraire, le garagiste récupère des pièces, et en particulier certains moteurs, alors Moteur serait une classe composante non dépendante de Voiture.

Remarque sur les liens inverses :

Le lien de référence est orienté de la classe composée vers la classe composante, à cause de son implémentation (pointeur logique sur les objets de la classe composante). Cela implique que, dans les requêtes, il est beaucoup plus facile et rapide d'aller de l'objet composé aux objets composants que l'inverse. Pour avoir un accès aisé dans les deux sens dans la base de données, il est utile de décrire le lien de composition et son **inverse**. Certains SGBDO assurent la contrainte inverse: l'objet composé dépend de ses composants. Les contraintes de dépendance définissent l'ordre selon lequel les objets doivent être créés (et détruits): d'abord les composants ou d'abord le composé.

b) Les associations (Relationship)

Pratiquement, les liens de composition servent à décrire le fait qu'un objet est composé d'autres objets, mais aussi tout autre lien de type association, car il n'existe pas dans le modèle orienté-objets de lien d'association générique proprement décrite en modèle relationnel. Les associations en BD-OO permettent de compléter la modélisation des objets. L'ODMG préconise le support d'associations binaires, bidirectionnelles de cardinalité (1:1), (1:N), ou (N:M), sans données. Une association de ClassA vers ClassB définit deux chemins de traversée inverses, A->B et B->A. Chaque chemin doit être défini en ODL au niveau du type d'objet source par une clause "RELATIONSHIP". L'association pointe vers un seul objet cible ou vers une collection. Elle porte un nom et son inverse doit être déclaré. La figure 12 illustre une association entre les classes Etudiant et Enseignant.

```
class Enseignant (extent  lesEnseignants)
{
   attribute short Numbureau;
   relationship set<Etudiant> tuteur_de inverse Etudiant :: tutoré_par;
   ……
};
class Etudiant  (extent lesEtudiants  key numInscription)
{
   attribute string numInscription;
   relationship Enseignant tutoré_par inverse Enseignant :: tuteur_de;
   ……
};
```

Fig. 12 : Exemple de définition d'une association

Commentaire :

On constate que la relation de tutorat qui relie les deux classes Enseignant et Etudiant est définie dans les deux sens dans les deux classes. Il en découle que le nombre de "RELATIONSHIP" dans un schéma ODL devrait être un nombre pair. La relation décrite dans l'exemple de la figure 12 ci-dessus se lit comme suit :

- Un enseignant est tuteur d'un ensemble (SET) d'étudiants, et inversement
- Un étudiant est tutoré par un enseignant.

2.2.2 Les liens de généralisation / spécialisation

Certains ensembles d'objets du monde réel ayant des caractéristiques communes peuvent être perçus comme comprenant un ou plusieurs sous-ensembles d'objets, chaque sous-ensemble ayant des propriétés particulières. Les modèles de données orientés objets permettent de décrire ce type de situation à l'aide du concept de généralisation/spécialisation (ou lien IS-A des modèles sémantiques) auquel ils ont ajouté un mécanisme d'héritage.

Un lien de généralisation/spécialisation, **CS –>CG,** est un lien binaire orienté entre deux classes, CS appelée sous-classe (ou classe spécifique) et CG appelée superclasse

(ou classe générique). Il est noté graphiquement sur les diagrammes par une flèche épaisse et il est défini par les trois règles suivantes :

- L'inclusion des populations : CS représente un sous-ensemble des entités du monde réel décrites par CG. Du point de vue conceptuel, la population de CS est incluse dans celle de CG.
- La substituabilité des objets de CG par des objets de CS : lors de la manipulation des objets, partout où l'on peut employer un objet de type CG, on peut aussi employer un objet de type CS.
- L'héritage : CS hérite des attributs et méthodes de CG (voir paragraphe 3.3.2). C'est ce concept d'héritage de la structure et des méthodes qui permet la réutilisation des classes.

Soit le diagramme de généralisation/spécialisation (G/S) ci-dessous en figure 13.

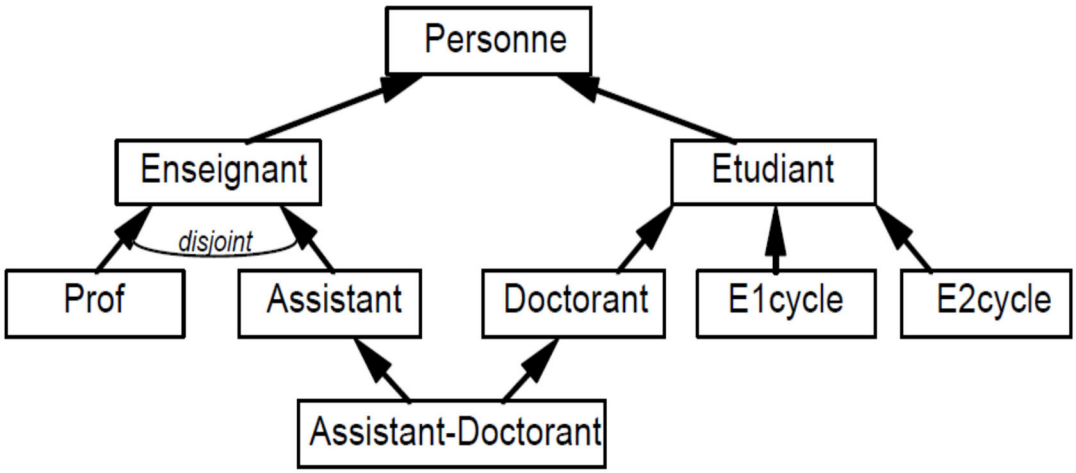

Fig. 13 : graphe de généralisation / spécialisation de classes

Le diagramme est traduit par une déclaration partielle fournie en figure 14. Il est à noter qu'en ODMG, le lien de G/S se déclare dans le schéma textuel par le caractère "**:**".

```
class Personne {
   attribute string nom;
   attribute string addresse;
   attribute date dateNais;
};
class Enseignant : Personne (extent  lesEnseignants) {
   attribute short numBureau;
   attribute statut : enum ("prof", "assistant")

   ....
};
class Etudiant  : Personne  (extent lesEtudiants) {
   attribute string numInscription;
   .....
};
```

Fig. 14 : Déclaration partielle d'un graphe de G/S

Dans ce code, tout Enseignant est une Personne et tout Etudiant et une Personne. En d'autres termes, tout objet Enseignant (ou Etudiant) a les attributs et méthodes de la classe générique Personne en plus de ses attributs et méthodes spécifique.

Note :

Le graphe engendré par la hiérarchie des classes ne doit pas contenir de cycle. Cette interdiction des cycles découle directement de la définition des liens IS-A avec inclusion de population. En effet si l'on décrivait par exemple les deux liens IS-A suivants:

C1 à C2 et C2 à C1

Cela signifierait que la population de C1 est contenue dans celle de C2 et vice versa. Donc, les deux classes devraient décrire exactement le même ensemble d'objets, elles devraient donc être confondues en une seule et même classe.

Contraintes :

Des contraintes d'intégrité de couverture, de disjonction et de partition peuvent être associées aux sous-classes d'une classe :

- Un sous-ensemble, S, des sous-classes immédiates d'une superclasse C est **disjoint** si toute occurrence de C peut être occurrence d'au plus une des classes de S. Dans le graphe de la figure 13, un assistant ne peut pas être aussi un professeur, on a donc une contrainte de disjonction entre ces deux sous-classes.
- Un sous-ensemble, S, des sous-classes immédiates d'une superclasse C est **couvrant** si toute occurrence de C est aussi occurrence d'une (au moins) des classes de S. Dans le graphe de la figure 13, si les seules personnes enregistrées dans la base sont les enseignants et les étudiants, alors il y a une contrainte de couverture: Enseignant et Etudiant constituent une couverture de Personne. Mais il n'y a pas de disjonction entre Enseignant et Etudiant, car il peut y avoir des assistant-doctorants (assistants inscrits en thèse).
- Enfin, un sous-ensemble, S, des sous-classes immédiates d'une superclasse C constitue une **partition** de C, si ce sous-ensemble est à la fois disjoint et couvrant.

Restrictions :

1) La plupart des SGBDO commercialisés limitent fortement les possibilités théoriquement offertes par la hiérarchie de généralisation / spécialisation. Souvent ils imposent aux sous-classes issues d'une même sur-classe soit d'être disjointes, soit que leur recouvrement ait été explicitement déclaré par une sous-classe commune. Cette restriction est due au fait que ces systèmes se basent sur le principe de l'instanciation unique des objets: chaque objet est stocké par le système dans une seule classe, qui est la classe la plus spécialisée le contenant (cette classe étant nécessairement unique).

2) Une seconde restriction, très fréquente elle aussi, est que la hiérarchie de G/S soit statique: une fois qu'un objet a été créé dans une classe, il ne peut plus en changer, ni acquérir de nouvelles sous-classes. Il ne peut pas être spécialisé, ni "dé-spécialisé", ni passer dans une autre sous-classe. Par exemple dans la hiérarchie du diagramme de la figure 13, pour ces systèmes, un objet de 1ercycle ne peut pas devenir un objet de 2emecycle. Pour cela, l'utilisateur devrait détruire l'objet de 1er cycle puis créer un nouvel objet (avec un nouvel OID!) dans 2emecycle.

Raffinement et redéfinition:

Certaines propriétés peuvent avoir des caractéristiques particulières propres à une sous-classe. Par exemple, l'attribut statut de Enseignant aura toujours la valeur « prof » dans la sous-classe Prof, et la valeur "assistant" dans la sous-classe Assistant. De même, il est parfois utile de définir pour une méthode héritée un nouveau comportement spécifique de la sous-classe. Par exemple, la méthode afficher() appliquée à une Personne affichera les nom, prénoms et adresse de la personne, appliquée à un Etudiant elle affichera en plus les cours suivis.

Pour cela, certains SGBD orientés objet offrent la possibilité de raffiner un attribut ou redéfinir une méthode héritée :

· raffiner dans une sous-classe un attribut hérité signifie déclarer que dans la sous-classe son domaine de définition (ou ses cardinalités) est réduit à un sous domaine de celui qu'il a dans la sur-classe. Cette réduction respecte le principe de substituabilité.

· redéfinir dans une sous-classe une méthode héritée signifie déclarer dans la sous-classe un nouveau comportement (code) tout en conservant sa signature (voir polymorphisme par redéfinition en section 3.3.3).

· D'autres variantes, telles le changement de signature d'une méthode, peuvent aussi être offertes.

Remarque :

L'héritage multiple est une variante dans les langages de programmation orientés objets et dans certains SGBD-OO. L'héritage multiple se traduit par le fait qu'une classe ait plusieurs classes mères. C'est le cas pour la classe Assistant-Doctorant. Les assistants doctorants sont à la fois assistants (et donc enseignants) et doctorants (et donc étudiants). Dans le cas d'héritage multiple, un conflit peut survenir lors de l'héritage des attributs et des méthodes: deux attributs ou méthodes de deux surclasses peuvent avoir le même nom et n'être pas identiques. Par exemple, pour Assistant-Doctorant, quelle est la bonne méthode afficher qu'il faut employer? Celle de Enseignant ou celle de Etudiant? Face à une telle situation, les SGBD-OO réagissent différemment:

* soit ils ne savent pas résoudre ce conflit, et refusent la définition d'un tel schéma **(cas d'ODMG)**

* soit ils demandent au concepteur, lors de la définition d'un tel schéma, qu'il choisisse la sur-classe "dominante" dont la sous-classe héritera;

* soit ils appliquent une règle déterministe pour ce choix (par exemple la première classe citée comme sur-classe dans la définition textuelle de la sous-classe).

3. ODL: le langage de définition de données [2, 4]

3.1 Hiérarchie des classes

Afin de mieux cerner le langage ODL, nous présentons d'abord la hiérarchie des classes prédéfinies d'ODMG. L'indentation représente les liens de généralisation / spécialisation.

```
Denotable_Object
    Object
        Atomic_Object
            Type
            Iterator
            …
        Structured_Object
            Collection(<T>)
                Set(<T>)
                List(<T>)
                Bag(<T>)
                Array(<T>)
        Structure<e1:T1; …; en:Tn>
    Literal
        Atomic_Literal
            Integer
            Float
            Boolean
            String
        Structured_Literal
            Immutable_Structure<e1:T1; …; en:Tn>
                Date
                Time
                …
            Immutable_Collection
                Enumeration
```

Fig. 15 : Hiérarchie de classes du langage ODL d'ODMG

Les classes créées par le concepteur de BD-OO suivant la norme ODMG sont automatiquement sous-classes de Object.

3.2 Les méthodes les plus utilisées associées aux classes

3.2.1 Méthodes associées à la classe Object

Object est la racine de la hiérarchie des classes définies par les utilisateurs. Les méthodes prédéfinies suivantes sont utilisées par le SGBD :

CREATE([nom_attribut : valeur, …]) → o:Oid

Cette méthode permet la création d'un objet avec initialisation éventuelle de sa valeur et passage en réponse de l'OID de l'objet créé.

DELETE()

Méthode pour la suppression de l'objet.

SAME_AS(o : OID) → b : Boolean

Test d'identité : teste si l'objet appelé et l'objet passé en paramètre sont identiques.

3.2.2 Méthodes associées à la classe Collection<T>

CREATE(<T>) → c : Collection<T> */ méthode **redéfinie**/*

Méthode permettant la création d'une collection dont les éléments seront de type T. Pratiquement, on ne peut pas créer de collection générique, mais une instance d'un de ses sous-types.

INSERT_ELEMENT(o : Denotable_Object)

REMOVE_ELEMENT(o : Denotable_Object)

SELECT_ELEMENT(predicate : String) → element : T

SELECT (predicate : String) → c : Collection<T>

3.2.3 Méthodes associées à la classe Set<T>

UNION(s1, s2 : Set<T>) → s3 : Set<T>

INTERSECTION(s1, s2 : Set<T>) → s3 : Set<T>

DIFFERENCE(s1, s2 : Set<T>) → s3 : Set<T>

IS_SUBSET?(s1, s2 : Set<T>) → b : Boolean

3.2.4 Méthodes associées à la classe List<T>

FIRST(T)

INSERT_ELEMENT_BEFORE(o : T, position : Integer)

INSERT_FIRST_ELEMENT(o :T)

RETRIEVE_ELEMENT_AT(position : Integer) → element : T

3.2.5 Méthodes associées à la classe Structure<e1:T1, … , en:Tn>

CREATE(<nomatt$_1$: T$_1$, … , nomatt$_n$: T$_n$>) → Structure<nomatt$_1$: T$_1$, … , nomatt$_n$: T$_n$>

GET_ELEMENT_VALUE(nomatt) → value : Denotable_Object

SET_ELEMENT_VALUE(nomatt, value : Denotable_Object)

CLEAR_ELEMENT_VALUE(nomatt)

3.3 Extrait partiel de la syntaxe ODL d'ODMG

3.3.1 Définition d'une classe

Class nom-classe1 (**Extent** nom-population, **Key** < clés>) {
Attribute < def-attribut> ;
Relationship nom-relation1 : [**List** / **Set** / **Bag** / **Array**] nom-classe2 [**Inverse** nom-classe2.nom-relation2] ;
<domaine-résultat> nom-méthode ([nom-paramètre : <domaine-paramètre> , ...]) ;
} ;

Fig. 16 : Définition abstraite d'une classe en syntaxe ODL

3.3.2 Définition d'un domaine :

Typedef nom-domaine
[**List** / **Set** / **Bag** / **Array**] <domaine-s/c>) ;

avec:
<domaine-s/c> :: <domaine-simple> / <domaine-complexe>

<domaine-simple> :: **String** / **Int** / **Float** / **Date**/ **Boolean** / **Enum** (…,valeur)

<domaine-complexe> :: **Struct** nom-domaine {<def-attribut>; ...}

Fig. 17 : Définition abstraite d'un domaine en syntaxe ODL

3.4 Exemple complet d'un schéma ODL

```
Class Voiture (extent lesVoitures key nveh) {
        attribute string nveh ;
        attribute string couleur ;
        attribute string marque ;
        attribute short km ;
        relationship Personne Appartient inverse Personne : : Possede ;
        short rouler (in short distance) ;  } ;
Interface Personne {
        attribute string nss;
        attribute string nom;
        attribute string prenom;
        attribute date datenaissance;
        relationship Appart habite inverse Appart : : loge ;
        relationship Voiture Possede inverse Voiture : : appartient ;
        short vieillir( ) ;
        void dormir ( )
        short age( ) ;   } ;
class Employé : Personne (extent lesEmployés key nss) {
        attribute enum fonct {ingénieur, secrétaire, analyste, programmeur} ;
        attribute float salaire ;
        attribute list<float> primes ;
        relationship Set<Employé> inférieur inverse supérieur ;
        relationship Employé supérieur inverse inférieur ;
        void travailler( ) ;   } ;
class Sportif : Personne (extent lesSportifs key nss) {
        attribute typesport {chrono, group, indiv} type;
        attribute niveausport {professionnel, debutant} niveau;
        relationship list<Sport> pratique inverse sport : : pratiqué_par ;
        void pratiquer (in Sport s) ;   } ;
class Appart (extent lesApparts) {
        attribute struct adresse (short etage, unsigned short numero, string rue,
        short code, string ville);
        relationship Set<Personne> loge inverse Personne: : habite;   };
class Sport (extent lesSports) {
        attribute string nomsport ;
        attribute integer nbrejoueurs ;
        relationship list<Sportif> pratiqué_par inverse Sport : : pratiquer ;   } ;
class EmployéSportif : Employé {
        attribute typesport {chrono, group, indiv} type;
        attribute niveausport {avancé, debutant} niveau;
        relationship list<Sport> pratique inverse sport : : pratiqué_par ;
        void pratiquer (in Sport s) ;
} ;
```

4. OQL: le langage de manipulation de données [3]

A l'instar du SQL employé dans l'interrogation des bases de données relationnelles, OQL est un langage déclaratif basé sur le standard du langage de requête SQL mais il n'est pas compatible avec lui. En SQL, la clause basique **SELECT-FROM-WHERE** sélectionne (SELECT) une liste d'attributs à partir (FROM) des tables spécifiées lorsque (WHERE) une condition est satisfaite. OQL adopte lui aussi le format familier de la clause SELECT-FROM-WHERE.

Puisqu'il est basé sur un modèle objet, sa clause FROM spécifie les collections d'objets dans lesquelles on effectue la recherche de données. Sa clause WHERE décrit les propriétés que doivent satisfaire les données en résultats de la requête. La clause SELECT en OQL définit la structure du résultat de la recherche.

OQL permet d'écrire des requêtes, mais pas des instructions de mise à jour (MAJ) contrairement à son prédécesseur. Les MAJ se font grâce aux méthodes associées aux classes. OQL peut être utilisé seul ou à partir d'un langage de programmation, tel que C++ ou Java. Ses requêtes peuvent appeler des méthodes, et inversement tout programme ou méthode peut contenir des requêtes OQL.

Si la définition des données de la base via ODL peut être apprise à partir de formules de prédicats aussi abstraites soient elles, il en va tout autrement de l'interrogation de la BD-OO. Le langage de manipulation de données, en l'occurrence OQL, est celui qui s'explique le mieux à travers des exemples. Nous allons faire l'étude d'OQL sur le schéma ODL de la section 3.4 tout en précisant à chaque fois le littéral.

4.1 Calcul d'expressions

OQL permet de calculer des expressions arithmétiques de base, qui sont donc comme des questions.

(Q0) ((STRING) 10*5/2**) ||** "BASESEDONNEES"

===> **LITTÉRAL STRING**

Ceci donne la chaîne « 25BASESEDONNEES » obtenue après calcul de l'expression 10*5/2 qui est 25, sa conversion en chaîne de caractères et sa concaténation avec la chaîne BASESEDONNEES. Cette requête montre à la fois le calcul d'expressions et les conversions de type. Son profil syntaxique est :

(<TYPE>**)** **<**EXPRESSION**>**

Où l'expression peut être calculée avec tous les opérateurs classiques (+, *, –, /, mod, abs, concaténation). Plus généralement, l'expression peut aussi être une collection d'objets, et donc une requête produisant une telle collection, comme nous en verrons beaucoup ci-dessous.

4.2 Accès à partir d'objets nommés

OQL permet de formuler des questions sous forme d'expressions simples, en particulier construites à partir du nom d'un objet. Si MAVOITURE est le nom de l'objet désignant ma voiture, il est possible de demander :

(Q1) MAVOITURE**.**couleur

===> LITTÉRAL STRING

Plus généralement, OQL permet de naviguer dans la base en parcourant des chemins monovalués, comme vu ci-dessus. Par exemple, la question (Q2) retourne le nom du propriétaire de ma voiture (en principe, mon nom !) :

(Q2) MAVOITURE**.**Appartient**.**nom

===> LITTÉRAL STRING

Nous appellerons de telles requêtes des extractions d'objets. Son profil syntaxique est :

<NOM OBJET**>.<**CHEMIN**>**

Où <CHEMIN> désigne une expression de chemins.

4.3 Sélection avec qualification

Nous retrouvons ici les expressions de sélection du SQL de base. La notation de SQL pour la définition de variable derrière le FROM (du style Sportifs AS B, ou Sportifs B) peut d'ailleurs être utilisée. La requête suivante retrouve les noms et prénoms des sportifs débutants :

(Q3) **SELECT** B**.**nom, B**.**prenom
FROM B IN lesSportifs
WHERE B**.**niveau = "debutant"

==> LITTÉRAL BAG<STRUCT(STRING,STRING)>

La syntaxe générale de sélections simples est:

SELECT [<VARIABLE**>.<**ATTRIBUT**>]+**
FROM <VARIABLE**> IN <**COLLECTION**>**
WHERE <VARIABLE**>.<**ATTRIBUT**> <**COMPARATEUR**> <**CONSTANTE**>**

4.4 Expression de jointures

OQL permet les jointures, tout comme SQL, avec une syntaxe similaire. Par exemple, la requête suivante liste les noms et prénoms des employés sportifs pratiquant des sports individuels :

(Q4) **SELECT** B.nom, B.prenom
FROM B **IN** lesSportifs**,** E **IN** lesEmployes
WHERE B.nss = E.nss A**ND** B.type = “indiv”

==> LITTÉRAL SET<STRUCT(STRING,STRING)>

Pourquoi SET à la place de GAB, parce que nous utilisant l’attribut nss qui est une clé.

Le profil syntaxique d’une requête de sélection avec un ou plusieurs critères de sélection et une ou plusieurs jointures est :

SELECT [<VARIABLE**>.<**ATTRIBUT**>]+**
FROM <VARIABLE**> IN <**COLLECTION**>,[<**VARIABLE**> IN <**COLLECTION**>]+**
[WHERE <VARIABLE**>.<**ATTRIBUT**> <**COMPARATEUR**>**
<VARIABLE**>.<**ATTRIBUT**>**
[AND <VARIABLE**>.<**ATTRIBUT**> <**COMPARATEUR**> <**VARIABLE**>.<**ATTRIBUT**>]***
[{AND|OR} <VARIABLE**>.<**ATTRIBUT**> <**COMPARATEUR**> <**CONSTANTE**>]]***

De manière générale, les qualifications peuvent faire intervenir, comme en SQL, des AND et des OR, et des expressions parenthésées contenant ces connecteurs logiques. Une question équivalente peut utiliser le type EmployéSportif comme suit :

(Q5) **SELECT ((**EmployéSportif**)**B**).**nom**, ((**EmployéSportif**)**B**).**prenom
FROM B **IN** lesSportifs
WHERE B.TYPE = “indiv”

==> LITTÉRAL BAG<STRUCT(STRING, STRING)>

L'évaluateur doit alors vérifier à l'exécution l'appartenance des sportifs individuels trouvés à la classe des EmployéSportif. Son profil syntaxique est :

SELECT [((<CLASSE**>)<**VARIABLE**>).<**ATTRIBUT**>]+**
FROM...
[WHERE...]

4.5 Parcours d'associations multivaluées par des collections dépendantes

Les collections dépendantes se parcourent par des variables imbriquées derrière le FROM. Ceci permet de traverser des chemins multivalués correspondant à des associations [1:N] ou [M:N]. Par exemple, la requête suivante retrouve le nom et le prénom des sportifs qui pratiquent le sport à un niveau professionnel :

(Q6) **SELECT** B.nom, B.prenom
FROM B **IN** lesSportifs, V **IN** B.pratique
WHERE V.niveau = "professionnel"

==> LITTÉRAL BAG<STRUCT(STRING, STRING)>

Le profil syntaxique général d'une requête de parcours de collections dépendantes est :

SELECT [<VARIABLE**>.<**ATTRIBUT**>]+**
FROM...,[<VARIABLE**> IN <**VARIABLE**>.<**ATTRIBUT**>]+**
[WHERE...]

Bien sûr, la clause WHERE peut être composée de sélections, de jointures, etc.

4.6 Sélection d'une structure en résultat

Il est possible de sélectionner une structure en résultat ; c'est d'ailleurs l'option par défaut que nous avions utilisée jusque-là, les noms des attributs étant directement ceux des attributs sources. La requête suivante permet de retrouver des doublons (Name, City, PostCode) pour chaque sportif individuel :

(Q7) **SELECT STRUCT (**NAME: B.nom, CITY: B.habite.adresse.ville,
POSTCODE: B.habite.adresse.code**)**
FROM B **IN** lesSportifs
WHERE B.type = 'indiv'

===> LITTÉRAL BAG <STRUCT(STRING,STRING,SHORT)>

Par similarité avec SQL, la collection résultat est un BAG. Il est aussi possible d'obtenir une collection de type SET en résultat ; on utilise alors le mot clé **DISTINCT**, comme en SQL :

(Q8) SELECT DISTINCT STRUCT (NAME: B.nom, CITY: B.habite.adresse.ville,
POSTCODE: B.habite.adresse.code**)**
FROM B **IN** lesSportifs
WHERE B.nom = ''LAHOULOU''

===> LITTÉRAL SET <STRUCT(STRING,STRING,SHORT)>

Par contre ici, on ne vise pas les sportifs individuels mais les ceux dont le nom est LAHOULOU. Le profil syntaxique est:

SELECT [DISTINCT][STRUCT]([<ATTRIBUT**>: <**VARIABLE**>.<**CHEMIN**>]+)**
FROM...
[WHERE...]

4.7 Calcul de collections en résultat

Plus généralement, le résultat d'une requête peut être une collection quelconque de littéraux ou d'objets. Par exemple, la requête suivante fournit une liste de structures :

(Q9) LIST(SELECT STRUCT(NOM: B.nom, CITY: B.habite.adresse.ville**)**
FROM B **IN** lesSportifs
WHERE B.**NOM = "**LAHOULOU**")**

===> LITTÉRAL LIST <STRUCT(STRING, STRING)>

Le profil syntaxique plus général est :

<COLLECTION**> (SELECT [DISTINCT]**
[STRUCT] ([<ATTRIBUT**>: <**VARIABLE**>.<**CHEMIN**>]+)**
FROM...
[WHERE...])

4.8 Manipulation des identifiants d'objets

Les identifiants sont accessibles à l'utilisateur. Il est par exemple possible de retrouver des collections d'identifiants d'objets par des requêtes du type :

(Q10) ARRAY(SELECT V
FROM V **IN** lesVoitures
WHERE B.marque = "HYUNDAI"**)**

===> LITTÉRAL ARRAY(<OID>)

dont le profil est :

<COLLECTION**> (SELECT <**VARIABLE**>**

FROM...

[WHERE...])

4.9 Application de méthodes en qualification et en résultat

La question suivante sélectionne les employés ainsi que leur ville d'habitation et leur âge, dont le salaire est supérieur à 10 000 et l'âge inférieur à 30 ans (age() est une méthode de la classe Personne) :

(Q11) SELECT DISTINCT E.nom, E.habite.adresse.ville, E.age()

FROM E **IN** lesEmployés

WHERE E.salaire > 10000 **AND** E.AGE() < 30

===> LITTÉRAL SET <STRING, STRING, INT>

Elle peut être généralisée au profil suivant :

SELECT [DISTINCT]...,

[<VARIABLE**>.<**CHEMIN**>.<**OPERATION**>([<**ARGUMENT**>]*)]***

FROM...

[WHERE....

[AND<VARIABLE**>.<**CHEMIN**>.<**OPERATION**>([<**ARGUMENT**>]*)**

<COMPARATEUR**>**

<CONSTANTE**>]*]**

4.10 Imbrication de SELECT en résultat

Afin de construire des structures imbriquées, OQL permet d'utiliser des SELECT imbriqués en résultat de SELECT. Par exemple, la question suivante calcule pour chaque employé une structure comportant son nom et la liste de ses inférieurs mieux payés :

(Q12) SELECT DISTINCT STRUCT (NOM : E.nom, INF_MIEUX_PAYES **:**

LIST (SELECT F

FROM F **IN** E.inférieur

WHERE F.salaire > E.salaire**))**

FROM E **IN** lesEmployés

===> LITTÉRAL SET <STRUCT (STRING, LIST<OID>)>

Voici un profil possible pour une telle question :

SELECT [DISTINCT][STRUCT](...,[<ATTRIBUT**>:<**QUESTION**>...]*)**
FROM...
[WHERE...]

4.11 Création d'objets en résultat

Il est possible de créer des objets dans une classe par le biais de constructeurs ayant pour arguments des requêtes. Bien sûr, des constructeurs simples peuvent être appliqués pour insérer des objets dans des extensions de classes, par exemple la requête :

(Q13) EMPLOYÉ (NSS:15603300036029, NOM:"Atidel", SALAIRE:100000).

Plus généralement, il est possible d'employer tous les sportifs qui ne sont pas employés par la requête suivante :

(Q14) EMPLOYÉ**(SELECT STRUCT(**NSS : B**.**nss, NOM: B**.**nom, SALAIRE : 4000**)**
FROM B **IN** lesSportifs
WHERE NOT EXISTS E **IN** lesEmployés : E**.**nss=B**.**nss**)**

==> LITTÉRAL SET<EMPLOYE> INSÉRÉ DANS lesEmployés

Notez que cette requête utilise un quantificateur EXISTS que nous allons expliciter ci-dessous. Le profil général de ces requêtes de création d'objets est donc :

<CLASSE**>(<**QUESTION**>).**

4.12 Quantification de variables

OQL propose des opérateurs de quantification universelle et existentielle directement utilisables pour composer des requêtes, donc des résultats ou des qualifications.

4.12.1. Le quantificateur universel(FOR ALL)

La question suivante retourne vrai si tous les employés ont moins de 60 ans :

(Q15) FOR ALL P **IN** lesEmployés : P**.**age() < 60

Son profil est :

FOR ALL <VARIABLE**> IN <**QUESTION**> : <**QUESTION**>**

4.12.2. Le quantificateur existentiel (EXISTS)

La requête suivante retourne vrai s'il existe une voiture de marque Hyundai possédée par une personne de plus de 20 ans :

(Q16) EXISTS V **IN SELECT** V
FROM V **IN** lesVoitures**,** B **IN** V**.**possede
WHERE V.marque = "HYUNDAI" **:** B**.**age() > 20.

Un profil généralisé possible est :

EXISTS <VARIABLE**> IN <**QUESTION**> : <**QUESTION**>**

4.13 Calcul d'agrégats et opérateur GROUP BY

Dans la première version, les agrégats étaient calculables par utilisation d'un opérateur fonctionnel GROUP applicable sur toute requête. La version 2 est revenue à une syntaxe proche de celle de SQL pour des raisons de compatibilité. Les attributs à grouper sont généralement définis par une liste d'attributs, alors que le critère de groupement peut être explicitement défini par des prédicats, ou implicitement par une liste d'attributs. Dans le dernier cas, le mot clé PARTITION peut être utilisé pour désigner chaque collection résultant du partitionnement. Les deux requêtes suivantes illustrent ces possibilités :

(Q17) SELECT E
FROM E **IN** lesEmployés
GROUP BY (BAS : E**.**salaire < 7000**,**
MOYEN **:** E**.**salaire > 7000 AND E**.** salaire < 21000**,**
HAUT **:** E**.** salaire ≥ 21000**)**

===> Littéral STRUCT(BAS: SET(EMP.), MOYEN:SET(EMP.),HAUT:SET(EMP.))

(Q18) SELECT STRUCT(VILLE: E**.**habite**.**adresse**.**ville,
MOY : **AVG**(SALAIRE))
FROM E **IN** lesEmployés
GROUP BY E.habite**.**adresse**.**ville
HAVING AVG (**SELECT** E.salaire **FROM** E **IN PARTITION**)> 5000

===> Littéral BAG <STRUCT (VILLE: STRING, MOY: FLOAT)>

Le profil syntaxique est :

SELECT...[<AGGREGAT>(<ATTRIBUT>)]*

FROM...

[WHERE...]

GROUP BY {<ATTRIBUT>+ | <PREDICAT>+}

HAVING <PRÉDICAT>

AGGREGAT désigne une fonction d'agrégats choisie parmi COUNT, SUM, MIN, MAX et AVG. PREDICAT désigne une expression logique de la forme <ATTRIBUT> <COMPARATEUR> <CONSTANTE> ou

<AGGREGAT>(PARTITION) <COMPARATEUR> <CONSTANTE>.

4.14 Expressions de collections

Il est possible de manipuler directement les collections en appliquant des opérateurs spécifiques. Les résultats de requêtes sont souvent des collections, si bien que les opérateurs peuvent être appliqués aux requêtes. Voici quelques exemples d'opérateurs.

4.14.1. Tris

La première syntaxe était fonctionnelle. La nouvelle est copiée sur SQL, comme suit :

(Q19) SELECT E.nom, E.salaire
FROM E **IN** lesEmployés
WHERE E.salaire > 21000
ORDER BY DESC E.salaire

Le profil est donc analogue à celui de SQL :

<COLLECTION**>**

ORDER BY {DESC|ASC} <ATTRIBUT**>+**

4.14.2. Union, intersection, différence

Comme en SQL, union, intersection et différence de questions sont possibles. Par Exemple la requête Q20 est une requête valide:

(Q20) lesEmployés **UNION** lesSportifs

===> Littéral BAG <OID>

Le profil général de ce type de requête est :

<COLLECTION**> {INTERSECT|UNION|EXCEPT} <**COLLECTION**>**

Les priorités sont selon l'ordre indiqué, les parenthèses étant aussi possibles.

4.14.3. Accesseurs aux collections

Il est possible d'extraire un élément d'une collection en utilisant une fonction d'accès générale, comme first ou last, ou plus généralement des fonctions d'accès spécifiques au type de collection. Les constructeurs de collections STRUCT, SET, LIST, BAG, ARRAY sont aussi utilisables pour construire des collections comme nous l'avons déjà vu. Ainsi :

(Q21) SELECT B.nom, **FIRST(**B.pratique**)**

FROM B **IN** lesSportifs

Sélectionne le premier sport pratiqué par le sportif.

Plus généralement :

FIRST(<COLLECTION>) |

LAST(<COLLECTION>) |

<FONCTION>(<COLLECTION>)

Sont des profils admissibles.

5. Exercice corrigé [2]

Soit le schéma ODL de la base de données « Institut_de_Formation » de la figure 18. Nous donnons en premier temps une liste de requêtes OQL pour lesquelles nous cherchons à comprendre : (1) ce que sera le résultat de l'exécution de chaque requête, et (2) le littéral de chaque résultat. Le lecteur confrontera ensuite ses réponses avec celles fournies en section 5.2.

```
Class Personne {
        attribute String nom ;
        attribute List <String> prénoms ;
        attribute Tadresse adresse ;
        Void afficher() ;
        Void nouvelle_adresse(nvadr : Tadresse)
} ;
Class Etudiant : Personne (Extent lesEtudiants, Key n°E) {
        attribute Int n°E ;
        attribute Date dateN ;
        attribute études : List Struct { Int année ; String diplôme  } Etude;
        relationship obtient List <CoursObtenu> Inverse CoursObtenu : : est_obtenu ;
        relationship suit Set <Cours> Inverse Cours : : est_suivi ;
        Void afficher() ;
        Void inscrire ( nvcours : Cours ) ;
        Void aobtenu ( nvcours : Cours , note : Float , année : Int ) ;
        Int age()
} ;
Class CoursObtenu {
        attribute Int année ;
        attribute Float note ;
        relationship fait_partie Cours Inverse Cours : : est_réussi ;
        relationship est_obtenu Etudiant Inverse Etudiant : : obtient ;
} ;
Class Enseignant : Personne (Extent lesEnseignants) {
        attribute Int tél ;
        attribute statut : Enum ( "prof", "Assist" ) ;
        attribute Struct RensBq { String banque ; String agence; Int compte } rb;
        relationship assure Set Cours Inverse Cours : :  est_assuré ;
        Void afficher() ;
        Void assure (nvcours : Cours) ;
        Void nassureplus (oldcours : Cours)
} ;
Class Cours (Extent lesCours, Key nomC) {
        attribute String nomC ;
        attribute Int cycle ;
        relationship est_assuré Enseignant Inverse Enseignant : : assure ;
        relationship est_suivi  Set <Etudiant> Inverse Etudiant : : suit ;
        relationship a-prérequis  Set <Cours> Inverse Cours : : est-prérequis ;
        relationship est-prérequis  Set <Cours> Inverse Cours : : a-prérequis ;
        relationship est_réussi Set <CoursObtenu> Inverse CoursObtenu : : fait_partie ;
        Void afficher() ;
        Int nb-inscrits()
} ;
Typedef Tadresse Struct (
        attribute String rue ;
        attribute short numéro ;
        attribute String ville ;
        attribute short NPA
) ;
```

Fig. 18 : Schéma ODL de la base « Institut_de_formation »

5.1 Liste des requêtes OQL

Numéro	Requête OQL
(Q1)	lesCours
(Q2)	e: Etudiant *//déclaration d'une variable e de type Etudiant* e = Etudiant(nom:"Mohamed", adresse:"Jijel", n°E:2456)
(Q3)	**SET**(1,2,3)
(Q4)	**LIST**("Amel", "Amina")
(Q5)	**BAG**(1,1,2,3,2)
(Q6)	**STRUCT**(nom:"Mohamed", prénoms:**LIST** ("Ali","Amine"), adresse:"Jijel", n°E:2456)
(Q7)	**SELECT DISTINCT STRUCT** (nom: c.nomC , prereq: (**SELECT** p.nomC **FROM** p **IN** c.a_prerequis)) **FROM** c **IN** LesCours ;
(Q8)	**SELECT** p **FROM** p **IN** lesPersonnes **WHERE** p.nom="Mohamed" **AND** p.prénoms.first()="Ali" ;
(Q9)	**SELECT DISTINCT** p.nom **FROM** e **IN** lesEtudiants, c **IN** lesCours, p **IN** lesEnseignants **WHERE** e.n°E=136 **AND** c **IN** e.suit **AND** p == c.assure ;
(Q10)	**SELECT DISTINCT** c.assure.nom **FROM** e **IN** lesEtudiants, c **IN** e. suit **WHERE** e.n°E=136 ,

(Q11)	**SELECT** e **FROM** e **IN** lesEnseignants **WHERE EXISTS** c **IN** e.assure **:** c.cycle=3 ;
(Q12)	**SELECT** e **FROM** e **IN** lesEnseignants, c **IN** lesCours **WHERE** c.nomC = "BDA" **AND** c **IN** e.assure ;
(Q13)	**COUNT** (**SELECT** e **FROM** e **IN** lesEnseignants **WHERE** e.statut="prof") ;
(Q14)	**GROUP** c **IN** lesCours **BY** (cycle : c.cycle) **WITH** (nbcours : **COUNT**(**PARTITION**));
(Q15)	(**SELECT** e **FROM** e **IN** lesEtudiants, c **IN** e.cours_obtenus **WHERE** c.cours.nomC='BDA') **EXCEPT** (**SELECT** e **FROM** e **IN** lesEtudiants, c **IN** e.cours_obtenus **WHERE** c.cours.nomC='ALGO') ;
(Q16)	**SELECT DISTINCT** e.nom **FROM** e **IN** lesEtudiants **WHERE** e.age()<25 ;
(Q17)	**SELECT** c.nomC **FROM** c **IN** lesCours **WHERE** c.nb_inscrits()>120 ;

5.2 Réponses (données recherchées et littéraux)

Numéro	Données recherchées / littéral
(Q1)	Fournit l'ensemble des objets Cours.
	SET<OID>
(Q2)	Crée un objet de type Etudiant dont l'OID est retourné dans la variable e. Ses attributs sont initialisés: nom à la valeur "Mohamed", adresse à la valeur "Jijel" et n°E à la valeur 2456. Quant à dateN, prénoms et études, ils sont initialisés à une valeur par défaut: ✓ nul pour un attribut monovalué, ✓ vide pour un attribut de type collection.
	<OID>
(Q3)	Crée une valeur de type ensemble de valeurs;
	SET<INT>
(Q4)	Crée une valeur de type liste de valeurs;
	LIST<STRING>
(Q5)	Crée une valeur de type multi-ensemble de valeurs.
	BAG<INT>
(Q6)	Crée une valeur complexe (et non un objet);
	STRUCT (STRING, LIST<STRING>, STRING, INT)
(Q7)	Crée un ensemble de valeurs structurées contenant pour chaque cours, son nom et l'ensemble des noms de ses cours pré-requis.
	SET(STRUCT(STRING ,SET<STRING>)
(Q8)	Donne les personnes de nom Mohamed et de premier prénom Ali. Ceci regroupe tous les Mohamed Ali qu'ils soient enseignants ou étudiants, ou ni l'un ni l'autre.
	<OID>

(Q9)	Donne les noms des enseignants assurant les cours suivis par l'étudiant de numéro 136
	SET<STRING>
(Q10)	Même chose que (Q9) mais (Q10) est optimisée
	SET<STRING>
(Q11)	Retrouve les enseignants qui donnent au moins un cours de 3ème cycle.
	SET<OID>
(Q12)	Retrouve les enseignants du cours "BDA".
	SET<OID>
(Q13)	Compte le nombre total de professeurs.
	INT
(Q14)	Donne en résultat un ensemble de couples (cycle:INT, nbcours:INT) qui définissent pour chaque cycle le nombre de cours de ce cycle.
	SET<INT, INT>
(Q15)	recherche les étudiants qui ont obtenu le cours 'BDA', mais pas le cours 'ALGO'
	SET<OID>
(Q16)	Donne les noms des étudiants de moins de 20 ans, (age() est une méthode de la classe Etudiant)
	SET<STRING>
(Q17)	Recherche les noms des cours qui ont plus de 120 inscrits (nb_inscrits() est une méthode de la classe Cours)
	BAG<STRING>

6. Conclusion

Les premiers SGBDO ont été construits à partir de langages de programmation orientés objets rendus persistants, c'est-à-dire que le système a offert une nouvelle fonction qui permet à l'utilisateur de rendre permanents les objets manipulés par le langage. Les fonctions usuelles des SGBD (fiabilité, partage simultané, optimisation des requêtes…) étaient très mal assurées. L'interface utilisateur était de type navigationnel uniquement.

Les SGBD-OO de la seconde vague sont eux aussi de "purs" SGBO, très inspirés des langages de programmation orientés objets. Mais ils offrent un meilleur éventail des fonctions des SGBD classiques, dont notamment un LMD déclaratif plus ou moins complet. Le groupe ODMG qui rassemble un ensemble de constructeurs de SGBDO de cette génération propose un ensemble d'interfaces standard comprenant entre autres un modèle de données et un LMD déclaratif.

A la fin des années 90, une nouvelle génération de SGBD appelés relationnels objets ("objectrelational") est née. Ce sont des SGBD basés sur le modèle relationnel étendu aux concepts principaux de l'orienté objets (extensibilité de l'ensemble des types, structures complexes, identité d'objet, héritage et méthodes). Ils sont aussi accompagnés de bibliothèques de classes (appelées selon les constructeurs datablades, cartridges ou extenders) spécialisées sur des thèmes, tels que le spatial, le temporel… Les grands constructeurs de SGBD relationnels proposent des SGBD de ce type et ont participé à l'élaboration de la norme de la nouvelle version de SQL, SQL3, qui est de type relationnel-objet. Il est probable que ce soit cette dernière génération de SGBD qui remporte la plus grande part du marché. Nous consacrons le chapitre suivant aux bases de données relationnelles objets.

PARTIE II

LES BASES DE DONNEES RELATIONNELLES OBJET (BD-RO)

Chapitre 4

Eléments de base sur la transition vers le Relationnel-Objet (RO)

- Pourquoi s'orienter plutôt vers le RO ?
- Objectifs des SGBD-RO
- Création de nouveaux types utilisateurs en SQL3
- Création de tables en SQL3
- Identité et attributs référence
- Hiérarchie des types OBJECT

1. Introduction

Les SGBD-RO sont nés du double constat de la puissance nouvelle promise par les SGBDOO et de l'insuffisance de leur réalité pour répondre aux exigences de l'industrie des BD classiques. Leur approche est plutôt d'introduire dans les SGBD-R les concepts apportés par les SGBD-OO plutôt que de concevoir de nouveaux systèmes.

Le Modèle relationnel-objet (appelé parfois modèle objet-relationnel) est le modèle relationnel étendu avec des principes objet afin d'en augmenter les potentialités.

2. Pourquoi s'orienter plutôt vers l'extension ? [5]

Les SGBD-OO ont été créés pour gérer des structures de données complexes, en profitant de la puissance de modélisation des modèles objets et de la puissance de stockage des BD classiques. Leurs objectifs peuvent être résumés dans les points suivants :

- ✓ Offrir aux langages de programmation orientés objets des modalités de stockage permanent et de partage entre plusieurs utilisateurs
- ✓ Offrir aux BD des types de données complexes et extensibles
- ✓ Permettre la représentation de structures complexes et/ou à taille variable

Ainsi, les SGB-DOO présentent un éventail d'avantages permettant de surpasser les limites des bases de données conventionnelles parmi lesquels on peut citer :

- ✓ Le schéma d'une BD objet est plus facile à appréhender que celui d'une BD relationnelle (il contient plus de sémantique, il est plus proche des entités réelles)
- ✓ L'héritage permet de mieux structurer certains éléments de modélisation
- ✓ La création de ses propres types et l'intégration de méthodes permettent une représentation plus directe du domaine
- ✓ L'identification des objets permet de supprimer les clés artificielles souvent introduites pour atteindre la troisième forme normale (3FN) et donc de simplifier le schéma
- ✓ Les principes d'encapsulation et d'abstraction du modèle objet permettent de mieux séparer les BD de leurs applications (notion d'interface).

Cependant, les SGBDOO apportent des innovations sur des aspects que les SGBD-R ne savent pas faire, mais sans être au même niveau sur ce que les SGBDR savent bien faire. Le revers de la médaille cache ainsi un certain nombre d'inconvénient tels que :

- ✓ Gestion de la persistance et de la coexistence des objets en mémoire (pour leur manipulation applicative) et sur disque (pour leur persistance) complexe
- ✓ Gestion de la concurrence (transactions) plus difficile à mettre en œuvre
- ✓ Interdépendance forte des objets entre eux
- ✓ Gestion des pannes
- ✓ Complexité des systèmes (problème de fiabilité)
- ✓ Problème de compatibilité avec les SGBD-R classiques

Il ressort de cette analyse que le modèle relationnel présente des points forts indiscutables, mais il a aussi des points faibles. L'objet répond à ces faiblesses d'où l'intérêt d'une intégration douce de l'objet dans le relationnel.

3. Objectifs des SGBD-RO [5]

- ✓ Gérer des données complexes (temps, géo-référencement, multimédia, types utilisateurs, etc.)
- ✓ Rapprocher le modèle logique du modèle conceptuel
- ✓ Réduire la dichotomie d'impédance (voir explication en section 5.5)
- ✓ Réduire les pertes de performance liées à la normalisation et aux jointures

4. Création de nouveaux types utilisateur en SQL3 [2, 3, 5]

Le concept central du RO est celui de type défini par l'utilisateur (TDU), correspondant au concept de classe en POO qui permet d'injecter la notion d'objet dans le modèle relationnel. Les apports principaux des nouveaux types utilisateurs sont :

- La gestion de l'imbrication (données structurées et collections)
- Les méthodes et l'encapsulation
- L'héritage et la réutilisation de définition de types
- L'identité d'objet (OID) et les références physiques
- Créer des tables contenant
    - soit des tuples en première forme normale (1NF) spécifique au relationnel, tolérant uniquement des domaines prédéfinis monovalués
    - soit des tuples en non première forme normale (Non First Normal Form : NF^2) tolérant des domaines complexes et/ou multivalués
    - soit des objets
- Etablir des liens de composition par des attributs référence qui contiennent l'OID de l'objet composant

SQL3 est la version étendue du SQL relationnel avec des fonctions orientées objets, les déclencheurs, le multimédia, le spatial, les séries temporelles..., tout en restant compatible avec le SQL classique, c-à-d que toute instruction de SQL est toujours valable en SQL3. Pour

étendre le langage de requêtes relationnel tout en restant compatible, l'idée a été de **conserver la notion de table inchangée**, et d'ajouter à côté de nouvelles notions objets, essentiellement, les types (avec structure complexe et méthodes) et les OID.

Les nouveaux types définis par l'utilisateur (TDU) sont créés grâce au constructeur "TYPE" dont il existe plusieurs formes :

- VARRAY et NESTED TABLE : permettent d'avoir des attributs multivalués
- OBJECT : a une double fonction car il permet de créer :
  - soit des valeurs structurées (notamment pour les attributs complexes)
  - soit des objets, c'est à dire des couples <valeur structurée, OID>.

4.1 Les types VARRAY

Les types VARRAY permettent de créer des attributs multivalués sous la forme de tableaux à une dimension. L'instruction suivante:

CREATE TYPE nom-type **AS VARRAY** (nb-max) **OF** nom-type2 ;

déclare un nouveau type de nom "nom-type" dont chaque élément est un tableau à une dimension de taille maximale "nb-max", et dont les éléments sont tous du type "nom-type2". Le type nom-type2 peut être n'importe quel type, prédéfini de SQL (VarChar, Date, Number…) ou défini par l'utilisateur.

4.2 Les types NESTED TABLE

Les types NESTED TABLE"(ou table emboîtée) permettent de créer des multivalués sous la forme de tables relationnelles qui vont être incluses dans les autres tables où elles serviront de domaine à un attribut. L'instruction suivante:

CREATE TYPE nom-type **AS TABLE OF** nom-type2 ;

déclare un nouveau type de nom "nom-type" dont chaque élément est une table relationnelle dont les tuples sont tous du type "nom-type2". Le type nom-type2 peut être n'importe quel type, prédéfini de SQL ou défini par l'utilisateur.

Lors de l'emploi de ce type, nom-type, comme domaine d'un attribut d'une autre table, par exemple de l'attribut A de la table T, le système Oracle crée physiquement une table annexe (la nested TABLE) dans laquelle il stockera les tuples de l'attribut A. L'utilisateur doit, lors de

la création de la table T, définir le nom de cette table annexe. Pour cela, l'instruction CREATE TABLE contient une clause supplémentaire afin que l'utilisateur puisse nommer cette table annexe :

NESTED TABLE nom-attribut- type-table-emboîtée **STORE AS** nom-table-annexe ;

On résume dans le tableau ci-dessous les points qui permettent de distinguer les VARRAY des NESTED TABLE lors de leur utilisation.

	VARRAY	**NESTED TABLE**
Indexation	non	oui
Taille	statique	dynamique
Eléments ordonnés	oui	non

4.3 Les types OBJECT

Les types OBJECT créent des valeurs structurées et/ou des objets. L'instruction suivante:

CREATE TYPE nom-type **AS OBJECT**

(nom-attribut$_1$ nom-type$_1$, nom-attribut$_2$ nom-type$_2$, , nom-attribut$_n$ nom-type$_n$)

déclare un nouveau type de nom "nom-type" dont chaque élément est un objet structuré comportant les attributs nom-attribut$_1$ (de domaine nom-type$_1$), nom-attribut$_2$ (de domaine nom-type$_2$),… et nom-attribut$_n$ (de domaine nom-type$_n$). Les types nom-type$_i$ peuvent être n'importe quel type, prédéfini de SQL ou défini par l'utilisateur.

Cette instruction permet aussi de déclarer un jeu de méthodes associées aux objets de ce type (en préfixant la déclaration de la méthode du mot clé "MEMBER").

4.4 Exemples

Le premier exemple illustre l'utilisation du type OBJECT pour la définition de valeurs structurées.

```
CREATE TYPE T-Voiture AS OBJECT
        ( numéro VARCHAR(9) ,
        marque VARCHAR(18) ,
        modèle VARCHAR(18) ,
        année NUMBER ) /
```

Fig. 19 : Utilisation du type OBJECT pour créer des valeurs structurées

Il est donc possible aux utilisateurs de créer des valeurs structurées de type T-Voiture, comme suit :

T-Voiture ('VD 1212', 'Hyundai', 'Accent', 2009)

Ou si l'on ne veut pas renseigner tous les attributs :

T-Voiture (numéro: 'VD 1212', année: 2009)

Remarque importante :

Chacun des TDU de SQL3 ne permet de définir que des structures contenant un seul constructeur : pas d'attribut complexe contenant des attributs complexes ou multivalués, pas d'attribut complexe et multivalué. Pour créer une structure contenant plusieurs constructeurs, il faut définir un type intermédiaire par constructeur. Par exemple, pour représenter une personne avec ses enfants, et pour chaque enfant ses prénoms, on devra créer les types suivants :

```
CREATE TYPE T-Personne AS OBJECT
        ( nom VARCHAR(20) ,
        prénoms T-ListePrénoms ,          // attribut de type varray (4)
        enfants T-ListeEnfants ) /        // attribut de type varray(10)
CREATE TYPE T-ListePrénoms AS VARRAY (4) OF VARCHAR(20) ;
CREATE TYPE T-Enfant AS OBJECT
        ( prénoms T-ListePrénoms ,
        genre CHAR ,
        dateNais DATE ) /
CREATE TYPE T-ListeEnfants AS VARRAY (10) OF T-Enfant ;
```

Fig. 20 : version 1 avec nombre d'enfants limité à 10

```
CREATE TYPE T-Personne AS OBJECT
        ( nom VARCHAR(20) ,
        prénoms T-ListePrénoms ,
        enfants T-ListeEnfants ) /              // attribut de type nested table
CREATE TYPE T-ListePrénoms AS VARRAY (4) OF VARCHAR(20) ;
CREATE TYPE T-Enfant AS OBJECT
        ( prénoms T-ListePrénoms ,
        genre CHAR ,
        dateNais DATE ) /
CREATE TYPE T-ListeEnfants AS TABLE OF T-Enfant ;

// Plus tard dans la création des tables :
CREATE TABLE PERSONNE ……..              // voir section suivante
NESTED TABLE enfants STORE AS table_annexe_enfants ;
```

Fig. 21 : version 2 avec nombre d'enfants non prédéfini

5. Création de Tables en SQL3

Les tables sont les récipients qui stockent les valeurs ou les objets. Les tables peuvent être :

- Des tables du relationnel classique (contenant des valeurs de type tuple en première forme normale), ou
- Des tables à structure complexe contenant des valeurs structurées (en non première forme normale), ou
- Des tables contenant des objets (qui eux-mêmes peuvent être structurés en non première forme normale).

5.1 Création d'une table relationnelle classique

On emploie l'instruction CREATE TABLE du relationnel classique:

CREATE TABLE nom-table (nom-attribut$_1$ nom-type$_1$, nom-attribut$_2$ nom-type$_2$, … , nomattribut$_n$ nom-type$_n$)

Où les nom-type$_i$ sont des domaines prédéfinis de SQL, tels que VARCHAR, NUMBER, FLOAT, ….., etc. Cette table contiendra des valeurs de type tuple en première forme normale.

5.2 Création d'une table relationnelle en non première forme normale

On emploie l'instruction CREATE TABLE du relationnel classique mais en utilisant un ou plusieurs types créés précédemment par un constructeur CREATE TYPE.

> CREATE TABLE nom-table (nom-attribut$_1$ nom-type$_1$, nom-attribut$_2$ nom-type$_2$, … , nomattribut$_n$ nom-type$_n$)

Où au moins un des nom-type$_i$ est le nom d'un type construit (VARRAY, NESTED TABLE ou OBJECT). Cette table contiendra des valeurs structurées qui ne sont pas en première forme normale.

5.3 Création d'une table d'objets

On emploie l'instruction:

> CREATE TABLE nom-table OF nom-type

Où nom-type est le nom d'un type OBJECT (qui doit avoir été créé auparavant). Le format de la table est celui de nom-type. Cette table contiendra des objets, identifiés par leurs OID et structurés selon le type "nom-type".

6. Identité et attributs-référence

Tout ce qui est de type OBJECT possède une identité (OID) et peut donc être référencé par un lien de composition. Pour référencer un objet, on utilise la clause "REF nom-type", comme dans l'exemple ci-après:

```
CREATE TYPE T-Personne AS OBJECT
( AVS NUMBER ,
nom VARCHAR(18) ,
prénom VARCHAR(18) ,
conjoint REF T-Personne,        //attribut référence qui contient un OID
                                // de type T-Personne
voiture T-Voiture,              //attribut à valeur complexe de type T-Voiture
…. ) /
```

7. Hiérarchie de types OBJECT

En Oracle à partir de la version 9, il est possible de créer des hiérarchies de types OBJECT, puis d'utiliser ces types et leurs sous-types lors des créations de tables. Pour créer un type pour lequel on va déclarer des sous-types, il faut ajouter la clause NOT FINAL à son instruction CREATE TYPE; sinon, par défaut le type créé sera considéré par Oracle comme "final", c'est-à-dire sans sous-type.

Pour créer un sous-type d'un type non final, il faut rajouter la clause UNDER nom-sur-type. Le sous-type héritera alors des attributs et méthodes du sur-type.

N.B. A l'instar d'ODMG, SQL3 n'autorise pas l'héritage multiple ; ce qui revient à dire qu'un type ne peut avoir qu'un seul sur-type.

Exemple :

```
CREATE TYPE T-Personne AS OBJECT
        ( AVS NUMBER ,
        nom VARCHAR(18) ,
        prénom VARCHAR(18) ,
        adresse VARCHAR(200) )
                                        NOT FINAL /
CREATE TYPE T-Etudiant UNDER T-Personne
        ( faculté VARCHAR(18) ,
        cycle VARCHAR(18) )
/
```

Fig. 22 : Exemple de hiérarchie de type Object

Le type T-Etudiant est un type OBJECT, qui contient 6 attributs, AVS, nom, prénom, adresse, faculté et cycle. On peut l'utiliser, comme les autres types, pour créer des tables d'objets de type TEtudiant ou comme domaine pour des attributs. Le type T-Personne peut lui aussi être employé pour créer des tables d'objets ou servir de domaine pour des attributs. Dans le cas d'une table d'objets de type T-Personne et dans celui d'un attribut de type T-Personne, suivant le principe de substituabilité, l'utilisateur peut y insérer des valeurs de type T-Personne ou de type T-Etudiant.

8. Exemples de requêtes en SQL3 [3, 6]

Comme nous l'avons déjà expliqué en introduction, SQL3 est compatible avec SQL, donc toutes les requêtes en relationnel peuvent s'exécuter en SQL3. Nous allons maintenant nous intéresser à uniquement la partie du SQL étendu. Le lecteur est renvoyé sur d'autres documents pour des rappels sur le SQL conventionnel.

Considérons le code SQL3 suivant :

```
CREATE TYPE PERSONNE
(NSS INT, NOM VARCHAR, PRÉNOMS LIST(VARCHAR), TEL SET(PHONE)).
CREATE TYPE VOITURE (NUMÉRO CHAR(9), COULEUR VARCHAR,
                     PROPRIÉTAIRE REF(PERSONNE))
CREATE TYPE CONDUCTEUR (CONDUCTEUR VARCHAR, AGE INT)
CREATE TYPE ACCIDENT (ACCIDENT INT, RAPPORT TEXT, PHOTO IMAGE)
CREATE TABLE ACCIDENTS (ACCIDENT INT, RAPPORT TEXT, PHOTO IMAGE)
CREATE TABLE POLICE (NPOLICE INT, NOM VARCHAR, ADRESSE ADRESSE,
          CONDUCTEURS SET(CONDUCTEUR), ACCIDENTS LIST(ACCIDENT)).
CREATE TABLE EMPLOYÉS OF EMPLOYÉ ;
CREATE TABLE JUS(NJ INT, FRUIT VARCHAR, TENEUR FLOAT(5.2))
CREATE TABLE JUSDEFRUIT UNDER JUS (FRUIT INT, QUALITÉ VARCHAR).
CREATE TABLE EMPLOYÉSSPORTIFS UNDER EMPLOYÉS (ADRESS ADRESSE).
```

Fig. 22 : Code SQL3 de la base relationnelle objet « Agence_Assurances_Auto »

La requête (Q1) retrouve le nom et l'âge des employés de moins de 35 ans :

```
(Q1)  SELECT E.NOM, AGE(E)
      FROM EMPLOYÉS E
      WHERE AGE(E) < 35;
```

La notation pointée appliquée au premier argument est aussi utilisable pour invoquer les fonctions comme dans (Q2) :

```
(Q2)  SELECT E.NOM, E..AGE()
      FROM EMPLOYÉS E
      WHERE E..AGE() < 35;
```

Il s'agit d'un artifice syntaxique, la dernière version utilisant la double notation pointée (..) pour les fonctions et attributs composés, la notation pointée simple (.) étant réservée au SQL de base (accès à une colonne usuelle de tuple).

Au-delà des fonctions, SQL3 permet aussi l'accès aux attributs composés par la notation pointée. Supposons par exemple une table d'employés localisés définies comme suit :

La requête suivante permet de retrouver le nom et le jour de repos des employés d'Alger habitant Bab Zouar :

(Q3) **SELECT** NOM, REPOS
FROM EMPLOYÉSLOC E
WHERE DEPT(E.ADRESS) = "ALGER"
AND E.ADRESS..VILLE = "BAB ZOUAR";

Notons dans la requête ci-dessus l'usage de fonctions pour extraire le département à partir du code postal et l'usage de la notation pointée pour extraire un champ composé.
Nous préférerons utiliser partout la notation pointée ; il faut cependant distinguer fonction et accès à attribut par la présence de parenthèses.

Afin d'illustrer plus complètement, supposons définis un type point et une fonction distance entre deux points comme suit :

TYPE POINT **(**ABSCISSE **X,** ORDONNÉE **Y,**

FUNCTION DISTANCE**(P1** POINT**, P2** POINT**) RETURN (REAL)) ;**

Considérons une implémentation des employés par la table :

EMPLOYÉS (MUNEMP **INT,** NOM **VARCHAR,** ADRESS **ADRESSE,** POSITION **POINT) ;**

La requête suivante (Q4) recherche les noms et adresses de tous les employés à moins de 100 mètres de distance de l'employé Ahmed :

(Q4) **SELECT** E.NOM, E.ADRESS
FROM EMPLOYÉS G, EMPLOYÉS E
WHERE G.NAME = "AHMED" **AND** DISTANCE(G.POSITION,E.POSITION) < 100.

Définissons le type CERCLE comme suit :

TYPE CERCLE **(**CENTRE **POINT,** RAYON **REAL,**

CONSTRUCTOR FUNCTION CERCLE**(**C **POINT,** R **REAL) RETURN (**CERCLE**))**

Ajoutons une fonction booléenne CONTIENT au type point :

CREATE FUNCTION CONTIENT **(**P **POINT,** C **CERCLE)**

{ CODE DE LA FONCTION **} RETURN (BOLEAN)**

END FUNCTION.

La question suivante (Q5) retrouve les employés dont la position est contenue dans un cercle de rayon 100 autour de l'employé Ahmed :

(Q5) **SELECT** E.NAME, E.ADRESS
FROM EMPLOYÉS G, EMPLOYÉS E
WHERE EMPLOYÉS.NOM = "AHMED" **AND**
CONTIENT(E.POSITION, CERCLE(G.POSITION,1));

Les deux requêtes précédentes sont de fait équivalentes et génèrent les mêmes réponses.

8.1 Le parcours de référence

Comme nous l'avons déjà expliqué en section 6, SQL3 permet aussi de traverser les associations représentées par des attributs de type références.

Ces attributs peuvent être considérés comme du type particulier référence, sur lequel il est possible d'appliquer les fonctions Ref pour obtenir la valeur de l'OID et DeRef pour obtenir l'objet pointé.

Considérons le type VOITURE déjà défini ci-dessus comme suit :

CREATE TYPE VOITURE **(**NUMÉRO **CHAR(9),** COULEUR **VARCHAR,**

PROPRIÉTAIRE **REF(**PERSONNE**)).**

Créons une table de voitures :

CREATE TABLE VOITURES **OF TYPE** VOITURE**.**

La requête (Q6) recherche les noms des propriétaires de voitures rouges habitant Jijel :

(Q6) **SELECT** V.PROPRIETAIRE→NOM
FROM VOITURES V
WHERE V.COULEUR = "ROUGE" **AND** V.PROPRIETAIRE→ADRESSE..VILLE = "JIJEL".

SQL3 permet de multiples notations abrégées. En particulier, il est possible d'éviter de répéter des préfixes de chemins avec la notation pointée suivie de parenthèses. Par exemple, la requête suivante recherche les numéros des voitures dont le propriétaire habite Ouled Aissa à Jijel et a pour prénom Ahmed :

(Q7) **SELECT** V.NUMÉRO
FROM VOITURES V
WHERE V.PROPRIÉTAIRE→(ADRESSE..(VILLE = "JIJEL"
AND RUE = "OULED AISSA") **AND** PRÉNOM = "AHMED").

8.2 La recherche en collections

Les collections de base sont donc les ensembles, listes et sacs. Ces constructeurs de collections peuvent être appliqués sur tout type déjà défini. Les collections sont rendues permanentes en qualité d'attributs de tables. La construction TABLE est proposée pour transformer une collection en table et l'utiliser derrière un FROM comme une véritable table. Toute collection peut être utilisée à la place d'une table précédée de ce mot clé TABLE. Par exemple, supposons l'ajout d'une colonne PASSETEMPS à la table des personnes valuée par un ensemble de chaînes de caractères :

ALTER TABLE PERSONNES **ADD COLUMN** PASSETEMPS **SET(VARCHAR).**

La requête suivante retrouve les références des personnes ayant pour passe-temps le vélo :

(Q8) **SELECT REF**(P)
FROM PERSONNES P
WHERE "VÉLO" **IN**
SELECT *
FROM TABLE (P.PASSETEMPS)**.**

Plus complexe, la requête suivante recherche les numéros des polices d'assurance dont un accident contient un rapport avec le mot clé « université de jijel » :

(Q9) **SELECT** P.NPOLICE
FROM POLICE P, **TABLE** P.ACCIDENTS A**, TABLE** A.RAPPORT..KEYWORDS M
WHERE M = "UNIVERSITE DE JIJEL".

Cette requête suppose la disponibilité de la fonction KEYWORDS sur le type TEXT du rapport qui délivre une liste de mots clés. Nous tablons tout d'abord la liste des accidents, puis la liste des mots clés du rapport de chaque accident. Ceci donne donc une sorte d'expression de chemins multivalués.

8.3 Recherche et héritage

L'héritage de tables est pris en compte au niveau des requêtes. Ainsi, lorsqu'une table possède des sous-tables, la recherche dans la table retrouve toutes les lignes qui qualifient au critère de recherche, aussi bien dans la table que dans les sous-tables Par exemple, considérons la table BUVEURS définie comme suit :

CREATE TABLE SPORTIFS **UNDER** PERSONNES **WITH** ETAT ENUM(NORMAL,IVRE)**.**

La recherche des personnes de prénom Ahmed retournera par la requête (Q10) à la fois les personnes et les sportifs de prénom Ahmed.

(Q10) SELECT NOM, PRÉNOM, ADRESSE
FROM PERSONNES
WHERE PRÉNOM = "AHMED"

9. Conclusion

SQL3 est un standard en évolution qui se situe comme un concurrent de l'ODMG. Il est supporté par tous les grands constructeurs, comme IBM, Oracle et Sybase, et est impliqué dans un processus de standardisation international. Au contraire, l'ODMG est un accord entre quelques constructeurs de SGBD objet autour d'un langage objet pur. Les deux propositions pourraient donc apparaître comme complémentaires.

EXERCICES

EXERCICES

EXERCICE 1

On veut traduire en orienté objets une base de données relationnelle qui décrit les prêts bancaires faits à des clients (qui peuvent être des personnes, des entreprises ou des banques) pour des achats de voitures. Le schéma relationnel de cette base de données est décrit ci-dessous:

Personne (n°AVS , nom , age , employeur1 , employeur2 , adresse , téléphone)
Voiture (n°Propriétaire , n°Voiture , type.propriétaire , modèle , année)
Prêt (n°Voiture , n°Client , type.client , banque , n°prêt , taux , montant)
Entreprise (nom , n°Entreprise)
Banque (nom , n°Banque)

Traduisez en ODL la définition des données de cette base "Prêts_bancaires". Lors de la traduction, on supprimera tous les identifiants qui étaient nécessaires en relationnel, mais qui sont inutiles en orienté objets.

EXERCICE 2

L'administration d'Enregistrement et d'immatriculation des Véhicules désire connaître les informations relatives aux propriétaires et aux transactions (achat/vente) effectuées sur les véhicules.

À chaque véhicule, elle assigne un numéro d'enregistrement. Il n'existe pas deux véhicules ayant le même numéro d'enregistrement. À tout moment, un véhicule n'appartient qu'à un seul propriétaire, qui est soit un constructeur, soit un garage, ou encore une personne

privée. Il peut avoir été possédé par plusieurs propriétaires (à des moments distincts). Un constructeur, un garage ou une personne privée est connu de l'administration d'Enregistrement des Véhicules, c'est à dire considéré comme faisant partie de l'ensemble des propriétaires s'il possède ou a possédé un véhicule.

Qu'il soit constructeur, garage ou personne privée, un propriétaire est caractérisé par un numéro l'identifiant. Pour un constructeur, on connaît son nom, son adresse ainsi que les garages avec lesquels il travaille (garages concessionnaires). Un garage est caractérisé par un nom, une adresse et un numéro de registre de commerce. On connaît le nom, le prénom et l'adresse d'une personne privée.

Pour toute transaction effectuée sur un véhicule, on connaît le vendeur (ancien propriétaire), l'acheteur (nouveau propriétaire), la date de transaction et le prix d'achat/vente. Un véhicule peut faire l'objet de plusieurs transactions (à des dates différentes). Il n'est pas exclu que deux transactions réalisées à des dates différentes puissent porter sur un même véhicule, un même vendeur et un même acheteur.

Un constructeur ne peut vendre ses véhicules à d'autres constructeurs, ni directement à des personnes privées. Il ne les vend qu'à ses garages concessionnaires. Il n'achète aucun véhicule. Un garage peut vendre ou acheter des véhicules à des personnes privées ou à des garages. Il peut, bien sûr, acheter également des véhicules aux constructeurs pour lesquels il est concessionnaire. Une personne privée ne peut vendre ou acheter des véhicules qu'à des personnes privées ou à des garages. Ceci signifie donc que seuls, les véhicules dont le propriétaire "du moment" est un constructeur, n'ont été l'objet d'aucune transaction.

→ Donnez le schéma ODL de la base de données décrite plus haut.

EXERCICE 3

On considère une base de données géographiques dont le schéma est représenté de manière **informelle** ci-après :

Régions (id char(5), pays string, nom string, population int, carte géométrie)
Villes (nom string, population int, région ref (Régions), position géométrie)
Forêts (id char(5), nom string, carte géométrie)
Routes (id char(4), nom string, origine ref (Villes), extrémité ref (Villes), carte géométrie)

Cette base représente des objets Régions ayant un identifiant utilisateur (id), un pays, un nom, une population et un une carte géométrique.

Les objets Villes décrivent les grandes villes de ces régions et sont positionnées sur la carte par une géométrie.

Les Forêts ont un identifiant utilisateur, un nom et une géométrie.

Les Routes sont représentées comme des relations entre villes origine et extrémité ; elles ont une géométrie.

Une géométrie peut être un ensemble de points représentés dans le plan, un ensemble de lignes représentées comme une liste de points ou un ensemble de surfaces. Chaque surface est représentée comme un ensemble de lignes. Sur le type géométrie, les méthodes Longueur, Surface, Union, Intersection, et Draw (pour dessiner) sont définies.

Q1 : Proposez une définition du schéma de la base en ODL.

Q2 : Exprimez les questions suivantes en OQL :

1. Sélectionner les noms, pays, populations et surfaces des régions de plus de 10000 km² et de moins de 50000 habitants.
2. Dessiner les cartes des régions traversées par la route nationale RN7.
3. Donner le nom des régions, dessiner régions et forêts pour toutes les régions traversées par une route d'origine Jijel et d'extrémité Tamanraset.

EXERCICE 4

On désire conserver dans une base de données géographique la description d'un réseau routier telle que décrite ci-après. Certaines de ces informations ne peuvent être représentées dans le modèle relationnel. L'une des solutions possibles est d'utiliser le standard ODMG pour définir cette base en orienté objet.

- Pour définir le réseau routier, on part de points d'intersection qui sont les points où plusieurs routes se croisent. Un point d'intersection est caractérisé par ses coordonnées géographiques (latitude *x* et longitude *y*).
- On considère alors des segments de route qui sont des parties situées entre deux points d'intersection. Un point peut être d'origine ou de destination.
- Une route est désignée par un identifiant (N4, E411, A602, . . .) et est décrite par un ensemble de segments.
- De plus pour chaque route on conserve la désignation de l'autorité (région, commune, . . .) qui la gère.
- Un schéma relationnel préliminaire possible serait :

POINTS (ID-P, LATITUDE, LONGITUDE)
SEGMENTS (ID-SEG, ID_P1, ID_P2)
ROUTES (ID-R, ID_LISTE_SEG, AUTORITE)
LISTES_SEGMENTS (ID-LISTE-SEG, ID_SEG)

Questions :

1. Sachant que les attributs soulignés sont les clés, proposez le schéma ODL pour cette base de données géographique.
2. Exprimez en OQL les deux requêtes suivantes :
   2.1 : Quel est le nombre de segments constituant la route RN44 ?
   2.2 : Quelle est la liste des autorités qui gèrent les routes reliant les deux points (40, 80) et (120, 275) ?

Rappel :

Le schéma relationnel est souvent donné à titre d'illustration. Il n'est pas généralement recommandé de le reprendre pour produire un schéma ODL.

EXERCICE 5

Soit le schéma ODL de la base « Agences_Location_Maisons » en annexe 1 :

→ Donner le résultat exact ainsi que le littéral générés par l'exécution de chacune des trois requêtes OQL ci-après :

Requête 1 :

```
SELECT f.numAgence
FROM f IN agences
WHERE f.adresse.ville = "Jijel";
```

Requête 2 :

```
SELECT struct (nom: e.nomComplet.nom, genre: e.genre, âge: e.lireAge)
FROM e IN personnelVente
WHERE s.TravailleÀ.adresse.ville = "Béjaia";
```

Requête 3 :

```
SELECT struct(numéroAgence, nombreDEmployés: COUNT(partition))
FROM e IN personnelVente
GROUP BY numéroAgence: e.TravailleÀ.numAgence;
```

EXERCICE 6

Examinez le code ODL de la base « Université » définie en annexe 2 :

1) Donner une description détaillée de la base « Université ».

2) Exprimer en OQL et de manière optimisée les requêtes suivantes :

Req1 : Tous les objets Département de l'université.

Req 2 : Les paires d'enseignants et d'étudiants dont ils sont tuteurs.

Req 3 : Les noms de cours proposés par le département d'informatique et leurs unités.

Req 4 : Les noms d'unités ainsi que le nombre de leurs enseignants et des étudiants qui les suivent.

Req 5 : Les noms d'enseignants avec leurs départements tels que ces enseignants ont le nombre maximal d'étudiants en tutorat.

- Maintenant, il va falloir repenser entièrement la conception de la base « Université » en relationnel-objet.

3) Donner le diagramme des tables (utiliser la notation UML).

4) Donner une définition de la même base de données en utilisant SQL3. Il faut prendre soin de supprimer ou de rajouter des classes et/ou des types utilisateur afin qu'aucune information ne soit perdue.

EXERCICE 7

On désire développer une base de données orientée objets nommée « Compagnie_Aérienne » dans laquelle on conservera la description de l'activité d'une compagnie aérienne telle que **partiellement** représentée par le schéma relationnel suivant :

PILOTE (NUMPIL, NOMPIL, ADR, NBHV, SAL)

AVION (NUMAV, NOMAV, CAPACITE, LOC)

VOL (NUMVOL, NUMPIL, NUMAV, VILLE_DEP, VILLE_ARR, H_DEP, H_ARR)

Les attributs sont définis comme suit :

NUMPIL: clé de PILOTE, nombre entier
NOMPIL: nom du pilote, chaîne de caractères
ADR: ville de la résidence du pilote, chaîne de caractères
NBHV : nombre d'heures de vol, nombre entier
SAL: salaire du pilote, variable par mois, nombre entier
NUMAV: clé de AVION, nombre entier
CAPACITE: nombre de places d'un avion, nombre entier
LOC: ville de l'aéroport d'attache de l'avion, chaîne de caractères
NUMVOL: clé de VOL, nombre entier
VILLE_DEP: ville de départ du vol, chaîne de caractères
VILLE_ARR: ville d'arrivée du vol, chaîne de caractères
H_DEP: heure de départ du vol
H_ARR: heure d'arrivée du vol

Nous nous intéresserons à la partie de la base de données qui contient des informations sur l'équipage, c-à-d., les pilotes, les copilotes, les hôtesses, les stewards et les mécaniciens.

- Les copilotes ont les mêmes informations à conserver dans la base, seul le statut (pilote / copilote) change.
- Dans un équipage, connu par son numéro, on trouve un seul pilote, deux copilotes et un nombre variable d'hôtesses, de stewards et de mécaniciens.
- Les hôtesses et stewards sont identifiés via leurs numéros, ont un rang (entre 1 et 5) et possèdent plusieurs numéros de téléphone.
- Le même équipage assure le vol d'aller et celui de retour.
- Il est désormais possible pour les pilotes (et uniquement eux) de cette compagnie aérienne d'embarquer jusqu'à deux de leurs enfants, dépassant 04 ans, avec l'équipage. On conservera alors les prénoms des enfants des pilotes et leurs âges.

- La compagnie offre des formations aux pilotes et copilotes. Elles dépendent du type d'avion et s'étalent sur des durées différentes.
- Un pilote/copilote suit la même formation une seule fois dans sa carrière dans la compagnie.

Q1) Donnez le schéma ODL de la base de données objets « Compagnie_Aérienne ».

Q2) Exprimer en OQL les requêtes suivantes :

1- Quels sont les pilotes (numéros et noms) qui ont au moins conduit un vol au départ de la ville de l'un des membres de l'équipage, dont l'avion peut accueillir plus de 300 passagers ?
2- Donner les numéros de téléphones de tous les stewards ayant déjà assuré un vol avec le copilote qui a perçu le plus bas salaire dans la compagnie.
3- Trouver le pilote qui a profité au maximum de l'option offerte par la compagnie ; à savoir celle d'embarquer ses enfants avec l'équipage.

EXERCICE 8

L'objectif de cet exercice est de modéliser une base de données relationnelle objet de gestion d'emprunt de ressources par des abonnés, en utilisant SQL3.

Description

Les ressources empruntées peuvent être des disques ou des livres identifiées par leur numéro ISBN et possédant un titre et une date de parution. Chaque ressource est éditée par un éditeur dont on sauvegarde le nom et l'adresse, et est écrite par un ou plusieurs auteurs. Les livres ont un numéro d'édition et un nombre de pages. Les disques peuvent être de type CD, DVD, VOD, etc. On sauvegarde pour chaque abonné la date de sa première inscription à la bibliothèque, son nom, ses prénoms, son adresse et ses numéros de téléphone.

Les abonnements sont payés par semestres (du 01 Janvier au 30 Juin, et du 01 Juillet au 31 Décembre). Les droits à payer changent par tranches d'âges. Les personnes entre 5 et 12 ans payent 50 DA par semestre. Cette somme est multipliée par deux successivement à chaque tranche entre 12 et 18 ans, entre 18 et 30 ans, et plus de 30 ans. Les moins de 5 ans sont exonérés de tous frais.

Lorsqu'un abonné emprunte une ressource, on mémorise la date d'emprunt grâce à laquelle les emprunts d'une même ressource par un même abonné sont distingués. La date de restitution est calculée en fonction de la fidélité de l'abonné, le nombre de jours est égal à quinze fois le nombre de semestres d'abonnement déjà payés. Toutefois, les pénalités de retard sont proportionnelles au nombre de jours de retard. Un jour de retard prive l'abonné d'emprunter pendant deux jours et ainsi de suite.

- Donner un schéma en utilisant la notation UML
- Ecrire les requêtes SQL3 de création des tables de la BD O-R correspondant aux objets Abonné, Livre, Disque et Emprunt.
- Utiliser le type utilisateur le plus adapté pour la représentation de chaque élément.
- Pour chacune des tables correspondant aux objets Abonné, Livre, Disque et Emprunt de votre BD, donner un exemple de requête SQL3 d'insertion d'une instance.

On suppose que les tuples correspondants aux auteurs et éditeurs du livre et du disque que vous insérez sont déjà enregistrés dans la base.

EXERCICE 9

Examinez le code SQL3 suivant:

```
CREATE TYPE t_prenoms AS  VARRAY(3) OF VARCAHR(20);
CREATE TYPE t_telephones AS  TABLE OF CHAR(10);
CREATE TYPE t_personne AS OBJECT   (nom VARCHAR(20),
                                    prenoms t_prenoms,
                                    adresse VARCHAR (40))  NOT INSTANTIABLE
                                                           NOT FINAL;
CREATE TYPE t_abonné UNDER t_personne (numero INT,
                                       date_inscrption DATE,
                                       telephones t_telephones) ;
CREATE TYPE t_ressource AS OBJECT (isbn INT,
                                   titre VARCHAR (50),
                                   parution DATE) NOT FINAL ;
CREATE TYPE t_livre UNDER t_ressource (num_edition INT,
                                       nombre_pages INT) ;
CREATE TYPE t_disque UNDER t_ressource (type_disque VARCHAR(7),
                                        capacite INT) ;
CREATE TYPE t_emprunt AS OBJECT (abonne REF t_abonne,
                                 ressource REF t_ressource,
                                 date_emprunt DATE,
                                 date_retour DATE) ;
CREATE TABLE Abonne OF t_abonne (PRIMARY KEY numero) ;
CREATE TABLE Livre OF t_livre (PRIMARY KEY isbn) ;
CREATE TABLE Disque OF t_disque (PRIMARY KEY isbn) ;
CREATE TABLE Emprunt OF t_emprunt (PRIMARY KEY
                                   (abonne.numero, ressource.isbn, date_emprunt)) ;
```

Questions

1. Donner une description littérale détaillée, tout en restant précis et concis, de la base définie ci-dessus. Appuyer votre description avec un graphique de type MCD.
2. Décrire ce qu'on cherche par les requêtes suivantes :

Requête 1 :

```
UPDATE Emprunt emp
SET emp.ref_Abonne = (SELECT REF(abo) FROM Abonne abo WHERE abo.numero = 50)
WHERE emp.date_retour = '01/02/2011';
```

Requête 2 :

```
SELECT DISTINCT(emp.ref_Abonne.numero), emp.ref_Abonne.nom,
emp.ref_Abonne.prenoms, emp.ref_Abonne.adresse
FROM Emprunt emp;
```

Requête 3 :

```
SELECT emp.ref_Abonne.numero, emp.ref_Abonne.nom, COUNT(*) "Nombre"
FROM Emprunt emp
GROUP BY (emp.ref_Abonne.numero, emp.ref_Abonne.nomero)
HAVING COUNT(*) =
(SELECT MAX(COUNT(*)) FROM Emprunt emp2
GROUP BY (emp2.ref_Abonne.numero, emp2.ref_Abonne.nomero));
```

EXERCICE 10

On considère une base de données composée des entités élémentaires suivantes :

CLIENT (NumClient, NomClient, Prénoms, telephones, rue, ville, code)

FOURNISSEUR (NumFourn, NomFourn, Prénoms, telephones, rue, ville, code)

COMMANDE (NumCom, NumClient, DateCom, NumResp)

PRODUIT (NumProd, NomProd, TypeProd, PrixUnitaire, NumFourn)

Tel que les champs soulignés représentent les clés,

NumResp est le numéro du responsable du suivi de la commande.

Questions :

(1) Définir le schéma ci-dessus en SQL3. Dans cette définition, on doit avoir les définitions de types et de tables. Il est demandé à ce que l'adresse soit définie comme objet, que les prénoms soient rassemblés dans un tableau, et que les numéros de téléphone soient inclus dans une table imbriquée.

(2) Exprimer les requêtes suivantes en SQL3 :

a) Noms et ensemble des prénoms des fournisseurs et des clients habitant chacun une ville différente, tel que le fournisseur ait vendu un produit au client plus de deux fois.

b) Noms et types de produits avec les noms des fournisseurs offrant ces produits avec la condition que le responsable numéro 5 ait déjà suivi au moins une commande contenant l'un de ces produits.

EXERCICE 11

Soit le schéma ODL de la base « Etudes_diplomantes » fourni en annexe 3:

Partie I :

Q1) Quelles sont les classes manquantes dans ce schéma ?

Q2) proposez une définition adéquate des ces classes afin de compléter le schéma ODL.

Q3) Donner le résultat exact ainsi que le littéral générés par l'exécution de chacune des requêtes OQL ci-après :

Requête A :

```
SELECT STRUCT (alias1 : s.pnom.nom, alias2 : s.pnom.prenom, alias3 : s.moy)
FROM s IN etudiants
WHERE s.nomDept.dnom = "Informatique" AND
      s.annee = 2
ORDER BY alias3 DESC, alias1 ASC, alias2 ASC;
```

Requête B :

```
SELECT STRUCT (name: STRUCT (alias1 : s.pnom.nom, alias2 : s.pnom.prenom),
              diplomas: (SELECT STRCUT (alias3 : d.intitule_diplome, alias4 : d.an,
                                                                  alias5 :  d.fac)
                                        FROM d IN s.diplomes)
FROM s IN departments.dchef.conseille;
```

Requête C :

```
EXISTS g IN
        (SELECT s
         FROM s IN etudiants_diplomés
         WHERE s.nomDept.dnom = "Informatique" AND
               s.annee = 2)
       : g.moy < 5;
```

Partie II :

En considérant toujours le même schéma ODL de l'annexe 3, donnez une définition de la même base de données « Etudes_diplomantes » en utilisant SQL3. Il est requis de donner les définitions des types utilisateurs (si nécessaires), les définitions des classes ainsi que les contraintes d'intégrité. Prenez soin de supprimer ou de rajouter des classes et/ou des types utilisateur afin qu'aucune information ne soit perdue.

ANNEXES

```
class Agence (extent agences key numAgence)
{
        attribute string numAgence;
        attribute struct AdresseAgence {string rue, string ville, string codePostal} adresse;
        relationship Gérant GéréePar inverse Gérant :: Gère;
        relationship set<PersonnelVente> Emploie inverse PersonnelVente :: TravailleÀ;
        relationship set<PropriétéALouer> Propose inverse PropriétéALouer ::
                                                                EstProposéePar;
        void inscrireUnePropriétéALouer(in string numPropriété)
                                                        raises(propriétéDéjàALouer);
};
class Personne {
        attribute struct NomCompletP {string prénom, string nom} nomComplet;
};
class Personnel : Personne (extent personnel key numPersonnel)
{
        attribute string numPersonnel;
        attribute enum TypeGenre {M, F} genre;
        attribute enum TypeFonction {Gérant, Superviseur, Assistant} fonction;
        attribute date dateNaissance;
        attribute float salaire;
        short lireAge();
        void augmenterSalaire(in float incrément);
};
class Gérant : Personnel (extent gérants)
        relationship Agence Gère inverse Agence :: GéréePar;
};
class PersonnelVente extends Personnel (extent personnelVente)
{
        relationship Agence TravailleÀ inverse Agence :: Emploie;
        void transférerPersonnel(in string deNumAgence, in string versNumAgence)
                                                raises(neTravaillePasDansCetteAgence);
};
```

Annexe 1 : Schéma ODL de la base « Agences_Location_Maisons »

```
class Personne {
   attribute string nom;
   attribute string addresse;
   attribute date dateNais;
};
class Enseignant : Personne (extent  lesEnseignants) {
   attribute short Numbureau;
   relationship set<Etudiant> tuteur_de inverse Etudiant :: tutoré_par;
   relationship Departement travaille_pour inverse Departement :: staff;
   relationship set<Unité> enseigne inverse Unité :: enseigné_par;
};
class Etudiant  : Personne  (extent lesEtudiants) {
   attribute string Numinscription;
   relationship Enseignant tutoré_par inverse Enseignant :: tuteur_de;
   relationship Cours inscrit_en inverse Cours :: a_etudiants;
   relationship  set<Unité> obtient inverse Unité :: obtenue_par;
};
class Departement (extent lesDepartements) {
   attribute string nom;
   relationship set< Enseignant > staff inverse Enseignant :: travaille_pour;
   relationship set<Cours> offre inverse Cours :: offert_par;
};
class Cours (extent lesCours) {
   attribute string nom;
   relationship Departement offert_par inverse Departement :: offre;
   relationship set< Etudiant  > a_etudiantsinverse Etudiant  :: inscrit_en;
   relationship set<Unit> a_unités inverse Unit :: partie_de;
};
class Unité (extent lesUnités) {
   attribute string nom;
   attribute string code;
   relationship set< Etudiant  > obtenue_par inverse Etudiant :: obtient;
   relationship set< Enseignant > enseigné_par inverse Enseignant :: enseigne;
   relationship set<Cours> partie_de inverse Cours :: a_unités;
};
```

Annexe 2 : Schéma ODL de la base « Université »

```
Class Personne (extent personnes key numassurance) {
    attribute struct pnom (string prenom, string nom);
    attribute string numassurance;
    attribute date datenaissance;
    attribute enum sexe {M, F} ;
    attribute struct adresse (short no, string rue, short code, string ville);
    short age ( );
};
Class Professeur : Personne (extent professeurs) {
    attribute string rang;
    attribute float salaire;
    attribute string tel;
    relationship Departement travaille_dans inverse Departement : : a_professeurs;
    relationship set<EtudiantDip> conseille inverse EtudiantDip : : conseillé_par;
    void augmenter (in float augmentation) ;
    void promouvoir (in string nouveau_rang) ;
};
Class Note (extent notes) {
    attribute enum valeursNotes {A, B, C, D, F, I, P} ;
    relationship Etudiant obtenue_par inverse Etudiant : : a_obtenu ;
};
Class Etudiant : Personne (extent etudiants) {
    attribute int annee ;
    attribute Departement nomDept ;
    relationship set<Note> a_obtenu inverse Note : : obtenue_par ;
    relationship set<SessionCourante> inscrits_a inverse SessionCourante : :
                                                                etudiants_inscrits ;
    float moy ( ) ; %calculer la moyenne annuelle de l'étudiant
    void changer_dept (in string dnom) ;
    void note_attribuee (in V valeurNotes);
};
Class Diplome {
    attribute string fac ;
    attribute string intitule_diplome ;
    attribute string an ;
} ;
Class EtudiantDip : Etudiant (extent etudiants_diplomés) {
    attribute set<Diplome> diplomes ;
    relationship Professeur conseillé_par inverse Professeur : : conseille ;
} ;
Class Departement (extent departements key dnom) {
  attribute string dnom ;
  attribute string dtelephone ;
  attribute string fac ;
  attribute Professeur dchef ;
  relationship set<Professeur> a_professeurs inverse Professeur : : travaille_dans ;
  relationship set<Cours> propose inverse Cours : : proposé_par ;
} ;
```

Annexe 3 : Schéma ODL de la base « Etudes_diplomantes »

REFERENCES BIBLIOGRAPHIQUES

Références Bibliographiques

[1] Thomas Connolly, Carolyn Begg, ‘’Systèmes de base de données - Approche pratique de la conception, de l'implémentation et de l'administration’’, éditeurs Eyrolles, Reynald Goulet, 2005.

[2] Christine Parent, ‘’Cours de bases de données avancées’’, disponibles sur : http://www.hec.unil.ch/gcampono/index.php/teaching/BDA

[3] Georges Gardarin, ‘’Bases de données’’, éd. Eyrolles, 2003.

[4] R. G. G. Gattell, "Bases de données orientées objets - 2ème édition", Addison Wesley Publishing Company, Inc., 1994.

[5] Stéphane Crozat, ‘’Le modèle logique relationnel-objet et son implémentation sous Oracle’’, disponible sur : bdd.crzt.fr/ro/pdf/ro.pdf

[6] Christian Soutou, ‘’Programmer objet avec Oracle ’’, Vuibert, 2004.