Evolution de Schéma et Adaptation des Instances

Jean-Hilaire Yapi[1,2], André Lasquellec[1], Bernard Gibaud[2], Patrick Bosc[3].

[1]Dept. Informatique. Ecole Nat. Sup. de Télécommunications de Bretagne. B.P. 832 - 29285 Brest Cedex.
[2]Laboratoire Signaux et Images en Médecine. Université de Rennes 1. [3]Ecole Nat. Sup. des Sciences Appliquées et Technologies de Lannion. e-mail : JH.Yapi@enst-bretagne.fr

Résumé

Cet article présente un mécanisme d'évolution de schéma qui offre les opérations de fusion et d'éclatement de classe et une forme particulière de suppression de classe. Ces opérations se caractérisent par les requêtes qu'elles intègrent pour faire migrer les instances des classes modifiées vers d'autres classes. Notre attention se porte sur une fonction proposée pour restructurer les instances des classes modifiées et réaliser les migrations d'instances nécessaires.

Abstract

We present a schema evolution mechanism offering some special operations, namely : a particular kind of class deletion, allowing the migration of existing instances to a superclass or a subclass of the deleted one, the fusion of two classes and the split of a class. These operations support the migration of instances from one class to another. We focus on a function provided to adapt the instances of the classes which have been modified using these operations.

1. Introduction

Les applications informatiques sont confrontées tout au long de leur cycle de vie à des problèmes d'évolution, soit pendant la phase de conception et de développement, soit plus tard pendant la phase d'exploitation. Celles qui reposent sur des systèmes de gestion de bases de données orientées objet (SGBDOO) rencontrent ces mêmes problèmes. Dans leur cas, les évolutions à prendre en compte pendant la phase de conception et de développement concernent uniquement l'évolution du schéma de la base de données. Il s'agit de la modification de la définition des classes : propriétés et hiérarchie d'héritage. Par contre, pendant la phase d'exploitation, il est nécessaire de supporter en plus l'évolution des données. Il s'agit, pour ce dernier point, de faire évoluer les instances pour les rendre conformes au nouveau schéma.

L'évolution du schéma pendant la phase de conception et de développement est généralement bien assurée par les SBGDOO et, vu sous cet angle, les SGBDOO actuels offrent des fonctionnalités sensiblement équivalentes. Il n'en est pas de même pour l'évolution de schéma pendant la phase d'exploitation car les fonctionnalités offertes pour assurer l'évolution des données sont différentes d'un système à l'autre. Comme les besoins des applications en termes d'évolution sont très variables, il n'est pas toujours possible de les prendre en compte de manière satisfaisante à l'aide des fonctionnalités disponibles dans un système donné, souvent parce qu'une fonctionnalité réellement nécessaire à l'application n'est pas disponible, ou encore parce que la sémantique qui lui est attachée ne correspond pas à celle qui serait souhaitable.

C'est précisément le cas pour une application du domaine médical, appelée projet Atlas qui fait coopérer une base de connaissances et une base de données. La base de connaissances comporte une description symbolique et prototypique des objets qui composent le cerveau, et la base de données stocke des données relatives à des exemplaires de cerveaux. Les connaissances relatives au cerveau sont incomplètes et évolutives et les modifications apportées à la base de connaissances doivent être répercutées sur la base de données. Il est indispensable, pour supporter l'évolution de l'Atlas, de disposer d'un mécanisme offrant des opérations particulières telles que, par exemple, la fusion et l'éclatement de classe qui permettent non seulement de modifier une classe, mais aussi de faire migrer ses instances vers une de ses sous-classes ou une

de ses super-classes. Par exemple, fusionner deux classes consiste à regrouper ces deux classes pour en former une seule; elle nécessite donc de faire migrer les instances de ces classes vers la classe résultante.

Cet article présente un mécanisme d'évolution de schéma qui intègre les opérations particulières nécessaires pour supporter les évolutions de l'Atlas et qui réalise par une fonction d'adaptation des instances les restructurations et les migrations d'instances nécessitées par les modifications du schéma. Cette fonction soulève en particulier le problème de la mise à jour des références vers les instances converties.

La suite de ce papier est organisée comme suit. La deuxième partie situe le contexte applicatif de notre étude et présente les besoins particuliers qui l'ont motivée. La troisième partie est une analyse des mécanismes existants afin de montrer leurs limites face aux besoins exprimés. La quatrième partie présente notre mécanisme d'évolution de schéma pour répondre aux besoins de l'Atlas. Le problème de la mise à jour des références est étudié dans la perspective d'une implantation du mécanisme au dessus d'un SGBDOO. La cinquième partie pose le problème de la mise à jour des références dans l'hypothèse d'une implantation à l'intérieur du système et compare les solutions envisageables avec la solution retenue dans la quatrième partie. La sixième récapitule les avantages et inconvénients du mécanisme proposé par rapport aux systèmes existants et dégage les perspectives d'évolution de notre travail.

2. Spécification du besoin : le projet Atlas

Le projet Atlas a pour objectif la conception d'un *environnement permettant de gérer des connaissances sur l'anatomie et le fonctionnement du cerveau humain.* Le projet est motivé principalement par le besoin de se référer à des connaissances sur le cerveau, besoin ressenti à divers niveaux par les médecins et les chercheurs.

2.1. Architecture générale

Ce système comporte deux composantes principales : *une base de connaissances* (BC) exprimant une description symbolique et prototypique des objets qui composent le cerveau, et une *base de données* appelée Base d'Informations Sujets (BIS) permettant d'engranger des données relatives à des exemplaires de cerveaux particuliers.

• La BC vise à inventorier et à caractériser les structures anatomiques qui composent le cerveau normal (par exemple : noyaux gris centraux, cortex cérébral, voies de propagation, système ventriculaire, vascularisation) ainsi qu'à représenter les liens qui les associent entre elles (en particulier les liens de généralisation/spécialisation de type "est un" ou de composition de type "fait partie de"). Elle contient donc les descriptions des structures du cerveau admises par l'ensemble du corps médical, et qui correspondent à un modèle de cerveau prototypique. Ce modèle sert par ailleurs de formalisme de description des structures identifiées sur les images provenant d'examens pratiqués sur des exemplaires de cerveaux particuliers appelés sujets.

• La BIS a pour but de permettre l'accumulation progressive de données morphologiques et fonctionnelles sur le cerveau, recueillies chez des sujets, en particulier sous la forme d'examens d'imagerie (Imagerie par Résonance Magnétique, angiographie, scanner) ou d'explorations neurophysiologiques (Magnéto-encéphalographie, Electro-Encéphalographie, Stéréo-Electro-Encéphalographie). Cette base d'informations comprend d'une part les données brutes, telles qu'elles sont recueillies au niveau des capteurs et d'autre part des données étiquetées, résultat d'un processus d'étiquetage effectué par un expert du domaine médical. Il s'agit par exemple de délimiter précisément une structure anatomique particulière au sein d'une série d'images constituant un volume 3D et de lui affecter un nom (une étiquette) référencé dans la base de connaissances. Ces données étiquetées ont pour but d'établir le lien entre les objets réels observés chez des sujets et les objets du modèle de connaissances et d'expliciter les relations entre les différentes structures. Dans le système proposé, la BIS doit jouer un double rôle. D'une

part, elle fournit des exemples concrets qui viennent compléter les connaissances générales disponibles dans la BC. D'autre part, l'accumulation de descriptions réalisées chez des sujets permet de faciliter l'émergence de nouvelles connaissances qui, une fois établies, peuvent être modélisées explicitement dans la BC.

2.2. Le besoin d'évolution de schéma

Le schéma de la base de données s'appuie sur le modèle de connaissances défini par la BC. Or celui-ci est appelé à évoluer pour deux raisons essentielles. La première est le caractère nécessairement progressif de la constitution de la base de connaissances. En effet, la variété des structures du cerveau et la diversité des points de vues possibles imposent de les intégrer dans l'Atlas au fur et à mesure des besoins exprimés par les utilisateurs. La deuxième raison de l'évolution est due à la nature évolutive intrinsèque de toute connaissance. C'est en particulier le cas pour le cerveau dont le fonctionnement et les relations structures/fonctions sont encore très mal connus.

Toute modification du modèle de connaissances doit être impérativement propagée sur le schéma de la base de données puis sur les données existantes pour les rendre conformes au nouveau modèle. La modification du schéma dans le contexte du projet Atlas fait appel aux opérations classiques d'évolution de schéma avec la contrainte que toute modification de la structure d'une classe s'accompagne de la restructuration de ses instances existantes. Cette modification de schéma fait aussi appel à des opérations telles que, par exemple, l'éclatement de classe et une forme particulière de la suppression de classe comme le montrent les exemples qui suivent.

1) Eclatement de classe : Considérons par exemple la description de la vascularisation du cerveau. A un stade préliminaire de développement du système, il peut être suffisant de définir un "frame" Vaisseau et une classe correspondante pour décrire de façon globale tous les vaisseaux sanguins du cerveau, en distinguant les veines des artères à l'aide de valeurs particulières des attributs. Ultérieurement, il peut s'avérer utile de spécialiser le "frame" Vaisseau en deux "frames" Artère et Veine. Cette évolution peut se traduire au niveau du schéma de la base de données par la création de deux sous-classes Artère et Veine de la classe Vaisseau. Dans ce cas, il est important pour l'utilisateur d'identifier parmi les instances de la classe Vaisseau celles qui correspondent à des artères et celles qui correspondent à des veines, de manière à les faire migrer dans leurs classes respectives.

2) Suppression de classe : Dans le cadre du projet Atlas, la suppression d'une classe peut être motivée par le souci de réorganiser la hiérarchie de classes et, dans ce cas, il est important pour l'utilisateur de conserver les instances de cette classe afin de préserver les informations qu'elles contiennent. Cet aspect particulier de la suppression de classe est d'autant plus important qu'identifier une structure anatomique dans des images et la décrire peut demander plusieurs heures de travail à un expert médical. L'opération de suppression de classe doit donc permettre de faire migrer les instances dans une sous-classe ou une super-classe de la classe supprimée.

Ces exemples montrent la nécessité d'opérations offrant la possibilité de faire migrer les instances d'une classe vers une autre classe. Cette forme de migration d'instances doit être prise en compte par le système afin de garantir la conservation de la plus grande partie possible des informations déjà acquises. Ces besoins particuliers peuvent se rencontrer également dans de nombreuses applications, en particulier lorsque la structure des données par le SGBD est définie par une base de connaissances évolutive.

3. Approches proposées par les SGBD actuels

Cette section a pour objectif d'analyser les solutions proposées dans la littérature et par les systèmes actuels pour répercuter les modifications du schéma sur les données existantes, et de les confronter aux besoins spécifiques de l'Atlas. Mis à part le cas marginal du système O2 [12]

dont la version actuelle laisse à la charge de l'utilisateur le soin de rendre utilisables les instances existantes après une modification du schéma, la plupart des systèmes (par exemple Orion [1], Gemstone [9], Encore [10]) prennent en compte ce problème en s'appuyant sur l'une des trois stratégies suivantes : l'émulation, la gestion de versions et la conversion.

3.1. L'émulation

L'émulation consiste à donner à l'utilisateur une vision des instances existantes conforme au nouveau schéma, sans apporter de modification réelle dans la structure physique de ces instances. Lors des accès aux instances existantes, les attributs supprimés de la classe sont cachés à l'utilisateur et les attributs ajoutés sont présentés avec une valeur par défaut. Cette approche, utilisée par le système Orion [1], conduit donc à deux catégories d'instances dans une classe modifiée : les nouvelles instances conformes au nouveau schéma et les anciennes qui ne le sont pas. Cette stratégie pose un problème lié à l'impossibilité de manipuler, au sein des instances existantes, les attributs ajoutés dans la classe en leur affectant des valeurs spécifiques. Dans la plupart des applications (en particulier dans l'Atlas), la manipulation des attributs ajoutés dans la classe est essentielle tant pour les nouvelles instances que pour les instances créées avant la modification de schéma. Ce problème montre que l'émulation n'est pas appropriée pour l'Atlas.

3.2. La gestion de versions

La gestion de versions consiste à générer une nouvelle version des instances chaque fois que la structure de la classe est modifiée, tout en maintenant les instances existantes conformément à l'ancienne version de la classe. Cette stratégie, adoptée dans de nombreux systèmes (par exemple Orion [6], Encore [10], Avance [3], Closql [8]), a généralement pour objectif de permettre à l'utilisateur d'avoir plusieurs visions d'une même instance à travers des versions de classes différentes. Le besoin de l'Atlas étant d'avoir une vision unique mais cohérente des instances, nous écarterons aussi de notre analyse la gestion de versions.

3.3. La conversion

La conversion consiste à restructurer physiquement les instances en intégrant en leur sein les attributs ajoutés à leur classe et en retirant les attributs supprimés. Les deux techniques de conversion habituellement proposées sont la conversion immédiate et la conversion différée.

3.3.1. Conversion immédiate

La conversion immédiate consiste à répercuter les modifications du schéma sur les instances dans une transaction unique au cours de laquelle l'accès aux instances est interdit aux utilisateurs. Cette approche, adoptée par les systèmes Gemstone [9] et ObjectStore [13], permet de mettre à la disposition de l'utilisateur une base de données dans laquelle toutes les instances sont conformes aux nouvelles définitions des classes. Elle entraîne une indisponibilité du système pendant une durée qui dépend du nombre d'instances à convertir. Cette stratégie est bien adaptée aux applications pour lesquelles, comme l'Atlas, une indisponibilité provisoire du système est supportable par les utilisateurs.

3.3.2. Conversion différée

La conversion différée ([4], [5]) consiste à répercuter les modifications du schéma sur les instances au fur et à mesure de leur accès par les utilisateurs. Cette approche permet donc d'éviter l'indisponibilité du système occasionnée par la stratégie de conversion immédiate, mais elle cause une perte de performance de leurs programmes. En effet, chaque fois qu'une instance est accédée, le système vérifie si elle a été restructurée et effectue, au besoin, sa restructuration. De plus, la restructuration d'une instance peut nécessiter d'accéder à une autre instance, ce qui va déclencher la restructuration de cette instance et ainsi de suite. Cette restructuration en chaîne peut affecter encore plus gravement les performances des programmes de l'utilisateur.

Cette stratégie permet à l'utilisateur de procéder à de nouvelles modifications du schéma alors que les précédentes n'ont pas encore été totalement répercutées. Cette situation impose de gérer l'historique des différentes modifications intervenues sur le schéma afin de les propager correctement sur les instances. De plus, à cause du caractère totalement imprévisible des accès effectués par les utilisateurs, il est difficile de déterminer à quel moment une modification est totalement répercutée et peut être effacée de l'historique. Dans ces conditions, la gestion de cet historique devient délicate et complexe et peut avoir des conséquences fâcheuses sur les performances des programmes des utilisateurs.

3.4. Récapitulatif des approches proposées

La conversion différée pose des problèmes complexes dont la mise en œuvre n'est pas vraiment justifiée par les besoins de l'Atlas. La conversion immédiate est donc, parmi les stratégies que nous venons d'examiner, la plus intéressante pour les besoins de l'Atlas si on la compare à l'émulation et la gestion de versions, et la plus simple en terme de mise en œuvre. Cependant, pour supporter les opérations de fusion, d'éclatement et de suppression de classes, il est essentiel que la conversion des instances prenne en compte la migration d'instances. Les SGBD actuels ne fournissent pas ces opérations; par conséquent les solutions qu'ils proposent se limitent souvent à restructurer les instances des classes dont la structure a été modifiée par l'application des opérations.

4. Mécanisme proposé

Pour satisfaire les besoins exprimés, nous avons étudié un mécanisme d'évolution de schéma qui, en plus des opérations classiques (par exemple : ajouter et supprimer un attribut dans une classe, ajouter une classe, ajouter et retirer une super-classe à une classe), offre des opérations prenant en compte le besoin de migration d'instances.

L'utilisation de ce mécanisme comporte deux étapes. Premièrement, il s'agit de modifier le schéma à l'aide des opérations fournies. Deuxièmement, la répercussion de ces modifications sur les instances est réalisée à l'initiative de l'utilisateur qui active, à la fin de la modification du schéma, une fonction d'adaptation des instances. Cette fonction est fournie par le mécanisme et réalise une conversion immédiate des instances.

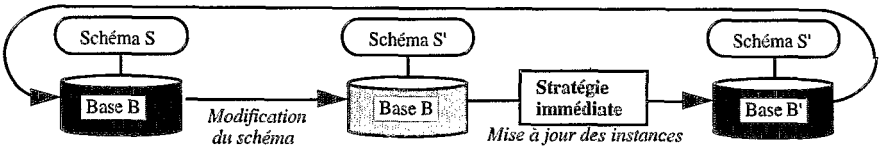


Figure 1: Déroulement d'une évolution.

Dans la suite de cette section, nous présentons la sémantique des opérations particulières proposées. Nous explicitons le problème de la mise à jour des références lié à la conversion (restructuration et migration) des instances et nous décrivons la solution que nous avons adoptée pour résoudre ce problème.

4.1. Sémantique des opérations

Les opérations particulières proposées par notre mécanisme sont la suppression, l'éclatement et la fusion de classes.

• Supprimer une classe : ***Delete_class(C, C_mig, req)***

Cette opération permet de supprimer une classe *C* et de faire migrer une partie de ses instances dans une classe *C_mig* qui est une sous-classe ou une super-classe de *C*. Ces instances sont

sélectionnées par une requête req spécifiée par l'utilisateur dans le langage de requête du système. Les autres sont supprimées en même temps que la classe *C*. Toutes les références à la classe *C* sont remplacées par des références à la classe *C_mig* spécifiée pour la migration des instances. Si aucune classe *C_mig* n'est spécifiée, toutes les références à *C* sont supprimées de même que toutes ses instances. Cette opération nécessite de convertir les instances des sous-classes de *C* pour leur retirer les attributs de locaux de *C* et aussi les instances de *C* qui migrent vers la classe *C_mig* pour leur supprimer les attributs de *C* et leur ajouter ceux de *C_mig*.

• Eclater une classe : ***Split_class(C1, C2, req)***

Cette opération permet de créer une nouvelle sous-classe *C2* d'une classe *C1*. Une requête de migration d'instances est attachée à cette opération pour sélectionner les instances de la classe *C1* à faire migrer dans sa nouvelle sous-classe *C2*. Les instances non sélectionnées demeurent dans la classe *C2*.

• Fusionner deux classes : ***Merge_classes(C1, C2, C3, req)***

Cette opération permet de fusionner les classes *C1* et *C2* au sein de leur super-classe *C3* ou pour former une nouvelle classe *C3*. Une requête de migration d'instances est attachée à cette opération pour sélectionner les instances des classes *C1* et *C2* qui doivent être conservées et donc transférées vers la classe C3. Les instances non sélectionnées sont supprimées en même temps que les classes *C1* et *C2*.

4.2. Problème de la mise à jour de références

La conversion opérée sur les instances par la fonction d'adaptation des instances comporte deux aspects : la restructuration des instances dont la structure de la classe d'appartenance a changé et la restructuration des instances concernées par une requête de migration. La migration d'une instance met en jeu la classe d'origine de l'instance et la classe cible vers laquelle elle doit être transférée. Dans les deux cas, il s'agit de (i) créer une nouvelle instance (de la classe cible pour la migration), (ii) d'initialiser cette nouvelle instance à partir des informations contenues dans l'ancienne instance et (iii) de supprimer l'ancienne instance.

Lorsque la nouvelle instance est créée, le système lui alloue un nouvel OID. Puisque les références à une instance sont faites à travers son OID, les références à l'ancienne instance doivent donc être reportées sur la nouvelle instance pour maintenir la cohérence de la base de données. De plus, cette mise à jour de références doit annuler les références qui existent sur des instances qui ont été supprimées en même temps que leur classe d'appartenance. Dans la suite de cet article, nous appellerons références valides les références qui sont faites à des instances existantes, et références invalides les références qui sont faites à des instances supprimées.

La mise à jour des références est un problème complexe qui peut altérer les performances de la fonction d'adaptation, en particulier si le nombre d'instances qui référencent chaque instance convertie est élevé. Dans cette partie, nous faisons l'hypothèse que la fonction d'adaptation des instances opère au niveau de l'interface du système et n'a pas accès aux fonctions internes du système permettant de manipuler les OID. Notre objectif est d'élaborer une solution pouvant servir à l'enrichissement d'un système existant. Nous verrons dans la cinquième partie comment ce problème se pose lorsque la fonction d'adaptation des instances opère à l'intérieur du système.

4.3. Solution au problème de la mise à jour de références

Afin de gérer plus facilement le problème de la mise à jour de références, nous associons à chaque instance *i* un objet permanent appelé représentant de *i* qui référence *i* par l'intermédiaire d'un attribut *refinst*. Inversement, l'instance *i* possède un attribut *repobj* qui lui permet de retrouver son représentant. Une référence à une instance est faite par l'intermédiaire de son représentant.

4.3.1. Mise à jour des références valides

Pour assurer la mise à jour des références valides, il suffit, lors de la conversion de l'instance *i*, de mettre à jour son représentant de manière que ce dernier référence la nouvelle instance *new_i* remplaçante de i comme le montre la figure suivante (ajout d'un attribut c dans une instance).

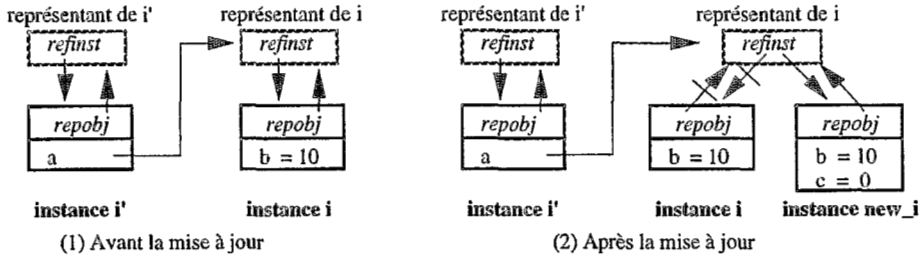


Figure 2: Mise à jour des références valides à une instance convertie.

Cette solution permet, par la seule mise à jour du représentant d'une instance, de remplacer toutes les références à cette instance par des références à la nouvelle instance qui la remplace. Cependant, cette solution nécessite de disposer de fonctions d'accès en lecture et en écriture qui tiennent compte de l'indirection introduite par la notion de représentant d'objet. Elles sont fournies par le mécanisme sous forme de méthodes attachées aux classes de l'utilisateur.

4.3.2. Annulation des références invalides

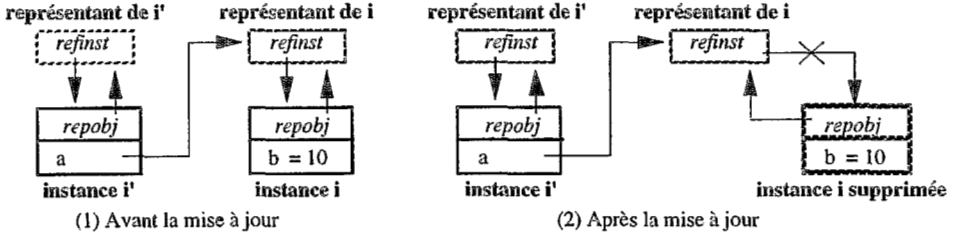


Figure 3: Annulation des références invalides à une instance supprimée.

L'annulation des références invalides consiste à faire en sorte que les instances qui sont supprimées suite à la suppression d'une classe ne soient plus référencées par des instances existantes. Cette annulation peut être effectuée par la fonction d'adaptation des instances lorsqu'une instance est supprimée. Il s'agit de donner une valeur nulle à l'attribut *refinst* du représentant de l'instance supprimée, comme le montre la figure 3.

Cette procédure permet de prémunir les utilisateurs contre un éventuel accès à une instance supprimée. Par exemple, la lecture de l'attribut *a* de *i'* va retourner une valeur nulle, ce qui est cohérent puisque l'instance *i'* initialement référencée a été supprimée. Cependant, le représentant de l'instance *i* est conservé et conduit à une indirection supplémentaire pour récupérer une valeur nulle. Cette situation ne pose aucun problème pour les futures modifications du schéma. Néanmoins, l'idéal serait que la lecture de l'attribut *a* de *i'* donne directement une valeur nulle. Deux solutions sont proposées pour résoudre ce problème : une mise à jour immédiate et une mise à jour différée.

• La mise à jour immédiate consiste à traiter les instances des classes qui référençaient précédemment une classe supprimée lors de la modification de schéma. Ce sont ces instances qui sont susceptibles de continuer à référencer des instances supprimées. La mise à jour consiste alors à affecter une valeur nulle à chaque attribut qui référence un représentant dont l'instance correspondante a été supprimée. Cette approche permet de libérer les représentants des instances supprimées et d'éviter l'indirection précédemment évoquée.

• La mise à jour différée consiste à modifier la fonction de lecture de l'attribut a de telle sorte qu'une valeur nulle soit affectée à cet attribut si le représentant référencé ne pointe sur aucune instance. Cette approche permet d'éliminer progressivement l'indirection supplémentaire et les représentants des instances supprimées. On économise ainsi le temps nécessaire pour la mise à jour immédiate.

4.4. Algorithme de la fonction d'adaptation des instances

Après une phase de modification du schéma, l'adaptation des instances consiste à considérer à tour de rôle les classes dont la structure est affectée par les modifications et faire les traitements suivants :

• On considère d'abord les requêtes de migration des instances de la classe vers d'autres classes. Il s'agit, pour chaque requête, de générer dynamiquement une fonction de conversion des instances de la classe vers la classe cible de la migration et d'utiliser cette fonction pour convertir effectivement les instances qui sont sélectionnées par la requête.

• Une fois les migrations effectuées, il ne reste dans la classe que les instances devant soit subir une restructuration, soit être supprimées. Si l'utilisateur a demandé la suppression de la classe, on la supprime ainsi que ses instances restantes. Si l'utilisateur n'a pas demandé la suppression de la classe, on génère, conformément aux modifications subies par la classe, une fonction de conversion afin de rendre les instances restantes conformes à la nouvelle définition de la classe.

4.5. Avantages et Inconvénients de la solution

La fonction d'adaptation des instances permet de rendre toutes les instances existantes conformes aux modifications intervenues sur le schéma. Elle prend en compte aussi bien la restructuration des instances d'une classe que leur migration dans une autre classe. La solution que nous avons adoptée pour résoudre le problème de la mise à jour des références sur les instances converties permet de donner un niveau de performance acceptable à la fonction d'adaptation des instances. En effet, l'accès et la mise à jour d'un seul représentant d'objet est suffisante pour mettre à jour toutes les références valides à une instance convertie et annuler toutes les références invalides à une instance supprimée. Il n'est donc pas nécessaire d'accéder à toutes les instances qui référencent cette instance pour les mettre à jour.

Le principal inconvénient de la solution adoptée concerne la performance des accès en lecture aux instances. En effet, celle-ci est affectée par la double indirection nécessaire pour accéder à une instance. Il faut également noter que la fonction d'adaptation des instances génère dynamiquement des fonctions et les exécute ensuite pour effectuer la conversion des instances. Elle ne peut donc être implantée qu'avec un système offrant une telle possibilité.

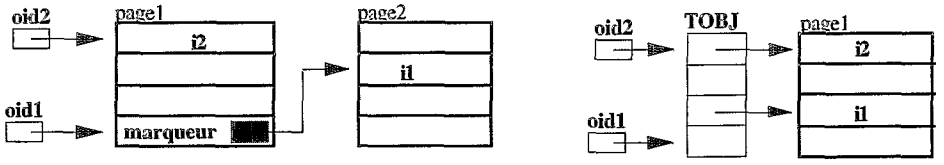
5. Implantation à l'intérieur du système

Le problème de la mise à jour de références que nous avons étudié dans la partie précédente vient du fait que la fonction d'adaptation des instances est réalisée au dessus de l'interface utilisateur du système et ne peut convertir une instance qu'en la remplaçant par une nouvelle instance identifiée par un nouvel OID. Selon Thévenin [11], les entités physiques de stockage des objets au niveau interne du système sont dotées de fonctionnalités d'interface qui permettent de les restructurer tout en préservant l'invariance des OID, grâce à un mécanisme

basé sur des OID physiques ou des OID logiques. Après avoir présenté succinctement les notions d'OID physiques et d'OID logiques développées par Thévenin [11], nous étudions le problème de la mise à jour de références et les solutions envisageables, que nous comparons ensuite avec celle adoptée dans le cas d'une implantation au dessus du système.

5.1. Mécanismes de gestion d'OID

• Avec le mécanisme d'OID physiques (voir la partie (1) de la figure 4), l'OID d'un objet correspond à son adresse en mémoire virtuelle qui est de la forme (n° de page, n° d'élément dans la page). Chaque fois que l'objet change de page (pour sa restructuration ou sa migration), sa nouvelle adresse est enregistrée à son premier emplacement utilisé comme marqueur.



(1) Gestion d'OID physiques (2) Gestion d'OID logiques

Figure 4: Mécanismes de gestion d'OID à l'intérieur du système.

• Avec le mécanisme d'OID logiques (voir la partie (2) de la figure 4), l'OID ne contient aucune information relative à la localisation physique de l'objet. L'invariance des OID est assurée par l'utilisation de tables d'indirections contenant les adresses des objets de la base de données. Un OID logique correspond à une entrée dans cette table. Si un objet change d'adresse, l'entrée correspondante dans la table d'indirections reçoit la nouvelle adresse.

	avant le changement de place	après le changement de place
OID physique	*(a+b)*	*2(a+b)*
OID logique	*(a+b+c)*	*(a+b+c)*

a : coût d'un accès à une page
b : coût d'un déplacement dans une page pour retrouver un objet
c : coût d'un déplacement dans la table d'indirection

Figure 5 : Coûts d'accès aux instances offerts par les mécanismes de gestion d'OID.

Lorsque l'objet n'a pas changé de place, le mécanisme d'OID physique est le plus performant. Lorsque l'objet change de place, c'est le mécanisme d'OID logique qui est le plus performant.

5.2. Problème de la mise à jour de références

Avec les mécanismes précédents, un objet dont l'adresse a changé reste accessible par les objets qui le référencent, au prix d'une indirection supplémentaire dans le cas des OID physiques et sans coût supplémentaire dans le cas des OID logiques. Le problème de la mise à jour des références valides ne se pose donc pas, mais celui de l'annulation des références sur un objet supprimé demeure.

Selon Bertino [2], certains mécanismes d'OID logiques intègrent dans l'OID d'une instance une référence à sa classe d'appartenance. La migration d'une instance dans une autre classe modifie donc son OID et rend incorrectes toutes les références valides à cette instance. Il faut les mettre à jour. Nous proposons dans la section qui suit une solution possible à ce problème.

5.3. Solution au problème de la mise à jour de références

Nous nous plaçons ici dans une situation où nous pouvons agir sur les éléments internes du système.

5.3.1. Annulation des références invalides

Dans le cas des OID physiques, il faut distinguer deux cas lors de la suppression d'un objet. Si l'objet a été déjà déplacé, il possède un marqueur; il suffit alors d'enregistrer une valeur nulle au niveau de ce marqueur et libérer la place occupée par l'objet dans la page. Si l'objet n'a pas été déplacé, il faut utiliser son emplacement comme marqueur et y stocker une valeur nulle. Dans le cas des OID logiques, il suffit également d'enregistrer une valeur nulle dans l'entrée de la table d'indirections correspondant à un objet. Lors d'une tentative d'accès à l'objet supprimé, la valeur nulle enregistrée permet de déterminer que l'objet a été supprimé et la référence servant à faire l'accès peut être annulée. La place occupée par cet objet est libérée par le système s'il n'est plus référencé.

5.3.2. Mise à jour des références valides lorsque l'OID logique référence la classe

Soit *i1* une instance (d'OID *oid1*) d'une classe qui migre vers une autre classe en devenant *new_i1* (d'OID *oid1'*). Pour que cette instance demeure accessible par les instances qui la référencent, une solution possible est d'enregistrer son nouvel OID dans l'entrée correspondante de la table d'indirections, comme le montre la figure 6.

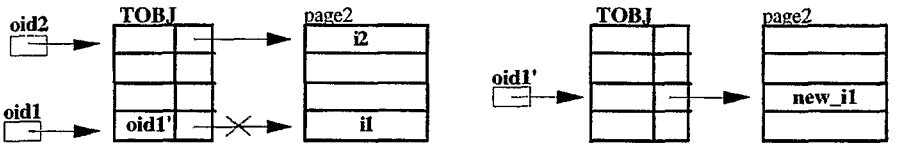


Figure 6: Migration de l'instance i1.

L'instance ainsi déplacée peut être atteinte à l'aide de son OID d'origine. Un déplacement dans la table d'indirections à l'aide de *oid1* permet de retrouver *oid1'*. Ce dernier est ensuite exploité pour retrouver l'instance *new_i* par un autre déplacement dans la table d'indirections suivi d'un déplacement dans la page.

Si une instance a migré n fois, il faut n déplacements dans la table d'indirections pour la retrouver, ce qui est consommateur de temps si les portions de la table d'indirections nécessaires ne sont pas déjà en mémoire. De plus, il faut maintenir toutes les entrées correspondantes de la table d'indirections pour offrir l'accès à cette instance. Cette situation conduit à une table d'indirections de plus en plus grande, en particulier, si les instances sont sujettes à de nombreuses migrations et/ou si le nombre d'instances ainsi déplacées est élevé. L'augmentation de taille de la table d'indirections diminue les performances des accès aux instances.

Il apparaît donc intéressant de mettre à jour les références valides sur une instance *i1* qui a migré afin, d'une part, d'éviter plusieurs accès systématiques à la table d'indirections avant de l'atteindre et, d'autre part de réduire la taille de la table d'indirections. Cette mise à jour nécessite l'accès effectif à chaque instance *obj* qui possède, par l'intermédiaire d'un attribut *a*, une référence à l'instance *i1*. Une mise à jour immédiate de toutes ces références peut être très coûteuse. La solution la plus intéressante est donc de différer cette mise à jour, lors de la lecture de l'attribut *a* de *obj*. La procédure précédemment décrite permet de retrouver l'instance *i1* et on en profite pour remplacer la valeur de l'attribut *a* de *obj* par le nouvel OID (*oid1'*) de *new_i1*. Les lectures suivantes de l'attribut *a* peuvent ainsi faire un accès direct à *new_i1* à l'aide de *oid1'*.

5.4. Comparaison avec l'implantation au dessus du système

Quelque soit le mécanisme de gestion d'OID utilisé, le problème de l'annulation des références invalides demeure et les solutions envisageables sont similaires à notre solution basée sur les représentants d'objets. Le problème de la mise à jour des références valides ne se pose que dans le cas d'un mécanisme d'OID logiques contenant des références aux classes et uniquement pour les migrations d'instances. Cette mise à jour nécessite d'accéder à chaque instance qui référence une instance convertie. Notre solution basée sur les représentants d'objets permet de réduire la mise à jour de toutes les références valides à une instance à la mise à jour de son seul représentant. Elle permet une stratégie immédiate alors que la solution à l'intérieur du système nécessite une approche différée pour éviter d'être trop coûteuse. La solution à l'intérieur du système nécessite de changer certaines structures internes utilisées pour la gestion des objets alors que notre solution les laisse intactes.

L'inconvénient de la solution au dessus du système concerne essentiellement les performances d'accès aux instances. En effet, l'accès à une instance nécessite l'accès supplémentaire à son représentant et occasionne le même coût avant et après sa restructuration ou sa migration. Si le système est basé sur le mécanisme d'OID physiques, le coût d'accès est de $(a+b)$ pour le représentant et de $(a+b)$ pour l'instance elle-même, soit un coût global de $2(a+b)$. Notre approche est donc au mieux au même niveau de performance que le système. Si le système est basé sur le mécanisme d'OID logiques, notre approche offre un coût d'accès de $2(a+b+c)$ double de celui du système.

6. Conclusion

Le mécanisme que nous proposons offre, en plus des opérations classiques d'évolution de schéma, les opérations de fusion, d'éclatement et de suppression de classe avec conservation des instances qui, bien que reconnues d'une grande utilité pour les applications ([2], [7]), ne sont pas fournies par les systèmes commerciaux actuels (Orion [1], GemStone [9] et O2 [12] par exemple). Ces opérations nécessitent de prendre en compte des requêtes de migration d'instances pour répercuter les modifications des classes sur leurs instances. Elles permettent à la fois de modifier l'organisation de la hiérarchie d'héritage et la répartition des données entre les classes. Les opérations traditionnelles sont limitées à la réorganisation du graphe d'héritage. Notre mécanisme fournit une fonction d'adaptation des instances qui réalise une conversion immédiate des instances suite à des modifications du schéma. Cette conversion prend en compte aussi bien la restructuration des instances d'une classe que leur migration dans une autre classe.

La principale difficulté de la mise en œuvre de cette fonction d'adaptation d'instances est due au problème de la mise à jour des références vers les instances converties. Notre solution effectue une mise à jour immédiate des références valides grâce à la notion de représentant d'objet qui minimise le nombre d'objets accédés pour la réaliser et réduit le temps d'indisponibilité du système. Concernant les références invalides, nous offrons à l'utilisateur de les annuler soit avec une stratégie immédiate, soit avec une stratégie différée. La stratégie différée adoptée ne génère aucun des problèmes d'incohérence d'exécution et d'hétérogénéité des données générés par les approches de conversion différée adoptées par Ferrandina [4] et Fontana [5]. En effet, les instances des classes sont converties avec une stratégie immédiate et sont donc homogènes. Supprimer une référence sur le représentant d'une instance supprimée peut se faire à tout moment sans perturber le fonctionnement du mécanisme, ni celui des programmes utilisateurs.

Notre fonction d'adaptation d'instances peut être implantée au dessus d'un système et permettre aux utilisateurs de ce système d'avoir une approche simple pour gérer la cohérence de leurs données si cette fonctionnalité n'est pas fournie par ce dernier. Les performances d'accès aux instances peuvent être améliorées en plaçant les représentants d'objets près des instances qu'ils représentent. De plus, l'implantation du mécanisme au dessus du système n'est pas totalement transparente à l'utilisateur car elle lui impose de manipuler certains attributs des instances à

l'aide de fonctions d'accès en lecture et en écriture particulières et non de manière naturelle avec le langage du système. Ce problème pourrait être résolu à l'aide d'un pré-compilateur qui remplace les instructions de lecture ou d'écriture faites de manière naturelle avec le langage du système par l'instruction appropriée faisant appel à nos fonctions de lecture et d'écriture.

Remerciements

Nos remerciements au Conseil Régional de Bretagne pour le soutien financier apporté à la réalisation de ce travail.

Références bibliographiques

[1] : Banerjee, W. Kim, K.-J. Kim, H. Korth. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases". In proc. of ACM SIGMOD Int. Conf., San Francisco, May 1987. pp 311-322.

[2] : Elisa Bertino, Lorenzo Martino. "Object-Oriented Database Management Systems : Concepts and Issues". In journal Computer, vol 24, Iss 4. USA, April, 1991. pp 33-47.

[3] : Björnerstedt, C. Hulten. "Version Control in an Object-Oriented Architecture". In Object-Oriented Concepts, Databases and Applications. W. Kim and F. Lochovsky, Eds., Addison-Wesley, Reading. Mass., 1989, pp. 451-485

[4] : Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari. "Implementing Lazy Database Updates for an object Database System". In proc. 20th Int. Conf. VLDB. Santiago, September 1994, pp. 261-272.

[5] : Edi Fontana, Yves Dennebouy. "Schema Evolution by using Timestamped Versions and Lazy Strategy". In proc. 11th Int. Conf. Advanced Databases, France, August 1995, pp. 43-60.

[6] : Wom Kim, Hong Tai Chou. "Versions of schema for Object-Oriented Databases". In proc. 14th Int. Conf. VLDB . Los Angeles, August 1988, pp. 148-159.

[7] : Quing Li, Dennis McLeod. "Conceptual Database Evolution through Learning". In OODB with Application to Case Networks and VLSI CAD. Rajiv Gupta, Elliss Horowitz Editors. Prentice Hall Series in Database and Knowledge Base Systems. 1991. pp 62-74.

[8] : Simon Monk, I Sommerville. "Schema Evolution in OODBs Using Class Versionning". In ACM SIGMOD Record, vol. 22, n° 3, September 1993.

[9] : Jason Penney, Jacob Stein. "Class Modification in the Gemstone Object-oriented Database Management System". In proc OOPSLA'87. Orlando, Florida. 1987. pp. 111-117.

[10] : Andrea H.Skarra, Stanley B. Zdonik. "The management of changing types in an object-oriented database". In proceedings of OOPSLA' 86, pp 483-491.

[11] : Jean-Marc Thevenin. "Architecture d'un SGBD Grande Mémoire". Thèse de Doctorat, Université de Paris 6, Décembre 1989.

[12] : Roberto Zicari. "A framework for O2 Schema updates". In IEEE Conf. on Data Engineering. 1991. pp 2-12.

[13] : Object Design Inc. *"ObjectStore User Guide"*. Chapter 9, 1993.