Utiliser l'objet Recordset ADO

Ecrit par J-M RABILLOUD de www.developpez.com.

Remerciements

J'adresse ici tous mes remerciements à l'équipe de rédaction de "developpez.com" et tout particulièrement à Etienne Bar, Sébastien Curutchet, Maxence Hubiche et Romain Puyfoulhoux pour le temps qu'ils ont bien voulu passer à la correction et à l'amélioration de cet article.

Introduction

Dans cet article nous allons étudier spécifiquement l'objet Recordset ADO. Cette étude va pourtant nous entraîner à travers une grande partie du modèle objet ADO.

En parcourant les forums Access et Visual basic de « developpez.com », je me suis rendu compte que si cet objet est beaucoup utilisé, il l'est souvent assez mal ce qui engendre un grand nombre de dysfonctionnements.

Ceci est d'ailleurs compréhensible puisque cet objet demande une connaissance correcte des mécanismes mis en jeux par les SGBD et les fournisseurs, ainsi que quelques astuces de programmation. En effet, ADO dans sa vision "universelle" d'accès aux données offre de nombreuses techniques. Malheureusement chaque fournisseur supportera ou non ces techniques. Ceci demande un travail important au développeur afin de bien connaître son fournisseur, sauf à utiliser un code très peu fonctionnel.

Les exemples de cet article ont été écrits pour la plupart avec Visual Basic 6 SP 5 et utilisent le moteur Jet 4.0. Pour rester cohérent, les exemples avec ADO.NET sont écrits en VB.NET.

Préambule

Le développement d'une application est souvent une suite de compromis entre performance, convivialité, ergonomie, etc.…

Par contre, lorsque l'on construit une application utilisant un SGBD, celle-ci ne doit **jamais** mettre en péril l'intégrité des données. Vous devez construire votre application en respectant cette règle. La moindre prise de risques concernant une éventuelle perte d'intégrité est à bannir.

Vous trouverez sûrement dans la suite de cet article que les jeux d'enregistrements (recordset) peuvent manquer de fonctionnalités, mais très souvent le fournisseur de données anticipe sur la règle énoncée précédemment en vous interdisant les opérations "à risques".

Toutefois, il est toujours possible de faire des erreurs, je ne saurais donc trop vous conseiller de commencer toujours par vous faire la main sur une base de test afin de bien connaître le fonctionnement du fournisseur et d'ADO.

Certains aspects de la programmation ADO sont assez techniques, aussi les ai-je regroupés sous le chapitre "Discussion autour des curseurs". N'hésitez pas à lire les exemples en fin d'article afin de mieux comprendre les concepts évoqués.

Bonne lecture.

Généralités

Définition

ADO (ActiveX Data Object) est un modèle d'objets définissant une interface de programmation pour OLE DB.

OLE DB est la norme Microsoft® pour l'accès universel aux données. Elle suit le modèle COM (Component Object Model) et englobe la technologie ODBC (Open DataBase Connectivity) conçue pour l'accès aux bases de données relationnelles. OLE DB permet un accès à tout type de source de données (même non relationnelles). OLE DB se compose globalement de fournisseurs de données et de composants de service.

Consommateur & fournisseur de données

Dans la programmation standard des bases de données en Visual Basic, il y a toujours une source de données (dans le cas de cet article, une base Access exemple : biblio.mdb).

Pour utiliser cette source de données, il faut utiliser un programme qui sait manipuler ces données, on l'appelle le fournisseur (dans certains cas serveur). Un fournisseur qui expose une interface OLE DB est appelé Fournisseur OLE DB.

Dans le cas qui nous intéresse, votre code, qui demande des données est le consommateur (ou client).

Attention, dans certains articles portant notamment sur les contrôles dépendants vous pourrez trouver ces termes avec une autre signification. En effet, si vous liez des zones de texte à un contrôle ADODC, le contrôle ADODC sera (improprement) appelé Fournisseur de données et vos zones de textes seront consommatrices.

Pourquoi est-ce impropre ?

Comme on ne voit pas le code que le contrôle ADODC utilise pour demander des informations, on tend à faire l'amalgame entre les deux. Mais c'est une erreur car le contrôle est le consommateur de données. Nous verrons l'importance de cela avec les curseurs et le paramétrage des contrôles de données.

Fournisseur & composant de service

Un fournisseur de service permet d'ajouter des fonctionnalités au fournisseur de données. Il y en a plusieurs dans ADO tel que "Microsoft Data Shaping Service" qui permet la construction de recordset hiérarchique ou "Microsoft OLE DB Persistence Provider" qui permet de stocker les recordsets sous forme de fichiers. L'appel de ces fournisseurs n'est pas toujours explicite (cf. Recordset persistant)

Un composant de service n'a pas d'existence propre. Il est toujours invoqué par un fournisseur (ou plus rarement par d'autres composants) et fournit des fonctionnalités que le fournisseur n'a pas. Dans cet article nous allons beaucoup parler du "Service de curseur pour Microsoft OLE DB" plus couramment appelé moteur de curseur.

Jeu d'enregistrement (Recordset)

Lorsque le fournisseur extrait des données de la source (requête SELECT), il s'agit de données brutes (sans information annexe) n'ayant pas un ordre particulier. Celles-ci ne sont pas très fonctionnelles, et il faut d'autres informations pour pouvoir agir sur la source de données. En fait, un recordset est un objet contenant des données de la base agencées de façon lisible, et des méta-données. Ces méta-données regroupent les informations connexes des données telle que le nom d'un champ ou son type et des informations sur la base telle que le nom du schéma.

Cette organisation est produite par un composant logiciel qui est le point central de la programmation ADO, le moteur de curseur. Le résultat ainsi obtenu est appelé curseur de données. Il y a dans ce terme un abus de langage qui explique bien des erreurs de compréhension.Le terme curseur vient de l'anglais "cursor" pour "CURrent Set Of Rows". On tend à confondre sous le même terme le

moteur de curseur qui est le composant logiciel qui gère l'objet Recordset, l'objet Recordset (données, méta-données et interface) et enfin le curseur de données qui n'est rien d'autre que l'enregistrement en cours. Cet amalgame vient de l'objet Recordset qui contient à la fois des informations destinées au moteur de curseur, des données et des méthodes qui lui sont propres. Dans la suite de cet article comme dans la majorité de la littérature disponible, vous trouverez sous la dénomination "curseur" le paramétrage de l'objet recordset vis à vis du moteur de curseurs.

Les Curseurs

Avec ADO, impossible de parler de recordset sans parler du curseur de données qui va créer ce recordset. La quasi-totalité des problèmes rencontrés lors de la programmation ADO sont dus à un mauvais choix ou à une mauvaise utilisation du curseur. Il faut dire à la décharge du développeur que ceux-ci peuvent être assez difficiles à programmer, d'où l'idée de cet article. Schématiquement le curseur doit gérer deux sortes de fonctionnalités :

- **Celles propres à la manipulation des données à l'aide du recordset**
  - Comme je l'ai déjà dit, les données renvoyées n'ont pas de notion d'enregistrement en cours ou de navigation entre enregistrements. Pour concevoir facilement l'objet Recordset il faut le voir comme une collection d'enregistrements. C'est le curseur qui permet de définir l'enregistrement en cours et s'il est possible d'aller ou pas vers n'importe quel autre enregistrement à partir de l'enregistrement en cours. C'est aussi lui qui permet de faire un filtrage, une recherche ou un tri sur les enregistrements du recordset.
- **La communication avec la source de données sous-jacente.**
  - Cette communication comprend plusieurs aspects tels que l'actualisation des données du recordset, l'envoi de modifications vers la base de données, les modifications apportées pas les autres utilisateurs, la gestion concurrentielle, etc.

Malheureusement il n'y a pas une propriété pour chacune de ces fonctionnalités. En fait, ces fonctionnalités se définissent par la combinaison de deux propriétés, le verrouillage (LockType) et le type du curseur (CursorType).

Mais avant de définir ces propriétés, il convient de choisir le positionnement du curseur et c'est là que commencent les problèmes.

Positionnement (CursorLocation)

Voilà le concept qui fait le plus souvent défaut. C'est pourtant un point fondamental. La position du curseur définit si les données du recordset sont situées dans le Process du client, c'est à dire votre application ou dans celui du serveur, c'est à dire celui du fournisseur. De plus le moteur du curseur et la bibliothèque de curseur seront donnés par le fournisseur si la position est côté serveur, alors qu'ADO invoquera le moteur de curseur client (composant de service) si la position est du côté client. La différence est primordiale car si vous n'avez pas à vous préoccuper du fonctionnement interne du fournisseur, il est utile de connaître le fonctionnement du moteur de curseur client. Nous verrons dans la discussion sur les curseurs clients tout ce que cela peut avoir d'important. La bibliothèque de curseur définit aussi quelles fonctionnalités sont accessibles comme nous le verrons un peu plus loin.

Certaines propriétés / méthodes ne fonctionnent que si le curseur est du côté client et d'autres que s'il est du côté serveur. Nous verrons plus loin quand choisir l'un ou l'autre et pourquoi.

Curseur côté serveur (adUseServer)

ADO utilise par défaut des curseurs Serveurs. Ceux-ci fournissent en des recordsets ayant moins de fonctionnalités que leurs homologues clients, mais ont les avantages suivants :

- Diminution du trafic réseau : Les données n'ayant pas à transiter vers le client
- Economie des ressources du client puisque celui-ci ne stocke pas les données.
- Efficacité de mise à jour. Comme c'est le fournisseur qui gère le recordset, celui-ci affiche les modifications en "temps réel" pour peu qu'un curseur ayant cette faculté ait été demandé.
- Facilité de programmation. Le fournisseur prenant en charge le curseur gère aussi directement les actions sur la source de données.

Ils sont très performants pour les petits recordsets et les mises à jour positionnées.

Il faut par contre garder à l'esprit :

- Ø Que chaque client connecté, va consommer des ressources côté serveur
- Ø Que le nombre de connexions, côté serveur, n'est pas illimité
- Ø La connexion doit constamment restée ouverte

Curseur côté client (adUseClient)

Les curseurs côté client présentent les avantages suivants :

- Ø Nombreuses fonctionnalités ADO (tri, filtrage, recherche …)
- Ø Une fois que les données sont dans le cache de votre application, celles-ci sont aisément et rapidement manipulables
- Ø Possibilité de travailler hors connexion, voire de stockage sur le disque

Par contre, ils présentent deux inconvénients importants :

- Ø La surcharge de la connexion est facile à atteindre, en cas de modifications fréquentes des données
- Ø Ils sont délicats à programmer comme vous allez avoir le plaisir de vous en rendre compte.

Fonctionnalités (bibliothèque de curseur)

Une fois définie la position, il vous faut choisir les fonctionnalités de votre recordset, et donc utiliser un des curseurs de la bibliothèque. Un curseur se définit en valorisant les propriétés LockType et CursorType de l'objet Recordset. Ces propriétés doivent être valorisées avant l'ouverture du recordset. Le curseur va donc définir :

Le défilement

L'accès concurrentiel

L'actualisation des données du recordset

Autant dire maintenant que le choix d'un mauvais curseur peut donner des comportements aberrants à votre application.

 **Piège n°1**

Lorsque vous demandez un curseur qui n'existe pas dans la bibliothèque, le moteur ou le fournisseur vous en attribuera un autre n'ayant pas les mêmes fonctionnalités sans toutefois déclencher d'erreur. Le choix de l'autre reste à mes yeux un mystère, puisque la documentation dit "le curseur le plus proche".

Verrouillage (LockType)

Le verrouillage (accès concurrentiel) est un élément indispensable des SGBD Multi-Utilisateurs. Le verrouillage consiste à interdire la possibilité a deux utilisateurs (ou plus) de modifier le même enregistrement en même temps. Il y a globalement deux modes de verrouillage :

Quel verrou choisir ?

Les considérations de ce paragraphe seront sur les curseurs côté serveur. La principale différence entre ces deux modes de verrouillage vient du comportement optimiste. En effet, un verrouillage pessimiste ne se préoccupe pas de la version de l'enregistrement. Un enregistrement est ou n'est pas verrouillé. S'il ne l'est pas rien n'empêche plusieurs utilisateurs de modifier successivement le même enregistrement. Dans le cadre d'un verrouillage optimiste il n'est théoriquement pas possible de modifier un enregistrement modifié par ailleurs. Cela est partiellement faux car sur un curseur qui reflète les modifications (KeySet par exemple), le numéro de version sera remis à jour à chaque rafraîchissement du Recordset. Le verrouillage optimiste ne sera donc que temporaire sur un curseur "dynamique". Ceci implique les observations suivantes :

Le verrouillage pessimiste interdit les modifications simultanées sur un enregistrement mais pas les modifications successives.

Le verrouillage optimiste peut n'être que temporaire et risque de mal jouer son rôle.

Regardons le code ci-dessous :

```
Private cnn1 As ADODB.Connection, MonRs As ADODB.Recordset

Private Sub Command1_Click()

MonRs.Fields("Author").Value = Text1.Text
MonRs.Update

End Sub

Private Sub Form_Load()

Set cnn1 = New ADODB.Connection
cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=d:\biblio.mdb ;User Id=Admin; Password="
cnn1.Properties("Jet OLEDB:Page Timeout") = 10000
Set MonRs = New ADODB.Recordset
MonRs.Open "SELECT * FROM Authors", cnn1, adOpenKeyset,
adLockOptimistic
Text1.Text = MonRs.Fields("Author").Value

End Sub
```

J'ai modifié la valeur de la propriété dynamique "Jet OLEDB:Page Timeout" afin qu'il y ait 10 secondes entre chaque rafraîchissement du recordset. Si j'exécute deux instances de ce programme simultanément, je n'aurais jamais d'erreur si j'attends au moins 10 secondes entre chaque modification quelle que soit l'instance. Sinon, une erreur de verrouillage optimiste peut se produire.

Comme souvent avec ADO, ces considérations peuvent forcer le choix du curseur. Par exemple, si vous voulez obtenir un verrouillage optimiste permanent avec MS Jet, vous devez utiliser un curseur statique. Comme Jet ne vous fournira pas un curseur statique acceptant les modifications côté serveur, vous devrez utiliser un curseur côté client

Le verrouillage pessimiste est configurable. Par défaut, si on tente d'accéder à un enregistrement verrouillé, il y aura déclenchement d'une erreur et la tentative d'accès échouera. Cependant on peut paramétrer la connexion, à l'aide des propriétés dynamiques "Jet OLEDB:Lock Delay" et "Jet OLEDB:Lock Retry" afin que la pose du verrou soit mise dans une boucle d'attente. Une telle boucle si elle empêche le déclenchement d'une erreur peut facilement encombrer le serveur.

Le verrouillage pessimiste (*adLockPessimistic*)

Ce mode de verrouillage prend effet au moment de l'édition de l'enregistrement. Le fournisseur pose un verrou sur l'enregistrement dès que l'utilisateur tente d'en modifier une valeur (on dit à l'édition). Le verrou dure jusqu'à la validation des modifications. Une erreur récupérable se produit lorsqu'un utilisateur tente de modifier un enregistrement verrouillé.

Le verrouillage optimiste (*adLockOptimistic*)

Ce mode de verrouillage prend effet lors de l'enregistrement des modifications.Ce n'est pas un verrou stricto sensu, mais une comparaison de valeurs.

Selon les SGBD et aussi selon la structure de la table, la vérification se fait sur un numéro de version, un champ date de modification (TimeStamp) ou la valeur des champs.

Prenons un exemple avec les numéros de version. Chaque recordset prend le numéro de version de l'enregistrement. Lors de la validation des modifications, le numéro de version change.Toutes les modifications d'autres utilisateurs ayant alors un mauvais numéro de version, celles-ci échouent.

D'autres modes de verrouillage

ADO supporte aussi un mode appelé optimiste par lot (**adLockBatchOptimistic**), qui marche comme expliqué ci-dessus à quelques variations près que nous verrons lors du traitement par lot.

La propriété LockType accepte aussi un mode ReadOnly (**adLockReadOnly**) qui interdit la modification des données du recordset.

Type de curseur (CursorType)

Dans cette propriété sont mêlées le défilement et la sensibilité aux modifications des données.

Le défilement est une fonctionnalité du curseur qui représente la faculté du jeu d'enregistrement à :

- refléter la position[1] de l'enregistrement en cours
- organiser le jeu de données
- se déplacer dans ce jeu.

La sensibilité représente la faculté du recordset de refléter les modifications, ajouts, suppressions effectués sur les données.

Je vais vous présenter ces curseurs en précisant à chaque fois leurs fonctionnalités

Il existe quatre types de curseur :

En avant seulement (*adOpenForwardOnly****)***

Position è Non

Défilement è En avant

Sensibilité è Pas de mise à jour des données modifiées certaines

Ce type de curseur est le plus rapide. Idéal pour la lecture de données en un seul passage.

Statique (*adOpenStatic****)***

Position è Oui

Défilement è Bidirectionnel

Sensibilité è Pas de mise à jour des données modifiées

Copie de données. Les curseurs côté client sont toujours statiques.

Jeu de clé (*adOpenKeyset****)***

Position è Oui

Défilement è Bidirectionnel

Sensibilité è Reflète les modifications de données mais ne permet pas de voir les enregistrements ajoutés par d'autres utilisateurs

Demande d'être utilisé avec des tables indexées. Très rapide car il ne charge pas les données mais juste les clés.

Dynamique (*adOpenDynamic****)***

Position è Oui

Défilement è Bidirectionnel

Sensibilité è Reflète toutes les modifications de données ainsi que les enregistrements ajoutés ou supprimés par d'autres utilisateurs

Ce poids lourd des curseurs, uniquement serveur, n'est pas supporté par tous les fournisseurs.

Héritage

La création d'un objet Recordset sous-entend toujours la création d'un objet Connection et d'un objet Command. Soit ces objets sont créés de façon explicite soit ils sont créés implicitement par ADO lors de l'ouverture du recordset.

La création implicite pose deux problèmes :

- Ø Ces objets cessent d'exister lors de la fermeture du Recordset et doivent être récréés à chaque exécution et on multiplie ainsi le nombre de connexions à la base.
- Ø On oublie de paramétrer correctement ces objets, voire on oublie qui plus est qu'on les a créés, mais le recordset héritant de certaines de leurs propriétés, on n'obtient pas ce que l'on désire.

Ces deux objets se retrouvent dans les propriétés **ActiveConnection** et **ActiveCommand** de l'objet Recordset. Cet héritage est important comme nous allons le voir dans la suite de cet article.

Un recordset dont la propriété ActiveConnection vaut Nothing et utilisant un curseur client est un recordset déconnecté.

Un recordset dont la propriété ActiveCommand vaut Nothing est dit "Volatile".

---

[1] i Par position, on entend la possibilité de déterminer la position de l'enregistrement en cours au sein du recordset.

Discussion autour des curseurs

Choisir le curseur dont on a besoin n'est pas très compliqué, dès lors que l'on a bien conscience de ce que l'on va obtenir en gérant le paramétrage. Pour savoir cela, il faut comprendre l'ensemble des mécanismes en jeu. Nous allons aborder quelques points techniques un peu ardus avant d'énoncer des règles générales qui devraient vous aider dans votre choix.

Le fait de choisir la position de son curseur entraîne de nombreuses conséquences, qu'on évalue souvent mal et qui pourtant vont énormément jouer sur le comportement du recordset. Les concepts que nous allons aborder dans cette discussion sont assez divers, aussi accrochez-vous.

Méta-données

Le recordset a donc besoin pour fonctionner de données autres que celles renvoyées par la requête, a fortiori si vous allez modifier la base par l'intermédiaire de ce recordset. Ces méta-données se divisent en deux blocs, celles qui sont stockées dans les propriétés statiques et dynamiques des objets Fields, et celles qui sont stockées dans la collection des propriétés dynamiques de l'objet Recordset. Celles stockées dans les propriétés statiques des objets Fields sont les mêmes quelle que soit la position du curseur.

Par contre, les propriétés dynamiques de l'objet recordset, ainsi que la collection Properties de chaque objet Field, sont très différentes selon la position du curseur.

Si on compare un recordset obtenu avec un curseur côté client de son homologue côté serveur, on constate qu'il possède beaucoup plus de propriétés dynamiques, alors que le second est le seul à posséder la série des propriétés "Jet OLE DB".

Ceci vient de la différence fondamentale de comportement du curseur. Lorsqu'on invoque un curseur côté serveur, le fournisseur OLE DB prend en charge la gestion de ce curseur. Il valorise quelques propriétés dynamiques qui lui sont propres, mais le recordset n'a pas besoin de stocker des informations de schéma puisque c'est le fournisseur qui va agir sur la base de données.

Données (Fields)

Pour stocker les données de la requête, les recordsets possèdent une collection Fields. Chaque objet Field représente un champ invoqué par la requête SELECT ayant créé le recordset. Un objet Field comprend :

La valeur de la donnée de l'enregistrement en cours

En fait, la valeur apparaît trois fois. La propriété Value contient la valeur existante dans votre Recordset, la propriété UnderlyingValue contient la valeur existante dans la source de données lors de la dernière synchronisation enfin la propriété OriginalValue contient la valeur de la base lors de la dernière mise à jour. Nous allons voir plus loin tout ce que cela permet et induit.

Les méta-données du champ (nom, type, précision)

Elles sont disponibles avant l'ouverture du recordset si la connexion est déjà définie dans l'objet Recordset et si la propriété source contient une chaîne SQL valide, sinon on ne peut y accéder qu'après l'ouverture du recordset.

Les informations de schéma

Elles se trouvent dans la collection Properties de l'objet Field. Elles existent toujours quel que soit le côté du curseur, mais il y en a plus du côté client. Attention, ces informations peuvent être fausses si le moteur de curseur n'en a pas besoin, pour un recordset en lecture seule par exemple.

Taille du cache

Régler la taille du cache peut être important pour l'optimisation des curseurs serveurs. En fait, lorsque vous demandez un jeu d'enregistrements au fournisseur de données, celui-ci n'est pas forcément complètement rapatrié dans l'espace client. Vous pouvez définir le nombre d'enregistrements qui seront transmis à chaque fois en réglant la propriété CacheSize de l'objet Recordset. Ce réglage ce fait en donnant le nombre d'enregistrements que le cache peut accepter dans la propriété CacheSize. S'il y a plus d'enregistrements retournés qu'il n'y a de place dans le cache, le fournisseur transmet uniquement la quantité d'enregistrements nécessaire pour remplir le cache. Lors de l'appel d'un enregistrement ne figurant pas dans le cache, il y a transfert d'un nouveau bloc d'enregistrements.

Ce mode de fonctionnement implique toutefois un piège. Lors des rafraîchissements périodiques de votre recordset (dynamique ou KeySet), les enregistrements du cache ne sont pas rafraîchis. Il faut forcer celui-ci en appelant la méthode Resync.

Il n'y a pas de règles absolues pour dire qu'elle taille de cache il faut utiliser. Je n'ai pour ma part pas trouvé d'autres méthodes que de faire des tests selon les applications.

Marshaling

Les objets ADO suivent les règles du marshaling COM pour la communication inter-process. Le marshaling (ou re-direction) représente la méthode de communication des objets ou/et des données au-delà des limites d'un thread ou d'un processus. Dans le cas de la communication in-process, ADO utilise la gestion de threads libres (non cloisonnés), il n'y a donc jamais de marshaling in-process avec les recordsets ADO. C'est le cas par exemple avec des applications mono-poste utilisant des SGBD qui ne sont pas Client / Serveur ou des sources de données particulières.

Dans tous les autres cas, il y a toujours marshaling. Néanmoins, il y a de profondes différences selon la position du curseur. Dans le principe, lorsque vous créez un recordset, celui-ci est construit dans l'espace du serveur. Il est ensuite transmis au process client. Si vous êtes côté serveur, la transmission se fait par référence et vous manipulez un pointeur vers le curseur qui se trouve dans le process du serveur. En ce sens, vous n'avez pas à vous préoccuper de comment les modifications du recordset sont répercutées sur la source de données. Seuls les enregistrements présents dans le cache sont effectivement transmis à l'espace client.

Par contre, lors de l'utilisation d'un curseur client, le Recordset est transmis par valeur, c'est à dire qu'une copie est créée dans l'application cliente (données et méta-données). Les informations de connexions ne sont pas transmises avec les données puisque vous les avez définies lors de la création de l'objet recordset. De cette observation découle deux concepts fondamentaux des curseurs clients :

- Comme le transfert peut être long, il ne serait pas sans risque de créer un recordset "dynamique" puisque le rafraîchissement périodique pourrait débuter avant la récupération complète du jeu d'enregistrement. Ainsi ADO ne vous fournira que des curseurs statiques côté client, quel que soit le fournisseur. Une fois le rapatriement effectué, le recordset n'est plus connecté, dans le sens ou tous les appels ultérieurs à la source de données seront explicites. Une fois le jeu d'enregistrement transmis au processus client, l'accès aux données va être très rapide.
- Pour mettre à jour des données vers la source, il va falloir re marshaler le recordset vers le processus du serveur. La communication d'un curseur client avec la source de données est toujours assez lente.

Pour en finir avec le marshaling ADO, sachez que le recordset et sa connexion doivent être dans le même processus.

Mécanismes de base

Nous avons maintenant le minimum requis pour comprendre ce qui se passe lors de la manipulation d'un objet Recordset. Je rappelle ici ce point fondamental, que nous reverrons en détails dans les rappels ADO, l'utilisation des propriétés dynamiques des objets ADO demandent toujours la définition du fournisseur. En clair, la propriété Provider doit être définie pour utiliser les propriétés dynamiques de l'objet Connection, et la propriété ActiveConnection doit être valorisée pour utiliser les collections Properties des objets Command et Recordset.

Nous allons regarder maintenant quelques opérations courantes sur un recordset pour voir ce qui se passe au niveau du moteur de curseur. Ces exemples seront principalement dans le cadre des curseurs clients, puisque le fonctionnement des recordsets côté serveur est géré par le fournisseur de données.

Création du jeu d'enregistrement

Généralement, pour ouvrir un Recordset, quelle que soit la méthode utilisée, vous allez consciemment ou non envoyer une requête SQL SELECT au fournisseur de données. Il n'est nullement nécessaire d'avoir la moindre connaissance de la structure de la base, pour peu que celle-ci possède des requêtes stockées (Vues), néanmoins on connaît en général le schéma de la source (au moins superficiellement). Fondamentalement, les différences de fonctionnement commencent lors de l'appel de la méthode open, mais souvent pour des raisons de marshaling, tout se passe dès le paramétrage de l'objet. Examinons le code suivant :

```
Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Mes
documents\jmarc\ADO\bdd\biblio.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseClient
MonRs.ActiveConnection = Cnn1
MonRs.Source = "SELECT * FROM Authors"
MonRs.Open , , adOpenStatic, adLockBatchOptimistic, adCmdText
```

Si j'espionne le recordset avant l'appel de la méthode Open, je vois que quelques propriétés dynamiques (informations de schéma) et toutes les propriétés statiques (méta-données) des objets Field du recordset sont correctement valorisées quelle que soit la position du curseur. La raison est simple, ADO a envoyé une requête de récupération de schéma lors de l'affectation de la propriété source. Si vous avez bien compris ce que nous avons vu précédemment, vous voyez déjà apparaître un des problèmes des curseurs clients. En effet, avec un curseur côté serveur, la requête de schéma valorise les méta-données de l'objet recordset (serveur) puis un pointeur d'interface est transmis vers l'application cliente. Dans le cas du curseur client, le moteur de curseur transfert un recordset contenant les méta-données vers l'objet Recordset qui est situé dans l'espace client. Il y une différence importante de temps de traitement entre les deux, alors que le volume des méta-données transmis est très faible. Il va y avoir ensuite une différence encore plus importante lors de l'appel de la méthode Open.

Côté serveur, le fournisseur exécute la requête et transmet les enregistrements nécessaires au remplissage du cache, pas besoin de re transmettre un pointeur puisque celui-ci existe déjà. Ce traitement peut être encore accéléré avec un curseur à jeu de clé (Key set) puisque celui ci ne valorise qu'un jeu de clé.

Le moteur de curseur côté client va par contre devoir récupérer les valeurs de la source renvoyées par la requête et transmettre le recordset vers le recordset de l'espace client. Ceci étant fait, comme le paramétrage précise que le Recordset doit être modifiable (Updatable), il va renvoyer une requête de schéma, évidemment différente de la première, afin de récupérer les informations nécessaires aux actions éventuelles sur la source de données.

Comme nous le voyons, la construction d'un recordset client est assez lourde, surtout si la requête doit renvoyer des milliers d'enregistrements.

Action sur le recordset

Que va-t-il se passer lorsqu'on fait une modification sur l'objet recordset ?

Pour bien comprendre l'influence de la position du curseur je vais imaginer une table sans clé, contenant des doublons parfaits (lignes dont toutes les valeurs sont identiques).

Exécutons le code suivant

```
Dim MaCnn As ADODB.Connection, MonRs As ADODB.Recordset

Set MaCnn = New ADODB.Connection
MaCnn.Open "Provider=Microsoft.Jet.OLEDB.4.0 ;Data
Source=D:\user\jmarc\bd6.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseServer
MonRs.Open "SELECT * From TabSansCle", MaCnn, adOpenKeyset,
adLockOptimistic, adCmdText
MonRs.Find "nom = 'bb'"
MonRs!numer = 21
MonRs.Update
```

Dans ma table j'ai plusieurs enregistrements dont le nom est "bb", certains d'entre eux étant des doublons parfaits. Si je regarde ma base après exécution, je constate qu'un de mes enregistrements a pris 21 pour valeur "numer".

Maintenant j'exécute le même code en positionnant côté client le recordset. A l'appel de la méthode Update, j'obtiens l'erreur "80004005 : Informations sur la colonne clé insuffisantes ou incorrectes ".

Ce petit exemple permet de démontrer la différence de fonctionnement entre les curseurs clients et serveurs. Dans le cas de l'opération côté serveur, c'est le fournisseur qui gère le curseur. Le fournisseur sait localiser physiquement l'enregistrement en cours du recordset dans la source de données et peut donc procéder à la modification.

Par contre, lorsqu'on procède du côté client, le moteur de curseur doit écrire une requête action pour le Fournisseur. Dans notre cas, le curseur ne trouve pas dans les méta-données de clé primaire pour la table (puisque la table n'en contient pas) ni d'index unique. Il se trouve donc dans l'impossibilité d'écrire une requête action **pertinente**, puisqu'il ne peut pas identifier l'enregistrement en cours, ce qui provoque la génération de l'erreur dont pour une fois le message est tout à fait explicite. Dans ce cas précis, le curseur porte mal son nom puisqu'il existe un enregistrement en cours dans le recordset, mais qu'il ne peut pas être identifié pour agir dessus. Néanmoins j'ai pris là un cas extrême, à la limite d'être absurde.

Dans le cas des recordsets côté client, il y a toujours tentative de construction d'une requête action pour traduire les modifications apportées à l'objet Recordset.

Echec ou réussite

Il est important de comprendre comment le curseur va réagir lors de l'échec ou de la réussite d'une action. Le moteur de curseur (client ou serveur) utilise les méta-données stockées dans le recordset pour valider une opération sur un recordset. Ce contrôle a lieu en deux temps :

- Lors de l'affectation d'une valeur à un champ

  La valeur saisie doit respecter les propriétés de l'objet Field concerné. Vous aurez toujours un message d'erreur si tel n'est pas le cas, mais vous n'aurez pas le même message selon la position du curseur. Si par exemple vous tentez d'affecter une valeur de type erroné à un champ, vus recevrez un message d'incompatibilité de type pour un curseur serveur, et une erreur Automation pour un curseur client.

- Lors de la demande de mise à jour

  Indépendamment de la position lorsqu'il y a un échec, une erreur standard est levée et il y a ajout d'une erreur à la connexion définie dans la propriété ActiveConnection de l'objet Recordset. La propriété Status de l'enregistrement concerné prendra une valeur censée refléter la cause de l'échec. Là encore, la pertinence de cette valeur va varier selon la position du curseur. Dans une opération côté serveur, si dans un accès avec verrouillage optimiste je cherche à modifier une valeur d'un enregistrement qui vient d'être modifiée par un autre utilisateur, j'obtiendrai une erreur "signet invalide" puisque le numéro de version ne sera plus le bon. Dans la même opération côté client, il y aura une erreur "violation de concurrence optimiste".

Le pourquoi de ces deux erreurs différentes pour une même cause va nous amener à comprendre comment le moteur de curseur du client construit ses actions.

En cas de réussite de la modification, l'opération est marquée comme réussie (dans la propriété Status) et il y a alors synchronisation.

Synchronisation

La synchronisation est l'opération qui permet de mettre à jour le recordset avec les données de la source de données sous-jacente. La synchronisation est normalement explicite, c'est-à-dire demandée par l'utilisateur à l'aide de la méthode Resync, mais elle suit aussi une opération de modification des données, elle est alors implicite.

- Synchronisation explicite
  - Elle peut porter sur tout ou partie d'un recordset ou sur un champ. Il existe deux stratégies différentes selon que l'on désire rafraîchir l'ensemble du recordset ou juste les valeurs sous-jacentes. Dans le premier cas, toutes les modifications en cours sont perdues, dans l'autre on peut anticiper la présence de conflit.
- Synchronisation de modification
  - Elle est plus complexe à appréhender. Elle suit les règles suivantes :
    - Elle porte **uniquement** sur les enregistrements modifiés dans **votre** recordset
    - Les champs sur lesquels la valeur a été modifiée sont **toujours** synchronisés
  - Sur un recordset côté serveur, la synchronisation qui suit une opération sur les données est toujours automatique et non paramétrable. Elle porte sur tous les champs de l'enregistrement modifié.
  - Par contre du côté client il est possible de spécifier le mode de synchronisation à l'aide de la propriété dynamique "Update Resync".

N'oubliez pas la règle d'or, les champs modifiés d'une opération réussie mettent la nouvelle valeur de la base dans leur propriété OriginalValue.

Modification d'un Recordset client

Ce que nous allons voir maintenant est propre aux curseurs côté client. Rappelez-vous que la gestion des jeux d'enregistrements côté serveur est faite par le fournisseur de données et qu'il n'est donc pas la peine de se pencher sur ce fonctionnement en détail.

Lors d'une modification des données de l'objet Recordset, le moteur de curseur va construire une ou plusieurs requêtes action permettant de transmettre les modifications de l'objet Recordset vers la source de données. En SQL, il n'existe que trois types de requêtes action pour manipuler les données, de même, il n'existe que trois méthodes de modification des données d'un recordset. Dans la suite de ce chapitre, nous regarderons l'application de traitement direct, les traitements par lots reposant sur le même principe (en ce qui concerne les requêtes SQL).

Si vous êtes étanche au SQL, le passage qui suit va vous sembler abscons. Je vous engage alors à commencer par consulter l'excellent cours de SQLPRO.

Modification (Update)

Commençons par un exemple simple.

```
Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Mes
documents\jmarc\ADO\bdd\biblio.mdb ;User Id=Admin; Password
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseClient
MonRs.ActiveConnection = Cnn1
MonRs.Open "SELECT * FROM Authors", , adOpenStatic,
adLockBatchOptimistic, adCmdText
MonRs.Find "Author = 'Gane, Chris'", , adSearchForward, 1
MonRs![year born].Value = "1941"
MonRs.Update
```

Lors de l'appel de la méthode Update, le moteur de curseur va construire une requête SQL UPDATE pour appliquer la modification. Par défaut le modèle de la requête est de la forme

UPDATE *Table*
SET *Champs modifiés=recodset.fields(Champs modifiés).Value*
WHERE *champs clés= recodset.fields(Champs clés).OriginalValue*

```
    AND champs modifiés= recodset.fields(Champs modifiés).OriginalValue
    AND éventuelle condition de jointure
```

Donc dans l'exemple que j'ai pris, la requête transmise par le moteur de curseur au fournisseur OLE DB sera :

```
UPDATE Author
SET [year born] = 1941
WHERE  Au_Id = 8139
   AND [year born] = 1938
```

Comment le moteur de curseur fait-il pour construire cette requête ?

Il va chercher pour chaque champ modifié la table correspondante, cette information étant toujours disponible dans la collection properties de l'objet Field correspondant, puis il va récupérer dans les méta-données la clé primaire de la table. Avec ces informations, il va construire sa requête sachant que les valeurs utilisées dans la clause WHERE sont toujours tirées des propriétés **OriginalValue** des objets Fields de l'enregistrement en cours.

Si d'aventure vous n'avez pas inclus le(s) champ(s) constituant la clé primaire, il est possible que le moteur ait quand même récupéré cette information. Si tel n'est pas le cas ou si votre table ne possède pas de clé primaire, le moteur lancera une requête d'information pour trouver un éventuel index unique lui permettant d'identifier de manière certaine l'enregistrement courrant. Si cela s'avère impossible, vous obtiendrez une erreur comme nous l'avons vu plus haut**. Il faut donc toujours veiller lors de l'utilisation d'un curseur client à donner les informations nécessaires au moteur de curseur si on désire effectuer des modifications de données**.

Cette requête est ensuite transmise au fournisseur OLE DB. Celui-ci retourne en réponse au moteur de curseur le nombre d'enregistrements affectés. Si celui-ci est égal à zéro, il y a déclenchement d'une erreur, sinon l'opération est marquée comme réussie. Il y a alors synchronisation de l'enregistrement.

Tout a l'air parfait mais il y a une faute de verrouillage. En effet, s'il y a eu modification ou verrouillage de l'enregistrement par un autre utilisateur, ce n'est pas stricto sensu par le changement de numéro de version du verrouillage optimiste que la requête échoue mais parce que les valeurs stockées dans les propriétés OriginalValue ne permettent plus d'identifier l'enregistrement en cours. Or cela peut être très différent. Imaginons le scénario suivant, dans le temps entre la création du recordset et l'appel de la méthode Update, un autre utilisateur a changé le nom en "Gane, Peter". Normalement, du fait du verrouillage, la modification de la date de naissance devrait échouer, pourtant il suffit de lire le code SQL correspondant pour voir qu'elle va réussir, puisque le champ 'Author' n'apparaît pas dans la clause WHERE.

Pour éviter ce problème ou pour contourner d'autres limitations, on peut paramétrer les règles de construction de la requête action du moteur de curseur à l'aide de la propriété dynamique "Update Criteria".

Il y a aussi un danger si on fait l'amalgame entre ADO et SQL. Avec une requête SQL de modification je peux m'affranchir de la valeur originelle du champ. Par exemple la requête SQL suivante est valide :

```
UPDATE Author
SET [year born] = [year born] + 1
WHERE  Au_Id = 8139
```

Mais du fait de la concurrence optimiste elle ne peut pas être exécutée par le moteur de curseur ADO avec un code comme celui ci-dessous :

```
MonRs.Find "Author = 'Gane, Chris'", , adSearchForward, 1
MonRs![year born].Value = MonRs![year born].Value + 1
MonRs.Update
```

Ce code ne fonctionnera que si la valeur du champ [year born] dans la base est la même que celle présente

Suppression (DELETE)

Ce sont ces requêtes qui posent le moins de problèmes avec ADO. Dans ce cas, la requête n'utilisera que le(s) champ(s) composant la clé primaire pour écrire la requête Delete SQL.

```
MonRs.Find "Author = 'Gane, Chris'", , adSearchForward, 1
MonRs.Delete adAffectCurrent
```

Sera transformé par le moteur en :

```
DELETE Authors WHERE Au_Id = 8139
```

Là pas de problème de critère ou de synchronisation.

Ajout (INSERT INTO)

Ces requêtes vont nous permettre de comprendre le principe de la synchronisation. Une remarque s'impose au préalable. Ne tentez pas d'entrer une valeur dans un champ à valeur automatique (NumeroAuto ou TimeStamp), vous obtiendrez alors une erreur systématique. De même, donnez des valeurs à tous les champs ne pouvant être NULL avant d'appeler la méthode Update. Prenons un exemple :

```
Dim Cnn1 As ADODB.Connection, MonRs As ADODB.Recordset, ValCle As
Long

Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Mes
documents\jmarc\ADO\bdd\biblio.mdb ;User Id=Admin; Password
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseClient
MonRs.ActiveConnection = Cnn1
MonRs.Open "SELECT * FROM Authors", , adOpenStatic,
adLockOptimistic, adCmdText
'MonRs.Properties("Update Resync") = adResyncAutoIncrement
With MonRs
    .AddNew
    .Fields("Author") = "RABILLOUD, Jean-Marc"
    .Fields("year born") = 1967
    .Update
End With
```

Comme vous le voyez, je n'entre pas de valeur pour le champ "Au_Id" puisqu'il s'agit d'un champ NumeroAuto. La requête action écrite par le moteur de curseur sera :

```
INSERT INTO Authors (Author, [year born])
        VALUES ("RABILLOUD , Jean-Marc",1967)
```

Comme vous le voyez, la requête ne contient pas de valeur pour le champ "Au_Id" celle-ci étant attribuée par le SGBD. Jusque là pas de problème. Mais que va-t-il se passer si j'ai besoin de connaître cette valeur dans la suite de mon code.

Il me suffit d'écrire :

```
ValCle = MonRs.Fields("AU_Id").Value
```

Bien que je n'aie pas spécifié le mode de mise à jour (ligne en commentaire), la valeur par défaut de synchronisation est "adResyncAutoIncrement" et le code fonctionne. Attention, cela ne fonctionnera pas avec une version d'ADO inférieure à 2.1 ni avec un fournisseur tel que JET 3.51.

Cela fonctionne car le moteur de curseur envoie des requêtes complémentaires afin de mettre à jour le nouvel enregistrement. Cela ne fonctionnera pas avec Jet 3.51 car il ne supporte pas la requête SELECT @@IDENTITY qui permet de connaître le nouveau numéro attribué, et sans celui-ci il n'est pas possible pour le moteur de curseur d'identifier l'enregistrement nouvellement créé.

Pour connaître les possibilités de paramétrage de la mise à jour, allez lire le paragraphe "Update Resync".

Requête avec jointure

Jusque là, nous avons regardé des cas simples ne portant que sur une seule table, mais dans les autres cas, cela peut être nettement plus compliqué.

Imaginons la requête suivante :

```
SELECT * FROM Publishers, Titles WHERE Publishers.PubID = Titles.PubID
```

Note SQL : Il est bien évidemment plus logique d'écrire une requête avec INNER JOIN, mais dans ce chapitre je vais faire mes jointures dans la clause WHERE afin de garder une écriture cohérente avec le moteur de curseur.

J'utilise le code suivant :

```
Dim Cnn1 As ADODB.Connection, MonRs As ADODB.Recordset

Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=d:\biblio.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
With MonRs
    .CursorLocation = adUseClient
    .ActiveConnection = Cnn1
    .Open "SELECT * FROM Publishers, Titles WHERE Publishers.PubID =
Titles.PubID", , adOpenStatic, adLockOptimistic, adCmdText
    .Find "Title='Evaluating Practice'"
    .Fields("Name") = "ALPHA BOOKS"
    .Fields("Title") = .Fields("Title") & "s"
    .Update
End With
```

Pour résumer, je recherche le livre dont le titre est 'Evaluating Practice' je veux changer son éditeur, par un autre éditeur existant dans la base et ajouter un 's' au titre du livre. Bien sûr, vu comme cela, la technique a l'air un peu branlante, mais c'est exactement ce que 95% des utilisateurs essayent de faire de manière transparente avec un Datagrid par exemple. Que va-t-il se passer si vous exécutez ce code ?

Le moteur de curseur va écrire autant de requêtes action que de tables concernées par les modifications dans notre cas :

```
UPDATE Publishers
SET Name =  'ALPHA BOOKS'
WHERE  PubId = 129 AND Name = 'ALLYN & BACON'
```

Et

```
UPDATE Titles
SET Title = 'Evaluating Practices'
WHERE  ISBN = '0-1329231-8-1' AND Title = 'Evaluating Practice'
```

Comme vous le voyez, ce n'est pas ce que nous voulions obtenir puisque l'éditeur 'ALLYN & BACON' a disparu de la base. La faute est énorme puisqu'il faut modifier la clé étrangère pour obtenir ce que nous voulons mais nous voyons que le moteur de curseur ne peut pas gérer ce style de programmation.

Dans ce cas pourtant, il existe des propriétés dynamiques qui permettent de contourner le problème. En effet vous pouvez définir à l'aide de la propriété "Unique Table", une table sur laquelle porteront les modifications dans une requête avec jointure, et à l'aide de la propriété "Resync Command" un mode de synchronisation pour ces mêmes requêtes. Pour voir leur utilisation, lisez l'exemple "propriétés dynamiques – Resync Command" du présent cours.

FireHose, un curseur particulier

Par défaut, le curseur fourni est un curseur "FireHose" qui a les caractéristiques suivantes :

- Côté serveur
- En avant seulement
- Lecture seule
- Taille du cache=1

Ce type de curseur est le plus rapide. Il possède une particularité dont il convient de se méfier. La connexion utilisée par un tel curseur est exclusive, ce qui revient à dire que la création d'un autre recordset utilisant la même connexion engendre la création d'une connexion implicite. Encore pire, si

vous fermez votre recordset avant d'avoir atteint la position EOF, la connexion sera toujours considérée comme étant utilisée. Il convient donc de faire attention avec ce type de curseur.

 Attention bien que la méthode MoveLast provoque un mouvement vers l'avant, elle déclenche une erreur sur les curseurs "En avant seulement".

Conseils pour choisir son curseur

En suivant les règles données ci-dessous vous ne devriez pas vous tromper souvent. Néanmoins ces règles ne sont pas intangibles et dans certains cas il vous faudra procéder à votre propre analyse pour déterminer le curseur nécessaire.

- Certaines propriétés/méthodes ne fonctionnent qu'avec un curseur d'un côté spécifique, si vous avez besoin de celles-ci, utilisez le curseur correspondant.
- Les fournisseurs ne donnent pas accès à toutes les combinaisons possibles. Etudiez d'abord les curseurs que le fournisseur peut mettre à disposition
- Sur une application où le SGBD est sur la machine du client et qui ne peut avoir qu'un utilisateur simultané, on utilise toujours un curseur côté serveur.
- Si vous devez modifier des données (mise à jour, ajout, suppression) utilisez un curseur côté serveur.
- Si vous devez manipuler un gros jeu d'enregistrements, utilisez un curseur côté client.
- Privilégiez toujours
  - Les curseurs en avant s'ils sont suffisants
  - Les curseurs en lecture seule
- Pour les applications Multi-Utilisateurs
  - Verrouillez en mode pessimiste les curseurs côté serveur
  - Travaillez par lot sur les curseurs côté clients
- Préférez toujours l'utilisation du SQL par rapport aux fonctionnalités du recordset.

Synchrone Vs Asynchrone

Avec ADO, on peut travailler de manière synchrone ou asynchrone. La programmation asynchrone présente évidemment l'avantage de ne pas bloquer l'application en cas de travaux longs du fournisseur mais demande de gérer une programmation événementielle spécifique. En fait, certains événements ADO se produiront quel que soit le mode choisi, d'autres ne seront utilisés qu'en mode asynchrone.

On peut travailler avec des connexions asynchrones et/ou des commandes asynchrones. Pour travailler de manière asynchrone vous devez utiliser un objet Connection (pas de connexion implicite). Le risque avec le travail asynchrone réside dans la possibilité de faire des erreurs sur des actions dites "bloquantes".

Opération globale (par lot)

Les opérations globales sont une suite d'instructions affectant plusieurs enregistrements ou manipulant la structure. Bien que l'on puisse y mettre n'importe qu'elle type d'instructions, on essaye de grouper des opérations connexes dans une opération par lot. Trois types d'opérations globales sont généralement connus.

Les transactions

La plupart des SGBDR gèrent le concept de transaction (on peut le vérifier en allant lire la valeur de Connection.Properties("Transaction DDL")).

Une transaction doit toujours respecter les règles suivantes :

- Atomicité : Soit toutes les modifications réussissent, soit toutes échouent
- Cohérence : La transaction doit respecter l'ensemble des règles d'intégrité
- Isolation : Deux transactions ne peuvent avoir lieu en même temps sur les mêmes données

- Durabilité : Les effets d'une transaction sont définitifs. Par contre toute transaction interrompue est globalement annulée.

Avec ADO, les transactions s'utilisent sur l'objet Connection

Les procédures stockées

Procédure écrite dans le SGBD qui fait l'opération. Elle présente de multiples avantages.

- Pas de trafic réseau
- Environnement sécurisé
- Pas de code complexe pour l'utiliser

Par contre, il faut qu'elle soit prévue dans la base ce qui est loin d'être toujours le cas.

Avec ADO elle s'utilise avec l'objet Command.

Gérée par le code

C'est le cas qui nous intéresse ici. Vous devez en principe, dans votre code, suivre les mêmes contraintes que pour une transaction. Pour ne pas rencontrer trop de problèmes je vous conseille d'écrire des opérations par lot assez petites.

Pour respecter ces règles vous devez vous assurer par le code que :

- La procédure ne démarre pas si un des enregistrements cible à un verrou
- Si une erreur ne peut pas être résolue pendant le traitement, tout doit être annulé
- Si une erreur provient d'une violation d'intégrité, tout doit être annulé
- Entre le démarrage et la fin de la procédure, les enregistrements concernés doivent être verrouillés.

Ces règles ne sont pas inviolables, l'essentiel est de bien vérifier que les modifications ne laissent pas la base dans un état instable.

Le piège "l'exemple Jet"

Nous avons vu que lorsque vous paramétrez votre Recordset vous émettez un souhaît de curseur. Rien ne vous dit que ce curseur existe ou est géré par le fournisseur, et rien ne vous le dira. En effet, le fournisseur cherche à vous fournir le curseur demandé, s'il ne l'a pas, il vous donnera un curseur s'approchant le plus, mais ne vous enverra jamais de messages d'erreurs. C'est là que le nid à bugs est dissimulé.

Selon les fournisseurs, la différence entre ce que vous désirez et ce que vous aurez peut être énorme. Cela explique la plupart des anomalies que vous constatez lors de l'exécution de votre code.

Nous allons voir tous les problèmes que cela peut poser en étudiant le cas des bases Access avec Microsoft Jet 4.0.

- Il n'est pas possible d'obtenir un curseur dynamique[2] avec JET.
- Les curseurs côté serveur sont soit :
  - En lecture seule avec tout type de curseur (sauf dynamique évidemment)
  - De type KeySet avec tout type de verrouillage
- Les curseurs côté client sont :
  - Toujours statiques et ne peuvent avoir un verrouillage pessimiste

Comme vous le voyez, voilà qui réduit fortement le choix de curseurs disponibles. Quelles que soient les options choisies, vous aurez un des curseurs donnés ci-dessus.

Vous aurez par contre toujours un curseur du côté choisi. Evidemment, le problème est équivalent lorsque vous utilisez un contrôle de données.

[2] Microsoft affirme que la déclaration d'un curseur dynamique (adDynamic) côté serveur renvoie un curseur KeySet qui aurait de meilleures performances sur les gros Recordset qu'un curseur déclaré KeySet. Loin de moi l'idée de mettre en doute une telle affirmation, mais pour ma part je n'ai jamais vu de différence notable.

Ce petit tableau vous donnera le résumé de ce qui est disponible.

CursorLocation	CursorType	LockType
adUserServer	AdOpenForwardOnly AdOpenKeyset adOpenStatic	AdLockReadOnly
	AdOpenKeyset	AdLockReadOnly AdLockPessimistic AdLockOptimistic AdLockBatchOptimistic
adUseClient	adOpenStatic	AdLockReadOnly AdLockOptimistic AdLockBatchOptimistic

Recordset Vs SQL

Beaucoup de développeurs expérimentés préfèrent utiliser des requêtes actions et du code SQL dès qu'il s'agit de modifier des données dans la base (j'entends par modifier, l'ajout, la suppression et la mise à jour), plutôt que d'utiliser les méthodes correspondantes de l'objet Recordset.

Nous avons vu pourquoi cette technique est fortement recommandable si vous maîtrisez un tant soi peu le SQL, car elle présente le net avantage de savoir exactement ce qu'on demande au SGBD. Néanmoins si vous ne connaissez pas bien le SQL, cela peut représenter plus d'inconvénients que d'avantages.

Attention, passer par des requêtes action n'est pas sans risque s'il y a plusieurs utilisateurs car on peut obtenir des violations de verrouillage.

Optimisation du code

C'est une question qui revient souvent sur les forums. Avant de chercher une quelconque optimisation de votre code ADO, gardez bien à l'esprit que la première chose qui doit être optimisée est votre source de données. Les index manquants, les clés mal construites et surtout une structure anarchique de vos tables engendreront une perte de temps qu'aucune optimisation ne sache compenser. Les axes de l'optimisation sont de deux sortes.

L'optimisation dans l'accès aux données

Celle-ci est propre à la programmation des bases de données.

Utiliser des requêtes compilées

Si vous devez exécuter plusieurs fois une requête. Celle-ci est soit une requête stockée dans la base, soit obtenue en utilisant la propriété "Prepared" de l'objet Command

Utiliser les procédures stockées

Si votre SGBD les supporte, utilisez toujours les procédures stockées plutôt qu'un code ADO équivalent. En effet, les procédures stockées permettent un accès rapide et sécurisé au données, bien au-delà de ce que pourra faire votre code.

Etre restrictif dans les enregistrements / champs renvoyés

En effet, on a trop souvent tendance pour simplifier l'écriture de la requête à rapatrier tous les enregistrements et toutes les colonnes. La plupart du temps, seul certains champs sont utilisés, donc méfiez-vous de l'utilisation du caractère "*". De même, filtrez à l'aide de la clause WHERE quand vous le pouvez.

Optimisation dans l'utilisation d'ADO

Celle-ci est orientée ADO. Attention toutefois, ces effets peuvent être plus ou moins visibles selon le SGBD attaqué.

Utiliser des objets explicites

Méfiez-vous des connexions et commandes implicites, car on a facilement tendance à les oublier ce qui utilise des ressources pour rien.

Fermer les connexions

Ne laissez pas vos connexions ouvertes quand vous n'en avez pas besoin. Cela occupe inutilement le serveur et certains SGBD fermeront une connexion inutilisée un certain temps sans vous envoyer de message d'erreur.

Utiliser une ou deux connexions / commandes

Pensez à réutiliser vos objets au maximum. En général, un objet connection et un objet Command suffisent.

Spécifier adExecuteNoRecords

Précisez toujours ce paramètre lorsqu'une requête ne renvoie pas d'enregistrements ou lorsque vous ne souhaitez pas les récupérer.

Spécifier la propriété CommandType

Si vous ne le précisez pas, ADO devra interpréter CommandText dans un processus assez lourd.

L'objet Recordset

Avant de nous lancer dans des exemples concrets, nous allons voir quelques propriétés, méthodes et événements intéressants de l'objet Recordset. Je vais faire assez bref, et nous verrons les concepts intéressants plus en détail dans la troisième partie de cet article.

Propriétés

Je ne vais pas reprendre celles qui définissent le curseur.

AbsolutePosition

Renvoie ou définit la position de l'enregistrement courant dans le Recordset. Toujours disponible avec les curseurs non en avant seulement.

Bookmark

Permet d'utiliser des signets pour marquer les enregistrements. Toujours disponible avec les curseurs non en avant seulement.

CacheSize

Renvoie ou définit le nombre d'enregistrements qui vont être stockés dans le cache.

EditMode

Indique le statut de modification de l'enregistrement en cours.

Peut prendre les valeurs **adEditNone**, **adEditInProgress**, **adEditAdd**, **adEditDelete**

Très utile pour savoir si des actions sont en cours sur le recordset, ce qui permet le cas échéant d'annuler ou de valider ces modifications.

Filter

Permet de filtrer les enregistrements d'un recordset selon les critères spécifiés. Le Recordset filtré se comporte comme un nouvel objet recordset. En général cette propriété ne s'utilise pas sur les recordsets côté serveur car elle tend à faire s'effondrer les performances. Par contre on l'utilise beaucoup lors des traitements par lot des curseurs clients. Le filtre peut être de deux sortes :

Filtre SQL : il s'utilise comme une clause WHERE sans le WHERE.

Une des valeurs prédéfinies

Constante	Valeur	Description
AdFilterAffectedRecords	2	Filtre les enregistrements affectés par le dernier appel d'une méthode Delete, Resync, UpdateBatch ou CancelBatch.
AdFilterConflictingRecords	5	Filtre les enregistrements n'ayant pas pu être modifiés lors de la dernière mise à jour
AdFilterFetchedRecords	3	Filtre les enregistrements présents dans le cache
adFilterNone	0	Enlève le Filtre ce qui rétabli le recordset d'origine
AdFilterPendingRecords	1	Filtre les enregistrements en attente de mise à jour. Applicable uniquement pour les traitements par lot.

Index

Définit l'index en cours dans le recordset. Uniquement utilisé avec la méthode Seek.

MarshalOptions

Définit le mode de mise à jour lors de la reconnexion d'un recordset déconnecté. Peut prendre les valeurs

AdMarshalAll, adMarshalModifiedOnly.

Clients

MaxRecords

Permet de définir un nombre maximum d'enregistrements que la requête peut renvoyer.

RecordCount

Renvoie le nombre d'enregistrements disponibles dans le recordset. Toujours disponible avec les curseurs qui ne sont pas de type "en avant seulement".

Sort

S'utilise avec une chaîne contenant le(s) nom(s) de(s) champ(s) suivi de ASC ou DESC, chaque champ étant séparé par une virgule. Préférez toujours un tri par le SQL qui est beaucoup plus performant sur les recordset côté serveur.

Source

Renvoie la source des données d'un recordset. Lorsqu'il s'agit d'un objet Command, renvoie la propriété CommandText de celui ci.

State

Renvoie toujours l'état du recordset (fermé / ouvert). Dans le cas d'une connexion asynchrone peut prendre les valeurs **adStateConnecting**, **adStateExecuting**, **adStateFetching.** Presque uniquement utilisé dans les cas asynchrones

Status

Cette propriété est très importante, nous la reverrons d'ailleurs plus loin dans cet article. Elle indique l'état de l'enregistrement en cours lors d'une opération globale (ex : mise à jour par lot). Peut prendre une ou plusieurs des valeurs suivantes :

Constante	Valeur	Description
adRecOK	0	Mise à jour de l'enregistrement réussie
adRecNew	1	L'enregistrement est nouveau
adRecModified	2	L'enregistrement a été modifié
adRecDeleted	4	L'enregistrement a été supprimé
adRecUnmodified	8	L'enregistrement n'a pas été modifié
adRecInvalid	16	L'enregistrement n'a pas été sauvegardé parce que son signet n'est pas valide
adRecMultipleChanges	64	L'enregistrement n'a pas été sauvegardé parce que cela aurait affecté plusieurs enregistrements
adRecPendingChanges	128	L'enregistrement n'a pas été sauvegardé parce qu'il renvoie à une insertion en attente
adRecCanceled	256	L'enregistrement n'a pas été sauvegardé parce que l'opération a été annulée
adRecCantRelease	1024	Le nouvel enregistrement n'a pas été sauvegardé à cause du verrouillage d'enregistrements existants
adRecConcurrencyViolation	2048	L'enregistrement n'a pas été sauvegardé à cause d'un accès concurrentiel optimiste
adRecIntegrityViolation	4096	L'enregistrement n'a pas été sauvegardé parce que l'utilisateur n'a pas respecté les contraintes d'intégrité
adRecMaxChangesExceeded	8192	L'enregistrement n'a pas été sauvegardé parce qu'il y avait trop de modifications en attente

adRecObjectOpen	16384	L'enregistrement n'a pas été sauvegardé à cause d'un conflit avec un objet de stockage ouvert
adRecOutOfMemory	32768	L'enregistrement n'a pas été sauvegardé parce que la mémoire de l'ordinateur est insuffisante
adRecPermissionDenied	65536	L'enregistrement n'a pas été sauvegardé parce que l'utilisateur n'a pas le niveau d'autorisation suffisant
adRecSchemaViolation	131072	L'enregistrement n'a pas été sauvegardé parce qu'il ne respecte pas la structure de la base de données sous-jacente
adRecDBDeleted	262144	L'enregistrement a déjà été supprimé de la source de données

Dans le cas d'un traitement par lot il est indispensable d'aller lire cette propriété pour les enregistrements n'ayant pu être mis à jour.

Propriétés dynamiques

On utilise ces propriétés en passant par la collection Properties de l'objet recordset. Elles dépendent principalement du fournisseur. Il en existe beaucoup ; je vous en cite quelques-unes que nous utiliserons dans des exemples. Les propriétés dynamiques se valorisent en générales après la définition du fournisseur pour l'objet, dans certains cas, après l'ouverture de l'objet.

IrowsetIdentity

Permet d'implémenter l'interface Rowset OLEDB sur les curseurs serveurs. Certains contrôles ont besoin de cette interface pour afficher les données. Attend une valeur booléenne.

Exemple d'utilisation

Serveurs

Optimize

Permet de créer un index sur un champ (objet Field) d'un recordset **côté client**. Cet index n'existe que dans le jeu d'enregistrement, mais n'est pas créé dans la base de données sous-jacente. L'ajout d'un index peut permettre un accroissement important des performances sur l'utilisation des recherches ou des filtres. Dans l'exemple suivant j'ajoute un index au champ "Author" avant d'exécuter le filtre :

```
Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=d:\biblio.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
With MonRs
    .CursorLocation = adUseClient
    .ActiveConnection = Cnn1
    .Open "SELECT * FROM Authors", , adOpenStatic, adLockOptimistic,
adCmdText
    .Fields("Author").Properties("Optimize") = True
    .Filter = "[ Author] like '*a'"
End With
```

Clients

Resync Command

Permet de synchroniser un recordset créé à partir d'une jointure, lors de la modification ou de l'ajout de la clé étrangère d'une des tables. S'utilise toujours en conjonction avec la propriété dynamique "Unique Table". Attend une chaîne SQL étant la commande de synchronisation.

Evidemment; on peu se passer de cette propriété en ré exécutant les requêtes plutôt que de les synchroniser. Il ne faut toutefois pas perdre de vue que le coût d'exécution n'est pas le même.

Comme vous devez spécifier la table qui subira les modifications, vous devez aussi préciser la clé primaire de celle-ci à l'aide du marqueur (?).

Reprenons l'exemple ébauché dans la discussion sur les curseurs, nous allons donc utiliser une requête similaire à :

```
SELECT * FROM Publishers, Titles WHERE Publishers.PubID = Titles.PubID
```

Sous la forme simplifiée à des fins de lisibilité :

```
SELECT Publishers.PubID, Publishers.Name, Titles.Title, Titles.ISBN, Titles.PubID
FROM Publishers , Titles
WHERE Publishers.PubID = Titles.PubID
```

Le but étant toujours de corriger le titre en changeant l'éditeur.

Nous allons utiliser le code suivant :

```
Dim Cnn1 As ADODB.Connection, MonRs As ADODB.Recordset

Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=d:\biblio.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
With MonRs
    .CursorLocation = adUseClient
    .ActiveConnection = Cnn1
    .Open "SELECT Publishers.PubID, Publishers.Name, Titles.Title,
Titles.ISBN, Titles.PubID FROM Publishers, Titles WHERE
Publishers.PubID = Titles.PubID", , adOpenStatic, adLockOptimistic,
adCmdText
    .Properties("Unique Table") = "Titles"
    'car les modifications réelles des données ne porteront que sur
cette table
    .Properties("Resync Command") = "SELECT Publishers.PubID,
Publishers.Name, Titles.Title, Titles.ISBN, Titles.PubID FROM
Publishers, Titles WHERE Publishers.PubID = Titles.PubID AND
Titles.ISBN=?"
    ' où Titles.ISBN=? précise la clé primaire de la table déclarée
dans unique table
    .Find "Title='Evaluating Practice'"
    .Fields("Titles.PubId") = 192 'Numéro de l'éditeur Alpha Books
    .Fields("Title") = .Fields("Title") & "s"
    .Update
    .Resync adAffectCurrent
    Debug.Print .Fields("Name").Value
End With
```

Nous voyons alors que la valeur apparaissant dans la fenêtre exécution est la bonne et que la modification désirée a bien eu lieu dans la table.

Clients

Unique Table, Unique Catalog & Unique Schema

Permet de désigner une table qui sera la cible unique des modifications d'un recordset issu d'une requête avec jointure. La clé primaire de cette table devient alors la clé primaire de tout le recordset.

Clients.

Update Criteria

Définit comment le moteur de curseur va construire l'identification de l'enregistrement cible d'une modification, c'est à dire les valeurs qui apparaîtront dans le WHERE de la requête UPDATE. Prend une des valeurs suivantes :

Constante	Valeur	Description
AdCriteriaKey	0	Utilise uniquement les valeurs des champs clés.
AdCriteriaAllCols	1	Utilise les valeurs de tous les champs. (identique à un verrou optimiste)
AdCriteriaUpdCols	2	Utilise les valeurs des champs modifiés et des champs clés (défaut)
AdCriteriaTimeStamp	3	Utilise le champ TimeStamp au lieu des champs clé.

Attention à l'utilisation de adCriteriaKey. Ceci implique que l'enregistrement sera identifié correctement, et que la valeur sera mise à jour même si elle a été modifiée par quelqu'un d'autre. N'oubliez pas que les valeurs utilisées par le moteur de curseur sont les valeurs stockées dans la propriété OriginalValue des objets Fields concernés.

Clients

Update Resync

Définit comment le recordset est synchronisé après un appel de la méthode Update ou UpdateBatch. N'oubliez pas que seuls les enregistrements modifiés ou ajoutés sont concernés par cette synchronisation. Prend une ou plusieurs des valeurs suivantes :

Constante	Valeur	Description
AdResyncNone	0	Pas de synchronisation
AdResyncAutoIncrement	1	Tente de renvoyer les valeurs générées automatiquement par le SGBD
AdResyncConflicts	2	Synchronise tous les enregistrements n'ayant pas pu être mis à jour lors du dernier appel de la méthode Update.
AdResyncUpdates	4	Synchronise tous les enregistrements mis à jour
AdResyncInserts	8	Synchronise tous les enregistrements ajoutés
AdResyncAll	15	Agit comme toutes les synchronisations précédentes.

Attention, le fait de paramétrer la propriété avec adResyncInserts ne suffira pas pour récupérer la valeur des champs à numérotation automatique. Dans ce cas vous devrez utiliser :

```
MonRs.Properties("Update Resync") = adResyncAutoIncrement +
adResyncInserts
```

Clients

Collection Fields

Chaque enregistrement d'un recordset est composé d'une collection Fields représentant les champs de l'enregistrement. Chaque objet Field possède un certain nombre de propriétés (type, taille, précision, etc...) qui sont propres au champ. Ils possèdent aussi tous une collection de propriétés dynamiques. Enfin ils possèdent aussi trois propriétés de "valeurs".

OriginalValue est la valeur du champ lors de la création du recordset. A chaque mise à jour réussie de l'enregistrement, cette propriété se synchronise avec la base. Nous verrons les problèmes que cela peut poser lors des traitements par lot.

UnderlyingValue est la valeur enregistrée dans la base du point de vue de l'utilisateur. C'est la propriété la plus difficile à appréhender. Si le curseur reflète les modifications de la base, cette propriété suit les modifications, sinon elle doit être synchronisée pour être juste.

Value est la valeur en cours du champ **dans** le recordset (pas nécessairement dans la base).

Nous verrons dans les exemples les utilisations de ces propriétés.

Méthodes

AddNew

Ajoute un nouvel enregistrement au recordset. De la forme

recordset.**AddNew** ***FieldList, Values***

Ou FieldList et Values sont des paramètres facultatifs permettant d'entrer directement des valeurs dans le nouvel enregistrement.

Dès l'appel de AddNew, l'enregistrement créé devient l'enregistrement en cours.

Attention, l'utilisation des paramètres lance un Update implicite. Par exemple :

```
With MonRs
    .AddNew
    .Fields("Author") = "RABILLOUD, Jean-Marc"
    .Fields("year born") = 1967
    .Update
End With
```

Est équivalent à :

```
MonRs.AddNew Array("Author", "year born"), Array("RABILLOUD, Jean-
Marc", 1967)
```

CancelBatch

Annule les modifications en attente dans un traitement par lot. De la forme :

recordset.**CancelBatch** ***AffectRecords***

Où AffectRecords détermine si la méthode porte sur un ou plusieurs enregistrements. Il peut prendre les valeurs adAffectAll, adAffectGroup ou adAffectCurrent.

Déclenche en fait un appel de la méthode CancelUpdate sur tous les enregistrements modifiés, ajoutés ou supprimés du recordset. Attention, comme toutes les opérations par lot, la méthode peut définir des erreurs dans la collection Errors de l'objet connection, sans interrompre le traitement.

CancelUpdate

Annule les modifications apportées sur l'enregistrement en cours, ou annule l'enregistrement crée par AddNew.

Clone

Duplique un objet Recordset. Les clones sont synchronisés avec l'original jusqu'à l'application de la méthode Requery sur celui-ci.

Delete

Supprime un ou plusieurs enregistrements. De la forme :

recordset.**Delete** ***AffectRecords***

Où AffectRecords détermine si la méthode porte sur l'enregistrement en cours ou sur tous les enregistrements qui correspondent à la propriété Filter.

Find

Recherche les enregistrements répondant aux critères spécifiés. Cette méthode ne permet une recherche que sur un seul champ. De la forme :

recordset.**Find** ***(criteria, SkipRows, searchDirection, start)***

Où *SkipRows* est une valeur facultative qui donne le décalage par rapport à la ligne en cours ou au paramètre Start, quand il est défini.

SearchDirection est une valeur facultative qui peut prendre les valeurs **adSearchForward** ou **adSearchBackward**

Start donne la position de l'enregistrement de démarrage de la recherche.

Criteria est l'expression du critère de recherche. Il se compose toujours du nom du champ suivi de l'opérateur de comparaison suivi de la valeur.

Le nom du champ doit être entre crochets s'il contient un espace

L'opérateur doit être (">" (supérieur à), "<" (inférieur à), "=" (égal) ">=" (supérieur à ou égal), "<=" (inférieur à ou égal), "<>" (différent de) ou "**Like**" (concordance avec un modèle).)

La valeur doit être entourée de simple quote ' s'il s'agit d'un texte, de dièse # si c'est une date ou de rien pour une valeur numérique.

Si aucun enregistrement correspondant n'est trouvé, le recordset se positionnera sur EOF ou BOF, selon le sens de la recherche.

N.B : Cette méthode est très coûteuse en ressources, essayez toujours de privilégier une méthode SQL sur les curseurs serveur.

GetRows

Cette méthode permet de convertir un Recordset en tableau à deux dimensions (champ, enregistrement) de la forme :

Tableau = recordset.**GetRows(*Rows*, *Start*, *Fields*)**

Où *Rows* représente le nombre d'enregistrements à convertir, si la valeur est –1 tous les enregistrements seront convertis à partir de la position donnée par le paramètre "Start", s'il est omis tous les enregistrements du Recordset seront convertis.

Start donne la position du premier enregistrement à convertir. Peut prendre les valeurs **adBookmarkCurrent, adBookmarkFirst, adBookmarkLast**

Fields est la liste des champs devant être converti.

Après l'appel de la méthode, le recordset se positionne sur le premier enregistrement non converti ou sur EOF.

GetString

Renvoie le recordset sous forme d'une chaîne. De la forme :

Set *Variant = recordset.***GetString*(StringFormat, NumRows, ColumnDelimiter, RowDelimiter, NullExpr)***

Où StringFormat est toujours **AdClipString**

Les lignes sont délimitées par ***RowDelimiter***, les colonnes par ***ColumnDelimiter*** et les valeurs NULL par ***NullExpr***. Ces trois paramètres sont valides uniquement avec **adClipString.**

MoveFirst, MoveLast, MoveNext, MovePrevious

Ces méthodes permettent de se déplacer dans le jeu d'enregistrements. A la différence des mêmes méthodes DAO, le déplacement de l'enregistrement en cours valide l'ensemble des modifications effectuées.

Open

Ouvre le Recordset. Nous verrons plus loin qu'il y a plusieurs méthodes pour obtenir un jeu d'enregistrements. De la forme :

recordset.**Open *Source*, *ActiveConnection*, *CursorType*, *LockType*, *Options***

CursorType et LockType sont les propriétés du même nom, ActiveConnection est un objet Connection (explicite) ou une chaîne de connexion (Implicite).

Options peut prendre une ou plusieurs valeurs (masque binaire) de CommandTypeEnum et de ExecuteOptionEnum qui sont des paramètres de la méthode Execute de l'objet Command.

CommandTypeEnum		
Constantes	Valeur	Description
AdCmdUnspecified	-1	Pas de spécification de type
AdCmdText	1	CommandText correspond à la définition textuelle d'une commande ou d'un appel de procédure stockée.
AdCmdTable	2	CommandText correspond au nom de table dont les colonnes sont toutes renvoyées par une requête SQL générée en interne.
AdCmdStoredProc	4	CommandText correspond au nom d'une procédure stockée.
AdCmdUnknown	8	Valeur utilisée par défaut. Le type de commande de la propriété CommandText est inconnu.
AdCmdFile	256	CommandText correspond au nom de fichier d'un Recordset permanent.
adCmdTableDirect	512	CommandText correspond à un nom de table dont les colonnes sont toutes renvoyées.

Ce paramètre doit être passé si le texte de la commande n'est pas une commande SQL, sous peine de ralentir le traitement.

ExecuteOptionEnum		
Constantes	**Valeur**	**Description**
adAsyncExecute	16	Indique que la commande doit s'exécuter en mode asynchrone
adAsyncFetch	32	Indique que les lignes restant après la quantité initiale spécifiée dans la propriété "Initial Fetch Size" doivent être récupérées en mode asynchrone. Si une ligne nécessaire n'a pas été récupérée, le chemin principal est bloqué jusqu'à ce que la ligne demandée devienne disponible.
adAsyncFetchNonBlocking	64	Indique que le chemin principal n'est jamais bloqué pendant la recherche. Si la ligne demandée n'a pas été recherchée, la ligne en cours se place automatiquement en fin de fichier. Cela n'a pas d'effet si le recordset est ouvert en mode adCmdTableDirect

Ce paramètre doit être passé si l'on travaille en mode Asynchrone.

Requery

Recrée le jeu d'enregistrements en ré-exécutant la requête. De la forme :

recordset.**Requery** ***Options***

Où options est le même paramètre que dans le cas de la méthode Open.

Comme la requête est exécutée à nouveau, il peut y avoir des enregistrements en plus ou en moins ou provoquant des conflits du fait des actions d'autres utilisateurs. L'appel de la méthode déclenche une erreur si l'enregistrement en cours est en mode édition.

Attention, il n'est pas possible de modifier les paramètres du curseur dans l'appel de Requery. Pour modifier le curseur, vous devez utiliser Close puis Open.

Resync

Met à jour le jeu d'enregistrements. Utilisée seulement sur des curseurs statiques ou en avant seulement ; si le curseur est côté client, il ne doit pas être en lecture seule De la forme

recordset.**Resync** ***AffectRecords, ResyncValues***

Où AffectRecords spécifie les enregistrements à mettre à jour et ResyncValues précise si les modifications sont écrasées. Si la valeur est **adResyncAllValues, toutes les modifications sont**

écrasées, on repart avec un recordset d'origine, si la valeur est **adResyncUnderlyingValues** les modifications en cours sont préservées.

Voyons comment cela fonctionne

Un objet Recordset peut être vu comme une collection d'enregistrements, chacun étant composé d'objet Field (champ). Lors de la création du recordset, chaque champ possède trois propriétés, OriginalValue, Value et UnderlyingValue qui représente la valeur du champ. Lorsque l'on fait des modifications on modifie la propriété Value, mais pas UnderlyingValue. Par comparaison des deux, le Recordset peut toujours "marquer" les enregistrements modifiés. Si on procède à une synchronisation des valeurs de la base, les propriétés UnderlyingValue reflètent les vraies valeurs de la base, mais les propriétés Valeurs peuvent ne pas être modifiées. Le fait de synchroniser le recordset peut donc permettre de voir par avance si des valeurs ont été modifiées par un autre utilisateur, en comparant les propriétés UnderlyingValue et OriginalValue avant de déclencher une mise à jour.

Save

Sauvegarde le Recordset comme un fichier. Un tel Recordset est appelé jeu d'enregistrement permanent. De la forme *recordset.***Save *FileName, PersistFormat***

Où PersistFormat, précise si le recordset est sauvegardé comme un fichier XML ou selon un format spécifique. Nous verrons dans les exemples comment on peut utiliser ces recordsets persistants, mais sachez dès à présent qu'il n'est pas nécessaire d'avoir le fournisseur qui a créé le recordset pour pouvoir l'ouvrir.

Seek

Permet de faire une recherche multi champs sur les recordsets indexés. Ne s'utilise que sur les curseurs serveurs. De la forme :

recordset.**Seek *KeyValues, SeekOption***

Où KeyValues est une suite de valeurs de type "Variant". Un index consiste en une ou plusieurs colonnes où le tableau contient autant de valeurs qu'il y a de colonne, permettant ainsi la comparaison avec chaque colonne correspondante.

SeekOption peut prendre les valeurs suivantes :

Constante	Valeur	Description
adSeekFirstEQ	1	Recherche la première clef égale à KeyValues
adSeekLastEQ	2	Recherche la dernière clef égale à KeyValues.
adSeekAfterEQ	4	Recherche soit une clef égale à KeyValues ou juste après l'emplacement où la correspondance serait intervenue.
adSeekAfter	8	Recherche une clef juste après l'emplacement où une correspondance avec KeyValues est intervenue.
adSeekBeforeEQ	16	Recherche soit une clef égale à KeyValues où juste avant l'emplacement où la correspondance serait intervenue.
adSeekBefore	32	Recherche une clef juste avant l'emplacement où une correspondance avec KeyValues est intervenue.

Pour utiliser la méthode Seek, il faut que le recordset supporte les indexes et que la commande soit ouverte en adCmdTableDirect.

Comme vous le verrez dans l'exemple le principe est simple, on ajoute un ou des indexes sur le(s) champ(s) cible(s) de la recherche, et on passe le tableau des valeurs cherchées à la méthode Seek.

Serveurs

Supports

Cette méthode est la meilleure amie du développeur ADO. Elle permet de savoir si un recordset donné va accepter une fonctionnalité spécifique. De la forme :

boolean = recordset.**Supports(*CursorOptions*)**

Où *CursorOptions* peut prendre les valeurs suivantes :

Constante	Valeur	Description
adAddNew	16778240	Vous pouvez utiliser la méthode AddNew pour ajouter de nouveaux enregistrements.
adApproxPosition	16384	Vous pouvez lire et définir les propriétés AbsolutePosition et AbsolutePage.
adBookmark	8192	Vous pouvez utiliser la propriété Bookmark pour accéder à des enregistrements spécifiques.
adDelete	16779264	Vous pouvez utiliser la méthode Delete pour supprimer des enregistrements.
adFind	524288	Vous pouvez utiliser la méthode Find pour localiser une ligne dans un Recordset.
adHoldRecords	256	Vous pouvez extraire plus d'enregistrements ou modifier la position d'extraction suivante sans valider toutes les modifications en attente.
adIndex	8388608	Vous pouvez utiliser la propriété Index pour donner un nom à un index.
adMovePrevious	512	Vous pouvez utiliser les méthodes MoveFirst et MovePrevious, et les méthodes Move ou GetRows pour faire reculer la position de l'enregistrement en cours sans recourir à des signets.
adNotify	262144	Indique que le fournisseur gère les événements du recordset
adResync	131072	Vous pouvez mettre à jour le curseur avec les données visibles dans la base de données sous-jacente, au moyen de la méthode Resync.
adSeek	4194304	Vous pouvez utiliser la méthode Seek pour localiser une ligne dans un Recordset.
adUpdate	16809984	Vous pouvez utiliser la méthode Update pour modifier les données existantes
adUpdateBatch	65536	Vous pouvez utiliser la mise à jour par lots (méthodes UpdateBatch et CancelBatch) pour transmettre des groupes de modifications au fournisseur.

On utilise dans le code la méthode Supports pour savoir comment va réagir le recordset et éviter de faire de la gestion d'erreurs a posteriori.

Update

Sauvegarde les modifications apportées à l'enregistrement en cours. De la forme :

recordset.**Update *Fields, Values***

Où Fields et Values sont la liste des champs et des valeurs à modifier (facultatif).

UpdateBatch

Met à jour toutes les modifications en attente (opération par lot). De la forme

recordset.**UpdateBatch *AffectRecords***

Où AffectRecords définit les enregistrements concernés.

Si toutes les modifications échouent, une erreur se produit, s'il n'y a que quelques échecs, ils sont ajoutés à la collection Errors de la connexion.

Evènements

Pour pouvoir utiliser les événements ADO, il faut déclarer la variable précédée du mot clé WithEvents.

Les événements de l'objet Recordset sont tous appariés sauf un. Ce qui veut dire qu'il se produit un événement avant l'action concernée et un autre après. Ces événements sont principalement utilisés en programmation asynchrone.

AdStatusEnum

Valeur que peut prendre le paramètre adStatus présent dans presque tous les événements.

Constante	Valeur	Description
adStatusOK	1	Indique que l'opération qui a déclenché l'événement s'est passée avec succès
adStatusErrorsOccurred	2	Indique que l'opération qui a déclenché l'événement n'a pas pu se terminer correctement suite à une erreur
adStatusCantDeny	3	Indique qu'il est impossible d'annuler l'opération en cours
adStatusCancel	4	Une demande d'annulation a été demandée sur l'opération en cours
adStatusUnwantedEvent	5	Empêche de nouvelles notifications pendant le traitement d'un événement

EndOfRecordset

Private Sub Recordset_EndOfRecordset(fMoreData As Boolean, adStatus As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)

Se produit lorsque le recordset arrive en position EOF.

Où *FMoreData* défini s'il reste des enregistrements après. Doit être valorisé à "True" si pendant le traitement de l'événement on ajoute des enregistrements

AdStatus est l'état du recordset. Vaut **adStatusOK** si l'opération s'est bien passée et **adStatusCantDeny** si l'opération qui a déclenché l'événement ne peut être annulée. Si vous devez modifier le recordset pendant la procédure, ce paramètre doit être mis à **adStatusUnwantedEvent** afin de supprimer les notifications non désirées.

Precordset est la référence à l'objet subissant l'événement.

FetchProgress & FetchComplete

Private Sub MonRs_FetchProgress(ByVal Progress As Long, ByVal MaxProgress As Long, adStatus As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)

Cet événement se déclenche périodiquement lors des opérations d'extractions asynchrones de longues durées. Les paramètres :

Progress représente le nombre de lignes extraites

MaxProgress le nombre de lignes qui doivent être extraites

Private Sub MonRs_FetchComplete(ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)

Se produit après la récupération du dernier enregistrement.

Où *PError* est un objet erreur ADO

WillChangeField & FieldChangeComplete

Private Sub MonRs_WillChangeField(ByVal cFields As Long, ByVal Fields As Variant, adStatus As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)

Private Sub MonRs_FieldChangeComplete(ByVal cFields As Long, ByVal Fields As Variant, ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)

Où *cFields* est le nombre de champs modifiés et *Fields* un tableau contenant ces champs.

La méthode **WillChangeField** est appelée *avant* qu'une opération en attente modifie la valeur d'un ou plusieurs objets Field dans le **Jeu d'enregistrements**. La méthode **FieldChangeComplete** est appelée *après* modification de la valeur d'un ou plusieurs objets **Field**.

Là il convient bien de ne pas se mélanger dans les événements. Ces événements se produisent lorsque la propriété value d'un ou plusieurs objets Field(s) est modifiée. Cela n'a rien à voir avec la transmission de valeur vers la base de données.

Pour annuler la modification il suffit de mettre le paramètre **adStatus** à la valeur **adStatusCancel** dans l'événement **WillChangeField**.

WillChangeRecord & RecordChangeComplete

```
Private Sub MonRs_WillChangeRecord(ByVal adReason As ADODB.EventReasonEnum,
  ByVal cRecords As Long, adStatus As ADODB.EventStatusEnum, ByVal pRecordset As
  ADODB.Recordset)

Private Sub MonRs_RecordChangeComplete(ByVal adReason As
  ADODB.EventReasonEnum, ByVal cRecords As Long, ByVal pError As ADODB.Error,
  adStatus As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)
```

La méthode **WillChangeRecord** est appelée *avant* qu'un ou plusieurs enregistrements dans le **Recordset** ne soient modifiés. La méthode **RecordChangeComplete** est appelée *après* qu'un ou plusieurs enregistrement sont modifiés.

adReason peut prendre les valeurs suivantes : **adRsnAddNew**, **adRsnDelete**, **adRsnUpdate**, **adRsnUndoUpdate**, **adRsnUndoAddNew**, **adRsnUndoDelete**, **adRsnFirstChange ou adRsnFirstChange.**

Ce paramètre indique l'action ayant provoqué le déclenchement de l'événement.

cRecords renvoie le nombre d'enregistrements modifiés.

Ces événements se produisent donc lorsqu'il y a modifications des recordsets ou mise à jour des données. Ce sont des événements qui peuvent modifier ou annuler des modifications dans la base de données.

WillChangeRecordset & RecordsetChangeComplete

```
Private Sub MonRs_WillChangeRecordset(ByVal adReason As
  ADODB.EventReasonEnum, adStatus As ADODB.EventStatusEnum, ByVal pRecordset
  As ADODB.Recordset)

Private Sub MonRs_RecordsetChangeComplete(ByVal adReason As
  ADODB.EventReasonEnum, ByVal pError As ADODB.Error, adStatus As
  ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)
```

La méthode **WillChangeRecordset** est appelée *avant* qu'une opération en attente modifie le **Recordset**. La méthode **RecordsetChangeComplete** est appelée *après* que le **Recordset** a été modifié.

adReason peut prendre les valeurs suivantes : **adRsnReQuery**, **adRsnReSynch**, **adRsnClose**, **adRsnOpen**

Ces événements se produisent lors de remises à jour du recordset, ce sont des événements de recordset, ils n'influent jamais directement sur la base de données.

WillMove & MoveComplete

```
Private Sub MonRs_WillMove(ByVal adReason As ADODB.EventReasonEnum, adStatus
As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)

Private Sub MonRs_MoveComplete(ByVal adReason As ADODB.EventReasonEnum,
ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, ByVal pRecordset As
ADODB.Recordset)
```

La méthode **WillMove** est appelée *avant que* l'opération en attente change la position actuelle dans le **Jeu d'enregistrements**. La méthode **MoveComplete** est appelée *après* modification de la position actuelle dans le **Recordset**.

adReason peut prendre les valeurs suivantes : **adRsnMoveFirst**, **adRsnMoveLast**, **adRsnMoveNext**, **adRsnMovePrevious**, **adRsnMove** ou **adRsnRequery**

Attention à l'utilisation de ces événements. De nombreuses méthodes provoquent une modification de l'enregistrement en cours ce qui déclenche systématiquement cet événement. De plus il est très facile d'écrire un événement en cascade.

Rappels ADO

L'objet Connection

Je ne vais pas me lancer ici dans une longue description de l'objet Connection. Je vais donc juste faire quelques rappels directement utilisés dans cet article.

La propriété CursorLocation

Définit ou renvoie la position de la bibliothèque de curseur à utiliser. Peut prendre les valeurs adUseClient ou adUseServer. La définition de cette propriété au niveau de la connexion permet de ne pas la redéclarer au niveau d'un recordset.

La propriété IsolationLevel

Cette propriété permet de définir le niveau d'isolation qu'auront les transactions faite par la connexion. Ceci permet de garantir le respect de la gestion des transactions. Elle peut prendre les valeurs :

Constante	Valeur	Description
adXactUnspecified	-1	Le fournisseur utilise un niveau d'isolation différent de celui qui est spécifié mais ce niveau ne peut pas être déterminé.
adXactChaos	16	Valeur utilisée par défaut. Vous ne pouvez pas écraser les changements en attente des transactions dont le niveau d'isolation est supérieur.
adXactBrowse	256	À partir d'une transaction, vous pouvez voir les modifications des autres transactions non encore engagées.
adXactReadUncommitted	256	Identique à adXactBrowse
adXactCursorStability	4096	Valeur utilisée par défaut. A partir d'une transaction, vous pouvez afficher les modifications d'autres transactions uniquement lorsqu'elles ont été engagées.
adXactReadCommitted	4096	Identique à adXactCursorStability.
adXactRepeatableRead	65536	À partir d'une transaction, vous ne pouvez pas voir les modifications effectuées dans d'autres transactions, mais cette nouvelle requête peut renvoyer de nouveaux jeux d'enregistrements.
adXactIsolated	1048576	Des transactions sont conduites en isolation d'autres transactions.
adXactSerializable	1048576	Identique à adXactIsolated.

Le niveau d'isolation à utiliser dépend de votre application. La valeur la plus stricte (sûre) est **adXactIsolated**

La propriété Mode

Cette propriété permet de gérer en partie l'accès concurrentiel dans les bases de données. Comme cette propriété porte sur la connexion, elle s'applique à la source de données. Elle peut prendre les valeurs :

Constante	Valeur	Description
adModeUnknown	0	Valeur utilisée par défaut. Indique que les autorisations n'ont pas encore été définies ou ne peuvent pas être déterminées.
adModeRead	1	Lecture seule
adModeWrite	2	Écriture seule
adModeReadWrite	3	Lecture et écriture
adModeShareDenyRead	4	Ouverture en lecture interdite pour les utilisateurs autorisés en

		lecture seule
adModeShareDenyWrite	8	Ouverture interdite pour les utilisateurs autorisés en écriture seule
adModeShareExclusive	12	Ouverture interdite aux autres utilisateurs
adModeShareDenyNone	16	Les autres utilisateurs peuvent ouvrir en lecture/écriture

Définir correctement cette propriété permet dans bien des cas de simplifier les éventuels conflits.

Collection Errors

Toutes les erreurs de fournisseur créent un objet Error qui vient s'ajouter à la collection Errors de l'objet Connection. Il faut bien garder à l'esprit que ce ne sont que les erreurs du **fournisseur,** les autres erreurs ADO se gèrent de façon standard. Cette collection Errors se remplie pour une opération, si une erreur survient lors d'une opération suivante, l'ancienne collection est remplacée par la nouvelle.

Dans certains cas, le fournisseur envoie des avertissements dans la collection Errors, il est souvent conseillé de vider la collection Errors à l'aide de la méthode Clear avant d'utiliser les méthodes Resync, CancelBatch et UpdateBatch sur l'objet Recordset.

Evènements de connexion

A l'ouverture de la connexion, on peut décider si celle-ci s'ouvre de façon asynchrone en valorisant le paramètre "Options" de la méthode Open à **adAsyncConnect.**

Les événements de l'objet Connection sont :

BeginTransComplete, CommitTransComplete, et RollbackTransComplete pour les transactions

WillConnect, ConnectComplete et Disconnect pour l'ouverture et la fermeture

InfoMessage pour la gestion des erreurs

WillExecute & ExecuteComplete

Ces événements se déclenchent avant et après l'exécution d'une commande, implicite ou non. En clair, ces événements se produisent sur l'appel de Connection.Execute, Command.Execute et Recordset.Open.

```
Private Sub Connection_WillExecute(Source As String, CursorType As ADODB.CursorTypeEnum, LockType As ADODB.LockTypeEnum, Options As Long, adStatus As ADODB.EventStatusEnum, ByVal pCommand As ADODB.Command, ByVal pRecordset As ADODB.Recordset, ByVal pConnection As ADODB.Connection)
```

```
Private Sub Connection_ExecuteComplete(ByVal RecordsAffected As Long, ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, ByVal pCommand As ADODB.Command, ByVal pRecordset As ADODB.Recordset, ByVal pConnection As ADODB.Connection)
```

pConnection doit toujours contenir une référence à une connexion valide.

Si l'événement est dû à Connection.Execute, pCommand et pRecordset sont à "Nothing".

Si l'événement est dû à Command.Execute, pCommand est la référence de l'objet Command et pRecordset vaut "Nothing".

Si l'événement est dû à Recordset.Open, pCommand vaut "Nothing" et pRecordset est la référence de l'objet Recordset.

L'objet Command

Pour ceux d'entre vous qui avez lu l'article "Utiliser le modèle ADOX avec Visual Basic" vous allez retrouver sensiblement le même chapitre, un peu plus étoffé et portant sur des axes différents.

Beaucoup de programmeurs n'utilisent pas l'objet Command avec ADO préférant travailler directement avec l'objet Recordset. Pourtant l'utilisation de cet objet peut être indispensable lorsqu'on souhaite travailler avec des requêtes paramétrées ou pour utiliser directement des requêtes action. Nous allons donc aborder l'objet Command de façon assez détaillée sous l'angle suivant :

- Utilisation de Command pour utiliser des requêtes paramétrées
- Utilisation de Command avec des requêtes action

Nous allons commencer par voir l'objet Command en détail.

Généralités

Un objet command représente une commande spécifique à exécuter sur une source de données. Ceci peut être une instruction SQL ou une procédure stockée. Cet objet peut être une requête/procédure stockée dans la base ou créée à l'exécution. Un objet Command dépend toujours d'une connection soit créée spécifiquement pour cette commande, soit d'une connection déjà existante.

Propriétés

ActiveConnection

Définit la connexion utilisé pour l'objet Command. Cette propriété doit être passée à Nothing avant de changer la connexion d'un objet Command. Comme les objets Command héritent de certaines propriétés de la connexion, faites attention au paramétrage de celle-ci.

CommandText

La propriété CommandText contient le texte de la commande à exécuter. Ce texte peut être le nom d'une requête stockée, une chaîne SQL etc…

CommandTimeout

La valeur est en secondes. S'applique en général sur l'objet Connection

CommandType

Donne le type de la commande. Cette propriété est très importante. En effet, il y aura une erreur récupérable si le paramètre donné est faux. De plus ne pas valoriser correctement cette propriété peut dégrader fortement les performances. Les valeurs peuvent être :

Constante	Valeur	Description
adCmdText	1	CommandText correspond à la définition textuelle d'une commande ou d'un appel de procédure stockée
adCmdTable	2	CommandText correspond au nom de table dont les colonnes sont toutes renvoyées par une requête SQL générée en interne.
adCmdTableDirect	512	CommandText correspond à un nom de table dont les colonnes sont toutes renvoyées.
adCmdStoredProc	4	CommandText correspond au nom d'une procédure stockée.
adCmdUnknown	8	Valeur utilisée par défaut. Le type de commande de la propriété CommandText est inconnu
adCmdFile	256	CommandText correspond au nom de fichier d'un Recordset permanent
adExecuteNoRecords	128	CommandText correspond à une commande ou une procédure stockée qui ne renvoie pas de ligne (par exemple, une commande qui insère uniquement des données). Si des lignes sont extraites, elles ne sont pas prises en compte et ne sont pas retournées. Toujours associée à adCmdText ou adCmdStoredProc

 Attention toutefois, une procédure stockée peut être différemment interprétée selon les SGBD. Voir aussi la rubrique "Parameters".

NamedParameters

Définit si le nom des paramètres sont passés au fournisseur, ce qui lui permet de ne pas avoir à les interpréter dans l'ordre de leur création.

Prepared

Détermine si une commande doit être Pré-compilée. N'est utile que si la commande doit être exécutée plusieurs fois. Attention, certains fournisseurs n'acceptent pas cette modification sans toutefois déclencher d'erreur.

State

Renvoie l'état de la commande (ouverte ou fermée)

Méthodes

Cancel

Annule l'exécution de la commande si celle-ci est asynchrone.

Execute

Execute la commande. De la forme :

command*.Execute *RecordsAffected*, *Parameters*, *Options

Où *RecordsAffected* renvoie le nombre d'enregistrements affectés par la commande. Ceci n'est vrai que pour une requête action ou pour l'appel d'une procédure d'action. Le paramètre n'est pas valorisé avec le nombre d'enregistrements extraits si la commande renvoie un recordset.

Parameters est un tableau contenant les paramètres (pour les requêtes/procédures paramétrées évidemment)

Option est identique à ce qui est décrit dans la méthode Open de l'objet Recordset.

Nous avons là plusieurs cas qui nous intéressent :

- Ø On utilise toujours Command.Execute ou Connection.Execute pour utiliser des requêtes action.
- Ø Pour récupérer un recordset de l'objet Command, on peut soit passer la commande dans la méthode Open du recordset, soit affecter les enregistrements renvoyés dans un recordset.

Regardons le code suivant :

```
Dim Cnn1 As ADODB.Connection, MonRs As ADODB.Recordset, MaCommand As
ADODB.Command

Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=d:\biblio.mdb ;User Id=Admin; Password="
Set MaCommand = New ADODB.Command
With MaCommand
    .ActiveConnection = Cnn1
    .CommandText = "SELECT * FROM Authors"
    Set MonRs = MaCommand.Execute
End With
MonRs.Close
Set MonRs = Nothing
Set MonRs = New ADODB.Recordset
MonRs.Open MaCommand, , adOpenKeyset, adLockOptimistic, adCmdText
```

J'utilise les deux méthodes que je vous ai données plus haut. Notez toutefois que dans le Command.Execute je n'ai pas défini le curseur. J'obtiens donc un curseur côté serveur en avant seulement et en lecture seule. Supposons que je paramètre mon Recordset au préalable :

```
Set MonRs = New ADODB.Recordset
With MonRs
    .CursorLocation = adUseClient
    .CursorType = adOpenStatic
    .LockType = adLockOptimistic
End With
Set MonRs = MaCommand.Execute
```

J'obtiendrais pourtant le même curseur qu'avec le code précédent. Cela vient du fait que lors de la création d'un recordset à l'aide de la méthode Execute, que ce soit sur l'objet Connection ou sur l'objet Command, le recordset hérite du curseur défini par la connexion. Selon la valeur de la propriété CursorLocation de l'objet Connection, vous obtiendrez soit un curseur en avant seulement en lecture seule (serveur) soit un curseur statique en lecture seule (Client). Le recordset sera toujours en lecture seule.

CreateParameter

Sert à créer un paramètre (voir explications plus loin).

De la forme *command*.**CreateParameter (*Name*, *Type*, *Direction*, *Size*, *Value*)**

Attention à ne pas faire une erreur courante avec cette méthode. Celle-ci crée un paramètre mais ne l'ajoute pas à la collection Parameters de l'objet Command. Ceci est d'ailleurs indispensable afin de pouvoir valoriser d'autres propriétés avant l'ajout.

Collection Parameters

Généralités

Il faut déjà faire attention, pour les utilisateurs d'autres SGBD qu'Access à bien faire la différence entre procédure stockée et requête paramétrée.

Quoique regroupés dans la même collection, il existe deux types de paramètres. Les paramètres d'entrée, attendus par la procédure/requête pour pouvoir s'exécuter, et les paramètres de sorties qui peuvent êtres renvoyés par une procédure. Il convient de faire attention avec ceux-ci, une connection n'acceptant jamais plus de deux objets Command ayant des paramètres de sorties.

Quelques méthodes de la collection.

Append

Ajoute un paramètre à la collection. De la forme

Command.Parameters .Append ***Name, Type, DefinedSize, Attrib***

Il est possible d'utiliser la méthode CreateParameter dans le Append. Vous devez avoir typé votre paramètre avant ou lors de l'ajout à la collection. Il est préférable de valoriser votre paramètre avant de l'ajouter à la collection afin de ne pas solliciter le fournisseur.

Delete

Enlève un paramètre de la collection. On doit spécifier soit le nom, soit l'index du paramètre à retirer.

Objet Parameter

Propriétés

Attributes

Précise si le paramètre accepte les valeurs signées, binaires ou NULL.

Direction

Définit si le paramètre est un paramètre d'entrée, de sortie ou de retour. Attention de ne pas confondre un paramètre de retour (adParamReturnValue) qui est l'entier éventuellement renvoyé par une procédure (dans le Return) d'un paramètre de sortie (adParamOutput) qui est le paramètre renvoyé par une procédure.

Name

Définit le nom du paramètre. N'est pas obligatoire.

NumericScale

Donne la précision (nombre de chiffres à droite de la virgule) d'un paramètre numérique.

Size

Donne la taille en octets d'un paramètre dont le type est potentiellement de longueur variable (par exemple String). Ce paramètre est obligatoire pour les types de longueur indéterminée (String, Variant).

Elle doit être toujours valorisée avant ou lors de l'ajout à la collection.

Type

Comme son nom l'indique !

Ne vous emmêlez pas les pinceaux entre les chaînes **adVarChar** et les chaînes Unicode **adVarWChar**

Value

De manière générale, valorisez cette propriété **après** l'ajout à la collection.

Méthode

AppendChunk

Permet d'accéder aux textes ou binaires longs.

Exemple

Requêtes paramétrées

Nous allons voir ici un exemple d'utilisation de requête paramétrée.
Prenons la requête stockée nommée ReqFerie définie comme :

```
PARAMETERS DateCib DateTime;
SELECT tblFerie.DateFer
FROM tblFerie
WHERE (((tblFerie.DateFer)>DateValue([DateCib])))
ORDER BY tblFerie.DateFer;
```

La fonction suivante crée un recordset en utilisant l'objet Command.

```
Private Sub Command5_Click()
Dim cnn1 As ADODB.Connection, Comm1 As ADODB.Command, Param1 As
Parameter, MonRecordset As ADODB.Recordset

Set cnn1 = New ADODB.Connection
With cnn1
    .Provider = "Microsoft.Jet.OLEDB.4.0;"
    .ConnectionTimeout = 30
    .CursorLocation = adUseClient
    .IsolationLevel = adXactChaos
    .Mode = adModeShareExclusive
    .Properties("Jet OLEDB:System database") =
"D:\User\jmarc\tutorial\ADOX\system.mdw"
    .Open "Data Source=D:\User\jmarc\tutorial\ADOX\baseheb.mdb ;User
Id=Admin; Password="
End With
Set Comm1 = New ADODB.Command
With Comm1
    .ActiveConnection = cnn1
    .CommandType = adCmdStoredProc
    .CommandText = "ReqFerie"
End With
Set Param1 = New Parameter
With Param1
    .Direction = adParamInput
    .Type = adDate
    .Name = "DateCib"
End With
Comm1.Parameters.Append Param1
Comm1("DateCib").Value = #1/1/2002#
Set MonRecordset = Comm1.Execute

End Sub
```

Requête action

On peut aussi utiliser l'objet Command pour exécuter des requêtes action sur la source de données. En général, comme celles-ci sont rarement utilisées plusieurs fois dans le même code, on utilise plus souvent l'objet Connection, ce qui permet de tout passer dans une ligne, tel que :

```
Cnn1.Execute "DELETE * FROM Authors WHERE [year Born]=1938", ,
adExecuteNoRecords
```

Néanmoins on peut aussi utiliser l'objet Command.

```
With Comm1
    .ActiveConnection = cnn1
    .CommandType = adCmdText
    .CommandText = "UPDATE Authors SET [Year Born]= [Year Born]+1"
```

```
End With
Comm1.Execute
```

On trouve par contre beaucoup plus souvent l'usage de requêtes stockées paramétrées avec l'objet command. Par exemple :

```
Dim cnn1 As ADODB.Connection, Comm1 As ADODB.Command, Param1 As
Parameter, Param2 As Parameter, MonRecordset As ADODB.Recordset

Set cnn1 = New ADODB.Connection
With cnn1
    .Provider = "Microsoft.Jet.OLEDB.4.0;"
    .ConnectionTimeout = 30
    .CursorLocation = adUseClient
    .Open "Data Source=D:\basehebdo.mdb ;User Id=Admin; Password="
End With
cnn1.BeginTrans
Set Comm1 = New ADODB.Command
With Comm1
    .ActiveConnection = cnn1
    .CommandType = adCmdText
    .CommandText = "PARAMETERS DateCib DateTime, BancCib Text;" &
"DELETE * From tblAnomalie WHERE tblAnomalie.NumBanc=[BancCib] AND
tblAnomalie.DatDimanche>=[DateCib];"
    .NamedParameters = True
End With
Set Param1 = New Parameter
With Param1
    .Direction = adParamInput
    .Type = adDate
    .Name = "DateCib"
End With
Set Param2 = New Parameter
With Param2
    .Direction = adParamInput
    .Type = adVarWChar
    .Size = 3
    .Name = "BancCib"
End With
Comm1.Parameters.Append Param1
Comm1.Parameters.Append Param2
Comm1("BancCib").Value = "S01"
Comm1("DateCib").Value = #1/1/2002#
Comm1.Execute
If MsgBox("valider la transaction", vbYesNo) = vbYes Then
    cnn1.CommitTrans
Else
    cnn1.RollbackTrans
End If
```

L'avantage de ce genre d'utilisation repose sur le fait qu'un nouvel appel d'Execute avec de nouvelles valeurs de paramètres est immédiatement accessible. Ainsi

```
Comm1.Execute , Array(#1/1/2002#, "S04"), adCmdText +
adExecuteNoRecords
```

est une réutilisation aisée de ma commande.

Exemple d'utilisation

Ouvrir un recordset

Dans l'exemple suivant, je vais ouvrir trois fois le même recordset avec trois méthodes différentes. Nous discuterons ensuite des différences.

```
Private Sub Form_Load()
Dim Cnn1 As ADODB.Connection, Cmd1 As ADODB.Command, MonRs As
ADODB.Recordset

Set Cnn1 = New ADODB.Connection
With Cnn1
    .Provider = "Microsoft.Jet.OLEDB.4.0;"
    .ConnectionTimeout = 30
    .Mode = adModeShareExclusive
    .Open "Data Source=D:\Biblio.mdb ;User Id=Admin; Password="
End With
'par Connection.Execute
Set MonRs = Cnn1.Execute("SELECT * From Authors", , adCmdText)
MonRs.Close
'par command.execute
Set Cmd1 = New ADODB.Command
With Cmd1
    .ActiveConnection = Cnn1
    .CommandType = adCmdText
    .CommandText = "SELECT * From Authors"
End With
Set MonRs = Cmd1.Execute
MonRs.Close
'par recordset.open
Set MonRs = New ADODB.Recordset
MonRs.Open "SELECT * From Authors", Cnn1, , , adCmdText

End Sub
```

Il n'y a pas de différence entre créer le recordset par la méthode Open ou par Command.Execute, dans les deux cas, l'objet Recordset à une référence à ActiveCommand et ActiveConnection. Par contre, le recordset créé par Connection.Execute n'a pas de référence à ActiveCommand. Cela vient du fait que la méthode Execute de l'objet Connection utilise une commande "volatile" qui ne crée pas d'objet Command.

Dans cet exemple je n'ai précisé aucun paramètre, j'ai donc trois fois un curseur serveur, en avant seulement et en lecture seule.

Nous allons voir maintenant l'ouverture d'un recordset configuré.

```
Private Sub Form_Load()
Dim Cnn1 As ADODB.Connection, Cmd1 As ADODB.Command, MonRs As
ADODB.Recordset

Set Cnn1 = New ADODB.Connection
With Cnn1
    .Provider = "Microsoft.Jet.OLEDB.4.0;"
    .ConnectionTimeout = 30
    .IsolationLevel = adXactIsolated
    .Mode = adModeReadWrite
    .Open "Data Source=D:\Biblio.mdb ;User Id=Admin; Password="
End With
'Set Cmd1 = New ADODB.Command
'With Cmd1
'    .CommandType = adCmdText
'    .CommandText = "SELECT * From Authors"
'End With
Set MonRs = New ADODB.Recordset
With MonRs
    .CursorLocation = adUseClient
    .LockType = adLockBatchOptimistic
    .CursorType = adOpenStatic
    .CacheSize = 30
    .Source = "SELECT * From Authors"
    Set .ActiveConnection = Cnn1
'    Set .ActiveCommand = Cmd1
    .Open , , , , adCmdText
End With

End Sub
```

Le code que je vous aie mis en commentaires déclenchera une erreur s'il est utilisé. En effet on ne peut pas affecter d'objet Command à la propriété ActiveCommand. Si je veux passer l'objet Command, j'utiliserai Recordset.Open avec le nom de l'objet Command comme source de mon Recordset.

En résumé donc, aucune méthode n'est équivalente. On utilise donc le choix selon les critères suivants :

Connection.Execute è Crée des requêtes volatiles (non réutilisables), en lecture seule

Command.Execute è Crée des jeux d'enregistrements toujours en lecture seule.

Recordset.Open è Permet l'accès aux curseurs disponibles. C'est la seule méthode à utiliser pour agir sur la source de données par l'intermédiaire d'un recordset.

Voyons enfin le cas de la création d'un jeu d'enregistrements par l'intermédiaire d'une requête paramétrée.

```
Dim Cnn1 As ADODB.Connection, MonRs As ADODB.Recordset, MaCommand As
ADODB.Command
Dim Param1 As ADODB.Parameter

Set Cnn1 = New ADODB.Connection
Cnn1.CursorLocation = adUseClient
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=d:\biblio.mdb ;User Id=Admin; Password="
Set MaCommand = New ADODB.Command
With MaCommand
    .ActiveConnection = Cnn1
    .CommandType = adCmdText
    .CommandText = "PARAMETERS Annee Long;SELECT Authors.Au_ID,
Authors.Author, Authors.[Year Born] FROM Authors WHERE [Year
Born]=Annee"
    Set Param1 = .CreateParameter("Annee", adInteger, adParamInput,
, 1941)
    .Parameters.Append Param1
End With
Set MonRs = New ADODB.Recordset
MonRs.Open MaCommand, , adOpenKeyset, adLockOptimistic
MonRs.ActiveCommand.Parameters(0).Value = 1940
MonRs.Requery
```

Comme nous le voyons, il n'est nécessaire de créer le paramètre qu'une fois. Après cela, on peut modifier la valeur par l'intermédiaire de la propriété ActiveCommand de l'objet Recordset. De même, nous voyons que je n'aie pas besoin de spécifier la propriété CursorLocation de mon Recordset puisqu'il va hériter de celle de la connexion.

Nombre d'enregistrement, position et signet

Dans cet exemple nous allons travailler sur la navigation sans les méthodes "Move" dans un Recordset. Pour cela nous allons utiliser deux propriétés du recordset : RecordCount et AbsolutePosition. Avant la version 2.6 d'ADO il fallait impérativement utiliser un curseur côté client pour utiliser certaines de ces propriétés, et cela reste encore le cas avec certains fournisseurs (Jet 3.51 par exemple).

Dans notre cas, la seule contrainte est d'obtenir un curseur bidirectionnel puisque les curseurs en avant seulement ne valorisent pas les propriétés RecordCount et AbsolutePosition.

Globalement la position dans un recordset est soit sur un enregistrement, soit en position BOF c'est-à-dire **avant** le premier enregistrement ou EOF c'est-à-dire **après** le dernier. La valeur de AbsolutePosition va de 1 à RecordCount.

Un signet (Bookmark) sert à identifier un enregistrement de manière à pouvoir le retrouver facilement. Ne confondez pas signet et position, ils ne sont pas identiques.

Dans cet exemple nous allons faire une ProgressBar qui indique la position de l'enregistrement courant en pourcentage. (équivalent au PercentPosition DAO)

J'ajoute donc à mon projet un contrôle DataCombo qui me permettra de sélectionner un auteur dans la liste et un contrôle ProgressBar. (COMCTL32.OCX)

```
Option Explicit

Private WithEvents Cnn1 As ADODB.Connection
Private WithEvents MonRs As ADODB.Recordset

Private Sub Form_Load()

Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\Biblio.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.Open "SELECT * From Authors", Cnn1, adOpenKeyset,
adLockReadOnly, adCmdText
With DataCombo1
    Set .RowSource = MonRs
    .ListField = "Author"
    .SelText = MonRs.Fields("Author").Value
End With

End Sub

Private Sub DataCombo1_Change()

    If IsNull(DataCombo1.SelectedItem) Then Exit Sub
    MonRs.Bookmark = DataCombo1.SelectedItem
    ProgressBar1.Value = CInt((MonRs.AbsolutePosition * 100) /
MonRs.RecordCount)

End Sub
```

Dans le "Load" de la feuille, je crée mon recordset. Notez que je n'ai pas précisé la position du curseur, il sera donc côté serveur. J'affecte ensuite mon recordset comme source de données du DataCombo, et je précise quel champ servira pour le remplissage de la liste. J'utilise enfin la propriété SelText du DataCombo, pour afficher le premier enregistrement.

Tout se passe donc dans l'événement Change du DataCombo. La première ligne me permet de sortir de la procédure quand aucun élément n'est sélectionné.

La deuxième ligne positionne l'enregistrement courant sur l'élément sélectionné. On utilise la propriété Bookmark du recordset car la propriété SelectedItem du DataCombo renvoie un signet sur l'élément choisi. En clair, le fait de choisir un élément dans un DataCombo ne change pas l'enregistrement courant. Par contre, comme j'affecte la valeur du signet à la propriété Bookmark du recordset, celui-ci modifie l'enregistrement courant.

Enfin je fais mon calcul en pourcentage.

Comparaison SQL Vs Recordset

Dans cet exemple nous allons faire un petit concours de vitesse qui va nous permettre de mieux considérer les raisons d'un choix entre les recordsets et l'utilisation du SQL.

Le concours est simple, nous allons extraire un jeu d'enregistrements correspondant à tous les auteurs dont le nom commence par "S" et le trier, puis extraire les enregistrements où le champ [année de naissance] est rempli. Pour cela je vais utiliser deux méthodes. D'un côté nous allons utiliser des clauses SQL, de l'autre uniquement des fonctionnalités de l'objet Recordset.

Pour mesurer le temps, je vais utiliser l'API GetTickCount qui renvoie le nombre de millisecondes écoulées depuis le début de la session Windows. Par différence, j'obtiendrai le temps nécessaire. Nous sommes en mode synchrone.

Je déclare donc mon API

```
Private Declare Function GetTickCount Lib "kernel32" () As Long
```

Pour rester sensiblement comparable, je n'utiliserai que des curseurs bidirectionnels en lecture seule. Dans le premier test je ne vais travailler qu'avec des ordres SQL. Je vais donc utiliser un curseur coté serveur. Je vais utiliser un curseur à jeu de clé (KeySet) qui est un peu plus rapide qu'un curseur statique. La connexion ne sera pas incluse dans le calcul du temps. Le code sera le suivant :

```
Private Sub Command1_Click()
Dim TStart As Long

Set Cnn1 = New ADODB.Connection
Cnn1.CursorLocation = adUseServer
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\Biblio.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseServer
TStart = GetTickCount
MonRs.Open "SELECT * From Authors WHERE Author LIKE 'S*' ORDER BY
Author, [Year Born] DESC", Cnn1, adOpenKeyset, adLockReadOnly,
adCmdText
MsgBox "temps = " & GetTickCount - TStart
MonRs.Close
TStart = GetTickCount
MonRs.Open "SELECT * From Authors WHERE Author LIKE 'S*' AND NOT
[Year Born] IS NULL ORDER BY Author, [Year Born] DESC", Cnn1,
adOpenKeyset, adLockReadOnly, adCmdText
MsgBox "temps = " & GetTickCount - TStart

End Sub
```

Tel que nous le voyons, je procède à deux extractions successives.

Dans mon second test, je ne vais travailler qu'avec les fonctionnalités du recordset. Je vais donc extraire l'ensemble de la table, puis travailler sur le jeu d'enregistrements. La propriété Sort demande un curseur côté client. De ce fait le curseur sera statique. Le code utilisé sera le suivant :

```
Private Sub Command2_Click()
Dim TStart As Long

Set Cnn1 = New ADODB.Connection
Cnn1.CursorLocation = adUseClient
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\Biblio.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseClient
TStart = GetTickCount
MonRs.Open "SELECT * From Authors", Cnn1, adOpenStatic,
adLockReadOnly, adCmdText
MonRs.Filter = "Author LIKE 'S*'"
MonRs.Sort = "Author ASC, [Year Born] DESC"
MsgBox "temps = " & GetTickCount - TStart
TStart = GetTickCount
MonRs.Filter = adFilterNone
MonRs.Filter = "Author LIKE 'S*' AND [Year Born] <> NULL"
MsgBox "temps = " & GetTickCount - TStart
```

```
End Sub
```

Chaque programme affichera donc deux temps, le temps qu'il lui faut pour obtenir un jeu d'enregistrements trié contenant tous les auteurs dont le nom commence par "S" et le temps qu'il met pour obtenir le même jeu dont le champ "année de naissance" est rempli.

Les temps ne seront jamais parfaitement reproductibles car de nombreux paramètres peuvent jouer, mais on constate que le code SQL va de 3 à 20 fois plus vite que le filtrage et le tri du recordset (premier temps). Ceci est dû au fait que le curseur client rapatrie l'ensemble des valeurs dans son cache, puis applique le filtre et tri les enregistrements. Il est à noter que le tri par la méthode sort du recordset est très pénalisant.

Par contre, pour le deuxième temps, le second code est toujours plus rapide (deux à trois fois plus). Cela vient du fait qu'il n'est pas nécessaire d'extraire à nouveau les données, celles-ci étant présentes dans le cache du client ce qui permet une exécution très rapide.

Tout cela montre bien qu'il ne faut pas a priori rejeter les fonctionnalités recordset comme on le lit trop souvent. Ce qui pénalise beaucoup les curseurs clients, c'est la nécessité de passer dans le cache l'ensemble des valeurs (Marshaling) mais une fois l'opération faite, le travail devient très rapide. Une fois encore, tout est question de besoin.

Les recherches

Il existe donc deux méthodes de recherches sur un jeu d'enregistrements, plus deux autres qui peuvent être apparentées à une recherche, les filtres.

Commençons par ces derniers. Filtrer un jeu d'enregistrements (en SQL ou par la méthode Filter) revient à faire une recherche de tous les enregistrements répondant à certains critères. Nous avons vu dans l'exemple précédent comment les utiliser, et nous verrons dans les traitements par lot d'autres exemples.

Dans cet exemple nous allons voir les deux méthodes de recherches standard d'ADO, Seek et Find.

Recherche avec Seek

Normalement cette méthode demande l'utilisation d'une table indexée qu'on ouvre avec l'option adCmdTableDirect, on précise l'index dans le recordset et on utilise la méthode Seek. Le code suivant montre un exemple.

```
Private Sub Command3_Click()
Dim MonRs1 As ADODB.Recordset, maprop As Property

Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb ;User Id=Admin; Password="
Set MonRs1 = New ADODB.Recordset
MonRs1.CursorLocation = adUseServer
MonRs1.ActiveConnection = Cnn1
MonRs1.Index = "PrimaryKey"
MonRs1.Open "Authors", Cnn1, adOpenKeyset, adLockReadOnly,
adCmdTableDirect
If MonRs1.Supports(adSeek) Then
    MonRs1.Seek InputBox("Entrez le numéro de l'auteur"),
adSeekFirstEQ
End If

End Sub
```

Dans ce code j'utilise la clé primaire comme index pour la recherche. Je fais un test pour vérifier que le recordset supporte bien la méthode Seek avant de l'utiliser.

Cette méthode de recherche est extrêmement rapide, puisqu'elle porte sur des champs indexés mais encore faut-il que ceux-ci le soient. Si je regarde ma table 'Authors' le seul champ indexé est la clé primaire 'Au_Id'. Or on connaît rarement la valeur du champ clé lorsque celui-ci est non

informationnel. Donc la possibilité d'utilisation de cette méthode repose sur la qualité de la source de données.

Recherche avec Find

Rappelons que la méthode Find ne fonctionne que sur **un** champ. A la différence de DAO, il n'existe pas de propriété NoMatch pour dire que la recherche a échoué. Lorsqu'il n'y a pas de correspondance, le recordset se place en position BOF ou EOF selon le sens de la recherche.

Le code suivant fait une recherche des enregistrements dont l'année de naissance est celle choisie par l'utilisateur.

```
Private Sub cmdFind_Click()

Dim AnneeCherchee As Integer

    AnneeCherchee = Val(InputBox("Entrez l'année de naissance", , 
1947))
    MonRs.Find "[year born]=" & AnneeCherchee, , adSearchForward, 1
    Do While Not MonRs.EOF
        MsgBox "nom de l'auteur -->" & MonRs!Author
        MonRs.Find "[year born]=" & AnneeCherchee, 1, 
adSearchForward
    Loop

End Sub
```

Ce code est l'équivalent d'une recherche DAO avec FindFirst et FindNext. Je pourrais écrire l'inverse avec une recherche à l'envers pour peu que le curseur soit bidirectionnel.

```
Private Sub cmdFind_Click()

Dim AnneeCherchee As Integer, Signet As Variant

    AnneeCherchee = Val(InputBox("Entrez l'année de naissance", , 
1947))
    MonRs.MoveLast
    Signet = MonRs.Bookmark
    MonRs.Find "[year born]=" & AnneeCherchee, , adSearchBackward, 
Signet
    Do While Not MonRs.BOF
        MsgBox "nom de l'auteur -->" & MonRs!Author
        MonRs.Find "[year born]=" & AnneeCherchee, 1, 
adSearchBackward
    Loop

End Sub
```

Attention de bien utiliser un signet ou la position courante pour démarrer la recherche, si vous utilisez une position votre code risque de ne pas se comporter comme vous le désirez.

On peut donc faire une analogie entre la méthode Find ADO et les méthodes DAO.

FindFirst	Find Critères, , adSearchForward, 1
FindLast	Find Critères, , adSearchBackward, Signet
FindNext	Find Critères, 1, adSearchForward
FindPrevious	Find Critères, 1, adSearchBackward

Enfin n'oubliez pas que l'opérateur du critère doit être =, <, >, <=, >=, <> ou Like. N'essayez donc pas une recherche du genre MonRs.Find "[year born] IS NOT NULL" qui déclenchera une erreur, utilisez MonRs.Find "[year born] <> NULL".

Si le recordset contient de nombreux enregistrements et que l'on travaille côté client, il peut être très rentable d'ajouter un index temporaire sur le champ de la recherche. Pour cela on utilise la

propriété dynamique "Optimize". Notez bien que cet index n'existe que dans l'objet Recordset, il n'est pas créé dans la source de données. Par exemple

```
Set cnn1 = New ADODB.Connection
cnn1.CursorLocation = adUseClient
cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb ;User Id=Admin; Password="
Set MonRs1 = New ADODB.Recordset
MonRs1.ActiveConnection = cnn1
MonRs1.Open "SELECT * FROM Authors", cnn1, adOpenStatic,
adLockReadOnly, adCmdText
MonRs1.Fields("Author").Properties("Optimize") = True
MonRs1.Find "Author='Boddie, John'"
```

Récupérer une clé auto-incrémentée

Lorsqu'on crée un nouvel enregistrement il peut être parfois intéressant de récupérer la valeur de la clé auto-incrémentée correspondante, pour pouvoir par exemple créer des enregistrements ayant cette valeur comme clé étrangère. Dans l'exemple suivant nous allons regarder le principe.

```
Private Sub Command4_Click()
Dim recup As Long

Set Cnn1 = New ADODB.Connection
Cnn1.CursorLocation = adUseServer
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\Biblio.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
With MonRs
    .CursorLocation = adUseServer
    .Open "SELECT * From Authors", Cnn1, adOpenKeyset,
adLockPessimistic, adCmdText
    .AddNew
        !Author = "Rabilloud ,J-M"
        ![year born] = 1967
    .Update
    recup = !Au_id
End With

End Sub
```

Dans ce cas il n'y à rien de particulier à faire puisque le curseur est côté serveur. Notons toutefois que le curseur doit être de type "KeySet" pour que cela fonctionne.

Avec un curseur côté client, nous avons vu qu'il faut utiliser la propriété dynamique donc ajouter dans le code :

```
MonRs.Properties("Update Resync") = adResyncAutoIncrement
```

Contrôles Visual Basic

C'est contrôles aussi contiennent leurs lots de pièges divers et variés. Nous allons regarder deux exemples pour parcourir certains de ces problèmes.

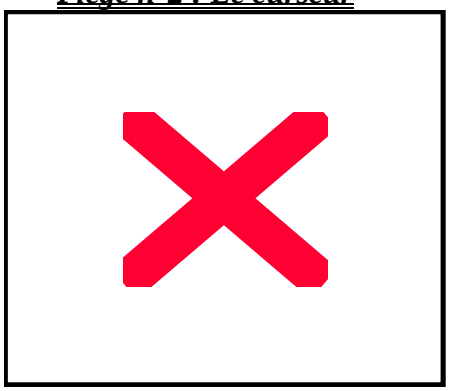
Le contrôle ADO (ADODC)

Ce contrôle possède quelques astuces de fonctionnement, mais son principal défaut est d'être utilisé à tort et à travers. En effet à l'origine, il est fait pour servir de fournisseur de données pour d'autres contrôles de la feuille tout en permettant la navigation par de simples clics. Or très souvent on le retrouve en utilisation invisible, et alors une simple classe ferait mieux l'affaire, comme fournisseur de recordset, et là il ne fonctionne pas.

Piège n°1 : Pas de recordset

Le contrôle ADO crée le recordset uniquement si au moins un autre contrôle de la feuille l'utilise comme DataSource **et** comme DataField. Dans le cas contraire, le recordset ne sera jamais créé, quand bien même cela serait explicitement demandé par le code.

Piège n°2 : Le curseur



Dans sa page de propriétés, le contrôle ADO contient toutes les informations qui lui sont utiles pour créer sa connexion et son recordset. Outre le piège standard de définir un curseur indisponible, il y a aussi la possibilité de définir une propriété à la création incompatible avec le code de manipulation du recordset. L'exemple suivant va expliciter mon propos.

Imaginons un contrôle ADODC configuré comme dans l'exemple ci-dessus. Si dans mon code je fais :

```
With Adodc1
    .RecordSource = "SELECT * FROM
Authors"
    .Refresh
End With
```

Je vais avoir une erreur "Erreur de syntaxe dans la clause FROM". Ceci vient du fait que la propriété CommandType du contrôle n'est plus valide.

Piège n°3 : L'asynchronisme

La troisième source d'erreurs classique vient du fait qu'un contrôle ADO est toujours asynchrone. Il faut donc gérer sa programmation événementielle. Ceci est indispensable car le changement de l'enregistrement courant valide l'ensemble des modifications effectuées sur l'enregistrement courant précédent.

Le contrôle DataGrid

Affichage des données

Je m'arrête sur ce point car il peut être assez déroutant. En utilisant comme fournisseur JET 3.51 tous les curseurs acceptant les signets (c'est à dire n'étant pas en avant seulement) peuvent être affectés comme source de données du contrôle DataGrid. Ainsi le code suivant affiche bien les données dans la grille :

```
Set Cnn1 = New ADODB.Connection
Cnn1.CursorLocation = adUseServer
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.3.51;Data
Source=D:\user\jmarc\bd6.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseServer
MonRs.Open "reqAdresse", Cnn1, adOpenKeyset, adLockPessimistic,
adCmdTable
Set DataGrid1.DataSource = MonRs
```

Si j'utilise le même code avec le fournisseur Jet 4.0 plus rien ne s'affiche dans la grille. Ceci vient du fait que les fonctionnalités du curseur ont changé. Pour qu'un curseur côté serveur puisse être utilisé comme source de données d'un contrôle DataGrid avec Jet 4.0, je dois utiliser la propriété dynamique IrowsetIdentity. Mon code deviendra :

```
Set Cnn1 = New ADODB.Connection
Cnn1.CursorLocation = adUseServer
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\user\jmarc\bd6.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseServer
MonRs.ActiveConnection = Cnn1
MonRs.Properties("IrowsetIdentity") = True
```

```
MonRs.Open "reqAdresse", , adOpenKeyset, adLockPessimistic,
adCmdTable
Set DataGrid1.DataSource = MonRs
```

Vous noterez que j'ai légèrement changé la structure de mon code. En effet, pour pouvoir valoriser une propriété dynamique, je dois :

- Ø spécifier le fournisseur
- Ø Donner la valeur avant l'ouverture de l'objet

Requête multi table

Le contrôle DataGrid permet pour peu qu'il soit correctement paramétré de faire des ajouts, suppressions, mises à jour dans la base de données. Si ceci fonctionne sans poser de problèmes lorsque la requête ne concerne qu'une table, il en va souvent autrement lorsque la grille affiche des données provenant de tables mises en relation.

Envisageons les deux tables suivantes

tab_Mere	
Champ	Type
Me_Id	NumeroAuto
Nom	Texte (50)
Prenom	Texte (50)

tab_Fille	
Champ	Type
Fil_Id	NumeroAuto
Adresse	Texte (255)
Ville	Texte (50)
Me_Id	Long

La requête reqAdresse de mon exemple précédent correspondant à la requête SQL :

```
SELECT tab_mere.Nom, tab_mere.Prenom, tab_fille.Adresse, tab_fille.Ville
FROM tab_mere
        INNER JOIN tab_fille
                ON tab_mere.Me_id = tab_fille.Me_id
```

Si dans mon contrôle DataGrid je modifie un enregistrement tout se passe bien, par contre si j'en ajoute un nouveau, j'obtiens l'erreur 6153 "Vous ne pouvez pas ajouter ou modifier un enregistrement car l'enregistrement associé est requis dans la table tab_mere".

Pour contourner ce problème il faut mettre en place toute une stratégie de récupération des valeurs saisies, pas nécessairement très lourde d'ailleurs. Néanmoins, la programmation du DataGrid peut vite devenir laborieuse sur les recordsets multi tables.

Nous allons voir ici une des méthodes de traitement possible pour les jeux d'enregistrements multi tables avec le contrôle Datagrid. Il en existe de nombreuses variantes, celle ci est peu coûteuse en ressources mais ne fonctionnera pas avec tous les fournisseurs.

J'ai donc sur ma feuille un contrôle ADO qui pointe sur ma base "biblio" avec la requête suivante :

```
SELECT Publishers.PubID, Titles.PubID, Publishers.Name, Publishers.[Company Name], Titles.Title,
Titles.ISBN, Titles.[Year Published]
FROM Publishers
        INNER JOIN Titles
                ON Publishers.PubID = Titles.PubID
```

J'ai configuré mon Datagrid afin que les deux champs PubID (clé primaire de la table Publishers et clé étrangère de la table Titles) n'apparaissent pas, puisque ce n'est pas à l'utilisateur de rentrer la valeur PubID de l'éditeur. Comme je l'ai dit plus haut, si un utilisateur entre une ligne dans le Datagrid, celui-ci va déclencher une erreur 6153. Je vais donc programmer l'événement Error du contrôle DataGrid.

```
Private Sub DataGrid1_Error(ByVal DataError As Integer, Response As
Integer)

Dim MaComm As ADODB.Command, Recup

If DataError = 6153 Then
    Set MaComm = New ADODB.Command
    With MaComm
        .ActiveConnection = Adodc1.Recordset.ActiveConnection
```

```
        .CommandText = "INSERT INTO Publishers (Name,[Company Name])
VALUES('" & Adodc1.Recordset!Name & "','" &
Adodc1.Recordset![Company Name] & "')"
        .Execute
        .CommandText = "SELECT @@IDENTITY"
        Recup = .Execute
        .CommandText = "INSERT INTO Titles (Title,ISBN,[Year
Published],PubId) VALUES('" & Adodc1.Recordset!Title & "','" &
Adodc1.Recordset!ISBN & "','" & Adodc1.Recordset![Year Published] &
"','" & Recup(0).Value & "')"
        .Execute
    End With
    Set DataGrid1.DataSource = Nothing
    Adodc1.Recordset.CancelUpdate
    Adodc1.Recordset.Requery
    Set DataGrid1.DataSource = Adodc1
    Response = 0
End If

End Sub
```

Comme vous le voyez, je vais utiliser un objet Command pour remplacer le moteur de curseur. Le recordset du contrôle ADO contient toutes les informations entrées par l'utilisateur dans son enregistrement en cours. En résumé, je passe la requête ajout dans la table parent, je récupère la valeur de la clé primaire et je passe ma seconde requête.

Ensuite je dois décrocher le DataGrid de sa source de données pour pouvoir annuler les modifications du contrôle ADO avant de refaire le lien.

Mise à jour

Une autre question qui revient souvent est au sujet de la non mise à jour de la grille par rapport au recordset sous-jacent. A cela, deux erreurs classiques reviennent souvent.

- Vous travaillez avec un curseur côté client donc statique. Comme le recordset ne reflète pas les modifications de la base, la grille n'est pas mise à jour.
- Vous utilisez le recordset que vous avez affecté au lieu du recordset sous jacent.

Explication du problème.

Regardons le code suivant :

```
Private Sub Form_Load()

Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\user\jmarc\bd6.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseServer
MonRs.ActiveConnection = Cnn1
MonRs.Properties("IrowsetIdentity") = True
MonRs.Open "tab_Mere", , adOpenKeyset, adLockPessimistic, adCmdTable
Set DataGrid1.DataSource = MonRs

End Sub
```

Ce code est le même que le précédent si ce n'est que la source est une table. A ce propos, vous vous demandez peut être pourquoi une requête stockée dans la base est ouverte avec le paramètre adCmdTable. La réponse se trouve là è "Utilisation d'ADOX"

```
Private Sub Command6_Click()

While Not MonRs.EOF
    If MonRs.Fields("nom").Value = "Durand" Then MonRs.Delete
adAffectCurrent
```

```
    MonRs.MoveNext
Wend

End Sub
```

Le code ci-dessus supprime tous les enregistrements dont le nom est Durand. Pourtant votre grille continue d'afficher les Durand. Cela vient du fait que plus rien ne lie le Recordset MonRs avec le contrôle grille. Pour que l'effet de cette suppression de la base soit reflété automatiquement par la grille il faut :

```
Private Sub Command6_Click()
Dim TempRs As ADODB.Recordset

Set TempRs = DataGrid1.DataSource
While Not MonRs.EOF
    If TempRs.Fields("nom").Value = "Durand" Then TempRs.Delete
adAffectCurrent
    MonRs.MoveNext
Wend

End Sub
```

Dans ce code, je récupère le recordset qui est dans la propriété DataSource de la grille. Dans ce cas là, celle-ci reflètera les modifications due à la manipulation du Recordset.

Programmation évènementielle

Celle-ci n'est pas plus complexe que la programmation événementielle des contrôles. Il faut toutefois garder à l'esprit que le changement de l'enregistrement courant valide les modifications. Comme un simple Find fait changer l'enregistrement courant, on devine facilement l'intérêt de la programmation asynchrone pour éviter des actions masquées.

Connection et command asynchrone

L'exemple suivant est une opération traditionnelle de modification de données. Comme elle peut être lourde, on la traite de manière asynchrone.

```
Private Sub Form_Load()

Set Cnn1 = New ADODB.Connection
Cnn1.CursorLocation = adUseServer
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\nwind.mdb
;User Id=Admin; Password=", , , adAsyncConnect
   ..........

End Sub

Private Sub Cnn1_ConnectComplete(ByVal pError As ADODB.Error,
adStatus As ADODB.EventStatusEnum, ByVal pConnection As
ADODB.Connection)

Dim MaCommand As ADODB.Command

    If adStatus = adStatusErrorsOccurred Then
        MsgBox "Erreur de connexion " & pError.Description
        Unload Me
    Else
        Set MaCommand = New ADODB.Command
        MaCommand.ActiveConnection = Cnn1
```

```
        MaCommand.CommandText = "UPDATE Clients Set Pays = 'USA'
WHERE Pays = 'États-Unis'"
        Set MonRs = MaCommand.Execute(, , adAsyncExecute)
    End If

End Sub

Private Sub Cnn1_ExecuteComplete(ByVal RecordsAffected As Long,
ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum,
ByVal pCommand As ADODB.Command, ByVal pRecordset As
ADODB.Recordset, ByVal pConnection As ADODB.Connection)

    MsgBox RecordsAffected & " modification(s) exécutée(s)"

End Sub
```

Le cheminement est simple, dans le load de la feuille je démarre une connexion asynchrone, lorsque celle-ci est complète elle lance s'il n'y a pas d'erreur la commande. Lorsque celle-ci est terminée, un message précise le nombre de modifications effectuées.

Extractions bloquantes & non bloquantes

Il y a ces deux sortes d'extractions asynchrones. Il s'agit juste d'une différence de comportement lorsqu'on fait appel à des enregistrements qui n'ont pas été encore extraits. Une extraction bloquante rend la main lorsque la réponse exacte peut être donnée, une extraction non bloquante rend la main immédiatement et une réponse temporairement exacte. Un petit exemple rendra cela plus clair.

```
Private Sub Command7_Click()
Dim Recup As String, MonSignet As Variant

Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb ;User Id=Admin; Password=" ', , ,
adAsyncConnect
Set MonRs = New ADODB.Recordset
With MonRs
    .CursorLocation = adUseClient
    .ActiveConnection = Cnn1
    .Properties("Initial Fetch Size") = 50
    .CursorType = adOpenStatic
    .LockType = adLockBatchOptimistic
    .Open "Titles", , , , adAsyncFetch  'adAsyncFetchNonBlocking
End With
MonRs.MoveLast
Recup = MonRs!Title
MonSignet = MonRs.Bookmark
Set DataGrid1.DataSource = MonRs
MonRs.Bookmark = MonSignet
MsgBox Recup & " " & MonRs.AbsolutePosition

End Sub
```

Selon le mode choisi, je récupérerai un signet sur le cinquantième enregistrement (adAsyncFetchNonBlocking) ou sur le dernier (adAsyncFetch). Aussi faut-il se méfier du type d'extraction que l'on choisit.

Suivre l'extraction

Nous allons, dans cet exemple, utiliser une ProgressBar pour suivre l'extraction d'un recordset asynchrone. Pour cela nous allons utiliser l'événement FetchProgress. Attention il faut travailler avec un curseur côté client. Le code est le suivant :

```
Private Sub cmdSuiv_Click()
    Dim MaComm As ADODB.Command

    ProgressBar1.Visible = True
    ProgressBar1.Min = 0
    Set MonRs = Cnn1.Execute("SELECT Count(Titles.Title) AS NbRes
FROM (Publishers INNER JOIN Titles ON Publishers.PubID =
Titles.PubID) INNER JOIN (Authors INNER JOIN [Title Author] ON
Authors.Au_ID = [Title Author].Au_ID) ON Titles.ISBN = [Title
Author].ISBN")
    ProgressBar1.Max = MonRs!NbRes
    MonRs.Close
    Set MonRs = Nothing
    Set MonRs = New ADODB.Recordset
    With MonRs
        .CursorLocation = adUseClient
        .ActiveConnection = Cnn1
        .Properties("Initial Fetch Size") = 100
        .Properties("Background Fetch Size") = 1000
        .Open "[All Titles]", , adOpenStatic, adLockReadOnly,
adCmdTable + adAsyncFetchNonBlocking
    End With

End Sub

Private Sub MonRs_FetchComplete(ByVal pError As ADODB.Error,
adStatus As ADODB.EventStatusEnum, ByVal pRecordset As
ADODB.Recordset)

    ProgressBar1.Visible = False

End Sub

Private Sub MonRs_FetchProgress(ByVal Progress As Long, ByVal
MaxProgress As Long, adStatus As ADODB.EventStatusEnum, ByVal
pRecordset As ADODB.Recordset)

    ProgressBar1.Value = Progress  'Debug.Print Progress
'Int((Progress / pRecordset.RecordCount) * 100) '

End Sub
```

Comme vous le voyez, j'utilise une requête pour déterminer le nombre d'enregistrements attendu afin de valoriser correctement le maximum de la ProgressBar. La propriété dynamique " Initial Fetch Size" doit être utilisée pour un comportement correct du code.

Gestion des modifications

Idéalement on utilise uniquement l'événement WillChangeRecord pour le contrôle des modifications. En effet en réalisant juste un test sur l'argument adReason on connaît la cause de la demande de validation. La structure du code est la suivante :

```
Private Sub MonRs_WillChangeRecord(ByVal adReason As
ADODB.EventReasonEnum, ByVal cRecords As Long, adStatus As
ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)

  Dim bCancel As Boolean

  Select Case adReason
    Case adRsnAddNew

    Case adRsnClose

    Case adRsnDelete

    Case adRsnFirstChange

    Case adRsnMove

    Case adRsnRequery

    Case adRsnResynch

    Case adRsnUndoAddNew

    Case adRsnUndoDelete

    Case adRsnUndoUpdate

    Case adRsnUpdate

  End Select

  If bCancel Then adStatus = adStatusCancel
End Sub
```

Le problème de ce style de programmation est la difficulté à bien identifier la cause de la modification. Ainsi, si vous changez deux champs d'un même enregistrement par l'intermédiaire d'une grille, l'événement se produira deux fois, dans le premier cas avec l'argument adRsnFirstChange, ensuite avec adRsnUpdate. C'est pour cela que dans certains cas on préfère travailler sur l'événement WillMove en testant la valeur EditMode du Recordset.

Dans cet exemple, nous allons afficher un message de confirmation de validation à chaque mouvement dans le jeu d'enregistrements.

```
Private Sub MonRs_WillMove(ByVal adReason As ADODB.EventReasonEnum,
adStatus As ADODB.EventStatusEnum, ByVal pRecordset As
ADODB.Recordset)

If pRecordset.EditMode = adEditInProgress Or adEditAdd Or
adEditDelete Then
    If MsgBox("Voulez vous enregistrer les modifictions", vbQuestion
+ vbYesNo) = vbYes Then
        pRecordset.Update
    Else
        pRecordset.CancelUpdate
    End If
End If

End Sub
```

Comme nous le voyons, ce sont des codes assez simples qui gèrent en général la programmation évènementielle.

Je ne vais pas vous donner plus d'exemple de programmation évènementielle car elle doit être adaptée à votre programme.

Recordset persistant

Un recordset persistant est en fait un fichier de type "datagram" ou XML que l'on peut créer à partir d'un recordset ou ouvrir comme un recordset. Ceci peut être très intéressant dans les cas suivants :

- Comme sécurité (comme nous allons le voir plus loin)
- Pour alléger la connexion (puisqu'on peut le remettre à jour par resynchronisation)
- Comme base de travail, si la mise à jour n'est pas primordiale

Dans le principe de fonctionnement c'est assez simple

```
Private Sub Form_Load()

Set Cnn1 = New ADODB.Connection
Cnn1.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb ;User Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.CursorLocation = adUseClient
MonRs.ActiveConnection = Cnn1
MonRs.Open "Authors", , adOpenStatic, adLockBatchOptimistic,
adCmdTable

End Sub

Private Sub cmdSave_Click()
    If Dir("d:\monrs.adtg") <> "" Then Kill "d:\monrs.adtg"
    MonRs.Save "d:\monrs.adtg", adPersistADTG
End Sub

Private Sub cmdOpen_Click()
    If MonRs.State = adStateOpen Then MonRs.Close
    MonRs.Open "d:\monrs.adtg", Cnn1, adOpenStatic,
adLockBatchOptimistic, adCmdFile
End Sub
```

Comme vous le voyez, il faut toujours s'assurer qu'il n'existe pas de fichier du même nom sur le disque. Notez aussi qu'un recordset persistant garde l'ensemble des données et des méta-données, on peut ainsi rendre persistant un recordset ayant des modifications en attente, puis le rouvrir pour appliquer ces modifications sur la base.

Dans mon exemple, j'utilise une connexion pour ouvrir mon Fichier. Cela n'est pas obligatoire, car ADO propose un fournisseur de service pour faire cela, je pourrais donc l'ouvrir avec :

```
MonRs.Open "d:\monrs.adtg", "Provider=MSPersist;", adOpenStatic,
adLockBatchOptimistic, adCmdFile
```

Quelques précisions sont encore nécessaires :

- Si le recordset a une propriété Filter définie sans que celle-ci soit une constante prédéfinie, seules les lignes accessibles avec ce filtre sont sauvegardées.
- Pour sauvegarder plusieurs fois votre recordset, précisez la destination seulement la première fois. Si la deuxième fois, vous redonnez la même destination vous obtiendrez une erreur, si vous en donnez une autre vous obtiendrez deux fichiers et le premier restera ouvert. La fermeture du fichier n'a lieu que lors de la fermeture du recordset.
- En programmation asynchrone, Save est toujours une méthode bloquante.

Ø Un recordset persistant garde ses possibilités de modifications sur la base. Toutefois, elles ne peuvent concerner qu'une table si le curseur est côté serveur. De plus, ce type de recordset ne pourra pas être synchronisé.

 Si votre recordset contient des champs de type "Variant" le recordset sauvegardé ne sera pas exploitable.

Synchronisation

Lorsqu'on utilise un Recordset sans mise à jour, de type statique ou en avant seulement, il est possible de réactualiser les données de celui-ci. On appelle cette technique la synchronisation. Celle-ci n'est pas possible sur les recordsets côté client en lecture seule. Cette méthode est souvent mal utilisée car son comportement peut sembler étrange, aussi beaucoup de développeurs préfèrent utiliser, à tort, la méthode Requery. Un des problèmes de cette méthode est qu'en fait, elle contient deux méthodes fondamentalement différentes selon les paramètres qu'on lui passe.

Synchronisation des données sous-jacentes

De la forme :

MonRs1.Resync adAffectAllChapters, adResync**Underlying**Values

Elle ne va concerner que les enregistrements dont vous avez modifiés au moins un champ. La valeur remise à jour sera la propriété UnderlyingValue des objets Fields. Dans certains cas, elle peut provoquer une erreur si la synchronisation n'est pas possible.

Synchronisation des données

De la forme :

MonRs1.Resync adAffectAllChapters, adResync**All**Values

Elle va porter sur tous les enregistrements. Par contre elle annule aussi toutes les modifications en cours de votre recordset.

Ces deux méthodes ne s'appliquent pas dans la même situation. Nous verrons le premier cas un peu plus loin dans les traitements par lot, le code suivant déclenche une alerte si un enregistrement a été supprimé par un autre utilisateur.

```
On Error Resume Next
MonRs1.Resync adAffectAllChapters, adResyncAllValues
If Err.Number = -2147217885 Then
    MonRs1.Filter = adFilterConflictingRecords
    MsgBox MonRs1.RecordCount & " enregistrements ont été
supprimés", vbCritical + vbOKOnly, "Erreur"
    Err.Clear
    Exit Sub
End If
```

Je vais profiter de ce chapitre, pour vous montrer une erreur assez facile à faire. Examinons le code suivant :

```
Set MonRs1 = New ADODB.Recordset
MonRs1.ActiveConnection = cnn1
MonRs1.Open "SELECT * FROM Authors", cnn1, adOpenStatic,
adLockOptimistic, adCmdText
MonRs1![Year born] = 1921
MonRs1.Resync adAffectCurrent, adResyncUnderlyingValues
```

Après l'exécution de la synchronisation, vous aurez soit une erreur de verrouillage, soit la valeur de l'année de naissance sera de 1921. Ceci vient du fait que l'appel de la méthode Resync crée toujours un repositionnement de l'enregistrement en cours, et comme nous l'avons vu, un changement de position valide les modifications en attente. Voilà pourquoi on utilise plutôt les synchronisations en mode Batch, que nous allons voir maintenant.

Traitement par lot

Comme nous allons le voir, voilà un passage "sensible" de la programmation ADO. L'intérêt du travail par lot n'est plus à démontrer mais il faut gérer soigneusement la mise à jour de la base afin de ne pas endommager celle-ci. Grosso modo, il existe trois sortes de traitement par lot :

- Ø Le traitement sécurisé : Il se fait par une connexion exclusive à la source de données (en général en dehors des heures d'accès). Il n'est dans ce cas pas nécessaire de gérer la concurrence et il peut contenir des procédures assez lourdes.
- Ø Le traitement sans contrôle : Il n'y a pas de contrôle sur les mises à jour en échec. Ce type de traitement est assez rare, mais ne demande aucune gestion par le code.
- Ø Le traitement transactionnel : C'est le cas le plus fréquent mais aussi le plus délicat à gérer. Nous allons donc étudier un exemple.

Remarque préliminaire :

Décomposition de Status

La valeur de la propriété Status peut être la somme de plusieurs des constantes données plus haut. Nous allons donc utiliser la fonction de décomposition donnée ci-dessous

```
Private Function Decomp_Status(ByVal ValStatus As Long) As Long()

Dim PEnt As Integer, Retour() As Long

ReDim Retour(0 To 0)
Do While ValStatus > 0
    PEnt = Int(Log(ValStatus) / Log(2))
    Retour(UBound(Retour)) = 2 ^ PEnt
    ReDim Preserve Retour(0 To UBound(Retour) + 1)
    ValStatus = ValStatus - 2 ^ PEnt
Loop
Retour(UBound(Retour)) = 0
Decomp_Status = Retour

End Function
```

Cette fonction renvoie un tableau de long contenant les valeurs des constantes de la propriété Status.

Transaction

On est parfois tenté d'englober le traitement par lot dans une transaction afin de pouvoir revenir en arrière en cas d'échec partiel du traitement. Malheureusement cette méthode assez simple ne fonctionne pas, du moins avec certains fournisseurs. On doit donc écrire l'ensemble du code qui gère l'intégrité de la source de données.

Règles de traitement

Avant toute écriture de code il faut avoir défini les règles de comportement que le traitement par lot doit suivre. Cette phase est indispensable afin de ne jamais pouvoir endommager la source de données. Dans l'exemple ci-dessous, je vais faire un traitement qui suit les règles suivantes.

- v Si un enregistrement concerné par le traitement a été supprimé, la mise à jour de cet enregistrement est annulée.
- v Si une mise à jour échoue de la faute du verrouillage, l'enregistrement sera retraité à la volée
- v Si une autre erreur se produit, toute l'opération sera annulée

Exemple

Pour voir le fonctionnement de la résolution des conflits, nous allons simuler par le code, une action de deux utilisateurs. Dans cet exemple nous allons voir de nombreuses techniques expliquées au-dessus, n'hésitez pas à vous y reporter de nouveau.

```
Dim Reponse() As Long, compteur As Long

'première connexion
Set AutreCnn = New ADODB.Connection
With AutreCnn
```

```
    .CursorLocation = adUseServer
    .Mode = adModeShareDenyNone
    .Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb ;User Id=Admin; Password="
End With
Set AutreRs = New ADODB.Recordset
With AutreRs
    .CursorLocation = adUseServer
    .ActiveConnection = AutreCnn
    .CursorType = adOpenKeyset
    .LockType = adLockPessimistic
    .Source = "SELECT * From Authors WHERE [year born] IS NOT NULL"
    .Open
End With
```

```
'seconde connexion
Set MaCnn = New ADODB.Connection
With MaCnn
    .CursorLocation = adUseClient
    .Mode = adModeShareDenyNone
    .Open "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb ;User Id=Admin; Password="
End With
Set MonRs = New ADODB.Recordset
With MonRs
    .CursorLocation = adUseClient
    .ActiveConnection = MaCnn
    .CursorType = adOpenStatic
    .LockType = adLockBatchOptimistic
    .Source = "SELECT * From Authors WHERE [year born] IS NOT NULL"
    .Open
End With
```

J'ai donc créé deux connexions, une côté serveur qui va me servir à perturber mon traitement, et une côté client que j'utilise pour le traitement par lot.

```
'modification de perturbation
While Not AutreRs.EOF
    'modification de quelques dates
    If AutreRs.Fields("year born").Value = 1947 Then
AutreRs.Fields("year born").Value = 1932
    AutreRs.MoveNext
Wend
AutreRs.MoveLast
'suppression du dernier enregistrement
AutreRs.Delete adAffectCurrent
'modification de mon utilisateur par lot
While Not MonRs.EOF
    If MonRs.Fields("year born").Value < 1951 Then
MonRs.Fields("year born").Value = MonRs.Fields("year born").Value +
1
    MonRs.MoveNext
Wend
```

Ici j'ai fais mes modifications, je procède maintenant au traitement par lot

```
'Vérification préliminaire
MonRs.Resync adAffectAllChapters, adResyncUnderlyingValues
MonRs.Filter = adFilterPendingRecords
```

```
'traitement préliminaire (non utilisé dans l'exemple)

'*****DEBUT TRAITEMENT*****
'While Not MonRs.EOF
'    If MonRs.Fields("year born").OriginalValue <>
MonRs.Fields("year born").UnderlyingValue Then
'        MsgBox "Au moins un enregistrement a été modifié par un
autre utilisateur" & vbCrLf & "Opération annulée", vbCritical +
vbOKOnly
'        MonRs.CancelBatch
'        Exit Sub
'    End If
'    MonRs.MoveNext
'Wend
'*****FIN TRAITEMENT*****

'création d'un recordset de sauvegarde
If Dir("d:\RsSauve.adtg") <> "" Then Kill "d:\RsSauve.adtg"
MonRs.Save "d:\RsSauve.adtg", adPersistADTG
'mise à jour + résolution des conflits
MonRs.Properties("Update Resync") = adResyncConflicts
On Error Resume Next
MonRs.UpdateBatch
If MonRs.ActiveConnection.Errors.Count > 0 Then
    MonRs.Filter = adFilterConflictingRecords
    Do While Not MonRs.EOF
        Reponse = Decomp_Status(MonRs.Status)
        For compteur = LBound(Reponse) To UBound(Reponse)
            Select Case Reponse(compteur)
                Case adRecOK, adRecModified 'mise à jour réussie

                Case adRecConcurrencyViolation, adRecCantRelease
'enregistrement modifié ou verrouillé
                    If MonRs.Fields("year born").UnderlyingValue <
1951 Then
                        MonRs.Fields("year born").Value =
MonRs.Fields("year born").UnderlyingValue + 1
                    End If

                Case adRecDBDeleted 'enregistrement supprimé
                    MonRs.CancelUpdate
                    Exit For

                Case Else 'Autres erreurs
                    Annulation = True
                    Exit For

            End Select
        Next compteur
        If Annulation Then Exit Do
        MonRs.MoveNext
    Loop
    If Annulation Then 'restaure les valeurs d'origine
        MonRs.Close
        Set MonRs = New ADODB.Recordset
        MonRs.Open "d:\RsSauve.adtg", MaCnn, , , adCmdFile
```

```
        MonRs.Resync adAffectAllChapters, adResyncUnderlyingValues
        MonRs.Filter = adFilterAffectedRecords
        Do While Not MonRs.EOF
            If MonRs.Fields("year born").UnderlyingValue =
MonRs.Fields("year born").Value Then
                MonRs.Fields("year born").Value = MonRs.Fields("year
born").OriginalValue
            End If
            MonRs.MoveNext
        Loop
    End If
    'exécute la validation ou l'annulation
    MonRs.Properties("Update Criteria") = adCriteriaKey
    MonRs.UpdateBatch adAffectGroup
End If
Err.Clear
On Error GoTo 0

End Sub
```

Regardons tout cela en détail. Je vous ai mis en commentaires au début un exemple de traitement préliminaire. Dans cet exemple pourtant simple, car les modifications ne portent que sur un champ d'une seule table, le codage est déjà assez lourd. Beaucoup de développeurs préfèrent garder un modèle transactionnel strict (atomicité) et rejettent l'ensemble du traitement si un enregistrement au moins ne peut être mis à jour. Dans ce cas, le traitement préliminaire permet de détecter une cause d'annulation avant le début des mises à jour.

Ensuite j'applique la stratégie générale qui consiste à garder une copie du recordset avant la transmission des modifications. Je vais donc créer un recordset persistant. Le fait de garder un recordset persistant peut paraître aberrant, mais il est obligatoire lors de traitement par lot avec un curseur client. Nous avons vu que le moteur de curseur utilise les propriétés OriginalValue des champs pour écrire ses requêtes action. Ces propriétés, qui pour le coup porte mal leur nom se remettent à jour lors de la réussite d'une modification. Après l'appel de UpdateBatch, il n'est donc plus possible de connaître la valeur qu'avait le champ avant la mise à jour. Le recordset persistant permet de pouvoir restaurer.

Avant d'appeler la méthode UpdateBatch je déclenche la récupération d'erreur. En effet nous avons dit que l'échec d'une mise à jour provoque l'apparition d'une erreur ainsi que la valorisation de la collection Errors de l'objet Connection. Si au moins une erreur apparaît dans la collection je rentre alors dans mon code de gestion.

La procédure est alors simple. Je filtre pour obtenir un recordset des enregistrements ayant échoué leurs mises à jour et je décompose leur propriété Status. Cette décomposition est obligatoire puisque tous les enregistrements seront au moins marqués comme la somme du statut modifié et de l'erreur. Je gère ensuite les cas.

Vous pouvez noter aussi que pour contourner le verrouillage je modifie la propriété "Update Criteria".

Cet exemple ne présente qu'une des techniques de gestion d'erreurs. Il est plus fréquent de rencontrer des traitements à l'aide de l'objet Command.

Mise en forme des données

Le moteur de curseur client implémente un fournisseur de service qui permet de construire des recordsets un peu particuliers, le "Data Shaping Service for OLE Db". Cette mise en forme des données peut souvent ressembler à une requête de jointure standard ou à l'utilisation d'agrégats, mais le résultat final est sensiblement différent. Je ne vais pas visiter en détail ici le langage Shape car sa syntaxe peut devenir rapidement complexe, aussi n'allons nous voir que quelques exemples. Toutefois dans le deuxième exemple nous verrons une technique de création de requête.

N.B : On peut utiliser le composant DataEnvironment pour générer des requêtes Shape.

Pour faire simple, disons qu'un recordset "Shape" est un recordset auquel on a ajouté un (des) champ(s), ce champ pouvant contenir une référence à un autre recordset, une valeur calculée sur la ligne ou sur une colonne d'un recordset, etc….

Cette mise en forme est faite par le langage Shape dont nous allons voir ici deux cas d'utilisation.

Tout d'abord, il faut spécifier deux fournisseurs à la connexion, le fournisseur de service dans la propriété Provider de la connexion, et le fournisseur de données dans la propriété dynamique "Data Provider" de l'objet Connection. Evidemment, on peut aussi passer les informations dans la chaîne de connexion. Ensuite l'astuce principale consiste à toujours travailler vis-à-vis du recordset enfant (ou secondaire) pour construire le recordset.

Recordset hiérarchique

Dans cet exemple nous allons construire un recordset hiérarchique. Ce recordset contiendra la liste de tous les auteurs, chacun de ces enregistrements contiendra la liste de tous les livres qu'ils ont écrits.

Pour affecter un recordset enfant à un recordset parent, on doit utiliser la commande Shape APPEND et préciser une clause RELATE qui permet de faire la relation entre le parent et l'enfant. Bien sur, la hiérarchie va dépendre de la structure de la base. Dans le cas de la base "biblio" je vais devoir utiliser trois tables pour construire mon recordset. Regardons le code suivant :

```
Dim cnn1 As ADODB.Connection, MonRs As ADODB.Recordset, RsLect As
ADODB.Recordset
Set cnn1 = New ADODB.Connection
cnn1.CursorLocation = adUseClient
cnn1.Provider = "MSDataShape"
cnn1.Properties("Data Provider") = "Microsoft.Jet.OLEDB.4.0"
cnn1.Open "Data Source=d:\biblio.mdb"
'autre méthode d'ouverture possible
'cnn1.open "Provider = MSDataShape;Data
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=d:\biblio.mdb ;User
Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.StayInSync = False
MonRs.Open "SHAPE {SELECT * FROM `Authors`}  AS cmdMere APPEND ((
SHAPE {SELECT * FROM `Title Author`}  AS cmdFille APPEND ({SELECT *
FROM `Titles`}  AS cmdPetFille RELATE 'ISBN' TO 'ISBN') AS
cmdPetFille) AS cmdFille RELATE 'Au_ID' TO 'Au_ID') AS cmdFille",
cnn1
'possibilité d'utiliser une grille hierarchique pour visualiser le
résultat
'Set MSHFlexGrid1.DataSource = MonRs
Set RsLect = New ADODB.Recordset
Set RsLect = MonRs("cmdFille").Value
Set RsLect = RsLect("cmdPetFille").Value
```

Dans ce cas je crée une relation composée avec ma commande Shape, ce qui revient à dire que mon recordset est composé de trois recordsets. Pour pouvoir lire les données de la table "Titles", je crée un recordset pour aller récupérer le recordset renvoyé par l'alias "cmdPetFille". Comme vous le voyez, rien de bien sorcier.

Agrégat

Les agrégats réalisés par la commande SHAPE sont sensiblement les mêmes que ceux réalisés à l'aide du SQL.

Dans cet exemple nous allons voir comment construire sans se tromper une commande Shape. L'objectif est de récupérer le nom de l'auteur et le nombre de livres qu'il a écrits. Ceci peut ressembler à l'exécution de la requête SQL suivante :

```
SELECT Authors.Author, Count([Title Author].ISBN) AS NbLivre
FROM Authors
        INNER JOIN [Title Author]
                ON Authors.Au_ID = [Title Author].Au_ID
GROUP BY Authors.Author;
```

Mais il y a une différence. Avec ma commande Shape je garde la possibilité d'accéder à la valeur du champ 'ISBN' ce que ne me permet pas la requête ci-dessus.

Nous allons utiliser le code suivant :

```
Dim cnn1 As ADODB.Connection, MonRs As ADODB.Recordset, RsLect As
ADODB.Recordset

Set cnn1 = New ADODB.Connection
cnn1.open "Provider = MSDataShape;Data
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=d:\biblio.mdb ;User
Id=Admin; Password="
Set MonRs = New ADODB.Recordset
MonRs.StayInSync = False
MonRs.Open " SHAPE {SELECT * FROM `Authors`}  AS cmdAuthor APPEND ((
SHAPE {SELECT * FROM `Title Author`}  AS cmdFille COMPUTE cmdFille,
COUNT(cmdFille.'ISBN') AS AgrISBN BY 'Au_ID') AS cmdGrISBN RELATE
'Au_ID' TO 'Au_ID') AS cmdGrISBN", cnn1
Set RsLect = New ADODB.Recordset
Set RsLect = MonRs("cmdGrISBN").Value
'affiche la valeur du compte de livre du premier enregistrement
Debug.Print RsLect!AgrISBN
Set RsLect = RsLect("cmdFille").Value
'affiche la valeur ISBN du premier enregistrement
Debug.Print RsLect!ISBN
```

Le code n'a rien de bien complexe aussi ne vais-je pas entrer dans le détail, par contre nous allons étudier la commande SHAPE.

Pour mieux comprendre regardons là écrite ainsi :

SHAPE {SELECT * FROM `Authors`} AS cmdAuthor APPEND ((SHAPE {SELECT * FROM `Title Author`} AS cmdFille COMPUTE cmdFille, COUNT(cmdFille.'ISBN') AS AgrISBN BY 'Au_ID') AS cmdGrISBN RELATE 'Au_ID' TO 'Au_ID') AS cmdGrISBN

Les règles de base sont les suivantes.

SHAPE est toujours lié avec un APPEND ou un COMPUTE. Il devra donc y avoir autant de fois SHAPE dans la commande que l'on trouvera de fois les termes Append et COMPUTE.

Chaque commande SHAPE ajoute une colonne au recordset dans lequel il est imbriqué

Les champs des relations doivent être présents dans la commande mère et la commande fille.

Je vous ai dit au début du chapitre de raisonner plutôt du coté fils. Dans ce cas, la construction se fait de la manière suivante.

1) Le but est d'obtenir le compte de ISBN, je construis donc l'agrégat de la forme COMPUTE Recordset1, Agrégat (Champs de l'agrégat) AS alias BY regroupement éventuel. Dans notre cas la commande en vert ou recordset1 est cmdFille.
2) Mon recordset1 doit contenir le champ utilisé dans l'agrégat (dans notre cas 'ISBN'). Pour nous le recordset vient de la table 'Title Author'. Mon recordset1 est donc une commande SHAPE de la forme SHAPE Requête As Alias COMPUTE Agrégat AS Alias final. Notez que

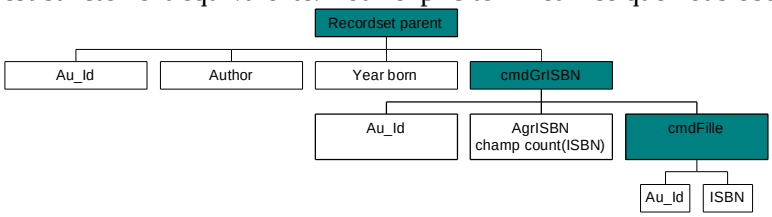
cette commande serait un recordset mis en forme valide si je ne souhaîtais pas l'imbriquer plus avant.

3) Mon Alias final doit contenir le champ à mettre en relation avec le recordset parent. Je n'ai pas besoin stricto sensu de cet alias intermédiaire, sauf pour des besoins de lecture. Je vais donc réutiliser le même comme alias dans le recordset parent. Je vais donc avoir SHAPE (recordset parent) AS Alias (facultatif) APPEND (Recordset imbriqué) AS Alias (facultatif) RELATE relation AS Alias

Comme on le voit j'utilise deux alias facultatifs et la commande

SHAPE {SELECT * FROM `Authors`} APPEND ((SHAPE {SELECT * FROM `Title Author`} AS cmdFille COMPUTE cmdFille, COUNT(cmdFille.'ISBN') AS AgrISBN BY 'Au_ID') RELATE 'Au_ID' TO 'Au_ID') AS cmdGrISBN

est strictement équivalente. Pour expliciter mieux ce que nous obtenons, visualisons la structure.



Mon recordset final est donc composé de trois recordsets.

Vers ADO.NET

Avec la nouvelle plate-forme .NET, Microsoft fournit une nouvelle architecture d'accès aux données appelée ADO.NET. Rassurez-vous tout de suite, il n'y a rien de révolutionnaire dans celle-ci et la plupart des concepts vus dans cet article restent fondamentalement les mêmes.

Nous allons parcourir les modifications engendrées par cette architecture vis-à-vis d'ADO.

N.B : Pour plus de renseignements sur les objets ADO.NET je vous invite à lire

Les objets Connection, Command et Datareader dans **ADO.NET**

Les objets DataAdapter et DataSet dans ADO.NET par leduke

DataSet

C'est probablement la modification la plus significative d'ADO.NET. L'objet recordset d'ADO a été supprimé pour être remplacé par le DataSet. Disons le tout de suite, le nouvel objet est autrement plus performant que le recordset qui souffre de plusieurs défauts, le principal étant d'utiliser un format propriétaire. L'objet DataSet stocke les données au format XML. Il permet la construction de schémas de données complexes avec des relations, plusieurs tables pouvant provenir de diverses tables etc…

On peut le voir comme un recordset issu du DataShaping, encore plus évolué. A tout point de vue, l'objet DataSet dépasse ce qu'il était possible de faire avec un recordset sous l'aspect mise en forme des données.

Dans une certaine mesure, on peut voir l'objet Recordset comme un Objet DataTable de l'objet DataSet. Le code suivant permet de remplir l'objet DataTable.

```
Dim strConnection As String = "Data Source=localhost;Initial
Catalog=Pubs;Integrated Security=True"
Dim cnn1 As SqlConnection = New SqlConnection(strConnection)
cnn1.Open(  )
Dim strSelect As String = "SELECT * FROM Authors"
Dim da As New SqlDataAdapter(strSelect, cnn1)
Dim EqvRst As New DataTable("Auteurs")
da.Fill (EqvRst)
```

Comme nous le voyons, pas de grande différence avec ADO à l'exception de l'objet DataAdapter. Une autre nouveauté intéressante est l'arrivée de la méthode ExecuteScalar qui permet de ne pas créer un DataSet lorsque la requête ne renvoie qu'une valeur.

Je vous ai donc dit qu'il n'y avait pas de révolution et pourtant, impossible de retrouver les curseurs dans ADO.NET. Rassurez-vous (surtout si vous n'avez rien compris à ce présent cours) ils ont été englobés dans deux objets DataReader (côté serveur) et DataAdapter (côté client) que nous allons voir maintenant. C'est là l'autre grande nouveauté d'ADO.NET mais elle soulève nettement moins mon enthousiasme que le DataSet.

DataReader

Cet objet est stricto sensu le curseur FireHose d'ADO. C'est dans ADO.NET le seul curseur serveur (connecté) disponible. Il garde les mêmes inconvénients que son homologue ADO, c'est-à-dire qu'il est exclusif sur la connexion qu'il utilise, qu'il ne connaît pas sa position absolue et qu'il ne permet de voir les mises à jour que sur les enregistrements non encore lus. Par contre il reste extrêmement rapide.

Comme son homologue ADO, il est fait pour faire de la lecture de données en un passage, typiquement pour le remplissage de contrôle visuel.

Il s'agit d'une grosse modification de stratégie quant à l'accès aux données. Il y a principalement deux raisons invoquées :

- Ø Les applications Web demandent une connexion minimale à la source de données
- Ø Il est plus performant d'utiliser les procédures stockées pour manipuler les données côté serveur.

Cet argumentaire est parfaitement recevable, mais à mon avis il ne prend pas le problème dans toute son ampleur.

Tout d'abord, la disparition des curseurs modifiables coté serveur entraîne la disparition de la gestion de l'accès concurrentiel par des mécanismes du SGBD. Cela va donc reporter une lourde charge sur le développeur. Ensuite il n'est pas toujours possible de stocker des procédures dans la base de données. Enfin et surtout, le fait de devoir travailler uniquement en mode déconnecté demande des utilisateurs autrement formés à la pratique de la programmation des bases de données. Nous allons voir cela dans des considérations sur l'objet DataAdapter.

DataAdapter

Voilà le nouvel enrobage du moteur de curseur client. Cet objet est défini comme étant l'intermédiaire entre la source de données et l'objet DataSet. Nous allons regarder en détail comment sont répercutées les modifications de l'objet DataSet vers la source.

Commençons par reprendre le schéma global de fonctionnement de l'objet. Il utilise une méthode Fill pour gérer le remplissage du DataSet. Cette méthode est paramétrable par l'intermédiaire de la méthode Select de l'objet DataTable. Dans l'autre sens on peut soit utiliser l'objet CommandBuilder, censé être l'équivalent, du point de vue du fonctionnement, du moteur de curseur, soit utiliser les méthodes de l'objet DataTable pour construire sa propre logique d'action.

Construire sa logique d'action

Nous allons uniquement étudier le cas des requêtes UPDATE.

Je fais un code similaire à l'appel ADO

```
Dim strConn, strSQL As String
strConn = "Provider=SQLOLEDB;Data Source=(local);Initial
Catalog=Pubs;Trusted_Connection=Yes;"
strSQL = "SELECT Au_Id, Author, [year born] FROM Authors WHERE Au_Id
= 8139"
Dim da As New OleDbDataAdapter(strSQL, strConn)
Dim tblAut As New DataTable("Auteurs")
da.Fill(tblAut)
'Je modifie une ligne comme dans l'exemple ADO
tblAut.Rows(1)("Year born") = CShort(tblAut.Rows(1)("Year born" )) +
1

'Envoi de la modification à la source.
Try
    da.Update(tblAut)
    Console.WriteLine("Modification effectuée")
Catch ex As Exception
    Console.WriteLine("Erreur lors de la modification :" &
ex.Message)
End Try
```

Ce code va provoquer une erreur. En effet, pour utiliser le DataAdapter, je **dois** avoir explicité auparavant ma logique de modification. Pour cela je vais utiliser une commande paramétrée "générique" qui indiquera à l'objet DataAdapter comment construire la requête action correspondante.

```
Dim strConn, strSQL As String
Dim cnn1 As OleDbConnection

'Je crée une connexion, un DataAdapter et un DataSet
strConn = "Provider=SQLOLEDB;Data Source=(local);Initial
Catalog=Pubs;Trusted_Connection=Yes;"
strSQL = "SELECT Au_Id, Author, [year born] FROM Authors WHERE Au_Id
= 8139"
cnn1 = New OleDbConnection(strConn)
```

```
Dim da As New OleDbDataAdapter(strSQL, cnn1)
Dim tblAut As New DataTable("Auteurs")
da.Fill(tblAut)
'Je défini la commande Update que va utiliser le DataAdapter
strSQL = "UPDATE Authors SET Au_ID = ?, Author = ?, [year born] = ?
WHERE Au_ID = ? ABD Author = ? AND [year born] = ?
Dim cmdUpdate As New OleDbCommand(strSQL, cnn1)
'Définition des paramètres en liaison au DataSet
Dim pcUpdate As OleDbParameterCollection = cmdUpdate.Parameters
Dim MonParam As OleDbParameter
MonParam =pcUpdate.Add("nAu_Id", OleDbType.Integer)
MonParam.SourceColumn="Au_Id"
MonParam.SourceVersion = DataRowVersion.Original
MonParam =pcUpdate.Add("nAuthor", OleDbType.VarChar,50)
MonParam.SourceColumn="Author"
MonParam.SourceVersion = DataRowVersion.Original
MonParam =pcUpdate.Add("nYearBorn", OleDbType.SmallInt)
MonParam.SourceColumn="Year Born"
MonParam.SourceVersion = DataRowVersion.Original
'Je modifie une ligne comme dans l'exemple ADO
da.UpdateCommand = cmdUpdate
tblAut.Rows(1)("Year born") = CShort(tblAut.Rows(1)("Year born" )) +
1
'J'envoie la modification
da.Update(tblAut)
```

Etudions ce code pour comprendre le fonctionnement.

Je définis ma requête générique en utilisant le caractère '?' pour signaler les paramètres. Dans le cas de cet exemple, je travaille en verrouillage optimiste puisque tous les champs apparaissent comme critères de la requête. Ma requête attend donc six paramètres. Pourtant je ne vais lui en passer que trois, puisque ces paramètres sont liés tout comme les propriétés Value et OriginalValue dans ADO.

A partir du moment où ces paramètres sont définis, l'objet DataAdapter saura construire une requête Update en récupérant les valeurs dans les enregistrements de mon DataTable.

Il est à noter que je pourrais procéder de plusieurs façons différentes pour augmenter la souplesse de fonctionnement au prix d'un code plus important.

Globalement, le fonctionnement caché d'ADO a été remplacé par un code, à la charge du développeur. Les bénéfices de ce changement sont nombreux. Tout d'abord je sais exactement quelle requête va être construite. Ensuite, je gère ce comportement à la création du projet et non plus à l'exécution ce qui représente un gain de temps important. Enfin, il est alors possible de faire des modifications plus complexes puisque je ne suis pas obligé d'utiliser des valeurs du DataSet comme paramètre de ma commande, de plus les requêtes relationnelles peuvent être correctement décrites.

Il y a toutefois deux inconvénients à ce système.

- Le développeur doit connaître le schéma de la source bien mieux que son homologue ADO.
- Le développeur doit surtout avoir une bien meilleure connaissance du SQL et des SGBD. En effet, les connaisseurs SQL ne seront pas sans remarquer qu'il y a une faute dans la déclaration de ma requête générique. Dans la définition de ma base, il est précisé que le champ "Year Born" peut être NULL. Si mon enregistrement avant modification ne contient pas de valeur dans ce champ, je vais passer un critère du type WHERE [year born] = NULL. Or on ne peut jamais faire de comparaison de NULL avec '='. Pour fonctionner correctement, mon code doit être.

```
 strSQL = "UPDATE Authors SET Au_ID = ?, Author = ?, [year born] = ?
WHERE Au_ID = ? ABD Author = ? AND ([year born] = ? OR ((? IS NULL)
AND ([year born] IS NULL)))
```

Néanmoins, les développeurs aguerris à la programmation des SGBD préfèreront cette nouvelle façon de procéder beaucoup plus claire et souple que la programmation du moteur de curseur client d'ADO.

Utiliser le CommandBuilder

C'est sensiblement l'équivalent du moteur de curseur. Je ne vais donc pas entrer dans les détails de son utilisation. Quelques remarques toutefois :

- Les requêtes créées sont de type optimiste strict (eq adCriteriaAllCols)
- Les modifications ne doivent concerner qu'une seule table
- La table doit contenir une clé primaire, celle-ci doit être présente dans la requête ayant rempli le DataSet

La présence de cet objet permet aux utilisateurs ayant une faible connaissance du SQL ou à ceux n'ayant pas d'informations sur le schéma de travailler comme avec ADO.

Conclusion

Voilà, j'espère que ce cours de découverte vous aura permis de mieux comprendre les mécanismes mis en jeu lors de la programmation des jeux d'enregistrements avec ADO. Comme nous l'avons vu, ce fonctionnement masqué peut engendrer de nombreux problèmes, j'espère que cet article vous aidera à les éviter.

Je n'ai fait que survoler ADO.NET et il y aurait beaucoup à dire sur le sujet. A l'exception de la disparition des curseurs serveurs modifiables, qui à mes yeux est un manque important, il faut reconnaître qu'il s'agit d'une évolution importante qui améliore profondément la programmation ADO.

Bonne programmation