PROGRAMMER ADO EN DELPHI

Ecrit par J-M RABILLOUD de www.developpez.com. Reproduction, copie, publication sur un autre site Web interdite sans autorisation de l'auteur.

Remerciements

J'adresse ici tous mes remerciements à l'équipe de rédaction de "developpez.com" et tout particulièrement à Sylvain James, Franck Farrugia, Maxence Hubiche et Samuel Dalichampt pour le temps qu'ils ont bien voulu passer à la correction et à l'amélioration de cet article.

TABLE DES MATIERES

INTRODUCTION **4**

PREAMBULE **4**

ADO: ACTIVEX DATA OBJECT **4**

DEFINITION 4
Consommateur & fournisseur de données *5*
Fournisseur & composant de service *5*
Jeu d'enregistrement (Recordset) *5*
Propriétés statiques & dynamiques (Properties) *6*
MODELE OBJET 6
LA CONNEXION 6
Le fournisseur *7*
La source de données *7*
Synchronisation *7*
Erreurs *7*
Transactions *8*
Mode d'ouverture *8*
LE RECORDSET (JEU D'ENREGISTREMENT) 9
Les Curseurs *9*
Méta-données *14*
Données (Fields) *14*
Mécanismes de base *15*
Modification d'un Recordset client *18*
FIREHOSE, UN CURSEUR PARTICULIER 22
CONSEILS POUR CHOISIR SON CURSEUR 22
SYNCHRONE VS ASYNCHRONE 22
OPERATION GLOBALE (PAR LOT) 23
Les transactions *23*
Les procédures stockées *23*
Gérée par le code (traitement par lot) *23*
LE PIEGE "L'EXEMPLE JET" 23
RECORDSET VS SQL 24
RECORDSET PERSISTANT 24
OPTIMISATION DU CODE 25
L'optimisation dans l'accès aux données *25*
L'OBJET COMMAND 25
Communication vers le SGBD *26*
Communication bidirectionnelle *26*
Création de requêtes et de procédures *26*

Collection Parameters ... *26*
TADOCONNECTION ... 27
Propriétés ... *27*
Propriétés dynamiques ... *30*
Méthodes ... *32*
Evènements ... *35*
TADOCOMMAND ... 36
Propriétés ... *36*
Méthodes ... *39*
OBJETS INCLUS: TBOOKMARK, TFIELD, TINDEXDEF ... 40
TBookmark ... *40*
TField ... *40*
TIndexDef ... *43*
TADOQUERY ... 44
Propriétés ... *44*
Propriétés dynamiques ... *51*
Méthodes ... *54*
Evènements ... *60*
TADOTABLE ... 63
Propriétés et méthodes spécifiques ... *63*
TADODATASET ... 64
TADOSTOREDPROC ... 64
EXEMPLES DE CODE ... 65
NOMBRE D'ENREGISTREMENT ET POSITION ... 65
COMPARAISON SQL VS RECORDSET ... 66
RECHERCHE SUCCESSIVES OU DIRECTIONNELLES ... 68
Les défauts de la méthode Locate ... *68*
Gestion des signets ... *68*
Programmation intrinsèque ... *71*
PROGRAMMATION ASYNCHRONE ... 74
Connection et command asynchrone ... *74*
Extractions bloquantes & non bloquantes ... *75*
Suivre l'extraction ... *76*
Gestion des modifications ... *77*
RECORDSET PERSISTANT ... 78
SYNCHRONISATION ... 79
GENERATION DE COMMANDES ... 80
Les trois valeurs de TField ... *80*
Similaire au moteur de curseur ... *81*
Commande paramétrée ... *84*
TRAITEMENT PAR LOT ... 88
Gestion standard des erreurs ... *91*
Actions correctives ... *93*
Utiliser une transaction ... *93*
ACCES CONCURRENTIEL ... 94
Les procédures stockées ... *95*
Le verrouillage ... *95*
Transactions et exclusivité ... *96*
CONCLUSION SUR ADO ... 96
ADOX: MICROSOFT ACTIVEX DATA OBJECTS EXTENSIONS ... 97
IMPORTER LA BIBLIOTHEQUE ADOX DANS DELPHI. ... 97
MODELE OBJET ... 98

RAPPELS ACCESS 98
Sécurité *98*
Paramétrage JET *98*
NOTIONS FONDAMENTALES 99
ADOX & Access *99*
Propriétaire *99*
ParentCatalog *99*
L'OBJET CATALOG 99
COLLECTIONS DE L'OBJET CATALOG 100
COLLECTION TABLES 100
COLLECTION PROCEDURES 101
COLLECTION VIEWS 101
COLLECTION GROUPS 101
COLLECTION USERS 101
L'OBJET TABLE 101
Collection Properties *102*
Collection Columns *102*
Objet Column *102*
COLLECTION INDEXES 106
Objet Index *106*
COLLECTION KEYS 107
Quelques notions *107*
Méthode Append *108*
Objet Key *108*
Exemples *109*
Conclusion sur les tables *110*
L'OBJET PROCEDURE 110
Création d'un objet procédure *111*
Modification d'un objet Procedure *111*
L'OBJET VIEW 112
CONCLUSION SUR LES OBJETS VIEW & PROCEDURE 112
GESTION DES UTILISATEURS 112
CAS PARTICULIER D'ACCESS 112
PROPRIETES ET DROITS 113
Propriétaire *113*
Administrateur *113*
Utilisateurs et groupes *113*
Héritage des objets *114*
OBJET GROUP 114
SetPermissions *114*
GetPermissions *117*
OBJET USER 118
ChangePassword *118*
GetPermissions & SetPermissions *118*
Properties *118*
EXEMPLE 118
TECHNIQUES DE SECURISATION 119
Modification *119*
Création *120*
Une autre solution : le DDL *120*
CONCLUSION SUR LA SECURITE 120
CONCLUSION 120

INTRODUCTION

Dans cet article nous allons regarder comment utiliser les accès à des sources de données en utilisant ADO (ActiveX Data Object) et ADOX (extension pour la structure et la sécurité) avec Delphi 7. Dans un premier temps nous découvrirons les technologies ADO ainsi que les concepts fondamentaux qu'il convient de connaître afin de les utiliser correctement. Dans la suite, nous verrons comment Delphi encapsule ces objets et permet de les mettre en œuvre en utilisant les composants ou le code, enfin nous regarderons quelques exemples afin de découvrir quelques astuces.

Pour ceux qui ont déjà lu d'autres articles, que j'ai écrit sur ADO pour VB, vous allez naviguer en terrain nouveau. En effet, du fait de l'encapsulation Delphi, la manipulation ADO est grandement modifiée, je dirais même améliorée pour de nombreux points. S'il n'est pas possible d'utiliser ADO sans connaître ses principes fondamentaux, il est tout aussi vain d'essayer de le manipuler comme en VB ou en C++.

PREAMBULE

Lorsqu'on est dans l'éditeur Delphi, on est en droit de se demander quel peut bien être l'intérêt de connaître les mécanismes de fonctionnement d'ADO. Il est tout d'abord évident qu'il s'agit de l'attaque d'une base de données autre qu'Interbase. Dans le cas de base de données de type Access ou Sql-Server, la meilleure stratégie consiste à utiliser les composants ADO intégrés dans Delphi depuis la version 6. Ils sont assez semblables aux contrôles standards et on peut penser qu'il n'y a rien de plus à connaître.

Le petit exemple qui va suivre va vous montrer pourquoi un tel apprentissage est indispensable pour peu que l'on veuille faire autre chose que de la simple consultation.

Imaginons une table de base de données contenant des doublons parfaits et n'ayant pas de contrainte d'unicité.

Dans ma feuille, j'ajoute un composant TADOConnection dont je paramètre la propriété ConnectionString afin de me connecter à la source de données, je crée un composant ADODataset (ou ADOTable) qui utilise cette connexion et au travers d'un composant DataSource, j'alimente un composant DBGrid. A l'exécution, ma table apparaît. Si je change une valeur de ma table dans une ligne qui n'est pas un doublon, pas de problème, la modification est répercutée dans la source de données. Par contre si cette ligne est un doublon, j'obtiens une erreur. Si je regarde mon composant ADODataset, il possède une propriété CursorLocation dont la valeur est clUseClient. En modifiant celle-ci à la valeur clUseServer, mes modifications se font sans erreur quelle que soit la ligne modifiée.

Bien sur une telle table est aberrante, mais ADO comporte de nombreux pièges que nous allons découvrir ensemble.

ADO: ActiveX Data Object

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, 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 champ (DBEdit) à un contrôle DataSource, le contrôle DataSource sera (improprement) appelé Fournisseur de données et vos zones de champs seront consommatrices.

Pourquoi est-ce impropre ?

Comme on ne voit pas le code que le contrôle DataSource 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 jeux d'enregistrements hiérarchiques ou "Microsoft OLE DB Persistence Provider" qui permet de stocker les données sous forme de fichiers.

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 (ou dataset) 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 du jeu d'enregistrement vis à vis du moteur de curseurs.

Propriétés statiques & dynamiques (Properties)

Dans le modèle ADO et a fortiori dans le modèle ADOX, de nombreux objets possèdent une collection un peu particulière : "Properties". Celle ci concerne des propriétés dites dynamiques par opposition aux propriétés habituelles des objets (statiques). Ces propriétés dépendent du fournisseur de données, elles ne sont en général accessibles qu'après la création de l'objet (et éventuellement l'application d'une méthode refresh sur la collection) voire après l'ouverture de l'objet.

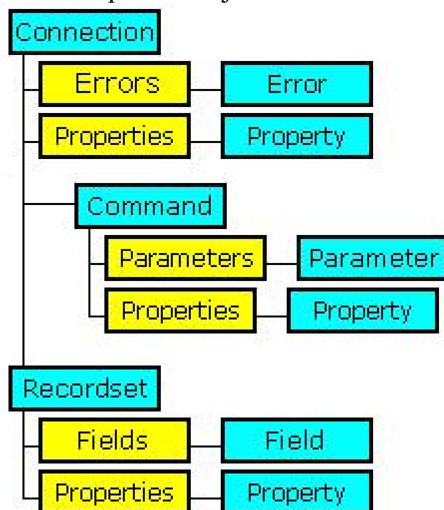
On ne peut pas accéder à une propriété statique par l'intermédiaire de la collection Properties.

Un objet **Property** dynamique comporte quatre propriétés intégrées qui lui sont propres, à savoir :

- La propriété **Name** qui est une chaîne identifiant la propriété
- La propriété **Type** qui est un entier spécifiant le type de donnée de la propriété.
- La propriété **Value** qui est un variant contenant la valeur de la propriété.
- La propriété **Attributes** qui est une valeur de type Long indiquant les caractéristiques de propriétés spécifiques au fournisseur.

Modèle objet

Le modèle objet ADO est fondamentalement assez simple tel que nous le voyons dans le schéma ci-dessous. Ce schéma est volontairement incomplet. En effet dans le cadre de ce document nous n'étudierons pas les objets Record et Stream.



La connexion

Contrairement à une idée reçue, avec ADO, il y a toujours une connexion au moins lors de la création des objets de données. Le fournisseur peut être un fournisseur de données ou un fournisseur de service, la source de données peut être distante ou locale, mais la connexion existe toujours. Elle peut être soit explicite c'est à dire déclarée par votre code et/ou vos composants, soit implicite c'est à dire créée par ADO. La connexion est une session unique d'accès à une source de données et dans le cas d'application client/serveur elle représente aussi une connexion au réseau. Globalement une connexion attend toujours au moins deux éléments, un fournisseur et une source de données. Le maintient de cette connexion active est souvent plus une question de ressource que de stratégie, comme nous le verrons dans l'étude du mode déconnecté.

Le fournisseur

ADO en lui-même n'est qu'une interface de haut niveau permettant d'utiliser OLE DB. La connexion doit donc impérativement définir le fournisseur utilisé. Il appartient au développeur de vérifier la présence de ce fournisseur.

N.B : Attention, certains empaquetages récents du MDAC de Microsoft(2.6+) ne contiennent pas certains composants ou fournisseurs comme Microsoft Jet, le fournisseur Microsoft Jet OLE DB, le pilote Desktop Database Drivers ODBC ou le pilote Visual FoxPro ODBC. Pour pouvoir utiliser pleinement les fonctionnalités ADO, il convient d'installer aussi une version plus ancienne (habituellement la version 2.1).

Le fournisseur peut être désigné sous la forme d'un DSN (Data Source Name) ou sous son nom. Je n'utiliserai pas les DSN dans cet article attendu que je déconseille vivement leur utilisation.

Dans la théorie ADO, il est conseillé de déclarer le fournisseur le plus tôt possible. En effet, vous avez remarqué que tous les objets ADO contiennent une collection "Properties". Celle-ci englobe ce que l'on appelle les propriétés dynamiques de l'objet. En fait chaque objet ADO contient quelques propriétés fixes, et de nombreuses propriétés données par le fournisseur, ce qui sous-tend que ces propriétés peuvent ne pas exister selon les fournisseurs. Elles ne sont donc pas accessibles tant que le fournisseur n'est pas défini. Ces propriétés peuvent parfois être indispensables au bon fonctionnement du code.

La source de données

La connexion doit aussi contenir une référence à la source de données. Il est évident que le fournisseur doit pouvoir manipuler cette source. Certaines sources peuvent être accédées par plus d'un fournisseur. Nous verrons avec ADOX qu'il est possible de créer la source de données par le code.

Synchronisation

Une connexion ADO peut être synchrone ou non. Les connexions ADO n'imposent jamais d'être synchrone, mais le développeur peut les forcer à être asynchrones. Je m'explique, une connexion ADO est soit demandée comme asynchrone, soit le fournisseur décide. Il faut donc toujours bien garder à l'esprit qu'une connexion peut ne pas être toujours synchrone, selon la charge du serveur par exemple. De manière générale, il est moins lourd pour le serveur de traiter des connexions asynchrones.

Erreurs

Lorsqu'une erreur ADO se produit, il y a ajout d'un objet "Error" à la collection Errors de la connexion active de l'objet, si tant est que celle-ci existe encore. Une erreur récupérable se produit également. Une erreur classique consiste à croire que la collection garde la trace de toutes les erreurs survenues lors de la session. En fait lorsqu'une erreur se produit, elle crée un ou plusieurs objets "Error" dans la connexion selon que l'erreur entraîne d'autres erreurs en cascade ou non. Si une autre erreur survient par la suite, la collection est vidée et le processus recommence. Certains passage de la programmation ADO comme la mise à jour par lot de jeu d'enregistrement déconnecté demande le traitement de ces erreurs (cf l'exemple Traitement par lot).

Transactions

Il s'agit là d'un vaste sujet, que nous allons survoler seulement pour l'instant. Pour plus de renseignement je vous conseille de lire :

- Les transactions par Henri Cesbron Lavau
- A quoi servent les transactions ? par frédéric Brouard (SQLPro)

L'utilisation des transactions est fortement recommandée, car elles assurent une cohésion à une suite de traitement nettement plus dur à réaliser avec le code client. D'autant plus qu'on peut imbriquer des transactions. Ceci permet de valider certaines modifications, tout en gardant la possibilité d'annuler l'ensemble à la fin, l'inverse présentant moins d'intérêt.

Dans certains cas, on peut remplacer les transactions par du code client, mais celui-ci doit être géré comme une transaction.

Comme vous l'avez lu dans l'article de Frédéric, le point de difficulté se situe autour de l'isolation. D'ailleurs, la grande difficulté de la programmation des SGBD repose sur la gestion des accès concurrentiels. Il faut toujours suivre les règles suivantes :

- Tout risque de lecture sale est à bannir
- La transaction doit être courte
- Dans le doute, la transaction doit échouer.

Notez enfin que certains fournisseurs ne gèrent pas tous les degrés d'isolation.

Nous verrons dans les exemples, ce qu'il faut faire et ne pas faire

Mode d'ouverture

Lorsqu'on ouvre une connexion ADO, on lui donne un certain nombre de privilèges d'accès à la source.

L'exclusivité définit une connexion qui doit être unique sur la source. Une telle connexion ne peut s'ouvrir s'il y a déjà une connexion sur la source, et aucune autre connexion ne pourra avoir lieu sur une source ayant une telle connexion.

L'exclusivité étant très pénalisante, la connexion possède une propriété Mode qui définit soit ses propres droits soit qui restreint les droits des autres connexions ; étant bien entendu qu'une connexion restrictive peut restreindre les droits d'une connexion déjà ouverte. Il convient donc d'être vigilant lors de la restriction des droits.

Le recordset (jeu d'enregistrement)

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 par 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 (clUseServer)

ADO utilise par défaut des curseurs Serveurs (mais Delphi des curseurs clients). Ceux ci fournissent des recordset qui ont moins de fonctionnalités que leurs homologues clients, mais ayant 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 demandée.
- 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 recordset et les mises à jour positionnées.

Il faut par contre garder à l'esprit :

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

Curseur côté client (clUseClient)

Les curseurs côté client présentent les avantages suivants :

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

Par contre, ils présentent deux inconvénients important :

- La surcharge de la connexion est facile à atteindre, en cas de modifications fréquentes des données
- 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 :

Le verrouillage pessimiste (ltPessimistic)

Pose un verrou sur les enregistrements dès que l'utilisateur tente de modifier une valeur (on dit à l'édition). Le verrou dure jusqu'à la validation des modifications. Si un enregistrement est déjà verrouillé, il se produit une erreur récupérable lorsqu'un autre utilisateur tente de le modifier

Le verrouillage optimiste (ltOptimistic)

Ce n'est pas un verrou stricto sensu, mais une comparaison de valeur. 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 d'un enregistrement, le numéro de version change, toutes les modifications d'autres utilisateurs ayant alors un mauvais numéro de version, celles-ci échouent.

ADO supporte aussi un mode appelé optimiste par lot (**ltBatchOptimistic**), 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 (**ltReadOnly**) qui interdit la modification des données du recordset

Quel verrou choisir?

Les considérations de ce paragraphe seront sur les curseurs côté serveurs. 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 :

```
procedure TForm1.FormActivate(Sender: TObject);
begin
with ADOConnection1 do begin
      ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=d:\biblio.mdb ;';
      Connected:=true;
      ConnectionObject.Properties['Jet OLEDB:Page
Timeout'].Value:=10000;
end;
with ADOQuery1 do begin
      SQL.Add('SELECT * FROM Authors');
      Connection:=ADOConnection1;
      CursorLocation:= clUseServer;
      LockType:= ltOptimistic;
      CursorType:=ctKeyset;
      Active:=true;
      Edit1.Text:=fields[1].Value;
end;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
with ADOQuery1 do begin
      Edit;
      fields[1].Value:=Edit1.Text;
      Post;
end;
end;
```

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.

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. Elle représente la faculté qu'a le jeu d'enregistrement de refléter la position de l'enregistrement en cours, ainsi que sa faculté à organiser et à se déplacer dans le jeu de données.

La sensibilité aux données représente la faculté qu'a le recordset de refléter les modifications, ajouts, suppressions effectués sur les données. Je vais vous présenter ces curseurs en précisant à chaque fois leurs fonctionnalités

ⓘPar position, on entend la possibilité de déterminer la position de l'enregistrement en cours au sein du jeu d'enregistrements.

Il existe quatre types de curseur :

En avant seulement (ctOpenForwardOnly**)**

Position ➨ Non

Défilement ➨ En avant

Sensibilité ➨ Pas de mise à jour des données modifiées certaines, seuls les enregistrements non accédés seront mis à jour.

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

Statique (ctStatic**)**

Position ➨ Oui

Défilement ➨ Bidirectionnel

Sensibilité ➨ Pas de mise à jour des données modifiées

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

Jeu de clé (ctKeyset**)**

Position ➨ Oui

Défilement ➨ Bidirectionnel

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

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

Dynamique (ctDynamic**)**

Position ➨ Oui

Défilement ➨ Bidirectionnel

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

Le poids lourd des curseurs. 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 on crée ces objets de façon explicite soit ils sont créés par ADO de façon implicite lors de l'ouverture du recordset. La création implicite pose deux problèmes :

- 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 connexion à la base.
- On oublie de paramétrer correctement ces objets, voire on oublie qui plus est qu'on les à 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 **Connection** et **Command** de l'objet DataSet. Cet héritage est important comme nous allons le voir dans la suite de cet article.

Un jeu d'enregistrement dont la propriété Connection vaut Nil et utilisant un curseur client est un jeu d'enregistrement déconnecté, un jeu d'enregistrement dont la propriété ActiveCommand vaut Nil est dit "Volatile".

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'enregistrement 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 DataSet. 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.

Etat, statut et déplacement

La notion d'état et de statut est toujours un petit peu complexe à appréhender. Un jeu d'enregistrement ADO possède un état (ouvert, fermé, en cours, etc…) et un statut qui lui correspond à son mode (édition, navigation, filtrage….). L'état est propre au jeu d'enregistrement tandis que le statut peut lui dépendre du statut de l'enregistrement en cours.

L'enregistrement courant possède uniquement un statut (erreur, nouveau, supprimé, etc…). Il peut subir des changements d'état mais il les transmet toujours au statut du jeu d'enregistrements auquel il appartient.

Normalement un changement d'état est toujours du à un appel explicite d'une méthode de changement d'état (Filtered, Edit…). Mais il convient de se méfier car certaines de ces méthodes induisent l'appel d'autres méthodes qui elles sont masquées, c'est le cas des changements de position courante.

Lorsque l'enregistrement courant est en mode d'édition, le changement de position va valider les modifications en cours sans appel explicite de la méthode Post. Cela est du à la transmission de l'état de l'enregistrement vers le statut du jeu de données. Je vais détailler quelque peu car les explications de l'aide sont un peu confuses.

Prenons l'exemple suivant:

```
adoRecordset:=TADOQuery.Create(Owner);
     with adoRecordset do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
          CursorLocation:=clUseClient;
          CursorType:=ctStatic;
          LockType:=ltOptimistic;
          SQL.Add('SELECT * FROM Authors');
          Active:=True;
          First;
          Edit;
          FieldValues['year born']:=1947;
          MoveBy(1);
     end;
```

Que se passe-t-il dans ce cas là ?

Lors de l'appel de la méthode First, le statut du jeu d'enregistrement est en navigation, le statut de l'enregistrement courant est 'UnModified'.

A l'appel de la méthode Edit il y a basculement de l'enregistrement et donc du jeu d'enregistrement en mode Edition. Le statut de l'enregistrement n'a pas encore été modifié.

Lors du changement de la valeur, le statut de l'enregistrement change (Modified).

Tout se passe lors de l'appel de la méthode MoveBy. Le jeu d'enregistrement vérifie s'il est en statut de navigation, c'est à dire qu'il invoque la méthode CheckBrowseMode. Cette méthode vérifie le statut de l'enregistrement en cours. Si celui-ci est différent du UnModified, la méthode Post est alors appelée, le statut de l'enregistrement change de nouveau et le jeu d'enregistrements bascule dans le mode navigation.

Vous voyez bien, que la validation des modifications peut être induite. Néanmoins une programmation évènementielle peut permettre un contrôle sur ce genre de phénomènes.

Méta-données

Le recordset à donc besoin pour fonctionner de données autres que celles renvoyés 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.

Il ne faut pas confondre ce fonctionnement avec l'appel Delphi des méta-données (GetTableNames, GetFieldNames, etc…) qui est lié à un appel spécifique de l'objet Connection (équivalent à l'appel de OpenSchema ADO).

Données (Fields)

Pour stocker les données de la requête, les recordset possèdent une collection Fields. Chaque objet Field représente un champ invoqué par la requête SELECT ayant créée 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.

Les objets Field ADO ne sont pas encapsulés dans Delphi. Il convient de ne pas confondre les champs ADO avec les objets champs fournis par Delphi.

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, l'utilisation des propriétés dynamiques des objets ADO demande 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 Connection 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 recordset côté serveur est géré par le fournisseur de données. Dans le cadre de Delphi je ne vais utiliser dans cette partie que les composants TADOConnection, TADOQuery et TADOCommand.

Création du jeu d'enregistrement

Généralement, pour ouvrir un jeu d'enregistrement, quelle que soit la méthode utilisée, vous allez consciemment ou non envoyer une requête SQL SELECT au fournisseur de données. Il 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 :

```
with ADOConnection1 do begin
     ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=d:\biblio.mdb ;';
     Connected:=true;
end;
with ADOQuery1 do begin
      Connection:=ADOConnection1;
      SQL.Add('SELECT * FROM Authors');
      CursorLocation:= clUseClient;
      LockType:= ltOptimistic;
      CursorType:= ctStatic;
      Active:=true;
     Edit1.Text:=fields[1].Value;
end;
```

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 SQL. 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 transfere 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. 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 les transmettre 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 lourd, 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 reprendre mon exemple du début avec une table sans clé, contenant des doublons parfaits (lignes dont toutes les valeurs sont identiques).

Exécutons le code suivant

```
procedure TForm1.FormActivate(Sender: TObject);
begin
with ADOConnection1 do begin
     ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\user\jmarc\bd6.mdb;';
     Connected:=true;
end;
    with ADOQuery1 do begin
     SQL.Add('SELECT * From TabSansCle');
     Connection:=ADOConnection1;
     CursorLocation:= clUseServer;
     LockType:= ltOptimistic;
     CursorType:=ctKeyset;
     Active:=true;
     //Recordset.Find('nom = ' + QuotedStr('bb'),0, 1, 1); //
     Locate('nom','bb',[]);
     Edit;
     //Recordset.Fields('numer').Value:=21; //
      FieldByName('numer').Value:=21;
     Post;
    end;
end;
```

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 à 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 identifier pour agir dessus. Néanmoins j'ai pris là un cas extrême, à la limite d'être absurde.

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

N.B : J'ai mis en commentaire une autre syntaxe de recherche basée sur la recherche ADO. Dans ce cas je devrais appeler différemment la valeur du champ (deuxième ligne de commentaire) pour que le fonctionnement soit correct. Nous verrons cela plus en détail dans la deuxième partie.

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 à 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, vous 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 Connection de l'objet DataSet. 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'obtiendrais 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 jeu d'enregistrement 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 conflits.
- 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 et 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'oublier 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 serveurs 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.

```
with ADOConnection1 do begin
      ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
      Connected:=true;
end;
with ADOQuery1 do begin
      SQL.Add('SELECT * From Authors');
      Connection:=ADOConnection1;
      CursorLocation:= clUseClient;
      LockType:= ltOptimistic;
      CursorType:= ctStatic;
      Active:=true;
      Locate('Author','Gane, Chris',[]);
      Edit;
      FieldByName('year born').Value:=1941;
      Post;
end;
```

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 telle 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 courant. 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 :

```
Locate('Author','Gane, Chris',[]);
Edit;
FieldByName('year born').Value:=FieldByName('year born').Value+1;
Post;
```

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.

```
Locate('Author','Gane, Chris',[]);
Delete;
```

Sera transformé par le moteur en :

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

Là pas de problème de critère ou de synchronisation.

NB : Dans le cas de la base biblio, une erreur sera déclenchée puisqu'il y aurait tentative de suppression en cascade

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 :

```
var Recup:integer;
begin
with ADOConnection1 do begin
      ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=c:\biblio.mdb;';
      Connected:=true;
end;
with ADOQuery1 do begin
      SQL.Add('SELECT * From Authors');
      Connection:=ADOConnection1;
      CursorLocation:= clUseClient;
      LockType:= ltOptimistic;
      CursorType:=ctStatic;
      Active:=true;
      //recordset.Properties['Update Resync'].Value:=1;
      Insert;
      FieldByName('Author').Value:='RABILLOUD, Jean-Marc';
      FieldByName('year born').Value:=1967;
      Post;
      Edit1.Text:=FieldByName('Au_Id').AsString;
end;
end;
```

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)
```

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 :

```
Recup :=FieldByName('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" (1) et le code fonctionne. Attention, ce code 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 Propriétés dynamiques - "Update Resync" de l'objet TADOQuery.

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 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 :

```
with ADOQuery1 do begin
     SQL.Add('SELECT * FROM Publishers, Titles WHERE
Publishers.PubID = Titles.PubID');
     Connection:=ADOConnection1;
     CursorLocation:= clUseClient;
     LockType:= ltOptimistic;
     CursorType:=ctStatic;
     Active:=true;
     Locate('Title','Evaluating Practice',[]);
     Edit;
     FieldByName('Name').Value:='ALPHA BOOKS';
     FieldByName('Title').Value:=FieldByName('Title').Value+'s';
     Post;
end;
```

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 sur, vue comme cela, la technique à 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 Requête action que de table concernée 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 fournit coté serveur 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. Ce curseur est exclusif sur sa connexion, 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 fermer 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 ou 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 plutôt un curseur côté serveur.
- Si vous devez manipuler un gros jeu d'enregistrement, 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 avec des transactions
- Préférez toujours l'utilisation du SQL par rapport aux fonctionnalités du DataSet.

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 évènementielle spécifique. En fait, certains évènements ADO se produiront quels que soient 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 quel 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 même 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 (traitement par lot)

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.

C'est 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 souhait de curseur. Rien ne vous dit que ce curseur existe ou est géré par le fournisseur, et qui pluis est, 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 bug 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[1] 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 clients sont :
  - Toujours statiques et ne peuvent avoir un verrouillage pessimiste

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

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

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

CursorLocation	CursorType	LockType
clUseServer	ctForwardOnly ctKeyset ctStatic	ltReadOnly
	ctKeyset	ltReadOnly ltPessimistic ltOptimistic ltBatchOptimistic
clUseClient	ctStatic	ltReadOnly ltOptimistic ltBatchOptimistic

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.

Recordset Persistant

Avec ADO il est possible de créer des jeux d'enregistrements persistants. Ceci consiste à stocker le jeu d'enregistrement sous la forme d'un fichier. ADO gère deux formats de fichier ADTG et XML. Si le fichier est amené à rester sur le poste client je vous conseille d'utiliser le format ADTG (format propriétaire), par contre si vous êtes amené à transférer ce fichier ou pour des applications Web, utilisez plutôt XML. Le fichier se manipule à l'aide des méthodes LoadFromFile et SaveToFile.

[1] 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.

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.

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, utiliser 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.

L'objet command

Cet objet sert à envoyer des requêtes SQL (DDL ou DML) vers le SGBD, ainsi qu'à la manipulation des procédures stockées. Si cet objet est fondamental dans la programmation d'ADO, c'est sûrement le moins bien utilisé, car le moins bien perçu des objets ADO. Cela vient du fait qu'ADO présente deux aspects très différents l'un de l'autre.

- Accès à une source de données "générique". Le SGBD cible n'est pas défini. On ne peut dès lors pas utilisé de SQL puisque rare sont les SGBD respectant strictement le standard SQL. C'est un cas de programmation assez rare car il demande une très bonne connaissance d'ADO pour être efficace. Dans un tel modèle de programmation l'objet Command est peu ou pas utilisable puisque même la génération de procédure stockée dans le modèle ADOX est généralement inaccessible.
- Accès à une source de données connue. C'est le cas le plus fréquent. On utilise ADO pour attaquer un SGBD que l'on connaît et dont en général on connaît le SQL. L'objet Command sert alors de moyen de communication avec le SGBD, généralement de votre application vers le SGBD, mais aussi dans l'autre sens dans le cas des procédures stockées. Nous verrons que Borland a profité de l'encapsulation pour séparer l'objet en deux ce qui augmente la lisibilité.

Même si vous ne le voyez pas, les jeux d'enregistrements encapsulent leurs propres objets Command que vous pouvez modifier indirectement sans être obligé de re instancier l'objet.

Communication vers le SGBD

L'objet Command doit toujours avoir une connexion valide et ouverte pour pouvoir agir sur la source. On distingue trois cas d'application.

- Attaque avec commande figée

Le texte de la commande est écrit en dur soit dans votre application, soit dans le SGBD. C'est de loin le cas le plus simple. Il n'y a pas de paramètres à gérer. Il est intéressant d'utiliser alors des ordres SQL compilés (ou préparés). Les procédures stockées sans paramètre et sans renvoi d'enregistrement ainsi que les requêtes stockées sans renvoi entrent dans cette catégorie.

- Attaque avec commande paramétrée

L'objet Command sert généralement de fournisseur de paramètre entre le code client et le SGBD. C'est une utilisation standard de l'objet Command. Vous appelez une requête ou une procédure présente dans la source de donnée, l'application cliente donne la valeur des paramètres et la commande est alors exécutée.

- Attaque avec commande modulable

C'est le concept le plus complexe à saisir, assez proche de l'approche ADO.NET du problème. Cela revient à mettre dans le code client des requêtes génériques, puis à faire appel à celle-ci en ne passant que les paramètres. Cette méthode étant très souple, vous trouverez un exemple dans la troisième partie.

Communication bidirectionnelle

Elle n'est normalement employée que pour les requêtes ou procédures paramétrées, stockées dans la source et donnant lieu à un renvoi d'information. Nous verrons dans l'étude détaillée de l'objet Command que le retour d'information est différent selon qu'il s'agit de paramètre ou d'enregistrement.

Création de requêtes et de procédures

L'objet Command est le seul objet présent dans le modèle ADO et dans ADOX. Dans ce dernier, il permet de créer dans la source de données des requêtes et/ou des procédures, paramétrées ou non.

Collection Parameters

Les utilisateurs d'autres SGBD qu'Access veilleront 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 sortie qui peuvent être renvoyés par une procédure. Il convient de faire attention avec ceux-ci, une connexion n'acceptant jamais plus de deux objets Command ayant des paramètres de sortie (le paramètre de retour n'ayant pas d'influence).

Les paramètres de cette collection représentent les valeurs attendues ou renvoyées par des requêtes/procédures stockées dans la source de données.

Lors de la définition d'un paramètre, il faut rigoureusement suivre les règles suivantes :

- Utiliser des noms explicites pour vos paramètres.
- Le paramètre doit toujours être correctement typé
- La propriété Size doit toujours être précisée si le paramètre est de type potentiellement variable (CHAR et VARCHAR)
- La propriété Direction doit toujours être précisée

TADOConnection

Encapsule l'objet Connection ADO. La liste suivante n'est pas exhaustive.

Propriétés

Attributes (int)

Permet de gérer la conservation des annulations ou des validations des transactions de l'objet Connection.

Constante	Valeur	Description
xaAbortRetaining	262144	Exécute un abandon avec sauvegarde : si vous appelez la commande RollbackTrans, une nouvelle transaction commence automatiquement. Certains fournisseurs ne prennent pas cette valeur en charge.
xaCommitRetaining	131072	Exécute une validation avec sauvegarde : si vous appelez la commande CommitTrans, une nouvelle transaction commence automatiquement. Certains fournisseurs ne prennent pas cette valeur en charge.

CommandCount (Del), DatasetCount (Del)

Permet de connaître le nombre d'objets Command / Dataset attachés à une connexion.

Commands(Del), DataSet (Del)

Renvoie la collection des objets Command / Dataset **actifs** de l'objet Connection

CommandTimeout (int)

Indique le temps d'attente avant que la commande appelée par la méthode Execute ne déclenche une erreur récupérable.

⚠Ne confondez pas cette propriété de l'objet Connection avec celle du même nom de l'objet Command. Les objets Command attachés à une connexion ne tiennent pas compte de la valeur de cette propriété.

Connected (Del)

Indique si la connexion est ouverte ou fermée.

ConnectionObject (Del)

Permet d'accéder directement à l'objet Connection intrinsèque ADO encapsulé dans le composant TADOConnection.

L'exemple suivant va vous montrer la technique d'utilisation.

```
begin
  with ADOConnection1 do begin
    Provider:= 'Microsoft.Jet.OLEDB.4.0;';
    DefaultDatabase:='D:\User\jmarc\tutorial\ADOX\baseheb.mdb';
    ConnectionObject.Properties['Jet OLEDB:System
database'].Value:='D:\User\jmarc\tutorial\ADOX\system.mdw';
    ConnectOptions := coAsyncConnect;
    Open('Admin','password');
  end;
end;
```

Dans cet exemple je vais définir à la volée le fichier de sécurité utilisé par Access. Pour cela je vais faire appel à une propriété dynamique de l'objet Connection "'Jet OLEDB:System database". Toutes les propriétés dynamiques ne sont pas encapsulées dans Delphi. Pour pouvoir les utiliser, je vais donc devoir atteindre l'objet Connection ADO Intrinsèque.

ConnectionString (int)

Cette propriété permet de passer une chaîne contenant tous les paramètres nécessaires à l'ouverture de la connexion plutôt que de valoriser les propriétés une à une. C'est cette propriété que l'on utilise avec le composant visuel TADOConnection, mais je vous déconseille de l'utiliser dans le code pour des raisons évidentes de lisibilité.

ConnectionTimeout (int)

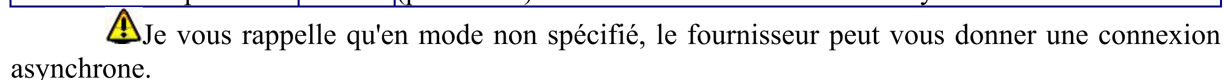
Tel que son nom l'indique. Attention, si CommandTimeout lève une exception, il n'en est pas de même systématiquement avec cette propriété. Selon les fournisseurs, il faut parfois aller tester la propriété Connected.

De même, si le fournisseur vous déconnecte pendant l'exécution du code, vous ne récupèrerez pas d'erreurs tant que vous n'essayerez pas d'utiliser la connexion.

ConnectOptions (Del)

Remplace le paramètre option de la méthode Open ADO.

Constante	Valeur	Description
coAsyncConnect	16	Ouvre la connexion en mode asynchrone. L'événement ConnectComplete peut être utilisé pour déterminer quand la connexion est disponible.
coConnectUnspecified	-1	(par défaut) Ouvre une connexion en mode synchrone.*



Je vous rappelle qu'en mode non spécifié, le fournisseur peut vous donner une connexion asynchrone.

CursorLocation (int)

Permet de définir la position du curseur des objets créés à partir de cette connexion. Par défaut il s'agit du coté client.

Errors (int)

Renvoie la collection des erreurs de la connexion. Voir l'exemple "Traitement par lot"

IsolationLevel (int)

Définit le degré d'isolation des transactions.

Constante	Valeur	Description
ilUnspecified	-1	Le serveur utilise un niveau d'isolation différent de celui demandé et qui n'a pu être déterminé..
ilChaos	16	Valeur utilisée par défaut. Vous ne pouvez pas écraser les changements en attente des transactions dont le niveau d'isolation est supérieur.
IlBrowse ou ilReadUncommitted	256	À partir d'une transaction, vous pouvez voir les modifications des autres transactions non encore engagées.
IlCursorStability ou ilReadCommitted	4096	A partir d'une transaction, vous pouvez afficher les modifications d'autres transactions uniquement lorsqu'elles ont été engagées.
ilRepeatableRead	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.
IlSerializable ou ilIsolated	1048576	Des transactions sont conduites en isolation d'autres transactions.

KeepConnection (Del)

Voilà une propriété dont il convient de se méfier. Dans le principe, toute connexion n'étant pas liée à un Dataset actif se ferme automatiquement. En mettant à Vrai cette propriété, on demande le maintien ouvert de la connexion. C'est assez dangereux car :

On a facilement tendance à oublier de fermer la connexion

Le serveur peut décider malgré tout de la fermer.

Il est souvent donc plus rentable coté client de laisser la connexion se fermer puis de la rouvrir pour envoyer les modifications.

Mode (int)

Précise les autorisations de modification de données de la connexion.

Constante	Valeur	Description
cmUnknown	0	Valeur utilisée par défaut. Indique que les autorisations n'ont pas encore été définies ou ne peuvent pas être déterminées.
cmRead	1	Lecture seule.
cmWrite	2	Ecriture seule.
cmReadWrite	3	Lecture Ecriture.
cmShareDenyRead	4	Ouverture interdite pour les utilisateurs autorisés en lecture seule.
cmShareDenyWrite	8	Ouverture interdite pour les utilisateurs autorisés en écriture seule.
cmShareExclusive	12	Ouverture interdite aux autres utilisateurs.
cmShareDenyNone	16	Empêche les autres utilisateurs d'ouvrir des connexions avec n'importe quelle permission.

L'utilisation de cette propriété est loin d'être triviale comme nous allons le voir avec l'exemple ci-dessous.

```
begin
with ADOConnection1 do begin
     ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
     Attributes:= [xaCommitRetaining];
     Mode :=cmReadWrite;
end;
with ADOConnection2 do begin
     ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
     Mode :=cmShareDenyWrite;
end;
ADOConnection1.Connected:=True;
ADOConnection2.Connected:=True;
with ADOQuery1 do begin
     SQL.Add('SELECT * From Authors');
     Connection:=ADOConnection1;
     CursorLocation:= clUseClient;
     LockType:= ltOptimistic;
     CursorType:=ctKeyset;
     Active:=true;
     Adoconnection1.BeginTrans;
     Filter:='author Like '+quotedstr('%Chris%') ;
     Filtered:=True;
     while not EOF do begin
         if FieldByName('year born').Value=Null then
         begin
            Edit;
            FieldByName('year born').Value:=1900;
            Post;
         end;
         Next;
     end;
```

```
      Adoconnection1.CommitTrans;
      while not EOF do begin
           if FieldByName('year born').Value=1900 then
           begin
               Insert;
               FieldByName('year born').Value:=Null;
               Post;
           end;
           Next;
      end;
      Adoconnection1.CommitTrans;
      Application.MessageBox('Terminé.','Look',0);
end;
end;
```

Dans ce code vous allez avoir une erreur lors de l'ouverture de la connexion 2 puisqu'elle ne peut pas restreindre les droits de la connexion 1 déjà ouverte. Imaginons maintenant que j'inverse l'ordre d'ouverture des connexions. L'erreur n'aura lieu qu'à la première rencontre de la méthode Post. En effet, le fait de se connecter après une connexion restrictive, restreint les droits de la nouvelle connexion sans avertissement. De plus, il n'y a jamais contrôle du droit de lecture et/ou écriture tant qu'ils ne sont pas réellement utilisés. L'erreur peut donc être assez difficile à cerner. Vous allez me dire, on s'en bat la rate avec un tentacule de poulpe pointillé puisqu'une erreur doit normalement se produire. Certes, mais imaginons maintenant que je referme ma connexion2 avant l'ouverture du DataSet. Seule ma connexion 1 est ouverte et pourtant je vais de nouveau avoir une erreur lors de l'appel de la méthode Post. Cela vient du fait qu'une fois que les droits ont été restreints, ils ne sont pas restaurés lors de la fermeture de la connexion restrictive.

Provider (int)

Permet de définir le fournisseur. Très utilisé lorsqu'on veut définir les propriétés dynamiques.

State (int)

Détermine l'état de la connexion.

Constante	Valeur	Description
StClosed	0	Valeur utilisée par défaut. Indique que l'objet est fermé.
StOpen	1	Indique que l'objet est ouvert.
StConnecting	2	Indique que l'objet est en train de se connecter
StExecuting	4	Indique que l'objet est en train d'exécuter une commande
StFetching	8	Indique que les lignes de l'objet sont en cours d'extraction

En lecture seule.

Propriétés dynamiques

L'ensemble des propriétés dynamiques ADO ne sont pas documentées. Il y en a beaucoup, mais je ne vais donner ici que les plus utilisées. Prenez l'habitude de les accéder au travers de ConnectionObject pour une meilleure lisibilité.

DataSource

Type chaîne. Chemin complet de la source de données

Extended Properties

Permet d'utiliser un fournisseur pour aller lire d'autres types de sources de données, typiquement Excel, texte formaté, etc…

```
with ADOConnection1 do begin
      Provider:='Microsoft.Jet.OLEDB.4.0;';
      ConnectionObject.Properties['Data
Source'].Value:='d:\configbase2.xls';
```

```
     ConnectionObject.Properties['Extended
Properties'].Value:='Excel 8.0';
     Open;
end;
```

Persist Security Info

Booléen. Indique si le fournisseur est autorisé à conserver des informations d'authentification confidentielles, comme un mot de passe, avec d'autres informations d'authentification.

Jet OLEDB:Create System Database

Utilisée avec la méthode Create de l'objet Catalog ADOX. Access uniquement.

Jet OLEDB:Database Locking Mode

Entier. Modèle à utiliser pour verrouiller la base de données. Vous remarquerez que l'on ne peut pas utiliser pour une base de données plusieurs modes d'ouverture à la fois. Le premier utilisateur qui ouvre la base de données détermine le mode de verrouillage utilisé tant que la base de données reste ouverte.

0 ➨ (JET_DATABASELOCKMODE_PAGE) Des verrous sont mis en place au niveau de la page.

1 ➨ (JET_DATABASELOCKMODE_ROW) Des verrous sont mis en place au niveau de la ligne (enregistrement).

Jet OLEDB:Database Password

Mot de passe utilisé pour ouvrir la base de données. Il est différent du mot de passe utilisateur. En effet, le mot de passe de base de données dépend du fichier alors que le mot de passe utilisateur dépend de l'utilisateur

Jet OLEDB:System database

Emplacement de la base de données du système Microsoft Jet à utiliser pour authentifier les utilisateurs. Cela annule la valeur définie dans le registre ou la clé de registre systemdb correspondante lorsque Jet OLEDB:Registry Path est utilisé. Peut inclure le chemin d'accès au fichier.

Jet OLEDB:Flush Transaction Timeout

Entier. Accessible après l'ouverture de la connexion. C'est le temps d'inactivité avant que le cache en écriture asynchrone ne soit vidé vers le disque. Ce paramètre prévaut sur les valeurs de Shared Async Delay et de Exclusive Async Delay

Jet OLEDB:Lock Delay

Entier. Accessible après l'ouverture de la connexion. Indique, en cas d'échec d'une tentative d'acquisition d'un verrou, le temps d'attente (en millisecondes) à respecter avant de faire un nouvel essai.

Jet OLEDB:Lock Retry

Entier. Accessible après l'ouverture de la connexion. Nombre autorisé de tentatives d'accès à une page verrouillée. Cette propriété s'utilise en générale avec la précédente pour gérer une boucle d'attente. Par exemple le code suivant qu'en cas de refus de connexion, celle ci va faire 10 tentatives pendant 10 secondes.

```
with ADOConnection1 do begin
      Open;
      Provider:='Microsoft.Jet.OLEDB.4.0;';
     ConnectionObject.Properties['Lock Delay'].Value:=1000;
     ConnectionObject.Properties['Lock Retry'].Value:=10;
```

```
end;
```

Jet OLEDB:Page Locks to Table Lock

Nombre de pages devant être verrouillées dans une transaction avant que Jet n'essaie de transformer les verrous en verrous de table exclusifs. La valeur zéro indique que Jet ne transformera jamais le verrouillage.

Jet OLEDB:Page Timeout

Délai d'attente, en millisecondes, que Jet doit respecter avant de vérifier si son cache est obsolète par rapport au fichier de base de données.

Méthodes

BeginTrans, CommitTrans, RollBackTrans (int)

Méthode utilisée pour la gestion des transactions. BeginTrans peut renvoyer un entier qui donne le niveau d'imbrication. Comme nous allons en voir dans de nombreux exemples, je ne m'étendrai pas dessus ici.

Cancel

Annule le dernier appel **Asynchrone** de la connexion. Ce n'est pas tellement que la méthode soit très utile mais elle va me permettre de vous parler d'autre chose. Regardons le code suivant :

```
var strUpdate : WideString;
compteur:Integer;
begin
with ADOConnection1 do begin
     ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
     Connected:=True;
     strUpdate:='UPDATE Publishers SET City = ' +
QuotedStr('Grenoble') + ' WHERE City = ' + QuotedStr('New York');
     BeginTrans;
     Execute(strUpdate,cmdText,[eoAsyncExecute]);
     for compteur:=1 to 100 do
     if State = [stExecuting] then
     begin
       Cancel;
       RollBackTrans;
     end
     else
       CommitTrans;
end;
end;
```

Bien que techniquement juste, ce code va aléatoirement déclencher des erreurs. Cela vient du fait que les transactions sur des connexions asynchrones doivent être gérées différemment comme nous le verrons dans l'étude des évènements de cet objet. Si vous enlevez les méthodes de transaction, ce code fonctionne correctement.

Close (int)

Ferme la connexion. Utilisez plutôt Connected:=False.

Execute (Del)

De la forme Connection.Execute(CommandText; CommandType; ExecuteOptions)

Permet d'exécuter une commande volatile à partir de l'objet Connection (voir l'exemple au dessus). A ne surtout pas confondre avec la fonction Execute de l'objet Connection ADO sous-jacent. La commande est volatile car elle n'est pas récupérable après exécution.

Le code Recup := Adoconnection1.Commands[0].CommandText; provoquera une erreur d'exécution.

Execute (int)

De la forme Connection.Execute(CommandText; RecordsAffected; ExecuteOptions)

Exécute aussi une commande volatile avec possibilité de récupérer le nombre d'enregistrements affectés par la commande.

Ces deux méthodes peuvent renvoyer un jeu d'enregistrement avec le curseur par défaut.

Open (int)

De la forme Connection.Open(UserId ; Password)

Permet l'ouverture de la connexion en fournissant les paramètres du login. Stricto sensu il n'y a pas de différence à utiliser cette méthode plutôt que la propriété Connected en mode synchrone, mais il faut utiliser Open en mode asynchrone

OpenSchema (int)

Permet de récupérer des méta-données contenant les informations de schéma de la source de données. Cette méthode est de la forme suivante: *connection***.** OpenSchema **(*QueryType***, ***Criteria***, ***SchemaID, DataSet*)**

Où QueryType donne le type d'information à récupérer, Criteria définit les contraintes sur ces informations

SchemaID n'est utilisé que si QueryType est adSchemaProviderSpecific.

Pour avoir la liste complète des possibilités reportez-vous à l'aide Delphi (Méthode OpenSchema (ADO)).

Comme nous le voyons, les informations renvoyées se trouvent dans un Dataset. Pour voir le principe de la méthode, observons le code suivant :

```
var dsSchemaTable, dsSchemaChamp : TADOdataset;
Fichier : TextFile;
compteur : Integer;
begin
with ADOConnection1 do begin
     ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
     dsSchemaTable:=TADODataset.Create(Owner);
     adoconnection1.OpenSchema(siTables,EmptyParam, EmptyParam,
dsSchemaTable);
     datasource1.DataSet:=dsSchemaTable;
     dsSchemaChamp:=TADODataset.Create(Owner);
     dsSchemaTable.First;
     AssignFile(Fichier,'d:\user\confbase.txt');
     Rewrite(Fichier);
     while not dsSchemaTable.Eof do begin
           writeln(Fichier,'Nom : ' +
dsSchemaTable.FieldByName('TABLE_NAME').Value + ' type: ' +
dsSchemaTable.FieldByName('TABLE_TYPE').Value + ' description : ' +
dsSchemaTable.FieldByName('DESCRIPTION').Value) ;

adoconnection1.OpenSchema(sicolumns,VarArrayOf([Unassigned,
Unassigned, dsSchemaTable.FieldByName('TABLE_NAME').Value,
Unassigned]), EmptyParam, dsSchemaChamp);
           for compteur:=1 to dsSchemaChamp.FieldCount-1 do

write(Fichier,dsSchemaChamp.Fields.FieldByNumber(compteur).DisplayTe
xt + chr(vk_tab));
           dsSchemaTable.Next;
```

```
      end;
      closefile(Fichier);
end;
end;
```

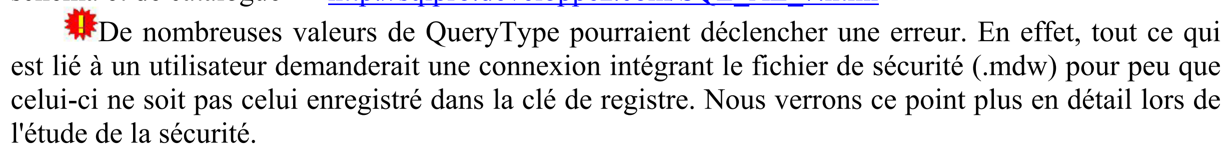
Ce code récupère le schéma des tables dans le dataset dsSchemaTable, et pour chaque table extrait le schéma des colonnes. Observons la ligne suivante :

```
adoconnection1.OpenSchema(sicolumns,VarArrayOf([Unassigned,
Unassigned, dsSchemaTable.FieldByName('TABLE_NAME').Value,
Unassigned]), EmptyParam, dsSchemaChamp);
```

Si nous regardons l'aide nous trouvons dans le tableau :

QueryType	Criteria
siColumns	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME

C'est donc pour obtenir la liste des champs d'**une** table que VarArrayOf([Unassigned, Unassigned, dsSchemaTable.FieldByName('TABLE_NAME').Value, Unassigned]). Les constantes QueryType n'étant pas toujours très claires, et de manière générale pour comprendre les notions de schéma et de catalogue ➨ http://sqlpro.developpez.com/SQL_AZ_7.html



De nombreuses valeurs de QueryType pourraient déclencher une erreur. En effet, tout ce qui est lié à un utilisateur demanderait une connexion intégrant le fichier de sécurité (.mdw) pour peu que celui-ci ne soit pas celui enregistré dans la clé de registre. Nous verrons ce point plus en détail lors de l'étude de la sécurité.

Evènements

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 évènements 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 évènements 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 Dataset.Open. (ou Active:=True)

TwillExecuteEvent = procedure (Connection:TADOConnection ; varCommandText:WideString ; varCursorType:TcursorType ; varLockType:TADOLockType ; varCommandType:TcommandType ; varExecuteOptions:TexecuteOption ; varEventStatus:TeventStatus ; constCommand: _Command ; constRecordset:_ Recordset) of object;

TExecuteCompleteEvent = procedure (Connection:TADOConnection ; RecordsAffected:Integer ; constError:Error ; var EventStatus:TeventStatus ; constCommand:_ Command ; constRecordset:_ Recordset) of object;

Connection doit toujours contenir une référence à une connexion valide.

Si l'événement est dû à Connection.Execute, _Command et _Recordset sont à "Nil".

Si l'événement est dû à Command.Execute, _Command est la référence de l'objet Command et _Recordset vaut "Nil".

Si l'événement est dû à la méthode Open du Dataset, _Command vaut "Nil" et _Recordset est la référence de l'objet Recordset.

TADOCommand

Encapsule une partie de l'objet ADO Command. Comme nous l'avons déjà vu, l'objet Command ADO sert à :

- Exécuter des commandes SQL vers le SGBD
- Utiliser les procédures stockées
- Créer des jeux d'enregistrements (pas recommandé).

Dans Delphi, il existe deux composants pour cela, TADOCommand que nous allons voir ici et TADOStoredProc que nous verrons après.

Propriétés

CommandObject (int)

Renvoie vers l'objet Command ADO intrinsèque

CommandText (int)

Texte de la commande à exécuter

ExecuteOptions (int)

ExecuteOptionEnum		
Constantes	**Valeur**	**Description**
eoAsyncExecute	16	Indique que la commande doit s'exécuter en mode asynchrone
eoAsyncFetch	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.
eoAsyncFetchNonBlocking	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
eoExecuteNoRecords	128	Une commande ou une procédure stockée qui ne renvoie rien. Si des lignes sont récupérées, elles ne sont pas prises en compte et rien n'est renvoyé.

Ce paramètre doit être passé si l'on travaille en mode Asynchrone.
Vous trouverez un exemple dans l'utilisation asynchrone de TADOQuery.

ParamCheck (Del)

Cette propriété permet un apport de souplesse par rapport à la manipulation classique ADO. En effet, lors d'un changement de la propriété CommandText il y a génération automatique des paramètres de la command si ParamCheck est vrai. Dans le cas ou vous utilisez d'autres paramètres il faut mettre cette propriété à False. Cela évidemment uniquement si votre commande contient des paramètres qui ne sont pas des paramètres ADO.

Parameters (int)

Permet d'accéder à la liste des paramètres ADO de la commande. Attention à ne pas vous mélanger entre une requête paramétrée (requête présente dans le SGBD et retournant ou non un jeu d'enregistrement) et une procédure stockée (suite d'ordre SQL présent dans le SGBD). Ici nous allons

voir les paramètres de requêtes paramétrées. Ces objets sont des Tparameter dont voici les propriétés les plus utiles.

Attributes

Précise si le paramètre accepte les valeurs signées (psSigned), binaires longues (psLong) ou NULL (psNullable).

Direction

Non utilisé pour les requêtes. Peut prendre les valeurs :

Constante	Valeur	Description
adParamUnknown	0	La direction du paramètre est inconnu.
adParamInput	1	Valeur utilisée par défaut. Paramètre d'entrée.
adParamOutput	2	Paramètre de sortie.
adParamInputOutput	3	Paramètre d'entrée et de sortie.
adParamReturnValue	4	Valeur de retour

Name

Définit le nom du paramètre. N'est pas obligatoire. En effet un paramètre non nommé apparaît sous la forme '?' et est accessible par sa position ordinale

NumericScale

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

Size

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

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

Type

Comme son nom l'indique !

Value

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

Et quelques méthodes

AppendChunk

Permet d'accéder aux textes ou binaires longs.

Assign

Permet de dupliquer les propriétés d'un paramètre dans un autre paramètre.

Comme un exemple vaut mieux qu'un long discours, étudions le cas suivant.
Soit la requête action suivante stockée dans Access sous le nom "ReqModif".

```
PARAMETERS Annee integer; UPDATE Authors SET Authors.[Year Born] = 1972
WHERE Authors.[Year Born]=[Annee];
```

Pour l'utiliser en partant du code :

```
var adoComModif: TADOCommand;
begin
     with ADOConnection1 do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
          Connected:=true;
          adoComModif:=TADOCommand.Create(owner);
          with adoComModif do begin
               Connection:=adoconnection1;
```

```
               Parameters.Clear;
               CommandText:='ReqModif';
               CommandType:=cmdStoredProc;
              Parameters.CreateParameter('Annee', ftInteger,
pdinput, 4, 1941);
               Execute;
          end;
     end;
end;
```

Ce n'est pas bien compliqué. On pourrait aussi passer par la méthode AddParameter. Vous voyez bien que dans ce cas, je ne sais pas combien de modifications ont été effectuées. Pour pouvoir récupérer ce paramètre je dois exécuter l'autre forme de la méthode Execute.

```
var adoComModif: TADOCommand;
nbChangement: Integer;
begin
     with ADOConnection1 do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
          Connected:=true;
          adoComModif:=TADOCommand.Create(owner);
          with adoComModif do begin
                Connection:=adoconnection1;
                CommandText:='ReqModif';
                CommandType:=cmdStoredProc;
                Execute(nbChangement,VarArrayOf([1941]));
          end;
     end;
end;
```

Comme cette forme d'écriture est plus simple, puisqu'on ne passe pas par la collection Parameters, certains développeurs ont tendance à la préférer. Il faut quand même voir qu'on travaille là directement sur l'objet Command ADO. Il y a une perte importante tant au niveau de la lisibilité que de la sécurité à utiliser ce style d'écriture.

Vous noterez aussi que bien qu'Access ne gère pas les procédures stockées, le CommandType est cmdStoredProc. Ceci permet à ADO de savoir que le texte de la commande est dans le SGBD et qu'il attend des paramètres sans renvoyer d'enregistrements.

Il est aussi possible de gérer la commande et les paramètres par le code.

```
var adoComModif: TADOCommand;
paramAnnee: TParameter;
begin
     with ADOConnection1 do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
          Connected:=true;
          adoComModif:=TADOCommand.Create(owner);
          with adoComModif do begin
               Connection:=ADOConnection1;
               CommandText:='PARAMETERS Annee integer; UPDATE
Authors SET Authors.[Year Born] = 1972 WHERE Authors.[Year
Born]=[Annee];';
               CommandType:=cmdText;
               Prepared:=True;
               ParamAnnee:= Parameters.AddParameter;
               with ParamAnnee do begin
                    DataType:=ftInteger;
                    Name:='Annee';
                    Direction:=pdInput;
               end;
               BeginTrans;
               Parameters.ParamByName('Annee').Value:=1941;
               Execute;
               CommitTrans;
```

```
                Execute(VarArrayOf([1939]));
            end;
        end;
end;
```

J'utilise une autre technique d'écriture afin de vous montrer le nombre de choix possibles. En toute rigueur ce code est très lourd. On ne trouve jamais ce type de notation dans ADO mais uniquement dans ADOX, vous trouverez habituellement la notation :

```
var adoComModif: TADOCommand;
begin
     with ADOConnection1 do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
          Connected:=true;
          adoComModif:=TADOCommand.Create(owner);
          with adoComModif do begin
               Connection:=ADOConnection1;
               CommandText:='PARAMETERS Annee integer; UPDATE
Authors SET Authors.[Year Born] = 1972 WHERE Authors.[Year
Born]=[Annee];';
               CommandType:=cmdText;
               Prepared:=True;
               Execute(VarArrayOf([1941]));
          end;
     end;
end;
```

Pour les paramètres, nous en resterons là pour l'instant.

Prepared (Int)

Permet de préparer la commande avant de l'exécuter. Il y a toujours une perte de temps lors de la préparation de la commande, mais celle-ci est beaucoup plus performante en cas d'appel multiple.

Méthodes

Assign (Del)

Duplique un objet Command

Execute (int & del)

Existe sous trois syntaxes différentes que nous avons vues dans les exemples.

Objets inclus: TBookmark, Tfield, TIndexDef

Avant de nous lancer dans l'étude des jeux d'enregistrement, nous allons regarder trois "objets" indissociables de ceux-ci, dont la compréhension est indispensable.

TBookmark

Il est possible de marquer certains enregistrements afin de pouvoir les localiser rapidement. On utilise pour cela des signets (BookMark). Ces signets sont en fait des pointeurs d'enregistrement. Une des applications standards consiste à sauvegarder un tableau de pointeurs afin de pouvoir récupérer un jeu d'enregistrement particulier. La valeur renvoyée est une chaîne, il est dès lors conseillé d'utiliser un objet liste pour construire son tableau de signets.

Si un enregistrement marqué est supprimé par un autre utilisateur, il est fréquent que le signet désigne alors un autre enregistrement sans déclencher d'erreur. Ceci est une source de Bug non négligeable.

TField

Représente un objet champ générique. On n'utilise jamais directement TField mais plutôt ses descendants.

On peut se représenter un jeu d'enregistrement comme un tableau d'objet TField à deux dimensions ayant :

- Une cohérence horizontale : Chaque ligne de champs représente un enregistrement, il n'est pas possible d'en dissocier les valeurs
- Une cohérence verticale : A une position ordinale donnée, les champs ont tous les mêmes attributs.

Concepts généraux

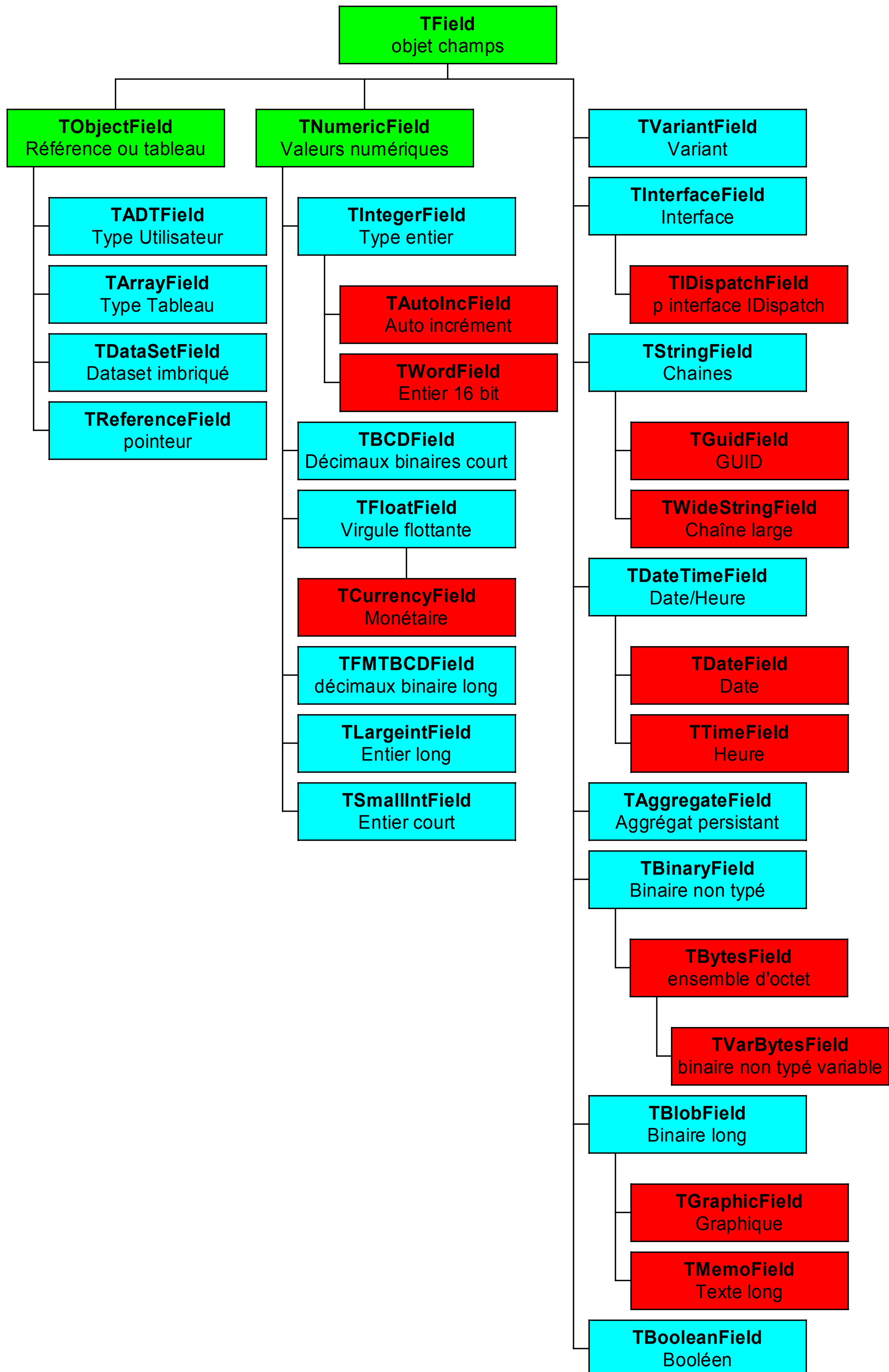
Champ dynamique. Par défaut les champs créés par un jeu d'enregistrements sont toujours des champs dynamiques. Cela veut dire que les champs représentent des champs de la source de données. Leur durée de vie est celle du jeu d'enregistrement. Un certain nombre de manipulation ne sont pas possibles sur les champs dynamiques. Leur intérêt est par contre certain pour des applications flexibles.

Champ persistant. Il est possible de définir ces champs à la création. On perd évidemment en dynamique de programmation, mais on augmente fortement la souplesse des champs ainsi créés.

Les champs renvoyés par les objets ADO n'étant pas différents des autres je ne vais pas détailler trop les concepts des objets champs, mais je vous recommande vivement de vous y intéresser plus avant afin de profiter pleinement des possibilités offertes.

Regardons maintenant la hiérarchie des descendants TField.

Hiérarchie des objets champs



Les descendants TField correspondent sensiblement à leur type de données.

Propriétés

Je ne vais détailler ici que les propriétés/méthodes utilisées dans cet article

AsFloat, AsString, etc…
Convertir la valeur d'un champ dans un autre type de données.

CurValue
Renvoie la valeur réelle du champ lors des erreurs liées à la mise à jour différée (optimiste).

Dataset
Identifie le dataset qui contient le champ

DefaultExpression
Spécifie une valeur par défaut pour le champ

DisplayName
Fournit le nom du champ dans une chaîne

DisplayText
Fournit la valeur dans une chaîne. Le champ apparaît formaté.

FieldKind
Permet d'identifier si le champ est de type calculé, agrégé, etc.….

Index
Renvoie ou définit la position ordinale du champ dans le jeu d'enregistrement. Sans effet sur la source de données.

IsIndexField
Permet de savoir si le champ fait partie d'un index.

IsNull
Permet de savoir s'il y a une valeur dans le champ

NewValue
Détermine la valeur en attente dans le cache.

OldValue
Définit la valeur initiale du champ. En lecture seule.
Les diverses propriétés values seront étudiées lors des mises à jour par lot.

Required
Précise si la saisie d'un champ est obligatoire

ValidChars
Permet de restreindre les caractères de saisie autorisés.

Value
Valeur du champ

Méthodes

AssignValue
Permet d'affecter une valeur à un champ tout en la transtypant.

Clear
Affecte la valeur NULL au champ.

IsValidChar
Détermine si un caractère est valide pour ce champ.

Evènements

OnChange
Se produit après l'envoie d'une valeur vers le cache.

TIndexDef

Cet objet représente un index sur une table ou un dataset. Attention, un index n'est pas une clé. Un index permet d'améliorer les recherches ou les tris, les clés sont des contraintes.

Il ne faut pas non plus abuser de l'indexation des champs, sous peine d'alourdir la base ou les composants de données.

Dans ADO on utilise les index sur des tables, en général obtenu par l'objet TADOTable. Seuls ceux-ci permettent d'utiliser la méthode de recherche Seek.

TADOQuery

Ce composant encapsule l'objet Recordset ADO. Son existence est très intéressante si vous avez l'habitude de manipuler l'objet Recordset ADO, sinon il vaut mieux utiliser TADOTable ou TADODataSet. Ce passage va être assez technique et tout ce que vous y trouverez sera aussi vrai pour le composant TADODataset.

Propriétés

Active (Del)

Ouvre le jeu d'enregistrements. Equivaut à l'appel de la méthode Open.

BOF, EOF (Del)

Détermine des positions particulières du DataSet. Si BOF est vrai, l'enregistrement actif est le premier, si EOF est vrai c'est le dernier. A ne surtout pas confondre avec les mêmes propriétés ADO ou BOF est avant le premier enregistrement et EOF après le dernier.

BookMark (Int)

Permet de gérer un signet sur l'enregistrement en cours.

CacheSize (Int)

Définit ou renvoie le nombre d'enregistrements placés dans le **Cache**. Il est souvent utile de faire un test de performance pour avoir une idée du réglage, tel que celui du code suivant.

```
procedure TForm1.Button1Click(Sender: TObject);
var temps, cmpt0,cmpt10,cmpt100 :Integer;
adoRecordset:TADOQuery;
adoConnect:TADOConnection;
strAuthor:WideString;
begin
  adoConnect:=TADOConnection.Create(Owner);
  with adoConnect do begin
    ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\biblio.mdb;';
    CursorLocation:=clUseServer;
    Connected:=True;
  end;
  adoRecordset:=TADOQuery.Create(Owner);
  with adoRecordset do begin
    Connection:=adoConnect;
    SQL.Add('SELECT * FROM Authors');
    Active:=True;
    First;
    temps:=windows.GetTickCount;
    while not Eof do begin
      strAuthor:=Fields[0].Value;
      Next;
    end;
    cmpt0:=windows.GetTickCount-temps;
    CacheSize:=10;
    First;
    temps:=windows.GetTickCount;
    while not Eof do begin
      strAuthor:=Fields[0].Value;
      Next;
    end;
    cmpt10:=windows.GetTickCount-temps;
```

```
    CacheSize:=100;
    First;
    temps:=windows.GetTickCount;
    while not Eof do begin
      strAuthor:=Fields[0].Value;
      Next;
    end;
    cmpt100:=windows.GetTickCount-temps;
    Close;
  end;
  adoConnect.Close;
  MessageDlg('Cache=1 : ' + IntToStr(cmpt0) + Chr(vk_return) +
'Cache=10 : ' + IntToStr(cmpt10) + Chr(vk_return) + 'Cache=100 : ' +
IntToStr(cmpt100),mtInformation,[mbOK],0);
end;
```

CanModify (Del)

Permet de savoir si le jeux d'enregistrement est modifiable (si le curseur n'est pas ReadOnly).

Connection (int)

Définit la connexion utilisée

CursorLocation, CursorType, LockType (int)

Je ne vais pas revenir dessus. Toutefois attention, la valeur par défaut de CursorLocation est clUseClient et le type par défaut est ctKeySet. Or un curseur client est toujours statique, il y a donc une légère contradiction dans les valeurs par défaut. Celle-ci n'est pas bien grave puisque le curseur utilisé sera effectivement statique.

DataSource (Del)

Permet d'utiliser un composant DataSource comme fournisseur dynamique de paramètre à votre recordset ADO ou du moins à son objet Command de construction.

DefaultFields (Del)

Indique si les champs d'un jeu d'enregistrements sont dynamiques ou persistants.

EnableBCD (Del)

Force le stockage de valeur numérique décimale comme Float ou comme Currency

Filter (Del)

Définit un filtre sur le jeu de données

FieldCount (Int)

Renvoie le nombre de champs présents dans le jeu d'enregistrements.

FieldDefList (Del)

Renvoie la liste des définitions des champs du jeu d'enregistrements. S'utilise principalement dans une création dynamique de composant

Fields (Int), AggFields (Del)

Renvoie la collection des champs du jeu d'enregistrements. Dans un jeu de champ dynamique, Fields contient tous les champs alors que dans un jeu persistant, c'est Fields et AggFields qu'il faut

utiliser, ces deux collections étant exclusives. Ceci revient à dire qu'aucun champ ne peut appartenir en même temps aux deux collections.

Dans la collection Fields, la position ordinale est définie par la source de donnée (la requête) pour des champs dynamiques et à la création pour des champs persistants.

Selon la valeur de la propriété ObjectView, la collection Fields des ensembles d'enregistrements maître/détail peut être hiérarchisée ou non.

En général on préfère utiliser la méthode FieldByName ou la propriété FieldValues pour utiliser les valeurs de champs, mais la collection Fields reste indispensable pour utiliser un code générique.

FieldValues (Del)

Renvoie un tableau Variant des valeurs de champs. Cette propriété est très pratique mais je vais y mettre un bémol. Le tableau renvoyé est un tableau de variant contenant les valeurs de tous les champs (données, agrégat, référence)

- Un tableau de variant est une structure extrêmement lourde. De plus la conversion des valeurs peut être assez lourde. Si votre jeu d'enregistrement est volumineux, le coût en temps peut être important.
- La programmation des variants demande de l'habitude. Il convient donc d'être assez vigilant.

La programmation par FieldValues est par contre très simple, voilà un exemple classique

```
adoRecordset.SQL.Add('SELECT [n° Commande],[Date Commande] FROM
Commandes');
  adoRecordset.Active:=True;
  adoRecordset.Locate('N° Commande',10273,[loCaseInsensitive]);
  VarRecup:=adoRecordset.FieldValues['Date Commande'];
  if vartype(varrecup)= varDate then begin
    adoRecordset.Edit;
    adoRecordset.FieldValues['Date
Commande']:=IncDay(adoRecordset.FieldValues['Date Commande'],2);
    adoRecordset.Post;
  end;
```

Pour mémoire la fonction IncDay augmente la date du nombre de jour spécifié en deuxième paramètre.

Filter (Del)

Les développeurs ADO doivent faire attention à ne pas confondre la propriété Filter ADO avec celle-ci. En effet, dans Delphi, deux propriétés Filter et FilterGroup remplacent la propriété Filter ADO.

Permet de filtrer les enregistrements d'un Dataset selon les critères spécifiés. Le Dataset filtré se comporte comme un nouvel objet DataSet. En général cette propriété ne s'utilise pas sur les Dataset 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 doit être de type filtre SQL, c'est à dire qu'il s'utilise comme une clause WHERE sans le WHERE.

```
procedure TForm1.FormActivate(Sender: TObject);
var adoRecordset:TADOQuery;
begin
     adoRecordset:=TADOQuery.Create(Owner);
     with adoRecordset do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
          CursorLocation:=clUseClient;
          CursorType:=ctStatic;
          LockType:=ltReadOnly;
          SQL.Add('SELECT * FROM Authors');
          Active:=True;
          DataSource1.DataSet:=adoRecordset;
     end;
end;

procedure TForm1.Button1Click(Sender: TObject);

begin
     DataSource1.DataSet.Filter :='[year born] <> null';
     DataSource1.DataSet.Filtered:=True;
end;
```

N'oubliez pas que Filtered doit toujours être faux avant une modification du filtre.

Filtered (Del)

Active ou désactive le filtre

FilterGroup (Del)

Cette propriété n'est utilisée que si le Dataset est en LockType ltBatchOptimistic.
Elle sert à filtrer les enregistrements selon leur statut de mise à jour.
Peut prendre une des valeurs prédéfinies suivantes :

Constante	Valeur	Description
fgNone	0	Enlève le Filtre ce qui rétabli le recordset d'origine. On obtient le même résultat en affectant la valeur false à la propriétéFiltered.
fgPendingRecords	1	Filtre les enregistrements en attente de mise à jour (UpdateBatch ou CancelBatch)
fgAffectedRecords	2	Filtre les enregistrements affectés par le dernier appel d'une méthode Delete, Resync, UpdateBatch ou CancelBatch.
fgFetchedRecords	3	Filtre les enregistrements présents dans le cache
fgPredicate	4	Filtre utilisé pour ne montrer que les lignes supprimées.
fgConflictingRecords	5	Filtre les enregistrements n'ayant pas pu être modifiés lors de la dernière mise à jour

J'ai lu dans un article sur le Web que la combinaison de ces deux propriétés différentes en Delphi donne un résultat différent des filtres ADO intrinsèque. Visiblement l'auteur n'a pas bien compris le fonctionnement des filtres ADO puisque c'est très exactement identique.

Regardons le code suivant.

```
var adoRecordset:TADOQuery;
begin
     adoRecordset:=TADOQuery.Create(Owner);
     with adoRecordset do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
          CursorLocation:=clUseClient;
          CursorType:=ctStatic;
          LockType:=ltBatchOptimistic;
          SQL.Add('SELECT * FROM Authors');
          Active:=True;
```

```
            Filter:='[year born] <> null';
            Filtered:=True;
            First;
            while not Eof do begin
                  if FieldByName('Year born').Value<1945 then
                     begin
                          Edit;
                          FieldByName('Year
born').Value:=FieldByName('Year born').Value+1;
                          Post;
                     end;
                  Next;
            end;
            FilterGroup :=fgPendingRecords;
            DataSource1.DataSet:=adoRecordset;
      end;
end;
```

Les données visibles sont bien celles des deux filtres ce qui est logique puisque le deuxième filtre affiche les valeurs modifiées déjà filtrées par le premier filtre.

Notez aussi que donner la valeur fgNone à la propriété FilterGroup supprime aussi le filtre de la propriété Filter.

FilterOptions

Précise si le filtre autorise le filtrage partiel et s'il est sensible à la Casse (Minuscule/Majuscule).

Found (Del)

Cette propriété renvoie Vrai si la dernière recherche a abouti. Notez que le comportement lors d'une recherche Delphi est différent de la recherche intrinsèque ADO.

IndexFieldCount, IndexFields, IndexName (Del)

Ces propriétés permettent de jouer sur les index dans une certaine mesure. En effet il est possible par leurs biais de spécifier l'index utilisé lors d'une recherche sur index (Seek).

MarshalOptions (int)

Coté client uniquement.

Spécifie si tous les enregistrements ou seuls les enregistrement modifiés sont renvoyés vers le serveur.

MaxRecords (int)

Permet de limiter le nombre d'enregistrements renvoyés par une requête. Pas de limitation quand la propriété vaut zéro. Dans certains cas, cette propriété permet de protéger le client contre le rapatriement d'un nombre trop important d'enregistrements.

ObjectView (Del)

Définit si la collection Fields se présente sous forme hiérarchisée ou non.

RecNo (Del)

Equivalent de la propriété AbsolutePosition ADO. Renvoie la position de l'enregistrement actif si elle peut être déterminée, sinon renvoie 0.

RecordCount (Int)

Renvoie le nombre d'enregistrement d'un Dataset lorsque celui ci peut être déterminé, sinon renvoie -1. Pour cette propriété comme pour la précédente, les curseurs "en avant seulement" ne peuvent déterminer la position en cours ni le nombre d'enregistrement. Elles renverront les valeurs que je vous ai signalées et non une erreur comme vous pourrez parfois le lire. Cette erreur qui apparaît parfois vient d'une erreur de curseur et non de la propriété RecordCount. Regardons le code suivant :

```
begin

  adoRecordset:=TADOQuery.Create(Owner);
  with adoRecordset do begin
    ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\biblio.mdb;';
    CursorLocation:=clUseServer;
    CursorType:=ctOpenForwardOnly;
    LockType:=ltReadOnly;
    SQL.Add('SELECT * FROM Authors');
    Active:=True;
    Locate('year born',1938,[loCaseInsensitive]);
    intRecup:= RecordCount;
    MessageDlg(IntToStr(intRecup),mtInformation,[mbOK],0);

  end;
```

Ce code déclenchera une erreur sur la ligne `intRecup:= RecordCount;` ce qui fait que l'on pense que la propriété RecordCount renvoie une Erreur. En fait cela est du à l'appel de Locate. Un curseur en avant seulement ne gère que Next comme méthode de déplacement.

Recordset (Int)

Désigne le recordset sous-jacent de l'objet Query.

RecordsetState (Del)

Equivalent de la propriété State de l'objet Connection.

RecordStatus (Int)

Permet de connaître l'état de la ligne en cours. Le traitement des états est souvent indispensable lors des mises à jours par lot. Peut prendre une ou plusieurs des valeurs suivantes :

Constante	Valeur	Description
rsOK	0	Mise à jour de l'enregistrement réussie
rsNew	1	L'enregistrement est nouveau
rsModified	2	L'enregistrement a été modifié
rsDeleted	4	L'enregistrement a été supprimé
rsUnmodified	8	L'enregistrement n'a pas été modifié
rsInvalid	16	L'enregistrement n'a pas été sauvegardé parce que son signet n'est pas valide
RsMultipleChanges	64	L'enregistrement n'a pas été sauvegardé parce que cela aurait affecté plusieurs enregistrements
RsPendingChanges	128	L'enregistrement n'a pas été sauvegardé parce qu'il renvoie à une insertion en attente
RsCanceled	256	L'enregistrement n'a pas été sauvegardé parce que l'opération a été annulée
RsCantRelease	1024	Le nouvel enregistrement n'a pas été sauvegardé à cause du verrouillage d'enregistrements existants
RsConcurrencyViolation	2048	L'enregistrement n'a pas été sauvegardé à cause d'un accès concurrentiel optimiste
RsIntegrityViolation	4096	L'enregistrement n'a pas été sauvegardé parce que l'utilisateur n'a pas respecté les contraintes d'intégrité
RsMaxChangesExceeded	8192	L'enregistrement n'a pas été sauvegardé parce qu'il y avait trop de modifications en attente
RsObjectOpen	16384	L'enregistrement n'a pas été sauvegardé à cause d'un conflit avec un objet de stockage ouvert
RsOutOfMemory	32768	L'enregistrement n'a pas été sauvegardé parce que la mémoire de l'ordinateur est insuffisante
RsPermissionDenied	65536	L'enregistrement n'a pas été sauvegardé parce que l'utilisateur n'a pas le niveau d'autorisation suffisant
RsSchemaViolation	131072	L'enregistrement n'a pas été sauvegardé parce qu'il ne respecte pas la structure de la base de données sous-jacente
RsDBDeleted	262144	L'enregistrement a déjà été supprimé de la source de données

Une bonne manipulation de cette propriété est indispensable pour un travail hors transaction.
Cette propriété n'est significative qu'en mode de traitement par lot (BatchOptimistic).

Sort (Int)

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. Utilisez une chaîne vide pour revenir à l'ordre de tri par défaut. Coté client il y a création d'un index temporaire si aucun index n'est présent dans le jeu de données.

SQL (Del)

Similaire à la propriété source ADO. Permet d'écrire ou de modifier la chaîne SQL de l'objet Command de l'objet Recordset. Cette propriété est de types listes de chaînes ce qui permet d'utiliser les propriétés/méthodes de ce type pour manipuler le SQL. Cette commande peut être paramétrée. Toute commande SQL valide est utilisable, cependant utiliser un Dataset pour n'exécuter que des requêtes actions est un non-sens.

State (Del)

Proche de la propriété EditMode ADO mais plus complet. Permet de connaître l'état du recordset. Peut prendre de nombreuses valeurs mais on utilise principalement :

Constante	Description
dsInactive	L'ensemble de données est fermé
dsEdit	L'enregistrement actif peut être modifié.
dsInsert	L'enregistrement actif est un tampon nouvellement inséré qui n'a pas été transmis. Cet enregistrement peut être modifié puis transmis ou abandonné
dsBrowse	Les données peuvent être visualisées, mais non modifiées. C'est l'état par défaut d'un ensemble de données ouvert.
dsOpening	Le DataSet est en cours d'ouverture mais n'a pas terminé. Cet état arrive quand l'ensemble de données est ouvert pour une lecture asynchrone.

Propriétés dynamiques

On utilise ces propriétés en passant par la collection Properties de l'objet recordset renvoyé par le DataSet. 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. Certaines de ces propriétés (Optimize par ex) ne sont plus utiles du fait de l'encapsulation Delphi.

Resync Command (ND)

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". Attends une chaîne SQL étant la commande de synchronisation.

Evidemment; on peut 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 :

```
procedure TForm1.FormActivate(Sender: TObject);
var adoRecordset:TADOQuery;
begin
     adoRecordset:=TADOQuery.Create(Owner);
     with adoRecordset do begin
```

```
            ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
            CursorLocation:=clUseClient;
            CursorType:=ctStatic;
            LockType:=ltBatchOptimistic;
            SQL.Add('SELECT Publishers.PubID, Publishers.Name,
Titles.Title, Titles.ISBN, Titles.PubID FROM Publishers , Titles
WHERE Publishers.PubID = Titles.PubID');
            Active:=True;
            Recordset.Properties['Unique Table'].Value:='Titles';
            Recordset.Properties['Resync Command'].Value:= 'SELECT
Publishers.PubID, Publishers.Name, Titles.Title, Titles.ISBN,
Titles.PubID FROM Publishers, Titles WHERE Publishers.PubID =
Titles.PubID AND Titles.ISBN=?';
            Locate('Title','Evaluating Practice',[loCaseInsensitive]);
            Edit;
            FieldValues['Titles.PubId']:=192;
            FieldValues['Title']:=FieldValues['Title'] +'s';
            Post;
             UpdateBatch(arCurrent);
            Recordset.Resync(1,2);
            DataSource1.DataSet:=adoRecordset;
      end;
end;
```

Nous voyons alors que la valeur apparaissant dans le TDBGrid est la bonne et que la modification désirée à bien eu lieu dans la table. Détaillons quelque peu.

Vous noterez tout d'abord que je travaille directement sur l'objet recordset sous-jacent à des fins de lisibilité et de cohérence.

Je stipule tout d'abord la table sur laquelle vont porter les modifications. Puis, je passe la commande de synchronisation. Celle-ci stipule que lorsque je ferais appel à la méthode Resync les enregistrements devront être mis à jour selon la clé primaire ISBN, ce qui revient à dire que l'enregistrement synchronisé ira re-extraire les données de la table 'Publishers' répondant à la condition de jointure de l'enregistrement considéré.

Enfin j'exécute mes modifications puis ma synchronisation.

Attention, la propriété dynamique 'Unique Table' ne fonctionne pas toujours bien dans Delphi selon les fournisseurs. Il reste prudent de faire attention de ne pas modifier les champs d'autres table avant d'avoir vérifié que le fournisseur la gère correctement.

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. Pas toujours supportée correctement dans d'autres modes que Delete.

Clients.

Update Criteria

Défini 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'autres. N'oubliez pas que les valeurs utilisées par le moteur de curseur sont les valeurs stockées dans la propriété OriginalValue des objets Fields concernés.

La gestion de cette propriété peut être très importante dans l'approche concurrentielle de votre application.

Envisageons le cas suivant.

J'exécute deux programmes simultanément, leur code dans le FormActivate étant identique

```
With ADOQuery1 do begin
    ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\biblio1.mdb;';
    CursorLocation:=clUseClient;
    CursorType:=ctStatic;
    LockType:=ltOptimistic;
    SQL.Add('SELECT * FROM Authors');
    Active:=True;
    Recordset.Properties['Update Criteria'].Value:=2;
    Locate('Au_Id',8139,[loCaseInsensitive]);
  end;
```

Dans ce cas la ligne 'Update Criteria' ne sert à rien puisque 2 (adCriteriaUpdCols) est la valeur par défaut.

Chaque feuille possède respectivement un bouton dont le code est soit

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with ADOQuery1 do begin
    Edit;
    FieldValues['Author']:='Gane John';
    post;
  end;
end;
```

Soit

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with ADOQuery1 do begin
    Edit;
    FieldValues['year born']:=1972;
    post;
  end;
end;
```

si je clique sur mes deux boutons, les deux modifications se déroulent correctement dans la base de données. Si je procède au même test en mettant la propriété dynamique 'Update Criteria' à 1 j'obtiens une erreur de concurrence optimiste lors du clic sur le deuxième bouton. Ce comportement dans un mode d'accès concurrentiel devrait être obligatoire puisqu'en toute rigueur l'enregistrement a été modifié. Mais il faut bien garder à l'esprit que ce n'est pas le mode par défaut du moteur de curseur client.

Clients

Update Resync

Défini 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 :

```
Recordset.Properties['Update Resync']:= 1 + 8;
```

Je reprends mon exemple précédent.

J'ajoute dans mes deux programmes un contrôle DataSource et un DBGrid pour visualiser mes modifications. Je laisse le 'Update Criteria' par défaut. Lors du clic sur le deuxième bouton il n'y a pas mise à jour correcte de la deuxième grille puisque la modification apportée par l'autre programme n'apparaît pas.

Par contre si j'ajoute dans les procédures FormActivate la ligne

```
Recordset.Properties['Update Resync'].Value:=4;
```

J'aurais un affichage correct.

Clients

Méthodes

Append (Del)

Similaire à la première syntaxe de la méthode AddNew ADO. Permet d'ajouter un enregistrement vide au Dataset. Cet enregistrement devient l'enregistrement actif.

AppendRecord (Del)

Identique à la deuxième syntaxe de la méthode AddNew. Permet l'ajout d'un enregistrement rempli. Cet enregistrement devient l'enregistrement actif. Cette méthode est souvent plus efficace que la précédente. Les valeurs données par le fournisseurs (valeur AutoInc par exemple) doivent être mises à "nil" lors de l'appel de la méthode. Par exemple

```
with adoRecordset do begin
    ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\biblio1.mdb;';
    CursorLocation:=clUseServer;
    LockType:=ltPessimistic;
    SQL.Add('SELECT * FROM Authors');
    Active:=True;
    adoRecordset.AppendRecord([nil,'Rabilloud',1967]);
end;
```

BookmarkValid (Del)

Permet de vérifier si un signet est toujours valide.

Cancel, CancelUpdates, CancelBatch (Del)

Ensemble des méthodes d'annulations de modifications d'enregistrement.
Cancel annule les modifications de l'enregistrement en cours avant qu'elles aient été validées.
CancelUpdates annule toutes les modifications dans les mises à jours par lot.
CancelBatch permet d'annuler les modifications en attente en spécifiant la cible.

ClearFields (Del)

Efface les valeurs des champs de l'enregistrement en cours, si celui-ci est en mode édition.

Clone (Int)

Duplique le jeu d'enregistrement. Le jeu ainsi obtenu est rigoureusement identique à l'original. Il est possible de forcer le clone à être en lecture seule quel que soit le verrouillage de l'original. Une modification apportée à l'original est répercutée sur tous les clones sauf si on a exécuté la méthode Requery sur l'original.

Close (Int)

Identique à Active:=False

CompareBookmark (Del)

Compare deux signets et renvoie 0 s'ils sont égaux. Attention, renvoie aussi 0 si un des deux signets vaut nil.

Delete (Int)

Supprime l'enregistrement actif, et tente de se repositionner sur le suivant, sur le précédent le cas échéant.
La méthode Delete ADO est assez sensible et peut engendrer facilement des dysfonctionnements. Pour éviter ceux-ci il convient de prendre la précaution suivante ; vider toujours le cache des modifications en attente même si elles concernent un autre enregistrement.

DeleteRecords (Del)

Supprime un groupe d'enregistrement selon le critère spécifié. En mode par lot la suppression n'est effective qu'après l'appel de la méthode UpdateBatch. N'utilisez pas cette méthode qui ne fonctionne pas correctement avec MS-Access, et préférez l'utilisation de l'ordre SQL équivalent.

Edit (Del)

Mais le jeu d'enregistrement en cours en cours en mode modification.

ExecSQL (Del)

Exécute le code SQL de la propriété SQL de l'objet. Cette méthode est utilisée lorsque la commande est une requête action. Une fois encore il n'est pas utile de se servir d'un TADOQuery pour faire des requêtes actions.

FieldByName (Del)

Renvoie l'objet Field dont on passe le nom à cette méthode.

FilterOnBookmarks (Del)

Filtre les enregistrements identifiés par un signet. Attends en paramètre un tableau de signets. Cette fonctionnalité est propre à ADO. Attention, utiliser cette méthode revient implicitement à réinitialiser les propriétés Filter et FilterGroup.

FindField (Del)

Similaire à FieldByName, mais ne déclenche pas d'erreur récupérable si le champ spécifié n'existe pas.

First, Last, Next, Prior (Del)

Méthodes de déplacement dans le jeu d'enregistrements. Les curseurs en avant seulement n'acceptent que la méthode Next.

GetBookmark, GotoBookmark, FreeBookmark (Del)

Méthodes de gestions des signets. Préférez l'emploi de ces méthodes plutôt que la propriété intrinsèque Bookmark ADO.

Insert, InsertRecord (Del)

Insère un nouvel enregistrement vide dans le jeu d'enregistrement. Dans la plupart des bases accédées par ADO, cela revient au même que d'utiliser les méthodes Append et AppendRecord.

IsEmpty (Del)

Permet de déterminer si l'ensemble d'enregistrement est vide. Si vous avez l'habitude de programmer ADO, il vous faut maintenant utiliser cette méthode du fait du changement de définition des propriétés BOF et EOF.

LoadFromFile, SaveToFile (Del)

Permet de gérer les jeux d'enregistrements persistants. L'appel de SaveToFile crée le fichier. Si celui-ci existe déjà une erreur survient. Utilisez plutôt un curseur client pour ces fonctionnalités, cela permet à ADO d'invoquer un fournisseur spécifique si le fournisseur ne gère pas les recordset persistant.

Locate (Del)

Méthode de recherche dans un jeu d'enregistrement implémentée par Delphi. Cette méthode est beaucoup plus performante que la méthode Find ADO car elle utilise les index s'ils existent et qu'elle permet une recherche multi critères. Si un enregistrement est trouvé, il devient l'enregistrement en cours.

Toutefois, à la différence de la méthode Find ADO, cette méthode ne permet pas de faire simplement des recherches successives avec le même critère. On peut contourner ce problème (voir l'exemple "recherches successives")

Lookup (Del)

Cette méthode est aussi une méthode de recherche mais elle diffère de Locate dans son approche et dans sa finalité.

Lookup ne modifie jamais l'enregistrement en cours, elle renvoie un tableau des valeurs désirées si la correspondance existe.

En cas d'utilisation de champs calculés persistants, la méthode Lookup force le recalcul avant le renvoi des valeurs. Cette méthode peut être très intéressante dans certains cas particulier mais elle contient un piège. Je vais écrire un exemple de code particulièrement mal que nous allons étudier ensemble.

```
procedure TForm1.FormActivate(Sender: TObject);
var adoRecordset:TADOQuery;
Signet:TBookmark;
begin
     adoRecordset:=TADOQuery.Create(Owner);
     with adoRecordset do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
          CursorLocation:=clUseClient;
          CursorType:=ctStatic;
          LockType:=ltOptimistic;
          SQL.Add('SELECT * FROM Authors');
          Active:=True;
          Locate('Author','Vaughn, William',[]);
          Edit;
          FieldByName('year born').Value:=1901;
          Signet :=GetBookmark;
          Locate('Author','Vaughn, William R.',[]);
          if FieldByName('year born').Value =1947 then
             begin
                  GotoBookmark(Signet);
                  Cancel;
             end
          else
              begin
                   GotoBookmark(Signet);
                   Post;
              end;
          DataSource1.DataSet:=adoRecordset;
     end;
end;
end.
```

Je suis sur que certains d'entre vous se demandent quel est l'intérêt d'étudier un tel code, mais dites-vous bien que je voie de nombreux codes d'un style identique et que malheureusement j'en verrais d'autres. Dans le concept, il s'agit de valider une modification en fonction de la valeur d'un autre enregistrement.

Quoique joyeusement délirant ce code compile et s'exécute sans erreur.

Voyons les erreurs de ce code.

Le curseur est à l'envers. Le seul intérêt d'un tel code étant de vérifier si la valeur du champ de décision n'as pas changé du fait d'un autre utilisateur, on doit travailler côté serveur puisque les curseurs clients sont statiques et qu'un curseur statique ne reflète pas les modifications des autres utilisateurs.

La modification sera toujours effectuée si le deuxième enregistrement existe puisque la méthode Locate provoquera un changement de l'enregistrement en cours ce qui induit un appel implicite de la méthode Post.

Quand bien même le deuxième enregistrement n'existerait pas, il y aura tout de même validation, car le simple Appel de la méthode Locate valide les modifications.

Enfin l'appel de GotoBookmark posera les mêmes problèmes que Locate.

Un code plus correct devrait être de la forme

```
with adoRecordset do begin
   ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
   CursorLocation:=clUseServer;
   CursorType:=ctKeySet;
   LockType:=ltOptimistic;
   SQL.Add('SELECT * FROM Authors');
   Active:=True;
   Locate('Author','Vaughn, William',[]);
   Edit;
   FieldByName('year born').Value:=1901;
   varTest:=adoRecordset.Lookup('Author','Vaughn, William R.','Year
born');
   if varTest =1947 then
       Cancel
   else
       Post;
   DataSource1.DataSet:=adoRecordset;
end;
```

Pourtant ce code ne fonctionnera pas mieux car bien que la méthode Lookup ne bouge pas l'enregistrement en cours en théorie, elle envoie tout de même les modifications du tampon vers la source.

MoveBy (Del)

Déplace l'enregistrement en cours d'un nombre d'enregistrement précisé en paramètre. Renvoie la valeur du déplacement effectif. En effet, si la valeur demandée excède le déplacement possible, le curseur se positionnera sur le premier (ou le dernier) enregistrement et enverra la valeur exacte du déplacement.

Open (Del)

Identique à passer la propriété Active à vrai.

Refresh, Requery (Int)

Utilisez plutôt Requery. Ré exécute la requête pour rafraîchir le jeu d'enregistrement. Il est possible de préciser un paramètre d'exécution pour une exécution asynchrone.
Attention, ces méthodes ferment puis ouvrent le jeu d'enregistrement ce qui peut exécuter le code programmé dans des événements liés à l'ouverture / fermeture.

Resync (Int - ND)

Attention, il existe une méthode interne Resync implémentée dans TdataSet et ses dépendants. Je vais moi vous parler de la méthode Resync intrinsèque ADO, il faut donc toujours spécifier Recordset dans votre ligne de code
Met à jour le jeu d'enregistrement. 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
Dataset.recordset.**Resync** ***AffectRecords, ResyncValues***
Où AffectRecords peut prendre les valeurs suivantes :

Constante	Valeur	Description
arAffectCurrent	1	Synchronise l'enregistrement courant.
arAffectGroup	2	Synchronise les enregistrements filtrés, soit par FilterOnBookmarks soit par FilterGroup
arAffectAll	3	Synchronise tous les enregistrements visibles dans le jeu.

et ResyncValues :

Constante	Valeur	Description

adResyncUnderlyingValues	1	N'écrase pas les modifications en cours
adResyncAllValues	2	Synchronise le jeu en supprimant les modifications

Par exemple

```
adoquery1.Recordset.Resync(3,2);
```

synchronise l'ensemble des enregistrements en écrasant les modifications en attente.

Fonctionnement

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 vrais 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 modifiés par un autre utilisateur, en comparant les propriétés UnderlyingValue et OriginalValue avant de déclencher une mise à jour.

Supports

Cette méthode reste la meilleure amie du développeur ADO. Elle permet de savoir si un jeu d'enregistrement donné va accepter une fonctionnalité spécifique. De la forme :

boolean := *Dataset*.**Supports(** ***CursorOptions*** **)**

ou *CursorOptions* peut prendre les valeurs suivantes :

Constante	Valeur	Description
coAddNew	16778240	Vous pouvez utiliser la méthode AddNew pour ajouter de nouveaux enregistrements.
coApproxPosition	16384	Vous pouvez lire et définir les propriétés AbsolutePosition et AbsolutePage.
coBookmark	8192	Vous pouvez utiliser la propriété Bookmark pour accéder à des enregistrements spécifiques.
coDelete	16779264	Vous pouvez utiliser la méthode Delete pour supprimer des enregistrements.
coFind	524288	Vous pouvez utiliser la méthode Find pour localiser une ligne dans un Recordset.
coHoldRecords	256	Vous pouvez extraire plus d'enregistrements ou modifier la position d'extraction suivante sans valider toutes les modifications en attente.
coIndex	8388608	Vous pouvez utiliser la propriété Index pour donner un nom à un index.
coMovePrevious	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.
coNotify	262144	Indique que le fournisseur gère les évènements du recordset
coResync	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.
coSeek	4194304	Vous pouvez utiliser la méthode Seek pour localiser une ligne dans un Recordset.

coUpdate	16809984	Vous pouvez utiliser la méthode Update pour modifier les données existantes
coUpdateBatch	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 Dataset et éviter de faire de la gestion d'erreurs a posteriori.

UpdateBatch

Met à jour toutes les modifications en attente (opération par lot). De la forme
recordset.**UpdateBatch** ***AffectRecords***
Où AffectRecords définit les enregistrements concernés (voir Resync).

Evènements

Les évènements 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 évènements sont principalement utilisés en programmation asynchrone.

TEventStatus

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

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

OnEndOfRecordset

EndOfRecordset(**DataSet**: TCustomADODataSet ;var **MoreData**: WordBool ; var **EventStatus**: TEventStatus)

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

EventStatus est l'état du recordset. Vaut **esOK** si l'opération c'est bien passée et **esCantDeny** si l'opération qui a déclenchée l'événement ne peut être annulée. Si vous devez modifier le recordset pendant la procédure, ce paramètre doit être mis à **esUnwantedEvent** afin de supprimer les notifications non désirées.

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

OnFetchProgress & OnFetchComplete

FetchProgress(**DataSet**: TCustomADODataSet;**Progress**, **MaxProgress**: Integer; var **EventStatus**: TEventStatus);

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

Progress représente le nombre de ligne extraite

MaxProgress le nombre de lignes qui doivent être extraites

FetchComplete(**DataSet**: TCustomADODataSet; const **Error**: Error; var **EventStatus**: TEventStatus);

Se produit après la récupération du dernier enregistrement.

Où *Error* est un objet erreur ADO

OnWillChangeField & OnFieldChangeComplete

WillChangeField(**DataSet**: TCustomADODataSet; const FieldCount: Integer; const **Fields**: OleVariant; var **EventStatus**: TEventStatus);

FieldChangeComplete(**DataSet**: TCustomADODataSet; const **FieldCount**: Integer; const **Fields**: OleVariant; const **Error**: Error; var **EventStatus**: TEventStatus);

Où FieldCount 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 TField 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 évènements. Ces évènements 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 **EventStatus** à la valeur **esCancel** dans l'événement **WillChangeField**.

OnWillChangeRecord & OnRecordChangeComplete

WillChangeRecord(**DataSet**: TCustomADODataSet; const **Reason**: TEventReason; const **RecordCount**: Integer; var **EventStatus**: TEventStatus);

RecordChangeComplete(**DataSet**: TCustomADODataSet; const **Reason**: TEventReason; const **RecordCount**: Integer; const **Error**: Error; var **EventStatus**: TEventStatus);

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

Constante	Valeur	Description
erAddNew	1	Une nouvelle ligne a été ajoutée
erClose	9	L'ensemble d'enregistrements a été fermé.
erDelete	2	Une ligne existante a été supprimée.
erFirstChange	11	Equivalent à la valeur ADO ad.
erMove	10	Le pointeur de ligne de l'ensemble d'enregistrements a été déplacé.
erMoveFirst	12	Le pointeur de ligne de l'ensemble d'enregistrements a été déplacé sur la première ligne.
erMoveLast	15	Le pointeur de ligne de l'ensemble d'enregistrements a été déplacé sur la dernière ligne.
erMoveNext	13	Le pointeur de ligne de l'ensemble d'enregistrements a été déplacé sur la ligne suivante.
erMovePrevious	14	Le pointeur de ligne de l'ensemble d'enregistrements a été déplacé sur la ligne précédente
erRequery	7	L'ensemble d'enregistrements a été rafraîchi par la méthode Requery
erResync	8	L'ensemble d'enregistrements a été synchronisé par la méthode Resync.
erUndoAddNew	5	Une opération d'insertion de ligne a été annulée.
erUndoDelete	6	Une opération de suppression de ligne a été annulée.
erUndoUpdate	4	Une opération de mise à jour à été annulée.
erUpdate	3	De nouvelles valeurs ont été saisies dans une ligne existante.

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

RecordCount renvoie le nombre d'enregistrements modifiés.

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

OnWillChangeRecordset & OnRecordsetChangeComplete

WillChangeRecordset(**DataSet**: TCustomADODataSet; const **Reason**: TEventReason; var **EventStatus**: TEventStatus);

RecordsetChangeComplete(**DataSet**: TCustomADODataSet; const **Reason**: TEventReason; const **Error**: Error; var **EventStatus**: TEventStatus);

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.

Reason peut prendre les valeurs suivantes : **erReQuery**, **erReSynch**, **erClose**, **erOpen**

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

OnWillMove & OnMoveComplete

WillMove(**DataSet**: TCustomADODataSet; const **Reason**: TEventReason; var **EventStatus**: TEventStatus);

MoveComplete(**DataSet**: TCustomADODataSet; const **Reason**: TEventReason; const **Error**: Error; var **EventStatus**: TEventStatus);

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**.

Reason peut prendre les valeurs suivantes : **erMoveFirst**, **erMoveLast**, **erMoveNext**, **erMovePrevious**, **erMove** ou **erRequery**

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

TADOTable

Les composants que nous allons voir maintenant sont tous similaires à ceux déjà vu sur de nombreux aspects. C'est donc sur leurs particularités que nous allons nous pencher.

Ce composant permet de n'accéder qu'à une seule table de la source de données. Cet accès permet un accès dit "physique", c'est à dire que les index de la table sont directement récupérables.

Propriétés et méthodes spécifiques

ReadOnly

Permet de mettre les données de la table en lecture seule.

TableDirect

Spécifie comment l'accès à la table est référencé. Dans certains cas, méthodes Seek ADO par exemple, il est impératif que cette propriété soit valorisée à vrai.

TableName

Indique le nom de la table ciblée. N'est pas accessible en écriture quand le composant est ouvert.

GetIndexNames

Renvoie une liste de tous les index disponibles pour une table.

Seek (int)

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

ADOTable.**Seek *KeyValues*, *SeekOption***

Où KeyValues est une suite de valeurs de type "Variant". Un index consiste en une ou plusieurs colonnes ou 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 index et que la commande soit ouverte en adCmdTableDirect.

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

Serveurs

Pour Access, le fournisseur Jet 3.51 ne gère pas cette méthode

Le code suivant recherche l'auteur ayant le code 54

```
with ADOTable1
   ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
   CursorLocation:=clUseServer;
   TableDirect:=True;
   TableName:='Authors';
   Active:=True;
   IndexName:='PrimaryKey';
   Seek(54,soFirstEq);
end;
```

TADODataset

C'est sensiblement le même objet que TADOQuery si ce n'est qu'il y a utilisation d'un objet Command en lieu et place du texte SQL.

TADOStoredProc

Il s'agit d'un Dataset un peu particulier. Il possède une propriété ProcédureName qui permet de définir le nom de la procédure utilisée. Il faut ensuite utiliser la collection Parameters comme pour l'objet Command.

Gardons quelques points importants à l'esprit.

- Cet objet sert à utiliser les procédures stockées paramétrées renvoyant un jeu d'enregistrements, sinon préférer l'utilisation de l'objet Command.
- Il existe trois types de paramètres
    - Les paramètres d'entrées sont ceux que la procédure attend pour s'exécuter correctement
    - Les paramètres de sorties sont ceux que la procédure vous fournis en réponse.
    - **Le** paramètre de retour est un paramètre particulier que certaines procédures renvoient.

Il est important de ne pas confondre paramètre de sortie et paramètre retour, donc de bien connaître la procédure appelée.

Exemples de code

Dans cette partie nous allons regarder quelques exemples de code un peu plus complexe. Comme dans la partie précédente, il m'arrive de passer directement la connexion dans le Dataset. Cela ne veut pas dire qu'il s'agit d'un exemple à suivre, c'est juste pour diminuer la complexité du code à la lecture.

Nombre d'enregistrement et position

Dans cet exemple nous allons travailler sur la navigation sans les méthodes "Move" dans un Recordset. Pour cela nous allons utiliser deux propriétés du Dataset : RecordCount et RecNo. 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 RecNo.

Globalement la position dans un Dataset est toujours sur un enregistrement. La valeur de AbsolutePosition va de 1 à RecordCount.

Remarque :

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.

J'ajoute donc à mon projet un contrôle TDataSource, un contrôle TDBLookupComboBox qui me permettra de sélectionner un auteur dans la liste et un contrôle ProgressBar.

```
unit uPosition;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms,
  Dialogs, ComCtrls, DB, ADODB, StdCtrls, DBCtrls;

type
  TForm1 = class(TForm)
    DataSource1: TDataSource;
    ProgressBar1: TProgressBar;
    DBLookupComboBox1: TDBLookupComboBox;
    procedure DBLookupComboBox1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;
  adoRecordset:TADOQuery;

implementation

{$R *.dfm}

procedure TForm1.DBLookupComboBox1Click(Sender: TObject);
begin
   ProgressBar1.Position:=adoRecordset.RecNo;
```

```
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
adoRecordset:=TADOQuery.Create(Owner);
  with adoRecordset do begin
    ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\biblio.mdb;';
    CursorLocation:=clUseServer;
    CursorType:=ctKeySet;
    LockType:=ltReadOnly;
    SQL.Add('SELECT * FROM Authors');
    Active:=True;
    ProgressBar1.Min:=1;
    ProgressBar1.Max:=RecordCount;
  end;
  Datasource1.DataSet:=adoRecordset;
  with DBLookupComboBox1
    ListSource:=Datasource1;
    KeyField:='Au_Id';
    ListField:='Author';
  end;

end;

end.
```

Dans le "FormCreate", je crée mon dataset. J'affecte ensuite mon dataset comme source de données du DataSource, et je précise quel champ servira pour le remplissage de la liste ainsi que le champ servant de clé.

Tout se passe donc dans l'événement Click du TDBLookupComboBox.

Un click sur une valeur dans la liste a pour effet de faire bouger l'enregistrement courant vers l'enregistrement dont la valeur est sélectionnée. Dès lors la récupération de la position est visible dans la ProgressBar.

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 recordset et l'utilisation du SQL.

Le concours est simple, nous allons extraire un jeu d'enregistrement correspondant à tous les auteurs dont le nom commence par "S" et le trier, puis ensuite extraire les enregistrements ou 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'obtiendrais le temps nécessaire. Nous sommes en mode synchrone.

Pour rester sensiblement comparable, je n'utiliserais 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. Le code sera le suivant :

```
procedure TForm1.btnSQLClick(Sender: TObject);
var tmpStart, tmpEnd:integer;
begin
with ADOQuery1 do begin
    ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
    CursorLocation:=clUseServer;
    CursorType:=ctKeySet;
    LockType:=ltReadOnly;
    SQL.Clear;
```

```
    SQL.Add('SELECT * From Authors WHERE Author LIKE ' +
QuotedStr('S*') + ' ORDER BY Author, [Year Born] DESC');
    tmpStart:=windows.GetTickCount;
    Active:=True;
    tmpEnd:=windows.GetTickCount;
    MessageDlg(IntToStr(tmpEnd-tmpStart),mtInformation,[mbOK],0);
    Close;
    SQL.Clear;
    SQL.Add('SELECT * From Authors WHERE Author LIKE ' +
QuotedStr('S*') + ' AND NOT [Year Born] IS NULL ORDER BY Author,
[Year Born] DESC');
    tmpStart:=windows.GetTickCount;
    Active:=True;
    tmpEnd:=windows.GetTickCount;
    MessageDlg(IntToStr(tmpEnd-tmpStart),mtInformation,[mbOK],0);
    Close;
end;
end;
```

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'enregistrement. La propriété Sort demande un curseur côté client. De ce fait le curseur sera statique. Le code utilisé sera le suivant :

```
procedure TForm1.btnRecordClick(Sender: TObject);
var tmpStart, tmpEnd:integer;
begin
with ADOQuery1 do begin
    ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
    CursorLocation:=clUseClient;
    CursorType:=ctStatic;
    LockType:=ltReadOnly;
    SQL.Clear;
    SQL.Add('SELECT * From Authors');
    tmpStart:=windows.GetTickCount;
    Active:=True;
    tmpEnd:=windows.GetTickCount;
    Filter:='Author Like ' + QuotedStr('S*');
    Filtered:=True;
    Sort:='Author ASC, [Year Born] DESC';
    MessageDlg(IntToStr(tmpEnd-tmpStart),mtInformation,[mbOK],0);
    tmpStart:=windows.GetTickCount;
    Filter:='Author Like ' + QuotedStr('S*')+ 'AND [Year Born] <>
NULL';
    Filtered:=True;
    tmpEnd:=windows.GetTickCount;
    MessageDlg(IntToStr(tmpEnd-tmpStart),mtInformation,[mbOK],0);
    Close;
end;
end;
```

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

Les temps ne seront jamais parfaitement reproductible car de nombreux paramètre peuvent jouer, mais on constate que le code SQL va plus vite que le filtrage et le tri du recordset (premier temps). Ceci est du 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 plus nécessaire de ré extraire les données, celles-ci étant présente 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 client, c'est la nécessite 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.

N.B : L'exemple choisi pénalise toutefois le recordset, mais une requête bien pensée (pas d'utilisation de *, limitation aux enregistrements nécessaires) optimise les ressources, et peut vous encourager à privilégier le recordset afin de bénéficier de ses fonctionnalités.

Recherche successives ou directionnelles

La méthode Locate permet de faire facilement une recherche, mais il est beaucoup plus difficile de faire avec des recherches successives ou directionnelles. Il existe quand même des techniques pour compenser le problème. La première est évidemment d'utiliser un filtre, c'est la plus simple et la plus efficace. Néanmoins elle ne peut pas toujours être utilisée puisqu'un autre filtre peut déjà être actif par exemple. Nous allons regarder deux techniques qui nous feront d'ailleurs dériver sur un ensemble de concepts un peu plus ardus. Et pour bien démarrer, nous allons regarder la fonction qui ne marche pas.

Les défauts de la méthode Locate

Avec Locate, il est possible de faire une recherche partielle (l'équivalent d'un Like) en spécifiant l'option loPartialKey. Toutefois cette méthode ne fonctionnera pas sur un jeu d'enregistrement filtré. Prenons l'exemple suivant :

```
with ADOQuery1 do begin
    ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=c:\biblio.mdb;';
    CursorLocation:=clUseClient;
    CursorType:=ctStatic;
    LockType:=ltReadOnly;
    SQL.Add('SELECT * FROM Authors');
    Active:=True;
    Filter:='Author like '+QuotedStr('s*');
    Filtered:=True;
    Locate('author','John',[loPartialKey]);
end;
datasource1.DataSet:=adoquery1;
```

Vous voyez que l'enregistrement sélectionné ne contient pas 'John'. Si je retourne le curseur (keyset, server) j'obtiens encore un enregistrement qui ne contient pas 'John' mais pas le même que précédemment. La méthode Locate présente donc ici un défaut majeur, et ne nous permet pas de répondre à notre problème.

La méthode ADO Seek ne répondra pas mieux à nos besoin puisqu'elle ne permet pas la recherche partielle.

Gestion des signets

Cette première méthode consiste à gérer la fonctionnalité voulue par un moyen détourné. J'ai assez souvent pratiqué ce genre de cautère et je ne peux que vous conseiller de bien réfléchir avant de vous lancer dans ce genre d'aventure. L'exemple qui suit va nous permettre de voir les implications de ce genre de programmation.

Comme les méthodes intégrées ne gèrent pas la recherche que je désire, je vais contourner l'obstacle en créant un tableau de signets des enregistrements cherchés.

Regardons le code de base suivant :

```
procedure TForm1.FormCreate(Sender: TObject);
var TabSignet:Array of TVarRec;
intElem : Integer;
begin
     with ADOQuery1 do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=c:\biblio.mdb;';
```

```
          CursorLocation:=clUseClient;
          CursorType:=ctStatic;
          LockType:=ltReadOnly;
          SQL.Add('SELECT * FROM Authors');
          Active:=True;
          Filter:='Author like '+QuotedStr('s*');
          Filtered:=True;
          First;
          intElem:=0;
          while not EOF do begin
            if Strpos(PChar(FieldByName('Author').AsString),
PChar('John'))<>nil then
              begin
              SetLength(TabSignet,intElem+1);
              Tabsignet[intElem].VType:=vtPointer;
              Tabsignet[intElem].VPointer:=GetBookmark;
              intElem:=intElem+1;
              end;
              Next;
          end;
          FilterOnBookmarks(TabSignet);
     end;
     datasource1.DataSet:=adoquery1;
end;
```

Ce code est particulièrement réducteur mais il répond au problème posé. Imaginons maintenant un code un peu plus passe-partout.

```
unit uFind;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms,
  Dialogs, StdCtrls, Mask, DBCtrls, DB, ADODB;

type
  TForm1 = class(TForm)
    DataSource1: TDataSource;
    Label1: TLabel;
    DBEdit1: TDBEdit;
    Label2: TLabel;
    DBEdit2: TDBEdit;
    Label3: TLabel;
    DBEdit3: TDBEdit;
    btnMvFirst: TButton;
    btnMvNext: TButton;
    btnMvLast: TButton;
    btnMvPrev: TButton;
    ADOQuery1: TADOQuery;
    procedure FormCreate(Sender: TObject);
    procedure btnMvFirstClick(Sender: TObject);
    procedure btnMvLastClick(Sender: TObject);
    procedure btnMvPrevClick(Sender: TObject);
    procedure btnMvNextClick(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
    procedure ConstruitSignet(ADOQuery1:TADOQuery);
  end;

var
```

```
  Form1: TForm1;
  TabSignet:Array of TVarRec;
  CurPos : Integer;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);

begin
     with ADOQuery1 do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=c:\biblio.mdb;';
          CursorLocation:=clUseClient;
          CursorType:=ctStatic;
          LockType:=ltReadOnly;
          SQL.Add('SELECT * FROM Authors');
          Active:=True;
          Filter:='Author like '+QuotedStr('s*');
          Filtered:=True;
     end;
     ConstruitSignet(ADOQuery1);
     datasource1.DataSet:=adoquery1;
end;

procedure TForm1.ConstruitSignet(ADOQuery1:TADOQuery);
var intElem:integer;
begin
with ADOQuery1 do begin
  First;
  intElem:=0;
  while not EOF do begin
     if
strpos(PChar(FieldByName('Author').AsString),PChar('John'))<>nil
then
      begin
        SetLength(TabSignet,intElem+1);
        Tabsignet[intElem].VType:=vtPointer;
        Tabsignet[intElem].VPointer:=GetBookmark;
        intElem:=intElem+1;
      end;
      Next;
  end;
end;
end;

procedure TForm1.btnMvFirstClick(Sender: TObject);
begin
CurPos:=low(TabSignet);
ADOQuery1.GotoBookmark(TabSignet[CurPos].VPointer);
end;

procedure TForm1.btnMvLastClick(Sender: TObject);
begin
CurPos:=high(TabSignet);
ADOQuery1.GotoBookmark(TabSignet[CurPos].VPointer);
end;

procedure TForm1.btnMvPrevClick(Sender: TObject);
begin
if CurPos>low(TabSignet)Then begin
  CurPos:=CurPos-1;
```

```
  ADOQuery1.GotoBookmark(TabSignet[CurPos].VPointer);
  End;
end;

procedure TForm1.btnMvNextClick(Sender: TObject);
begin
if CurPos<high(TabSignet)Then begin
  CurPos:=CurPos+1;
  ADOQuery1.GotoBookmark(TabSignet[CurPos].VPointer);
  End;
end;

end.
```

Ce code résout correctement le problème pose. Une petite modification de la procédure "ConstruitSignet" permettrai de laisser l'utilisateur saisir la chaîne de recherche. Il reste tout de même deux problèmes conséquents.

Le coût en terme de temps d'exécution est très important. Chaque modification du filtrage ou de l'ordre de tri devra engendrer une reconstruction du tableau de signet. Comme celui-ci est basé sur le parcourt l'ensemble du Dataset, la méthode est extrêmement lourde.

Ce prix peut être toutefois acceptable. Cependant la fonction contient un piège majeur. En mode Multi-Utilisateurs, Le tableau de signet peut être faux du fait des modifications apportées par d'autres utilisateurs (ajout, modification ou suppression). Coté serveur, il faudrait coupler la fonction a la synchronisation. Dès lors, le temps d'exécution du code est éliminatoire sauf sur des jeux d'enregistrements très petits. Coté client comme le curseur est forcément statique, c'est le programme qui gère la synchronisation il suffira donc d'intégrer la procédure au code de synchronisation.

Cette reconstruction du tableau de signet est indispensable. En effet plusieurs types d'erreurs peuvent se produire lors de l'appel d'un signet faisant référence à un enregistrement supprimé.

Programmation intrinsèque

Une autre méthode malheureusement méconnue consiste à programmer directement le Recordset ADO sous-jacent avec ses méthodes propres, étant bien entendu que cela n'est utile que pour des méthodes n'étant pas implémentées dans Delphi. Regardons le code suivant.

```
procedure TForm1.FormCreate(Sender: TObject);

begin
     with ADOQuery1 do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=c:\biblio.mdb;';
          CursorLocation:=clUseClient;
          CursorType:=ctStatic;
          LockType:=ltReadOnly;
          SQL.Add('SELECT * FROM Authors');
          Active:=True;
          Filter:='Author like '+QuotedStr('s*');
          Filtered:=True;
     end;
     datasource1.DataSet:=adoquery1;
end;

procedure TForm1.btnMvFirstClick(Sender: TObject);
begin
ADOQuery1.Recordset.Find('Author like '+QuotedStr('*John*'),0,1,1);
end;

procedure TForm1.btnMvLastClick(Sender: TObject);
var SearchBack:integer;
begin
SearchBack:=-1;
ADOQuery1.Recordset.MoveLast;
```

```
ADOQuery1.Recordset.Find('Author like
'+QuotedStr('*John*'),0,SearchBack,EmptyParam);
end;

procedure TForm1.btnMvPrevClick(Sender: TObject);
var SearchBack:integer;
begin
SearchBack:=-1;
ADOQuery1.Recordset.Find('Author like
'+QuotedStr('*John*'),1,SearchBack,EmptyParam);
end;

procedure TForm1.btnMvNextClick(Sender: TObject);
begin
ADOQuery1.Recordset.Find('Author like
'+QuotedStr('*John*'),1,1,EmptyParam);
end;

end.
```

Il est beaucoup plus simple que le précédent. Vous voyez que j'utilise la méthode Find ADO dont la définition est la suivante.

Find (Non documentée)

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 (1)** ou **adSearchBackward (-1)**

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.

En l'état le code ci-dessus ne fonctionnera pas, qui plus est le code des contient une erreur possible en cas de non-correspondance. Etudions les problèmes successivement.

Si je clique sur mes boutons, l'enregistrement affiché ne change pas. Par contre si je regarde l'enregistrement en cours dans le recordset sous-jacent, je constate que le déplacement a eu lieu. Cela vient du fait qu'une fois le Dataset créé à partir du Recordset, il n'y a pas synchronisation entre les enregistrements courants des deux objets. Je vais donc devoir gérer cette synchronisation.

Il existe plusieurs techniques possibles pour gérer cette synchronisation, mais la plus simple consiste à utiliser la méthode de déplacement MoveBy. En effet, le Recordset et le Dataset sont ordonnés et filtrés de façon identique. Il y a identité entre la propriété ADO AbsolutePosition et la propriété RecNo Delphi. Dès lors le calcul du déplacement est très simple. Attention toutefois de convertir la propriété AbsolutePosition en Integer Delphi pour éviter de faire des opérations sur deux données de types différents.

Je dois aussi gérer le fait que la méthode Find va positionner l'enregistrement en bout de recordset en cas d'échec de la recherche. Comme les positions EOF et BOF ne sont pas équivalentes

entre le recordset et le Dataset, je vais obtenir une erreur si je ne gère pas le cas. Le code final de mes boutons sera :

```
procedure TForm1.btnMvFirstClick(Sender: TObject);
var PosLocal:OLEVariant;
begin
PosLocal:=ADOQuery1.Recordset.Bookmark;
ADOQuery1.Recordset.Find('Author like '+QuotedStr('*John*'),0,1,1);
if ADOQuery1.Recordset.EOF then
    ADOQuery1.Recordset.Bookmark:=PosLocal
else
    ADOQuery1.MoveBy(Integer(adoquery1.Recordset.AbsolutePosition)-
ADOQuery1.RecNo);
end;

procedure TForm1.btnMvLastClick(Sender: TObject);
var SearchBack:integer;
PosLocal:OLEVariant;
begin
SearchBack:=-1;
PosLocal:=ADOQuery1.Recordset.Bookmark;
ADOQuery1.Recordset.MoveLast;
ADOQuery1.Recordset.Find('Author like
'+QuotedStr('*John*'),0,SearchBack,EmptyParam);
if ADOQuery1.Recordset.BOF then
    ADOQuery1.Recordset.Bookmark:=PosLocal
else
    ADOQuery1.MoveBy(Integer(adoquery1.Recordset.AbsolutePosition)-
ADOQuery1.RecNo);
end;

procedure TForm1.btnMvNextClick(Sender: TObject);
var PosLocal:OLEVariant;
begin
PosLocal:=ADOQuery1.Recordset.Bookmark;
ADOQuery1.Recordset.Find('Author like
'+QuotedStr('*John*'),1,1,EmptyParam);
if ADOQuery1.Recordset.EOF then
    ADOQuery1.Recordset.Bookmark:=PosLocal
else
    ADOQuery1.MoveBy(Integer(adoquery1.Recordset.AbsolutePosition)-
ADOQuery1.RecNo);
end;

procedure TForm1.btnMvPrevClick(Sender: TObject);
var SearchBack:integer;
PosLocal:OLEVariant;
begin
SearchBack:=-1;
PosLocal:=ADOQuery1.Recordset.Bookmark;
ADOQuery1.Recordset.Find('Author like
'+QuotedStr('*John*'),1,SearchBack,EmptyParam);
if ADOQuery1.Recordset.BOF then
    ADOQuery1.Recordset.Bookmark:=PosLocal
else
    ADOQuery1.MoveBy(Integer(adoquery1.Recordset.AbsolutePosition)-
ADOQuery1.RecNo);
end;
```

Quelques remarques encore sur ce code.

- Pour la recherche arrière j'utilise la variable SearchBack car le passage direct de la valeur –1 à la méthode Find déclenche une erreur de compilation.

- J'utilise la propriété ADO Bookmark pour marquer la position plutôt que la méthode Delphi GetBookmark. En effet, lorsqu'on travaille sur le recordset, il convient de n'utiliser que des propriétés / méthodes de celui-ci.

La programmation intrinsèque pourrait encore faire l'objet de nombreux exemples. Je ne saurais trop vous recommander de n'utiliser celle-ci que lorsque l'objet Dataset ne peut pas répondre à vos attentes, car elle demande une bonne connaissance de la programmation ADO et elle peut engendrer de grave dysfonctionnement.

Programmation asynchrone

Celle-ci n'est pas plus complexe que la programmation évènementielle 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.

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Déclarations privées }
    procedure MaConnectComplete(Connection: TADOConnection;
      const Error: Error; var EventStatus: TEventStatus);
    procedure MaConnectionExecuteComplete(Connection:
TADOConnection;
      RecordsAffected: Integer; const Error: Error;
      var EventStatus: TEventStatus; const Command: _Command;
      const Recordset: _Recordset);
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;
  adoConnection:TADOConnection;
  adoCommand:TADOCommand;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
     adoConnection:=TADOConnection.Create(nil);
     with adoConnection do begin
          ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\NWind.mdb;';
          ConnectOptions:= coAsyncConnect;
          OnConnectComplete:=MaConnectComplete;
          OnExecuteComplete:=MaConnectionExecuteComplete;
          Connected:=True;
     end;
end;

procedure TForm1.MaConnectComplete(Connection: TADOConnection;
  const Error: Error; var EventStatus: TEventStatus);
begin
```

```
     adoCommand:=TADOCommand.Create(nil);
     if EventStatus=esErrorsOccured then
        MessageDlg('Erreur lors de la connexion',mtError,[mbOK],0)
     else
     begin
          with adoCommand do begin
               Connection:=adoConnection;
               CommandText:='UPDATE Clients Set Pays = ' +
QuotedStr('États-Unis')+' WHERE Pays = '+QuotedStr('USA');
               ExecuteOptions:=
[eoAsyncExecute]+[eoExecuteNoRecords];
               Execute;
          end;
     end;
end;

procedure TForm1.MaConnectionExecuteComplete(Connection:
TADOConnection;
  RecordsAffected: Integer; const Error: Error;
  var EventStatus: TEventStatus; const Command: _Command;
  const Recordset: _Recordset);
begin
     MessageDlg(IntToStr(RecordsAffected)+' modifications
effectués',mtInformation,[mbOK],0);
end;

end.
```

Le cheminement est simple, à la création 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. Il y a trois avantages important avec ce style de programmation.

- Votre code peut exécuter autre chose pendant que les opérations de longue durée ont lieu.
- Vous êtes moins dépendant de la charge éventuelle du serveur.
- Les événements contiennent tous les paramètres nécessaires à la construction d'un code robuste.

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.

```
procedure TForm1.Button1Click(Sender: TObject);
var Signet:TBookmark;
begin
with ADOConnection1 do begin
   ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
   CursorLocation:=clUseClient;
   ADOConnection1.ConnectOptions:=coAsyncConnect;
   Connected:=True;
end;
with ADODataset1 do begin
     Connection:=ADOConnection1;
     CommandText:='Titles';
     CommandType:= cmdTable;
     ExecuteOptions:=[eoAsyncFetchNonBlocking];
     //Properties['initial Fetch Size'].Value:=50;
     Active:=True;
     Last;
```

```
     Signet:=GetBookmark;
end;
datasource1.DataSet:=ADODataset1;
ADODataset1.GotoBookmark(Signet);

end;
```

Selon le mode choisi, je récupérerai un signet sur le cinquantième enregistrement (eoAsyncFetchNonBlocking) ou sur le dernier (eoAsyncFetch). Aussi faut-il se méfier du type d'extraction que l'on choisit. J'ai mis en commentaire la ligne permettant la valeur du rapatriement initial car avec Delphi elle déclenche une erreur de violation d'accès.

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 :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with ADOConnection1 do begin
   ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
   CursorLocation:=clUseClient;
   ADOConnection1.ConnectOptions:=coAsyncConnect;
   Connected:=True;
end;
with ADODataset1 do begin
     Connection:=ADOConnection1;
     CursorLocation:=clUseClient;
     CursorType:=ctStatic;
     CommandText:='all Titles';
     CommandType:= cmdTable;
     ExecuteOptions:=[eoAsyncFetch];
     //Recordset.Properties['Background Fetch Size'].Value:=1000;
     Active:=True;
end;
end;

procedure TForm1.ADODataSet1FetchProgress(DataSet:
TCustomADODataSet;
  Progress, MaxProgress: Integer; var EventStatus: TEventStatus);
begin
Progressbar1.Position:=Trunc(Progress /MaxProgress *100);
Application.ProcessMessages;
end;

procedure TForm1.ADODataSet1FetchComplete(DataSet:
TCustomADODataSet;
  const Error: Error; var EventStatus: TEventStatus);
begin
datasource1.DataSet:=ADODataset1;
end;
```

Rien de bien compliqué avec ce type de codage.

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 :

```
procedure TForm1.ADODataSet1WillChangeRecord(DataSet:
TCustomADODataSet;
  const Reason: TEventReason; const RecordCount: Integer;
  var EventStatus: TEventStatus);
begin
     Case Reason of
          erAddNew:;
          erClose:;
          erDelete:;
          erFirstChange:;
          erMove:;
          erRequery:;
          erResynch:;
          erUndoAddNew :;
          erUndoDelete:;
          erUndoUpdate:;
          erUpdate:;
     end;
end;
```

Le problème de se 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'enregistrement.

```
procedure TForm1.ADODataSet1WillMove(DataSet: TCustomADODataSet;
  const Reason: TEventReason; var EventStatus: TEventStatus);
begin
if (Dataset.State =dsEdit) or (Dataset.State =dsInsert) then
   if MessageDlg('Voulez vous enregistrer les modifications?',
      mtConfirmation,[mbYes,mbNo],0)=mrYes then
          Dataset.UpdateBatch(arCurrent)
      else
          Dataset.CancelBatch(arCurrent);
end;

end.
```

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

Nous allons ici nous reposer un peu en parlant des recordset persistants. Ceux-ci sont très simples à utiliser.

Un recordset persistant est en fait un fichier de type "datagram" (format propriétaire) ou XML que l'on peut créer à partir d'un Dataset ADO ou ouvrir comme un Dataset. 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 re synchronisation)
- Comme base de travail, si la mise à jour n'est pas primordiale

Le principe de fonctionnement est le suivant :

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with ADODataset1 do begin
    ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\biblio.mdb;';
    CursorLocation:=clUseClient;
    CursorType:=ctStatic;
    CommandText:='Authors';
    CommandType:= cmdTable;
    Active:=True;
end;
Datasource1.DataSet:=ADODataset1;
end;


procedure TForm1.btnSauveClick(Sender: TObject);
var NomFichier:string;
begin
     NomFichier:= 'd:\monJeu.dtg';
     if FileExists(NomFichier) then
        DeleteFile(NomFichier);
     ADODataset1.Filter:='Author Like '+QuotedStr('s*');
     ADODataset1.Filtered:=True;
     ADODataset1.SaveToFile(NomFichier,pfADTG);
     ADODataset1.Close;
     Datasource1.DataSet:=nil;
end;

procedure TForm1.btnChargeClick(Sender: TObject);
var MonDataset:TADODataset;
begin
     MonDataset:=TADODataset.Create(Owner);
     MonDataset.LoadFromFile('d:\monJeu.dtg');
     Datasource1.DataSet:=MonDataset;
end;
```

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 un Dataset pour ouvrir mon Fichier. Vous voyez que je ne lui passe aucun paramètre de connexion. Il est donc parfaitement déconnecté. Il est possible de le connecter à la source en lui passant une chaîne de connexion.

Quelques précisions sont encore nécessaires :

- Si le recordset à une propriété Filter définie sans que celle-ci soit une constante prédéfinies, 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.
- 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 jeu d'enregistrements 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 Dataset côté client en lecture seule. Cette méthode est souvent mal utilisée car son comportement peut sembler étrange, aussi beaucoup de développeurs préfère 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 :

Dataset.Recordset.Resync adAffectAllChapters, adResync**Underlying**Values

Elle ne va concerner que les enregistrements dont vous avez modifié au moins un champ. La valeur remise à jour sera la propriété UnderlyingValue des objets Fields (CurValue en Delphi). Dans certains cas, elle peut provoquer une erreur si la synchronisation n'est pas possible. On utilise cette synchronisation principalement sur les curseurs statiques.

Synchronisation des données

De la forme :

Dataset.Recordset.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.

```
try
  adoRecordset.Recordset.Resync(3,2);
except
     on EOLEException do begin
       adoRecordset.FilterGroup:=fgConflictingRecords;
       adoRecordset.Filtered:=True;
       MessageDlg(IntToStr(AdoRecordset.recordcount)+'
enregistrements ont été supprimés',mtInformation,[mbOK],0);
     end;
end;
```

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

```
ADOQuery1.First;
ADOQuery1.Edit;
ADOQuery1.FieldByName('year born').Value:=1921;
ADOQuery1.Recordset.Resync(1,1);
```

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 attentes. Voilà pourquoi nous utiliserons plutôt les synchronisations en mode Batch, que nous allons voir dans l'exemple du traitement par lot.

Synchronisation induite

Il y a deux autres cas de synchronisation qui ne sont pas explicitement demandés par le développeur.

La mise à jour des curseurs dynamique : Elle est déclenchée par la valeur du Page TimeOut. Seule les valeurs sous jacentes sont rafraîchies. Cette opération ne repositionne pas le jeu d'enregistrements. Les enregistrements dans le cache ne sont pas synchronisés.

La mise à jour après modification. Nous avons vu comment la définir à l'aide de la propriété dynamique 'Update Resync'.

Génération de commandes

Dans cet exemple nous allons créer une fonction utilisant l'objet TADOCommand afin de remplacer l'objet Dataset dans ces actions sur la source de données. Comme je l'ai dit auparavant, il est souvent préférable d'agir par le biais de commande SQL plutôt que par le moteur de curseur. Cependant, tel que nous allons pratiquer dans cet exemple, cela ne changera globalement rien car nous allons baser nos fonctions sur l'objet Dataset. Ceci aura pour intérêt de voir comment fonctionne le moteur de curseur. Par la suite, nous verrons une approche pas les commandes paramétrées beaucoup plus proches des techniques utilisées par ADO.NET.

Les trois valeurs de TField

Un champ contient donc toujours trois valeurs qui représentent les divers états possibles.

- **OldValue** (*OriginalValue ADO*) ⇨Cette valeur représente la valeur du champ à la création du jeu d'enregistrements. Elle ne se modifie que lorsque le champ est mis à jour du fait du jeu d'enregistrement qui le contient. Ceci revient à dire que cette valeur n'est modifiée que lorsque l'enregistrement du Dataset qui contient ce champ subit une modification avec succès.
- **CurValue** (*UnderlyingValue ADO*) ⇨ Cette valeur représente la valeur présente dans la base de données. Avec certains curseurs (Dynamique, jeu de clés) c'est cette valeur qui se synchronise avec la source de données.
- **NewValue** (*Value ADO*) ⇨ C'est la valeur courante du champ dans **votre** Dataset. Lorsqu'on met un enregistrement est mis en mode modification, c'est sur cette valeur que l'on agit.

Par comparaison on peut identifier différents états pour l'enregistrement en cours.

Ces états doivent être envisagés différemment selon les cas. En mode non rafraîchis, que ce rafraîchissement soit dû au curseur ou au code, la propriété CurValue ne représente rien d'autre qu'une copie de la propriété OldValue. Par contre lorsqu'il y a synchronisation on peut distinguer les états suivants :

Les trois valeurs sont égales ⇨ c'est l'état stable. Aucune modification n'est en cours ni de votre fait, ni du fait d'un autre utilisateur.

OldValue = CurValue <>NewValue ⇨ L'état est en cours de modification dans votre application. La modification n'a pas encore été envoyée au fournisseur. Selon le mode de traitement (direct ou non) l'état changera à l'appel de la méthode Post ou UpdateBatch.

OldValue = NewValue <> CurValue ⇨ C'est l'état externe. L'enregistrement a été modifié par un autre utilisateur / jeu de données. Il faut en générale se synchroniser, sauf si cet enregistrement n'est pas la cible de vos modifications.

OldValue <> CurValue =NewValue ⇨ Etat instable. Bien que très rare, cet état est signe d'une grave désynchronisation de votre jeu d'enregistrements. Normalement, vous n'allez plus pouvoir synchroniser correctement. En général il faut ré exécuter la requête.

Les trois valeurs sont différentes ⇨ c'est l'état d'erreur. On obtient cet état en synchronisant les valeurs sous-jacentes avant une opération par lot. Cet état signifie que l'appel d'une méthode de mise à jour va provoquer une erreur.

Similaire au moteur de curseur

Dans ce premier exemple nous allons créer une fonction qui se comporte quasiment comme le moteur de curseur. Cette fonction va attendre deux paramètres, un pointeur sur votre jeu d'enregistrement et un paramètre de comportement, similaire à la propriété "Update Criteria". La fonction doit rendre le jeu de données dans un état stable. Après la fonction nous allons étudier plus en détail certains points du code.

Cette fonction en l'état fait les mêmes actions que le moteur de curseur client, à l'exception de la valeur du Where qui est basée sur les valeurs courantes plutôt que sur les valeurs d'origine.

```
procedure TForm1.ComAjout(var ptrMonJeu: TADODataset; const Critere:
integer);
Var strTable, strCommand: WideString;
strNomChamp, strValChamp: TStrings;
compteur, NbModif: Integer;
MaCommande: TADOCommand;
ValTemp:WideString;
begin
   ptrMonJeu.Filtered:=False;
   ptrMonJeu.FilterGroup:=fgPendingRecords;
   ptrMonJeu.Filtered:=True;
   if ptrMonJeu.RecordCount=0 then
      exit;
   ptrMonJeu.First;
   MaCommande:=TADOCommand.Create(nil);

strTable:=ptrMonJeu.recordset.Fields[0].Properties['BASETABLENAME'].
Value;
   while not ptrMonJeu.Eof do begin
      ptrMonJeu.Recordset.Resync(1,1);
      if (rsNew in ptrMonJeu.RecordStatus) then
         begin
            strNomChamp:=TStringList.Create;
            strValChamp:=TStringList.Create;
            try
              for compteur := 0 to ptrMonJeu.FieldCount-1 do begin
                 if ptrMonJeu.Fields[compteur].DataType<> ftAutoInc
then begin
                    strNomChamp.Add('[' +
ptrMonJeu.Recordset.Fields[compteur].Name + ']');
                 if (ptrMonJeu.Fields[compteur].DataType= ftString)
or (ptrMonJeu.Fields[compteur].DataType= ftWideString) then

strValChamp.Add(QuotedStr(vartostr(ptrMonJeu.Recordset.Fields[compte
ur].UnderlyingValue)))
                 else if ptrMonJeu.Fields[compteur].DataType= ftDate
then
                    strValChamp.Add('#' +
vartostr(ptrMonJeu.Recordset.Fields[compteur].UnderlyingValue)+ '#')
                 else
```

```
strValChamp.Add(vartostr(ptrMonJeu.Recordset.Fields[compteur].Underl
yingValue));
                 end;
              end;
              strCommand:=StringReplace('INSERT INTO [' + strTable +
'] (' + strNomChamp.CommaText +') VALUES(' +
strValChamp.DelimitedText +')','"','',[rfReplaceAll]);
            finally
              strNomChamp.Free;
              strValChamp.Free;
            end;
         end
      else if (rsDeleted in ptrMonJeu.RecordStatus) then
         begin
            strNomChamp:=TStringList.Create;
            strValChamp:=TStringList.Create;
            try
              for compteur := 0 to ptrMonJeu.FieldCount-1 do begin
                 if
strtobool(vartostr(ptrMonJeu.recordset.Fields[compteur].Properties['
KEYCOLUMN'].Value)) then
                    strNomChamp.Add('[' +
ptrMonJeu.Recordset.Fields[compteur].Name + ']');
                 if (ptrMonJeu.Fields[compteur].DataType= ftString)
or (ptrMonJeu.Fields[compteur].DataType= ftWideString) then

strValChamp.Add(QuotedStr(vartostr(ptrMonJeu.Recordset.Fields[compte
ur].UnderlyingValue)))
                 else if ptrMonJeu.Fields[compteur].DataType= ftDate
then
                    strValChamp.Add('#' +
vartostr(ptrMonJeu.Recordset.Fields[compteur].UnderlyingValue)+ '#')
                 else

strValChamp.Add(vartostr(ptrMonJeu.Recordset.Fields[compteur].Underl
yingValue));
              end;
              strCommand:='DELETE FROM ' + strTable + ' WHERE ' +
strNomChamp[0] +'='+strValChamp[0] ;
              if strNomChamp.Count>1 then
                 for compteur:=1 to strNomChamp.Count-1 do
                    strCommand:=strCommand + ' AND ' +
strNomChamp[0] +'='+strValChamp[0] ;
            finally
              strNomChamp.Free;
              strValChamp.Free;
            end;
         end
      else
         begin
            strNomChamp:=TStringList.Create;
            strValChamp:=TStringList.Create;
            try
              for compteur := 0 to ptrMonJeu.FieldCount-1 do begin
                 Valtemp:='[' +
ptrMonJeu.Recordset.Fields[compteur].Name + '] = ';
                 if
ptrMonJeu.Fields[compteur].CurValue<>ptrMonJeu.Fields[compteur].NewV
alue then begin
                    if (ptrMonJeu.Fields[compteur].DataType=
ftString) or (ptrMonJeu.Fields[compteur].DataType= ftWideString)
then
```

```
strNomChamp.Add(Valtemp+QuotedStr(vartostr(ptrMonJeu.Recordset.Field
s[compteur].Value)))
                    else if ptrMonJeu.Fields[compteur].DataType=
ftDate then
                       strNomChamp.Add(Valtemp+'#' +
vartostr(ptrMonJeu.Recordset.Fields[compteur].Value)+ '#')
                    else

strNomChamp.Add(Valtemp+vartostr(ptrMonJeu.Recordset.Fields[compteur
].Value));
                 end;
                 case Critere of
                 1: begin// juste les champs clés
                       if
strtobool(vartostr(ptrMonJeu.recordset.Fields[compteur].Properties['
KEYCOLUMN'].Value)) then begin
                          if (ptrMonJeu.Fields[compteur].DataType=
ftString) or (ptrMonJeu.Fields[compteur].DataType= ftWideString)
then

strValChamp.Add(Valtemp+QuotedStr(vartostr(ptrMonJeu.Recordset.Field
s[compteur].UnderlyingValue)))
                          else if
ptrMonJeu.Fields[compteur].DataType= ftDate then
                             strValChamp.Add(Valtemp+'#' +
vartostr(ptrMonJeu.Recordset.Fields[compteur].UnderlyingValue)+ '#')
                          else

strValChamp.Add(Valtemp+vartostr(ptrMonJeu.Recordset.Fields[compteur
].UnderlyingValue));
                       end;
                    end;
                 2: begin// juste les champs clés et les champs
modifiés
                       if
(strtobool(vartostr(ptrMonJeu.recordset.Fields[compteur].Properties[
'KEYCOLUMN'].Value))) or
(ptrMonJeu.Fields[compteur].CurValue<>ptrMonJeu.Fields[compteur].New
Value) then
                          if (ptrMonJeu.Fields[compteur].DataType=
ftString) or (ptrMonJeu.Fields[compteur].DataType= ftWideString)
then

strValChamp.Add(Valtemp+QuotedStr(vartostr(ptrMonJeu.Recordset.Field
s[compteur].UnderlyingValue)))
                          else if
ptrMonJeu.Fields[compteur].DataType= ftDate then
                             strValChamp.Add(Valtemp+'#' +
vartostr(ptrMonJeu.Recordset.Fields[compteur].UnderlyingValue)+ '#')
                          else

strValChamp.Add(Valtemp+vartostr(ptrMonJeu.Recordset.Fields[compteur
].UnderlyingValue));
                    end;
                 3: begin// tous les champs
                      if (ptrMonJeu.Fields[compteur].DataType=
ftString) or (ptrMonJeu.Fields[compteur].DataType= ftWideString)
then

strValChamp.Add(Valtemp+QuotedStr(vartostr(ptrMonJeu.Recordset.Field
s[compteur].UnderlyingValue)))
```

```
                    else if ptrMonJeu.Fields[compteur].DataType=
ftDate then
                       strValChamp.Add(Valtemp+'#' +
vartostr(ptrMonJeu.Recordset.Fields[compteur].UnderlyingValue)+ '#')
                    else

strValChamp.Add(Valtemp+vartostr(ptrMonJeu.Recordset.Fields[compteur
].UnderlyingValue));
                    end;
                end;
             end;
             strCommand:='UPDATE ' + strTable + ' SET ' +
strNomChamp[0];
                if strNomChamp.Count>1 then begin
                   for compteur:=1 to strNomChamp.Count-1 do
                       strCommand:=strCommand + ' AND ' +
strNomChamp[compteur];
                end;
                strCommand:=strCommand + ' WHERE ' +
strValChamp[0];
                if strValChamp.Count>1 then begin
                   for compteur:=1 to strValChamp.Count-1 do
                       strCommand:=strCommand + ' AND ' +
strValChamp[compteur];
                end;
            finally
              strNomChamp.Free;
              strValChamp.Free;
            end;
         end;
      with MaCommande do begin
         Connection:=ptrMonJeu.Connection;
         CommandType:=cmdText;
         CommandText:=strCommand;
         Try
            Execute(NbModif,EmptyParam);
         Except
            ptrMonJeu.CancelBatch(arCurrent);
         end;
      end;
      ptrMonJeu.CancelBatch(arCurrent);
      ptrMonJeu.Next;
   end;
   MaCommande.Free;
end;
```

Comme vous le voyez, je ne passe aucune information à ma fonction qui n'est pas contenu dans le DataSet. Le principe de fonctionnement est simple. Je filtre pour n'avoir que les enregistrements ayant une modification en cours. Attention, les informations de schéma contenues dans le Dataset ne sont pertinentes que parce que j'utilise un curseur coté client avec une mise à jour par lot. Je vais ensuite balayer mon Dataset est écrire une commande reflétant l'action demandée.

Le code peut sembler un peu abscons, mais il ne s'agit que de l'écriture d'une requête action.

Il s'agit là évidemment d'un exemple assez simple. En ajoutant quelques informations de schéma, je pourrais écrire un générateur de requête plus efficace que celui-la qui permettrait une écriture de requête multi-table.

Commande paramétrée

Cette deuxième approche est beaucoup plus claire. Elle ressemble à l'approche ADO.NET de programmation. Dans ADO.NET on utilise des commandes paramétrées pour agir sur la source de données. Il existe de nombreuses méthodes pour aborder ce type de stratégies que j'aborderai dans un autre article sur ADO.NET mais nous allons étudier le principe général ainsi que les pièges à éviter.

A la base, soit on travaille avec une commande "à la volée" soit on passe par des fonctions. Les deux techniques présentent des avantages, mais dans le cas qui m'intéresse, je vais utiliser des fonctions. Dans ADO.NET on gère autant de commande que de requête action, il n'y a alors plus qu'à passer les paramètres à la commande pour gérer celle-ci. Dans une approche plus évoluée, je peux générer une commande entièrement modulable, donc similaire au moteur de curseur en utilisant un code de type ADO.NET.

Prenons l'exemple suivant qui gère une requête de modifications.

```
procedure TForm1.btnTraiteClick(Sender: TObject);
var adoRecordset:TADODataset;
adoConnection:TADOConnection;
RecValue: integer;
TabParam:array of variant;
begin
  randomize;
  adoConnection:=TADOConnection.Create(Owner);
  with adoConnection do begin
     ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=c:\Biblio.mdb;';
     CursorLocation:=clUseClient;
     Connected:=True;
  end;
  adoRecordset:=TADODataset.Create(Owner);
  with adoRecordset do begin
     Connection:=adoConnection;
     CursorType:=ctStatic;
     LockType:=ltBatchOptimistic;
     CommandType:= cmdText;
     CommandText:='SELECT * FROM Authors';
     Active:=True;
     Recordset.Properties['Update Resync'].Value:=2;
     First;
     MaCommand:=TADOCommand.Create(nil);
     MaCommand.Connection:=adoConnection;
     TabParam:=vararrayof(['bidou',fieldvalues['Au_Id']]);
     RecValue:=CommUpdate(adoconnection,TabParam);
     if recvalue=0 then
        showmessage('pas de correspondance');
  end;
end;

function TForm1.CommUpdate(const ListeParam: array of
variant):integer;
var ParamRequete array of variant;
compteur: integer;
begin
   With MaCommand do begin
       CommandText:='UPDATE Authors SET [author]=:NewValChamp WHERE
Au_Id=:ValCle';
       CommandType:=cmdtext;
       Execute(result,vararrayof(ListeParam));
   end;
end;
```

Je suis là dans un codage standard ADO.NET qui gère une commande par fonction. C'est un code très sur mais pas très souple puisque si je veux pouvoir modifier de nombreuses tables je vais me retrouver à gérer autant de fonctions que de tables pour un seul type d'action. Je peux donc concevoir une fonction beaucoup plus puissante, répondant aux modifications de plusieurs tables. Toujours dans le même ordre d'idée, je vais m'astreindre à travailler dans le respect strict de la concurrence optimiste.

Donc j'utilise une fonction comme celle ci dessous

```
function TForm1.CommUpdate(const ListeParam: array of variant; const
GestParam: integer):integer;
var ParamRequete: variant;
```

```
compteur: integer;
begin
   MaCommand.CommandType:=cmdtext;
   if GestParam=0 then
       MaCommand.Execute(result,vararrayof(ListeParam))
   else
       begin
         MaCommand.CommandText:='UPDATE ' + ListeParam[0] + ' SET ';

ParamRequete:=VarArrayCreate([0,Trunc(length(ListeParam)/2)-
1],varVariant);
         compteur:=1;
         MaCommand.CommandText:=MaCommand.CommandText+ '[' +
ListeParam[compteur]+ ']=:param' + IntToStr(compteur);
         compteur:=compteur+1;
         While compteur<=GestParam do begin
             MaCommand.CommandText:=MaCommand.CommandText+ ' AND ['
+ ListeParam[compteur]+ ']=:param' + IntToStr(compteur);
             compteur:=compteur+1;
         end;
         MaCommand.CommandText:=MaCommand.CommandText+ ' WHERE [' +
ListeParam[compteur]+ ']=:param' + IntToStr(compteur);
         compteur:=compteur+1;
         While compteur<=Trunc(length(ListeParam)/2) do begin
             MaCommand.CommandText:=MaCommand.CommandText+ ' AND ['
+ ListeParam[compteur]+ ']=:param' + IntToStr(compteur);
             compteur:=compteur+1;
         end;
         for compteur:= 0 to Trunc(length(ListeParam)/2)-1 do

ParamRequete[compteur]:=listeparam[Trunc(length(ListeParam)/2)+1+com
pteur];
         MaCommand.Execute(result,ParamRequete)
       end;
end;
```

Une telle fonction attend comme paramètre :

Un tableau de variant contenant le nom de la table puis les noms des champs modifiés puis les noms des champs, puis la liste des valeurs de ces champs dans le même ordre.

Un entier représentant le nombre de champs modifiés. Si cet entier vaut zéro, le tableau de variant ne contiendra que la liste des valeurs.

Je sais bien que ce style de programmation semble affreusement tordu, mais comme vous l'avez maintenant remarqué, j'adore ça. Plus sérieusement, cela est aussi très efficace. Bon, où en étais-je avant d'être brutalement interrompu par votre remarque sur ma façon de coder.

Ah! oui. Un appel explicite de cette fonction dans le cadre de ma table 'Authors' pourrait être :

```
TabParam:=VarArrayOf(['authors','author','Au_Id','author','year
born','bidou',fieldvalues['au_id'],fieldvalues['author'],fieldvalues
['year born']]);
     RecValue:=CommUpdate(TabParam,1);
```

Voilà qui devrait finir de déprimer ceux qui ont pu lire jusque là. Evidemment dans le code on n'écrit pas chaque appel comme celui-là, à moins de vouloir se transformer rapidement en chèvre.

Prenons de l'avance sur mon exemple suivant et imaginons que nous souhaitions ajouter une valeur aléatoire à l'année de naissance.

La modification est de la forme

```
CommandText:=''SELECT * FROM Authors WHERE [year born] IS NOT NULL';
     Active:=True;
     Recordset.Properties['Update Resync'].Value:=2;
     First;
     MaCommand:=TADOCommand.Create(nil);
     MaCommand.Connection:=adoConnection;
     while not Eof do begin
```

```
         Edit;
         FieldByName('year
born').Value:=adoRecordset.FieldByName('year
born').Value+RandomRange(1,5);
         Post;
         Next;
     end;
```

Dans le cas de ma fonction je devrais utiliser

```
CommandText:='SELECT * FROM Authors WHERE [year born] IS NOT NULL';
Active:=True;
Recordset.Properties['Update Resync'].Value:=2;
First;
MaCommand:=TADOCommand.Create(nil);
MaCommand.Connection:=adoConnection;
SetLength(TabParam,3+fieldcount*2);
tabparam[0]:='authors'; // nom de la table
tabparam[1]:='year born'; // nom du champ modifiés
tabparam[2+fieldcount]:=FieldByName('year
born').Value+RandomRange(1,5);
for compteur:=0 to fieldcount-1 do begin
    tabparam[compteur+2]:=recordset.Fields[compteur].Name;
    tabparam[compteur+3+fieldcount]:=Fields[compteur].AsVariant;
end;
RecValue:=CommUpdate(TabParam,1);
if recvalue=0 then
   showmessage('pas de correspondance');
Next;
setlength(TabParam,4);
while not Eof do begin
  tabparam[0]:=FieldByName('year born').Value+RandomRange(1,5);
  for compteur:=0 to fieldcount-1 do
      tabparam[compteur+1]:=Fields[compteur].AsVariant;
  RecValue:=CommUpdate(TabParam,0);
  if recvalue=0 then
      showmessage('pas de correspondance');
  Next;
end;
```

Le code est certes un petit peu plus long, mais c'est loin d'être insurmontable. Avec ce type de fonction je remplace complètement le moteur de curseur par une action par des commandes. Qui plus est, je ne modifie aucune des valeurs de mon Dataset.

Seulement cette fonction ne va pas toujours fonctionner, pour les mêmes raisons que leurs homologues ADO.NET. En effet, que va-t-il se passer si une des valeurs est NULL.

Ma fonction écrit une chaîne de commande, dans mon exemple, qui est :

```
'UPDATE authors SET [year born]=:param1 WHERE [Au_ID]=:param2 AND
[Author]=:param3 AND [Year Born]=:param4'
```

Si j'envisage le cas du premier enregistrement de ma base je vais générer la commande suivante :

```
'UPDATE authors SET [year born]=3 WHERE [Au_ID]=1 AND [Author]=
'Jacobs, Russell' AND [Year Born]=NULL'
```

Là je vais avoir 0 comme valeur de retour car la notation [Year Born]=NULL ne permet pas d'identifier l'enregistrement. Donc je dois modifier ma fonction de la façon suivante :

```
function TForm1.CommUpdate(const ListeParam: array of variant; const
GestParam: integer):integer;
var ParamRequete: variant;
compteur: integer;
begin
   MaCommand.CommandType:=cmdtext;
   if GestParam=0 then
       MaCommand.Execute(result,vararrayof(ListeParam))
   else
```

```
        begin
          MaCommand.CommandText:='UPDATE ' + ListeParam[0] + ' SET ';

ParamRequete:=VarArrayCreate([0,Trunc(length(ListeParam)/2)-
1],varVariant);
          compteur:=1;
          MaCommand.CommandText:=MaCommand.CommandText+ '[' +
ListeParam[compteur]+ ']=:param' + IntToStr(compteur);
          compteur:=compteur+1;
          While compteur<=GestParam do begin
              MaCommand.CommandText:=MaCommand.CommandText+ ' AND ['
+ ListeParam[compteur]+ ']=:param' + IntToStr(compteur);
              compteur:=compteur+1;
          end;
          MaCommand.CommandText:=MaCommand.CommandText+ ' WHERE ([' +
ListeParam[compteur]+ ']=:param' + IntToStr(compteur) + ' OR [' +
ListeParam[compteur]+ '] IS NULL)';
          compteur:=compteur+1;
          While compteur<=Trunc(length(ListeParam)/2) do begin
              MaCommand.CommandText:=MaCommand.CommandText+ ' AND (['
+ ListeParam[compteur]+ ']=:param' + IntToStr(compteur)+ ' OR [' +
ListeParam[compteur]+ '] IS NULL)';
              compteur:=compteur+1;
          end;
          for compteur:= 0 to Trunc(length(ListeParam)/2)-1 do

ParamRequete[compteur]:=listeparam[Trunc(length(ListeParam)/2)+1+com
pteur];
          MaCommand.Execute(result,ParamRequete)
        end;
end;
```

Et voilà une fonction correctement écrite. Ce passage était sûrement un peu fastidieux, mais maintenant vous voyez exactement comment fonctionne le moteur de curseur et les commandes paramétrées, nous allons pouvoir rentrer dans des sujets un peu plus complexes.

Traitement par lot

Le traitement par lot est fondamentalement une opération déconnectée. Dans le principe, il s'agit de pratiquer un certain nombre d'opérations sur les données au sein de l'application cliente, puis de transmettre l'ensemble de ces modifications à la source de données. Le traitement par lot se gère très différemment selon qu'il pourra y avoir ou non concurrence.

Le traitement par lot doit suivre quelques règles de base

- **Des données connexes**

On peut sensiblement tout mettre lors d'un traitement par lot. Il faut pourtant s'astreindre à ne mettre dans un lot que des opérations qui aient une logique d'ensemble.

- **Unicité du type de traitement**

On travaille soit avec un code client (modification du Dataset et/ou commande SQL), soit avec des procédures stockées, mais pas avec les deux en même temps.

- **Intégrité de l'opération**

Un traitement par lot n'est pas une transaction. En cela il n'est pas nécessairement atomique. Néanmoins, les erreurs doivent être gérées au sein du traitement afin de garantir la cohérence des modifications

Dans l'exemple suivant, nous allons regarder un cas ou la concurrence ne peut pas arriver. Globalement un traitement par lot peut conduire à trois états.

- Réussite : Toutes les opérations sont correctement exécutées.
- Réussite partielle : Certaines opérations ne peuvent pas être exécutées. Ceci n'est pas gênant si le cas est prévu.
- Rejet : L'ensemble du lot doit être rejeté. La décision de rejet global doit être prise avant le début du traitement (Cancel) ou à la fin (Restauration).

Le cas de l'exemple est le suivant. Je vais ajouter entre un et cinq ans à l'année de naissance de tous les auteurs qui ont une année de naissance définie de façon aléatoire. Dans le même temps je dois supprimer tous les auteurs dont l'année de naissance est 1935. Je veux une opération de type transactionnelle c'est à dire que si toutes les modifications ne sont pas acceptées, toutes doivent être rejetées.

Ce cas est particulièrement abominable mais il va nous permettre de saisir plusieurs concepts intéressant.

```
procedure TForm1.btnTraiteClick(Sender: TObject);
var adoRecordset:TADODataset;
adoCommand:TADOCommand;
adoConnection:TADOConnection;
compteur:Integer;
adoErreurs:Errors;
TestErr:Boolean;
begin
  randomize;
  adoConnection:=TADOConnection.Create(Owner);
  with adoConnection do begin
     ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=D:\Biblio.mdb;';
     CursorLocation:=clUseClient;
     Connected:=True;
  end;
  adoRecordset:=TADODataset.Create(Owner);
  with adoRecordset do begin
     Connection:=adoConnection;
     CursorType:=ctStatic;
     LockType:=ltBatchOptimistic;
     CommandType:= cmdText;
     CommandText:='SELECT * FROM Authors WHERE [year born] IS NOT
NULL';
     Active:=True;
     Filter:='[year born] = 1938';
     Filtered:=True;
     if RecordCount>0 then
        begin
           First;
           while not Eof do
              begin
                 Delete;
                 Next;
              end;
        end;
     Filtered:=False;
     Filter:='[year born] <> 1938';
     Filtered:=True;
     First;
     while not Eof do begin
         Edit;
         FieldByName('year
born').Value:=adoRecordset.FieldByName('year
born').Value+RandomRange(1,5);
         Post;
         Next;
     end;
     SaveToFile('d:\temp.adtg',pfADTG);
     Filtered:=False;
     TestErr:=False;
     try
        UpdateBatch(arAll);
     except
           adoErreurs:= adoConnection.Errors;
```

```
           adoCommand:=TADOCommand.Create(nil);
           For compteur:=0 to adoErreurs.Count - 1 do
               if adoErreurs[compteur].Number <>-2147467259 Then
TestErr:=True;
           if Not TestErr then begin
              FilterGroup:=fgConflictingRecords;
              Filtered:=True;
              if RecordCount>0 then

MessageDlg(IntToStr(adoRecordset.Recordset.RecordCount),
mtInformation,[mbOK], 0);
              Filtered:=False;
              FilterGroup:=fgPendingRecords;
              Filtered:=True;
              if RecordCount>0 then begin
                 First;
                 while not Eof do begin
                    if rsDeleted in adoRecordset.RecordStatus then
                       adorecordset.CancelBatch(arCurrent);
                    With adoCommand do begin
                        Connection:=AdoConnection;
                        CommandType:=cmdText;
                        CommandText:='DELETE * FROM Titles WHERE
ISBN IN (SELECT ISBN FROM [Title Author] WHERE Au_Id =' +
adoRecordset.FieldByName('Au_Id').AsString + ')';
                        Execute;
                        CommandText:='DELETE * FROM [Title Author]
WHERE Au_Id= ' + adoRecordset.FieldByName('Au_Id').AsString;
                        Execute;
                        adorecordset.Delete;
                        adorecordset.UpdateBatch(arCurrent);
                    end;
                    Next;
                 end;
              end;
           end
           else begin
              Close;
              LoadFromFile('d:\temp.adtg');
              Connection:=adoConnection;
              First;
              adoCommand.Connection:=AdoConnection;
              adoCommand.CommandType:=cmdText;
              adoCommand.CommandText := 'UPDATE Authors SET [year
born] =:ValCible WHERE Au_Id = :ValCle AND [year born]= :ValSource';
              while not EOF do begin
                    if (rsModified in adoRecordset.RecordStatus)
then begin
                       Cancel;
                       Recordset.Resync(1,1);
                       if FieldByName('year
born').OldValue<>FieldByName('year born').CurValue then

adoCommand.Execute(VarArrayOf([FieldByName('year
born').OldValue,FieldByName('Au_Id').CurValue,FieldByName('year
born').CurValue]));
                    end;
                    next;
              end;
           end;

     end;
  end;
```

```
end;
```

Rien que du bonheur.

Rentrons dans l'étude détaillée. Jusqu'à l'appel de la méthode UpdateBatch rien de nouveau. Tout le code de l'exception vient de la volonté d'avoir une opération globalement acceptée ou globalement rejetée.

Je commence toujours par mettre les erreurs ado dans une collection indépendante car

- Une nouvelle erreur effacerait l'ancienne collection
- Le parcours de la collection sur la connexion tend à planter l'exécution.

Je parcours ensuite ma collection avec le code :

```
if adoErreurs[compteur].Number <>-2147467259 Then TestErr:=True;
```

En cherchant une erreur de numéro –2147467259, je recherche une suppression illégale de type tentative de suppression en cascade. Attention, ce numéro d'erreur pourrait aussi être obtenu avec une erreur de définition de fournisseur, mais une telle erreur se produirai avant le Try. Si seul ce type d'erreur est présent sur la connexion, je peux poursuivre l'opération sinon je vais devoir restaurer.

Il est important de noter qu'à ce point, les opérations valides ont déjà été effectuées.

Le premier filtrage, par les enregistrements en erreur, qui est la façon standard de procéder est mise ici à titre d'exemple car elle ne fonctionnera pas dans ce cas. En effet, ce type d'erreur n'est jamais attribué à un enregistrement spécifique, ce qui est d'ailleurs une incohérence ADO. Regardons quand même ici le cas standard de gestion des erreurs

Gestion standard des erreurs

La technique utilisée par la gestion standard consiste à filtrer les enregistrements en erreurs, et à gérer les cas en fonction de leurs statuts.

```
try
  UpdateBatch(arAll);
except
  if adoConnection.Errors.Count>0 then begin
     adoCommand:=TADOCommand.Create(nil);
     FilterGroup:=fgConflictingRecords;
     Filtered:=True;
     First;
     while not EOF do begin
       Case trunc(power(2,trunc(log2(Recordset.Status)))) of
         16: showmessage('invalide'); //rsInvalid
         256: showmessage('invalide'); // RsCanceled
         1024: showmessage('invalide'); // RsCantRelease
         2048:// RsConcurrencyViolation
            begin;
              With adoCommand do begin
                Connection:=AdoConnection;
                CommandType:=cmdText;
                CommandText:='Update Authors SET [year
born]='+vartostr(fieldbyname('year born').NewValue) +' WHERE [year
born]='+vartostr(fieldbyname('year born').CurValue)+' AND
Au_Id='+vartostr(fieldbyname('Au_Id').CurValue);
                Execute;
              end;
            end;
         4096: showmessage('invalide'); // RsIntegrityViolation
       end;
     Next;
  end;
end;
```

Notez que je travaille sur la propriété Status de l'objet Recordset ADO et non sur la propriété RecordStatus exposée par Delphi ; ceci afin de pouvoir utiliser une structure Case. En effet, Case attend un entier pour fonctionner alors que RecordStatus renvoie un jeu de réponse.

C'est d'ailleurs aussi le cas de la propriété Status qui renvoie la somme des statuts constatés pour l'enregistrement. La ligne `trunc(power(2,trunc(log2(Recordset.Status))))`

sert à évaluer le statut de plus haute valeur, ce qui permet de voir les erreurs avant le statut de modification. Je m'explique.

Si on regarde le tableau des valeurs de la propriété Status, on constate qu'il y a un ordre dans les statuts. Les valeurs de 0 à 8 correspondent au statut de modification, les valeurs supérieures aux statuts d'erreur. Un enregistrement en erreur à toujours un statut égale à la somme de son statut de modification et du statut de l'erreur. Il peut parfois y avoir plusieurs erreurs pour le même enregistrements. Dès lors le code strict de traitement devrait être de la forme :

```
try
   UpdateBatch(arAll);
except
  if adoConnection.Errors.Count>0 then begin
       FilterGroup:=fgConflictingRecords;
       Filtered:=True;
       First;
       while not EOF do begin
         Statut:= trunc(power(2,trunc(log2(Recordset.Status))));
           while Statut>8 do begin
             Case Statut of
               16: showmessage('invalide');//RsInvalid
               64: showmessage('invalide'); // RsMultipleChanges
               128: showmessage('invalide');// RsPendingChanges
               256: showmessage('invalide'); // RsCanceled
               1024: showmessage('invalide'); // RsCantRelease
               2048: showmessage('invalide'); //
RsConcurrencyViolation
               4096: showmessage('invalide'); //
RsIntegrityViolation
               8192: showmessage('invalide'); //
RsMaxChangesExceeded
               16384: showmessage('invalide'); // RsObjectOpen
               32768: showmessage('invalide'); // RsOutOfMemory
               65536: showmessage('invalide'); // RsPermissionDenied
               131072: showmessage('invalide'); // RsSchemaViolation
               262144: showmessage('invalide'); // RsDBDeleted
                 end;
                 Statut:=Recordset.Status-Statut;
             end;
             Next;
           end;
       end;
   end;
end;
```

Néanmoins je ne suis pas obligé d'utiliser une structure Case. Le même code pourrait travailler directement sur la propriété RecordStatus avec une structure if.

```
while not EOF do begin
  if (RsInvalid in Recordstatus) then
     showmessage('invalide');//RsInvalid
  if (RsMultipleChanges in Recordstatus) then
     showmessage('invalide');//RsMultipleChanges
  if (RsPendingChanges in Recordstatus) then
     showmessage('invalide');//RsPendingChanges
  if (RsCanceled in Recordstatus) then
     showmessage('invalide');//RsCanceled
  if (RsCantRelease in Recordstatus) then
     showmessage('invalide');//RsCantRelease
  if (RsConcurrencyViolation in Recordstatus) then
     showmessage('invalide');//RsConcurrencyViolation
  if (RsIntegrityViolation in Recordstatus) then
     showmessage('invalide');//RsIntegrityViolation
  if (RsMaxChangesExceeded in Recordstatus) then
     showmessage('invalide');//RsMaxChangesExceeded
```

```
   if (RsObjectOpen in Recordstatus) then
      showmessage('invalide');//RsObjectOpen
   if (RsOutOfMemory in Recordstatus) then
      showmessage('invalide');//RsOutOfMemory
   if (RsPermissionDenied in Recordstatus) then
      showmessage('invalide');//RsPermissionDenied
   if (RsSchemaViolation in Recordstatus) then
      showmessage('invalide');//RsSchemaViolation
   if (RsDBDeleted in Recordstatus) then
       showmessage('invalide');//RsDBDeleted
adoRecordset.Next;
end;
```

Actions correctives

Dès lors que je dois gérer des actions correctives selon mes erreurs, il y a globalement trois décisions possibles.

Pas de gestion

Je peux ignorer l'erreur, considérant que celle-ci ne porte pas à conséquence. C'est un choix qu'il convient de bien peser. Ce genre de pratique est à éviter à mes yeux, mais dans certains cas c'est le seul choix judicieux surtout lorsqu'il y a possibilité de concurrence. Par exemple, si un enregistrement a été supprimé par un autre utilisateur, il y aurait incohérence à recréer celui-ci par une correction.

Avec l'objet Command

C'est en général la méthode que j'utilise. Au lieu de passer par le moteur de curseur client on crée un objet Command pour appliquer directement les modifications sur la source.

Il n'est pas conseillé d'utiliser la commande intégrée à votre objet DataSet pour faire c'est modifications car cela pourrait influer sur votre jeu de données, mais cela est possible.

Il est utile alors d'avoir créé une connexion indépendante ce qui permet de placer plusieurs objets ADO dessus. Toutefois il faut alors récupérer la collection des erreurs de la connexion dans un objet indépendant avant d'utiliser la commande. En effet, si la commande engendre une erreur sur la connexion du jeu d'enregistrements, l'erreur effacera la collection d'erreurs précédentes, et vous pourrez dire adieu à une éventuelle lecture de cette collection, les statuts quant à eux n'étant pas modifiés.

Il est possible selon les besoins de récupérer le nombres de modifications effectivement réalisées pour détecter un éventuel problème masqué.

On travaille d'une manière assez similaire au moteur de curseur. On détermine une commande (généralement paramétrée) avec les champs que l'on souhaite utiliser comme critères, il faut donc faire attention aux éventuelles fautes de concurrence.

Il est évidemment possible d'utiliser une fonction en passant les paramètres adéquats, comme nous l'avons vu précédent.

Par le biais du Dataset

C'est une méthode que je ne vous recommande pas. Cela consiste à travailler uniquement avec le dataset et à boucler sur celui-ci jusqu'à ce que toute les modifications soit validées. Pour cela il faut synchroniser à chaque passage dans la boucle puis adapter les modifications au fur et à mesure. Outre l'extrême lourdeur de ce genre de code, on s'empêtre souvent dans des situations ou la restauration devient impossible.

Utiliser une transaction

Vous devez vous dire que mon exemple est bien compliquer puisque je n'aurais qu'à utiliser une transaction plutôt que de me torturer l'esprit avec mon code interminable. Et bien soit utilisons une transaction.

```
procedure TForm1.btnTraiteClick(Sender: TObject);
var adoRecordset:TADODataset;
adoConnection:TADOConnection;
begin
  randomize;
  adoConnection:=TADOConnection.Create(Owner);
  with adoConnection do begin
     ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Biblio.mdb;';
     CursorLocation:=clUseClient;
     Connected:=True;
  end;
  adoRecordset:=TADODataset.Create(Owner);
  with adoRecordset do begin
     Connection:=adoConnection;
     CursorType:=ctStatic;
     LockType:=ltBatchOptimistic;
     CommandType:= cmdText;
     CommandText:='SELECT * FROM Authors WHERE [year born] IS NOT
NULL';
     Active:=True;
     Filter:='[year born] = 1938';
     Filtered:=True;
     if RecordCount>0 then
        begin
           First;
           while not Eof do
              begin
                 Delete;
                 Next;
              end;
        end;
     Filtered:=False;
     Filter:='[year born] <> 1938';
     Filtered:=True;
     First;
     while not Eof do begin
         Edit;
         FieldByName('year
born').Value:=adoRecordset.FieldByName('year
born').Value+RandomRange(1,5);
         Post;
         Next;
     end;
     Filtered:=False;
     Connection.BeginTrans;
     try
        UpdateBatch(arAll);
        Connection.CommitTrans;
     except
        Connection.RollbackTrans;
     end;
  end;
end;
```

Comme nous le voyons, c'est en effet nettement plus simple. Il serait d'ailleurs possible d'imbriquer des transactions pour pouvoir valider des modifications partielles.

Accès concurrentiel

C'est là un immense sujet, bien trop vaste pour être traité en détail dans cet article. On peut concevoir l'aphorisme suivant de Frédéric Brouard comme une bonne base de départ.

S'il peut y avoir concurrence, il va y avoir concurrence.

Partant de ce principe, voyons quels sont les moyens qui sont mit à notre disposition pour combattre les risques engendrés par l'accès concurrentiel. En fait ces moyens ne sont pas inhérents à ADO mais plutôt fonctions des SGBD qui les gèrent et permettent ou non leurs mises en œuvre. Comme cet article se limite à la connaissance d'ADO, je ne présenterais pas de code dans ce chapitre, mais seulement quelques points à ne pas perdre de vues.

Les procédures stockées

Les procédures stockées consistent à utiliser du code, stockée dans le SGBD et agissant comme une source unique d'action. Ce code est souvent dans le langage du SGBD, parfois en SQL. L'intérêt, au niveau de la sécurité n'est pas évident contrairement à ce que vous allez souvent lire. Il est important de tordre le cou de ce canard définitivement, une procédure stockée n'est pas conceptuellement sûre. Comme il ne s'agit de rien d'autre que de code stocké sur le serveur, et dans le SGBD, celui-ci n'est sur que s'il a été **correctement conçu**.

Si vous n'avez pas écrit la procédure (parfois aussi si vous l'avez écrite mais là le problème est différent) et si vous n'avez pas lu son code, ne considérez jamais une procédure stockée comme un moyen sûr dans une vision concurrentielle. Cette légende vient d'un amalgame atroce entre procédure stockée et transaction, tout comme de nombreux développeurs se mélangent encore entre index et contraintes. Donc à priori, une procédure stockée, du point de vue de la concurrence n'est pas une garantie. Ne perdez pas cette vision de l'esprit, la supposée sûreté des procédures stockée à engendrée bien plus de problèmes d'intégrité que les verrous mortels et autre moment de bonheur que nous allons aborder.

Le verrouillage

Nous avons déjà vu ce qu'il en était de la théorie du verrouillage, au moins superficiellement car le verrouillage est une technique particulièrement piégeuse, tel que je vous l'ai déjà montré dans quelques exemples. Rappelons-nous tout d'abord que les verrous optimistes ne sont pas stricto sensu des verrous. D'autant plus qu'ADO dans ces curseurs avec rafraîchissement va se faire un malin plaisir de vous les faire sauter.

Les limitations du verrouillage optimiste

Croyez le bien, avec ce type de verrouillage c'est bien souvent le développeur qui est optimiste. Déjà il ne faut jamais utiliser de tel verrou sur un curseur dynamique ou à jeu de clé car alors il est impossible de savoir de quelle version de l'enregistrement on dispose et donc si le verrou va fonctionner ou non. De plus comme nous l'avons vu, le moteur de curseur ou une commande mal écrite peut contourner sans problème ce type de verrouillage sur certains SGBD. Néanmoins le verrouillage optimiste sollicite la connexion à un moment donné, il faut bien que les données soient envoyées au serveur, mais en tout cas, il sollicite beaucoup moins de ressources de la part du serveur SGBD qu'un verrouillage pessimiste et peut être utilisé en mode déconnecté. Il convient donc d'être rigoureux sur le paramétrage du recordset (propriété dynamique 'Update Crtiteria') et sur l'appel des synchronisations.

Les inconvénients du verrouillage pessimiste

Le verrou pessimiste est un 'vrai' verrou. Cela veut dire que l'enregistrement est 'physiquement' verrouillé et qu'il n'est plus accessible aux autres process tant que le verrou existe. Ces verrous sont très efficaces, parfois même trop. Il y a toutefois une chose à ne pas perdre de vue. De tel verrou sont des verrous en écriture, ce qui veut dire que l'enregistrement reste accessible en lecture. Il y a donc possibilité de démarrer une action sur un ensemble d'enregistrement dont certains sont verrouillés sans que l'erreur se produise immédiatement.

Le véritable problème des verrous pessimistes dans ADO vient de l'utilisation de la concurrence optimiste dans votre code. En effet, en mode client, il n'y aura pas d'interrogation du fournisseur pour vérifier si l'enregistrement est effectivement verrouillé tant qu'il n'y a pas vidage du cache des modifications vers la source. Autrement dit, c'est seulement lors d'une opération Update ou UpdateBatch que vous risquez de buter sur le verrou.

Enfin, on n'est jamais à l'abri de réaliser l'étreinte mortelle (DeadLock) qui peut planter définitivement le SGBD. Celle-ci se réalise lorsque deux process tente de poser des verrous sur les mêmes enregistrements mais dans un ordre différent.

N.B : Certaines versions d'Access utilisent le verrouillage par page. C'est la cerise sur le gâteau. Cela revient à dire que lors du verrouillage d'un enregistrement, d'autres peuvent l'être aussi sans aucune raison.

Transactions et exclusivité

Il s'agit là des vrais techniques de gestion sécurisée de la concurrence. Je ne vais pas redire ici ce que j'ai déjà dit plus haut sur les transactions et sur l'isolation de celles-ci. Sachez toutefois que vous **devez** travailler avec des transactions lorsqu'il y a risque de concurrence. Cette règle ne doit pas être contournée.

L'exclusivité consiste à restreindre les droits des nouveaux entrants, au niveau de leurs connexions ou à verrouiller une ou plusieurs tables le temps de faire ses modifications. L'une comme l'autre de ces techniques est à déconseiller car au lieu de régler le problème de la concurrence elles créent implicitement des accès privilégiés, ce qui ne doit être fait qu'en toute connaissance de cause

Conclusion sur ADO

Vous voilà maintenant au fait de la programmation ADO avec Delphi. Pour être efficace, il faut savoir dans le même temps utiliser les composants que Delphi met à votre disposition, tout en sachant programmer les objets sous-jacents pour quelques astuces indispensables.

Vous trouverez dans ADO un compagnon puissant pour l'accès aux bases de données dès lors que vous aurez correctement assimilé le principe des curseurs. Nous allons maintenant nous intéresser à l'aspect structure de la source de donnée, grâce à ADOX.

ADOX: Microsoft ActiveX Data Objects Extensions

Ce chapitre va aborder la programmation du modèle objet ADOX (Microsoft® ActiveX® Data Objects Extensions) pour le Langage de Définition des Données *(DDL = Data Definition Language and Security).*

Dès à présent, il faut bien comprendre que la bibliothèque ADOX sert pour la création ou la modification d'une base de données. Pour une simple consultation, il est beaucoup plus efficace d'utiliser la méthode OpenSchema d'ADO.

Avant de pouvoir l'utiliser, il faut importer la bibliothèque "Microsoft ADO Ext. 2.x for DDL and Security" selon une procédure que je vais vous donner ci-après.

Il n'est pas toujours évident de voir la limite entre les modèles ADO et ADOX. Pour simplifier, on utilise ADOX pour créer ou modifier un élément de la base de données, et ADO pour manipuler les données. De plus ADOX est le seul moyen de gérer les utilisateurs et les groupes.

Importer la bibliothèque ADOX dans Delphi.

Pour pouvoir utiliser le modèle objet ADOX il vous faut importes la bibliothèque de type dans Delphi.

Pour cela suivez la procédure suivante :

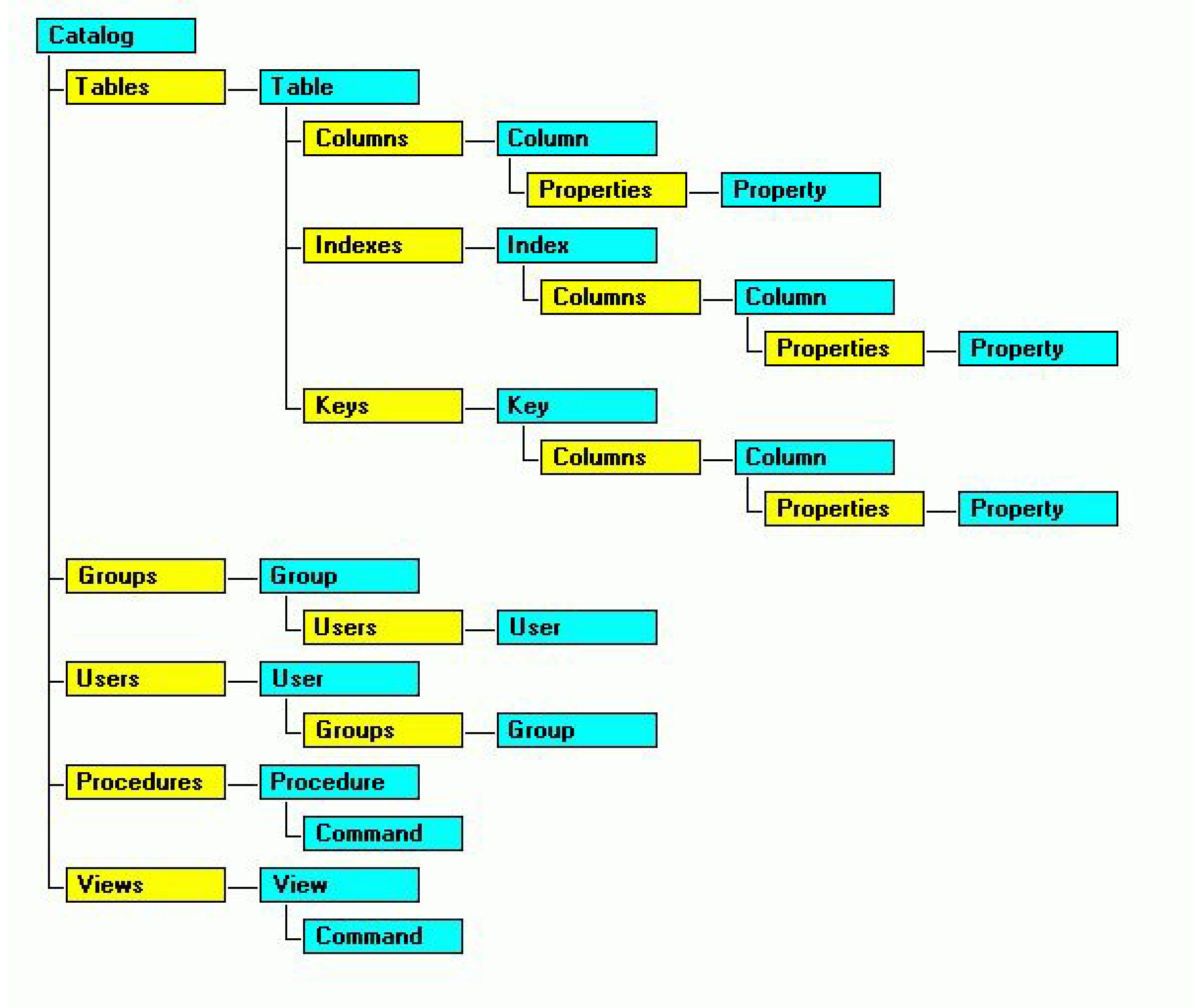
- Dans l'IDE faites : "Projet" – "importer une bibliothèque de type".
- Sélectionnez "Microsoft Ext. 2.5 for DDL and Security.
- Dans la fenêtre des noms de classes modifier :
- TTable -> TADOXTable
- Tcolumn -> TADOXColumn
- TIndex-> TADOXIndex
- TKey -> TADOXKey
- TGroup -> TADOXGroup
- TUser -> TADOXUser
- TCatalog -> TADOXCatalog
- cliquez sur le bouton installer, puis sur OK, puis sur Oui deux fois.

Dans votre projet il y a ajout d'une unité ADOX_TLB.pas, et les contrôles ont été ajoutés à l'onglet ActiveX. Pour des raisons de commodités, je vous conseille de déplacer ces contrôles vers l'onglet ADO (propriétés de la palette).

Vous voilà fin prêt pour affronter ADOX avec des composants graphiques. Comme dans le cas d'ADO, je vais passer pour ma part par du code intrinsèque, mais fondamentalement le principe est le même.

Modèle objet

Le modèle objet que nous allons regarder est le suivant :



Comme vous le voyez, il n'y à pas de similitude entre les deux modèles objet, si ce n'est de nouveau la présence des collections "Properties" de propriétés dynamiques. Cependant, si vous avez bien assimilez ce que nous avons vu jusque là, vous n'allez rencontrer aucune difficultés pour utiliser le Modèle ADOX.

Rappels Access

Sécurité

Il y a deux types de sécurité Access : la sécurité au niveau partage, c'est à dire l'utilisation d'un mot de passe pour ouvrir la base de données et la sécurité au niveau utilisateur semblable aux autres SGBD. Celle-ci repose sur l'identification de l'utilisateur, celui-ci ayant un certain nombre de droits définis. La différence d'Access vient de l'utilisation d'un fichier externe pour gérer cette sécurité. Les deux niveaux de sécurité sont cumulables.

Paramétrage JET

Par défaut, le paramétrage du moteur Jet se trouve dans la base de registre à la clé :
HKLM\Software\Microsoft\Jet\4.0\Engines\Jet 4.0.

Ce paramétrage pouvant être différent d'un poste à l'autre, il convient d'utiliser le code pour toujours mettre ce paramétrage aux valeurs désirées. Pour cela deux approches sont possibles : la

modification des valeurs du registre ou le paramétrage de la connexion (méthode que nous verrons plus loin).

Notions Fondamentales

ADOX & Access

Dans ce modèle, il y a une certaine redondance. Dans le cas d'Access, une requête non paramétrée renvoyant des enregistrements est considérée comme une table de type "VIEW" (vues). Une telle requête apparaîtra donc comme membre de la collection "Table" (un peu différente des autres tables) et comme membres de la collection "Views" mais pas comme membre de la collection procédure.

Attention, dans Access il faut considérer une procédure comme une requête paramétrée ou une requête ne renvoyant pas d'enregistrement (Requête action par exemple) ; n'attendez pas de récupérer des procédures VBA dans cette collection.

Propriétaire

Cette notion de propriété est très importante. En effet seul le propriétaire d'un objet peut faire certaines modifications sur celui-ci et attribuer les droits sur ces objets. Par défaut, le propriétaire d'un objet est son créateur. Cela implique deux choses :

- Ne pas laisser aux utilisateurs le droit de créer des objets dans vos bases sans un contrôle car vous n'auriez pas de droit de modifications sur ces objets sans en reprendre la propriété.
- Gérer strictement les droits sur la base.

Nous y reviendrons plus en détail lorsque nous aborderons la sécurité.

ParentCatalog

Lorsque l'on crée des objets, ils ne sont reliés à aucun catalogue. Dès lors, comme ils ne connaissent pas leurs fournisseurs, il n'est pas possible de valoriser leurs propriétés. On définit donc la propriété ParentCatalog d'un objet dès que l'on souhaite valoriser ses propriétés. Ceci pourtant dissimule un problème d'un fort beau gabarit ma foi. Outre le fournisseur, le catalogue transmet aussi ses droits de propriétaire. Si on n'y prend pas garde, il se peut que l'objet se retrouve ayant des droits non souhaités.

L'objet Catalog

C'est l'objet racine du modèle. Attention, nous n'allons utiliser que de la programmation intrinsèque.

ActiveConnection

C'est la seule propriété de l'objet Catalog, qui lui permet de définir le fournisseur de données, la base de données et éventuellement les paramètres de sécurité. Cette connexion est stricto sensu une connexion ADO, on peut donc parfaitement utiliser une connexion existante. Attention toutefois, ce sont les paramètres de la connexion qui définissent en partie les possibilités du fournisseur, il est donc vivement recommandé de créer sa connexion pour la gestion de la sécurité.

Le code suivant permet de créer un catalogue sur une base existante :

```
procedure TForm1.Button1Click(Sender: TObject);
var MonCat: _catalog;
begin
     MonCat:=coCatalog.Create;
     moncat.Set_ActiveConnection('Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\tutoriel\biblio.mdb;');
end;
```

L'exemple d'après crée une base de donnée Access

```
procedure TForm1.Button1Click(Sender: TObject);
var MonCat: _catalog;
begin
     if FileExists('C:\tutoriel\nouv.mdb') then
        DeleteFile('C:\tutoriel\nouv.mdb');
     MonCat:=coCatalog.Create;
     moncat.create('Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\tutoriel\nouv.mdb;');
end;
```

Create (Mth.)

Permet la création d'une nouvelle base de données selon les paramètres de la chaîne de connexion passée en paramètre. Attention, le nom de la base de données ne doit pas être une base existante. Le paramètre à passer à la méthode doit être une chaîne de connexion valide.

GetObjectOwner & SetObjectOwner (Mth.)

De la forme

Catalog.SetObjectOwner ObjectName, ObjectType , OwnerName [, ObjectTypeId]

Owner = Catalog.GetObjectOwner(ObjectName, ObjectType [, ObjectTypeId])

Permet de renvoyer ou de définir le propriétaire d'un objet de la base, celui-ci pouvant être la base de données, une table, un champ…..

Ne fonctionne qu'avec un fichier de sécurité défini.

Les collections

L'objet Catalog renvoie aussi cinq collections qui représentent la structure de la base. Nous allons maintenant étudier ces objets

Collections de l'objet Catalog

Dans ADOX, les collections ont normalement une propriété Count, et quatre méthodes. Attention le premier index d'une collection est zéro.

Append

Ajoute un objet à la collection. La méthode Append est propre à chaque collection, aussi allons nous la voir en détail plus loin. Notons toutefois qu'un objet créé est librement manipulable, tant qu'il n'a pas été ajouté à la collection correspondante. Après l'ajout, un certain nombre de propriétés passe en lecture seule.

Item

Renvoie un élément d'une collection par son nom ou son index.

Delete

Retire un élément d'une collection. Il faut faire très attention à la suppression de certains éléments qui peut mettre en danger l'intégrité des données voire endommager la base

Refresh

Met à jour la collection.

Collection tables

Représente l'ensemble des tables, au sens large du terme, qui compose la base de données. Ces tables peuvent être rangées dans les familles suivantes :

- ✓ Table système
- ✓ Table Temporaire
- ✓ Table de données
- ✓ Vues

En générale on ne travaille que sur les tables de données à l'aide de l'objet Table.

Append

De la forme Cat.Tables.Append *NomTable*. Lors de la création tous les objets appartenant à la table doivent être créés avant l'ajout de celle-ci à la collection

Collection Procedures

Représente l'ensemble des requêtes définies dans la base de données à l'exclusion des requêtes non paramétrées renvoyant des enregistrements.

Append

De la forme Cat.Procedures.Append *NomRequete*, *Command*. Command représente un objet command qui représente la requête.

Collection Views

Représente l'ensemble des requêtes non paramétrées renvoyant des enregistrements. On les retrouve d'une certaine façon dans la collection Tables avec Access.

Append

De la forme Cat.Views.Append *NomRequete*, *Command*. Command représente un objet command qui représente la requête.

Collection Groups

Représente l'ensemble des groupes définis dans la base. A ne pas confondre avec la collection Groups de l'objet User qui est le groupe auquel appartient un utilisateur.

Append

De la forme Cat.Groups.Append *NomGroupe*. Ceci représente l'ajout d'un groupe à la base de données. Les permissions doivent être définies lors de l'ajout du groupe à la collection. Le groupe doit être ajouté à la collection avant la création d'un utilisateur du groupe.

Collection Users

Représente l'ensemble des utilisateurs définis dans la base. A ne pas confondre avec la collection Users de l'objet Group qui est l'ensemble des utilisateurs définis dans un groupe.

Append

De la forme Cat.Users.Append *NomUtilisateur* [**,** *MotDePasse*]. Ceci représente l'ajout d'un utilisateur à la base de données.

L'objet Table

Cet objet représente une table de la base de données. Les tables de type "VIEW" seront vues dans le chapitre du même nom. Nous allons donc regarder les tables Standard.

Une table contient des champs (colonnes), des index, des clés et des propriétés propres au fournisseur.

Seules deux propriétés intéressantes sont statiques : « Name » qui représente le nom de la table et est obligatoire, et « type » qui porte bien son nom mais qui est en lecture seule. Ceci fait qu'il n'y a pas de questions existentielles à se poser, lorsqu'on crée une table avec ADOX, elle est forcément standard.

La table n'existe réellement dans la base de données que lorsque

- Son catalogue est défini
- Elle est ajoutée à la collection Tables.
  Il convient donc de créer entièrement la table avant de l'ajouter à la collection.

Collection Properties

Cette collection n'est utilisée que dans le cas des tables liées. N'essayez pas de créer des tables liées en partant du catalogue de votre base, bien que techniquement possible cela n'est pas conseillé. Le code suivant crée une table liée

```
procedure TForm1.Button1Click(Sender: TObject);
var MonCat: _catalog;
MaTable: _table;
begin
     MonCat:=coCatalog.Create;

moncat.Set_ActiveConnection('Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\tutoriel\tuto.mdb;Jet OLEDB:System
database=c:\tutoriel\system.mdw;User Id=Admin; Password=');
     MaTable:=coTable.Create;
     With MaTable do begin
        Name:='Auteurs';
        ParentCatalog:=MonCat;
        Properties['Jet OLEDB:Create Link'].Value:=True;
        Properties['Jet OLEDB:Link Datasource'].Value:=
'C:\tutoriel\biblio.mdb';
        Properties['Jet OLEDB:Link Provider
String'].Value:=';Pwd=password';
        Properties['Jet OLEDB:Remote Table Name'].Value:='Authors';
     end;
     moncat.Tables.Append(matable);
end;
```

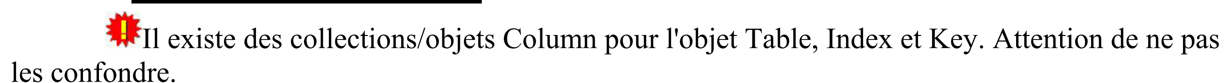
Lorsque j'écris que c'est déconseillé, ce n'est pas dû à la complexité du code mais pour les deux raisons suivantes :

- Il faut s'assurer que le fournisseur utilisé pour faire la liaison sera présent sur le poste qui exécute.
- Il faut bien savoir si les paramètres de sécurité de liaison seront ou ne seront pas stockés de façon permanente dans la base.

Regardons ce code. Après avoir défini un catalogue nous créons un objet Table. Dans cet exemple, le nom est différent du vrai nom de la table dans l'autre base, mais cela n'a aucune importance. Ensuite je lui attribue un catalogue afin de pouvoir valoriser ses propriétés dynamiques. Notez bien que j'utilise l'objet Catalog de ma base car je crée une table liée dans ma base.

Je valorise ensuite mes propriétés dynamiques, notons que je ne mets pas de fournisseur dans `"Jet OLEDB:Link Provider String"` car je continue d'utiliser Jet mais je pourrais tout à fait changer le fournisseur. Enfin j'ajoute ma table à la collection. A partir de cet ajout, la table liée est créée mais un certain nombre de ses propriétés sont passées en lecture seule.

Collection Columns



Il existe des collections/objets Column pour l'objet Table, Index et Key. Attention de ne pas les confondre.

La collection Columns regroupe les champs de la table. Elle possède les propriétés/méthodes d'une collection normale.

Append (méth.) : De la forme

Columns.**Append** *Column* [, *Type*] [, *DefinedSize*]

Où Column est l'objet colonne à ajouter.

Objet Column

Représente dans ce cas un champ de la table. A ne pas confondre avec l'objet Column de l'index. Cet objet n'a pas de méthode.

Propriétés

Attributes

Défini si la colonne est de longueur fixe et si elle accepte les valeurs NULL. En général avec Delphi il est plutôt conseillé d'utiliser les propriétés dynamiques Nullable et Jet OLEDB:Allow Zero Length.

DefinedSize

Défini la taille maximum du champ pour une chaîne de caractères.

Name

Nom de la table. Obligatoire. L'unicité d'un nom dans une collection n'est pas obligatoire, mais dans une même base, il peut pas y avoir deux tables portant le même nom.

NumericScale

A ne pas confondre avec la propriété Précision. Donne l'échelle (nombre de chiffre après la virgule)d'un champ dont le type est adNumeric ou adDecimal.

ParentCatalog

Qu'on ne présente plus!

Précision

Défini la précision (nombre maximal de chiffre pour représenter la valeur)d'un champ numérique.

Type

Défini le type du champ. Il existe beaucoup de types définis mais le tableau suivant ne vous donnera que les principaux types.

Type	Valeur	Commentaire
adSmallInt	2	Type Entier court
adInteger	3	Type Entier long (Access)
adSingle	4	Valeur décimale à simple précision
adDouble	5	Valeur décimale à double précision
adCurrency	6	Type monétaire. Utilise normalement 4 chiffres après la virgule
adDate	7	Stocké comme un double. La partie entière étant le nombre de jours depuis le 30/12/1899
adBSTR	8	Pointeur de BSTR. Pour plus de précisions sur ce type, consulter la Référence du Programmeur OLE DB.
adError	10	Code d'erreur 32-bit (DBTYPE_ERROR).
adBoolean	11	Type **Variant Boolean**. 0 est faux et ~0 est vrai.
adDecimal	14	Type variant décimal. Pour plus de précisions sur ce type, consulter la Référence du Programmeur OLE DB (DBTYPE_DECIMAL).
adTinyInt	16	Valeur numérique exacte, de précision 3 et d'échelle 0 (DBTYPE_I1).
adUnsignedTinyInt	17	Version non signée de **adTinyInt** (DBTYPE_UI1).
adUnsignedSmallInt	18	Version non signée de **adSmallInt** (DBTYPE_UI2).
adUnsignedInt	19	Version non signée de **adInteger** (DBTYPE_UI4).
adBigInt	20	Valeur numérique exacte, de précision 19 et d'échelle 0 (DBTYPE_I8).
adUnsignedBigInt	21	Version non signée de **adBigInt** (DBTYPE_UI8).
adFileTime	64	Valeur 64-bit value représentant le nombres d'intervalle de 100-nanosecond depuis le 1 Janvier 1601 (DBTYPE_FILETIME).
adGUID	72	Identificateur Globalement Unique (GUID = Globally Unique Identifier) (DBTYPE_GUID).
adBinary	128	Données binaires de longueur fixe (DBTYPE_BYTES).
adChar	129	Chaîne de caractères de longueur fixe (DBTYPE_STR).
adWChar	130	Chaîne de caractères unicode de longueur fixe (DBTYPE_WSTR).
adNumeric	131	Type numérique. Pour plus de précisions sur ce type, consulter la Référence du Programmeur OLE DB (DBTYPE_NUMERIC).
adUserDefined	132	Type de donnée de longueur variable défini par l'utilisateur (DBTYPE_UDT).
adDBDate	133	Structure de date pour base de données (yyyymmdd) (DBTYPE_DBDATE).
adDBTime	134	Structure de temps pour base de données (hhmmss) (DBTYPE_DBTIME).
adDBTimeStamp	135	Structure de marquage de temps pour base de données (yyyymmddhhmmss plus une fraction) (DBTYPE_DBTIMESTAMP).
adChapter	136	Indique une valeur 4-bit de chapitre qui identifie une ligne d'un jeu d'enregistrement enfant (DBTYPE_HCHAPTER).
adVarWChar	202	Chaîne de caractères (type Text Access)
adLongVarWChar	203	Champs mémo (Access) et lien hypertexte
adLongVarBinary	205	Type OLE Object Access

Collection Properties

Les propriétés dynamiques de l'objet Column intéressantes sont :
Autoincrement : Permet de créer les colonnes à numérotation automatique.
Default : Définit la valeur par défaut d'un champ.
Description : Donne accès à la description de la colonne, si le SGBD les gèrent.
Nullable : Définit si la colonne accepte des valeurs NULL.
Fixed Length : Définit si la colonne est de longueur fixe.
Seed : Détermine la prochaine valeur qui sera fourni par un champ de type numéro Automatique.
Increment : Définit le pas d'incrément d'un champ NuméroAuto.
Jet OLEDB:Column Validation Text : Définit le message d'alerte si une règle de validation n'est pas respectée.
Jet OLEDB:Column Validation Rule : Définit une règle de validation pour un champ.
Jet OLEDB:Allow Zero Length : Définit si les chaînes vides sont autorisées.

Exemple

Dans l'exemple suivant, nous allons créer une table contenant trois champs, 'Id' de type NuméroAuto, 'Nom' de type Texte et obligatoire, 'Num Tel' de type texte et facultatif.

```
var MonCat: _catalog;
MaTable: _table;
ColId, ColNom, ColTel: _column;
begin
     MonCat:=coCatalog.Create;

moncat.Set_ActiveConnection('Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\tutoriel\tuto.mdb;Jet OLEDB:System
database=c:\tutoriel\system.mdw;User Id=Admin; Password=');
     MaTable:=coTable.Create;
     MaTable.Name:='TestColonne';
     ColId:=coColumn.Create;
     ColId.ParentCatalog:=MonCat;
     With ColId do begin
         Name:='Id';
         Type_:=adInteger;
         Properties['Nullable'].Value:=False;
         Properties['Autoincrement'].Value:=True;
         Properties['Seed'].Value:=1;
         Properties['Increment'].Value:=1;
     end;
     ColNom:=coColumn.Create;
     ColNom.ParentCatalog:=MonCat;
     With ColNom do begin
         Name:='Nom';
         Type_:=adVarWChar;
         DefinedSize:=50;
         Properties['Nullable'].Value:=False;
         Properties['Jet OLEDB:Allow Zero Length'].Value:=False;
     end;
     ColTel:=coColumn.Create;
     ColTel.ParentCatalog:=MonCat;
     With ColTel do begin
         Name:='Num Tel';
         Type_:=adVarWChar;
         DefinedSize:=14;
         Properties['Nullable'].Value:=True;
         Properties['Jet OLEDB:Allow Zero Length'].Value:=True;
     end;
     MaTable.Columns.Append(ColId,Unassigned,Unassigned);
```

```
    MaTable.Columns.Append(ColNom,Unassigned,Unassigned);
    MaTable.Columns.Append(ColTel,Unassigned,Unassigned);
    moncat.Tables.Append(matable);
end;
```

Je commence donc par créer le catalogue de la base de données. Pour cela j'utilise une chaîne de connexion qui est obligatoire pour pouvoir utiliser la méthode Create de catalogue. Ensuite je crée mon objet Table. Comme je n'utilise pas sa collection Properties, je ne valorise pas sa propriété ParentCatalog.

Pour chaque colonne je définis une taille, un nom, un type et éventuellement ses attributs. Notons que comme la colonne n'est pas encore ajoutée à la collection Columns, elle n'existe pas encore. Dès lors, l'ordre de valorisation des propriétés n'a aucune importance. Dans le cas de la colonne "Id" je vais créer un champ NuméroAuto. Pour cela, je dois valoriser des propriétés dynamiques, donc je dois définir la propriété ParentCatalog. Enfin j'ajoute la colonne à la collection Columns puis la table à la collection Tables.

Notez aussi que tous ce code revient au même, à l'exception de la création de la base que la commande DDL pour peu que le DDL normalisé soit supporté par le SGBD.

```
"CREATE TABLE PremTable(Id COUNTER, Nom TEXT 50 NOT NULL, Tel TEXT 14)"
```

Collection Indexes

Représente les index de la table. Faites bien attention à ne pas confondre index et clé. Un index sert à augmenter l'efficacité du SGBDR lors des recherches, de la mise en relation etc….

Normalement devrait toujours être indexés

- Les colonnes composantes d'une clé
- Les colonnes ayant des contraintes de validité ou d'unicité
- Les colonnes étant la cible de nombreuses recherches.

Les propriétés/méthodes de cette collection sont celles des collections standard si ce n'est :

Append

De la forme : *Indexes*.**Append** *Index* [**,** *Columns*]

Où Columns renvoie le ou les noms des colonnes devant être indexées. En général, on définit l'objet colonnes avant son ajout à la collection

Objet Index

Cet objet n'a pas de méthodes.

Attention, on ne peut utiliser un objet Column avec cet objet que si la colonne appartient déjà à la collection Columns de l'objet Table.

Clustered

Indique si l'index est regroupé. Un index regroupé signifie qu'il existe un ordre physique pour les données. C'est en général l'index de la clé primaire.

IndexNulls

Définit le comportement de l'index vis à vis des colonnes ayant une valeur NULL.

PrimaryKey

Précise si l'index représente la clé primaire de la table.

NB : C'est là que je vous disais de faire attention. Une clé primaire (ou non d'ailleurs) est définie par la collection Keys. Néanmoins on définit en général aussi un index sur la colonne de clé primaire. En mettant à vrai cette propriété, la création de l'index implique la création de la clé. Donc si vous créez une clé primaire de façon implicite n'essayez pas de la recréer après dans la collection Keys et vice versa.

Unique

Définit la contrainte d'unicité.

Collection Properties

Comme toujours, la propriété ParentCatalog de l'index doit être valorisé avant l'appel d'un membre de cette collection. La plupart de ces propriétés étant en lecture seule ou redondante avec les propriétés statiques de l'objet, nous n'irons pas plus avant dans notre exploration.

Collection Columns et objet Column

Définit les colonnes sur lesquelles porte l'index. N'oubliez pas qu'un index porte sur un ou plusieurs champs mais qu'il appartient à l'objet Table.

L'objet Column doit impérativement exister dans la table avant de pouvoir être ajouter à la collection Columns de l'index.

Les colonnes des index utilisent une propriété supplémentaire

SortOrder

Définit l'ordre de tri d'une colonne d'index.

Exemple

Dans mon exemple précédent je vais rajouter un index sur la colonne Nom, avec un tri croissant.

```
MonIndex:=coIndex.Create;
     with MonIndex do begin
          IndexNulls:=adIndexNullsDisallow;
          Name:='ind1';
          Columns.Append(ColNom.Name, ColNom.type_,
ColNom.DefinedSize);
          Columns['Nom'].SortOrder:= adSortAscending;
     end;
     MaTable.Indexes.Append(MonIndex,EmptyParam);
```

Ce code se situe après l'ajout des colonnes à l'objet Tables mais avant l'ajout de l'objet Table. Comme précédemment, je manipule mon objet avant son ajout à la collection.

Collection Keys

Nous allons aborder maintenant un des passages un peu "sensibles" de la programmation ADOX, c'est à dire les clés. Les clés sont le fondement de l'intégrité référentielle il convient de bien connaître le sujet avant de se lancer dans leur programmation. Normalement, vous avez travaillé sur le modèle de données avant de construire votre base donc il suffira de suivre celui-ci.

Quelques notions

Les clés sont parfois appelées, improprement d'ailleurs, contraintes. Ceci vient du fait qu'en SQL, 'Primary Key' est une contrainte. Pourtant si les clés peuvent être des contraintes, toutes les contraintes ne sont pas des clés.

Clé primaire

Cette clé devrait être composée d'une seule colonne mais ce n'est pas une obligation. Elle sert à identifier chaque enregistrement de manière unique. La clé primaire n'accepte pas les valeurs NULL. Beaucoup de gens utilisent un champ à numérotation automatique pour définir la clé primaire, cela n'est en rien obligatoire. Une table n'a qu'une clé primaire.

Clé étrangère

Champ de même type/sous-type de données que la clé primaire, qu'elle référence dans la table enfant. Une table peut posséder plusieurs clés étrangères.

Intégrité référentielle

Mécanisme de vérification qui s'assure que, pour chaque valeur de clé étrangère, il y a une valeur de clé primaire lui correspondant. Ce mécanisme se déclenche lors de l'ajout d'une nouvelle valeur de clé étrangère, lors de la suppression d'une clé primaire ou lors de la modification de l'une quelconque des deux clés.

Opération en cascade

Mode opératoire dépendant de l'intégrité référentielle. Celui-ci détermine éventuellement le comportement du SGBDR en cas de suppression/modification d'un enregistrement parent.

Méthode Append

De la forme *Keys*.**Append** *Key* [**,** *KeyType*] [**,** *Column*] [**,** *RelatedTable*]
Nous examinerons plus tard le détail des paramètres (dans les propriétés de l'objet Key).

Objet Key

Représente une clé primaire, étrangère ou unique.

DeleteRule

Définit les règles à appliquer lors de la suppression d'une valeur de la clé primaire. On distingue 4 cas :

AdRINone
Aucune action (équivalent du mode de gestion SQL ON DELETE NO ACTION). Cette valeur interdit toute suppression d'un enregistrement tant qu'il existe un enregistrement dépendant dans la base.

AdRICascade
Modifications en cascade (SQL ON DELETE CASCADE). Attention à l'utilisation de cette valeur. Elle supprime tous les enregistrements liés lors de la suppression de l'enregistrement père. Ceci peut être très coûteux pour le SGBD en termes de performances et doit être limité au sein d'une transaction

AdRISetNull & adRISetDefault
Attribue la valeur nulle ou la valeur par défaut à la clé étrangère. Ceci se justifie rarement sauf pour gérer des traitements par lots à posteriori.

Name

Tel que son nom l'indique

RelatedTable

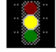
Si la clé est une clé étrangère, définit le nom de la table contenant la clé primaire correspondante.

Type

Détermine s'il s'agit d'une clé primaire, étrangère ou unique. Une clé unique est une clé sans doublons.

UpdateRule

Equivalent pour les modifications à DeleteRule pour les mises à jour.

Attention de nombreux SGBDR n'acceptent pas toutes les règles de modification/suppression. Je vous rappelle aussi que ces règles s'appliquent sur la clé étrangère ; les définir pour la clé primaire ne sert à rien.

Collection Columns et objet Column

Définit la ou les colonne(s) appartenant à la clé. Celles-ci doivent déjà appartenir à la collection Columns de l'objet Table. L'objet Column de l'objet Key utilise la propriété **RelatedColumn**. Celle-ci définit la colonne en relation dans la table définie par RelatedTable.

Exemples

Nous allons voir maintenant un exemple de création de clé toujours dans ma table.

```
MaCle:=coKey.Create;
     with MaCle do begin
          Name:='Prim Cle';
          type_:=adKeyPrimary;
          RelatedTable:='';
          Columns.Append(ColId.Name,ColId.type_,ColId.DefinedSize);
          Columns['Id'].RelatedColumn:='';
     end;
     MaTable.Keys.Append(MaCle,adKeyPrimary,EmptyParam,'','');
```

Je crée ici ma clé primaire de façon explicite mais c'est souvent plus rapide de le faire en même temps que l'on crée l'index.

Amusons nous maintenant à créer une deuxième table afin de la mettre en relation. Je vais donc créer une deuxième table 'TestRelat' dans ma base.

Tout d'abord dans notre première table pour ajouter un index unique sur la colonne 'Id'. Ensuite nous allons créer une deuxième fonction de la forme.

```
var MaTable: _table;
ColCle, ColLocal, ColResp: _column;
MonIndP, MonIndF: _index;
MaCle: _key;
begin
     MaTable:=coTable.Create;
     MaTable.Name:='TestRelation';
     MaTable.ParentCatalog:=MonCat;
     ColCle:=coColumn.Create;
     ColCle.ParentCatalog:=MonCat;
     With ColCle do begin
         Name:='NumCle';
         Type_:=adInteger;
         Properties['Nullable'].Value:=False;
         Properties['Autoincrement'].Value:=True;
         Properties['Seed'].Value:=1;
         Properties['Increment'].Value:=1;
     end;
     ColLocal:=coColumn.Create;
     ColLocal.ParentCatalog:=MonCat;
     With ColLocal do begin
         Name:='Local';
         Type_:=adVarWChar;
         DefinedSize:=50;
         Properties['Nullable'].Value:=False;
         Properties['Jet OLEDB:Allow Zero Length'].Value:=False;
     end;
     ColResp:=coColumn.Create;
     ColResp.ParentCatalog:=MonCat;
```

```
      With ColResp do begin
          Name:='Num_Resp';
          Type_:=adInteger;
          Properties['Nullable'].Value:=False;
      end;
      MaTable.Columns.Append(ColCle,Unassigned,Unassigned);
      MaTable.Columns.Append(ColLocal,Unassigned,Unassigned);
      MaTable.Columns.Append(ColResp,Unassigned,Unassigned);
      MonIndP:=coIndex.Create;
      with MonIndP do begin
           IndexNulls:=adIndexNullsDisallow;
           Name:='indP';

Columns.Append(ColCle.Name,ColCle.type_,ColCle.DefinedSize);
           PrimaryKey:=True;
           Unique:=True;
      end;
      MonIndF:=coIndex.Create;
      with MonIndF do begin
           IndexNulls:=adIndexNullsDisallow;
           Name:='indF';

Columns.Append(ColResp.Name,ColResp.type_,ColResp.DefinedSize);
      end;
      MaTable.Indexes.Append(MonIndP,EmptyParam);
      MaTable.Indexes.Append(MonIndF,EmptyParam);
      MaCle:=coKey.Create;
      with MaCle do begin
           Name:='Etr Cle';
           type_:=adKeyForeign;
           DeleteRule:= adRICascade;
           UpdateRule:= adRICascade;
           RelatedTable:='TestColonne';
           Columns.Append(ColResp.Name, ColResp.type_,
ColResp.DefinedSize);
           Columns['Num_Resp'].RelatedColumn:='Id';
      end;

MaTable.Keys.Append(MaCle,Unassigned,EmptyParam,Unassigned,Unassigne
d);
      moncat.Tables.Append(matable);
end;
```

Donc je crée une table de trois colonnes, la troisième (num_resp) contenant la clé étrangère. Le seul passage nouveau étant la création de la clé étrangère.

Conclusion sur les tables

Comme nous l'avons vu au cours de ce chapitre, le schéma de construction est assez simple. Voici les quelques règles à se rappeler :

- On crée les tables les unes après les autres en commençant toujours par les tables parents (celles qui n'ont pas de clés étrangères)
- On ajoute un objet à sa collection lorsqu'on l'a entièrement défini.
- On crée les champs de la table avant de définir les index et les clés.
- On indexe toujours les champs intervenants dans les jointures et/ou cible de recherche.
- La définition des clés lors de la création de la base augmente la sécurité des données.

L'objet Procedure

Cet objet peut représenter plusieurs choses selon les cas.

- Une requête Action
- Une requête paramétrée

➢ Une procédure stockée (n'existe pas dans Access)

En soi cet objet est très simple à utiliser puisqu'il ne s'agit que d'un objet Command ayant un nom dans la collection Procedures. Comme nous avons vu plus haut l'objet Command suffisamment en détail, nous allons nous pencher dans ce chapitre sur l'ajout ou la modification de l'objet Procedure.

Création d'un objet procédure

Elle se fait en générale en utilisant la méthode Append de la collection Procedures de la forme

Catalog.Procedures.**Append** *Name*, *objCommand*

Où *objCommand* est un objet Command **ADO**. Tout se fait donc dans la définition de l'objet Command. Attention de ne pas passer l'objet TADOCommand, mais bien l'objet Command qu'il contient.

Dans le cas d'une requête action, la création est facile puisque le texte de la commande est le code SQL. Mais dès lors qu'il y a utilisation de paramètres il faut faire attention.

Pas d'objet Parameter

Nous avons vu que pour l'appel des procédures stockées ou des requêtes paramétrées, nous utilisions les objets Parameters de la commande. Lors de la création cela n'est pas possible. La propriété CommandText de l'objet Command Doit contenir l'ensemble de la requête / procédure telle qu'elle est effectivement écrite dans le SGBD.

A part cela, la création d'un objet Procedure ne pose aucun problème.

Exemple

Dans la suite de mon exemple je peux ajouter

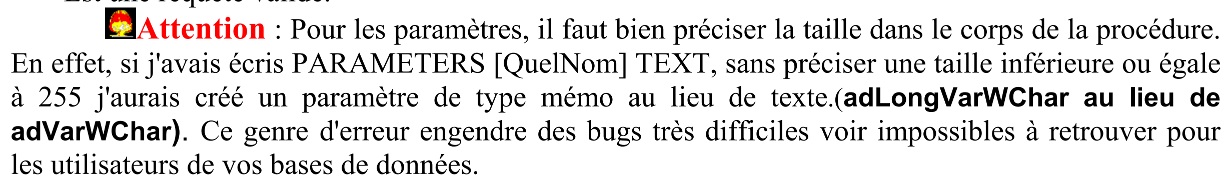
```
    var MaCommand:TADOCommand;
begin
   MaCommand:=TADOCommand.Create(nil);
   MaCommand.CommandText:= 'PARAMETERS [QuelNom] TEXT(50);SELECT *
FROM TestColonne WHERE Nom = [QuelNom]';
   MonCat.Procedures.Append('Cherche par nom',
MaCommand.CommandObject);
end;
```

Ce code ajoute une requête paramétrée nommée 'Cherche par nom' attendant le paramètre "QuelNom" de type texte.

Il est tout à fait possible de créer une requête action paramétrée. Par exemple

```
PARAMETERS DateCib DateTime, BancCib Text(255);DELETE * FROM
tblAnomalie WHERE NumBanc=[BancCib] AND DatDimanche>=[DateCib];
```

Est une requête valide.



Attention : Pour les paramètres, il faut bien préciser la taille dans le corps de la procédure. En effet, si j'avais écris PARAMETERS [QuelNom] TEXT, sans préciser une taille inférieure ou égale à 255 j'aurais créé un paramètre de type mémo au lieu de texte.(**adLongVarWChar au lieu de adVarWChar)**. Ce genre d'erreur engendre des bugs très difficiles voir impossibles à retrouver pour les utilisateurs de vos bases de données.

Modification d'un objet Procedure

La modification d'un objet Procedure par le code est assez ardue du fait qu'elle ne peut être faite directement. Il y a nécessité de passer par des interfaces Automation pour pouvoir récupérer un objet Command ADO manipulable. La technique est la suivante :

```
var MaCommand:_command;
RecIDisp:IDispatch;
begin
    RecIDisp:=MonCat.Procedures.Item['Cherche par nom'].Get_Command;
    MaCommand:=RecIDisp as _command;
    macommand.CommandText:='PARAMETERS [QuelLocal] TEXT(255);SELECT
* FROM TestRelation WHERE Local = [QuelLocal]';
```

```
    Moncat.Procedures.Item['Cherche par
nom'].Set_Command(MaCommand);
end;
```

« Mais alors, pourquoi ne pas écrire ? »

```
MonCat.Procedures.Item['Cherche par nom'].Command.CommandText=
"PARAMETERS [QuelLocal] TEXT(255);SELECT * FROM DeuxTable WHERE
Local = [QuelLocal]"
```

La raison est la suivante :

Intrinsèquement, un objet Command est volatile, c'est à dire qu'il n'est pas lié à la base de données (à contrario des objets DAO QueryDefs). Si nous utilisions le code ci-dessus, la procédure se comporterait comme attendu au cours de la même session de programme mais les modifications ne seraient pas enregistrées dans la base de données et ces modifications seraient perdues. C'est pourquoi il convient de réaffecter explicitement un objet Command à l'objet Procedure.

N.B : Le code ci-dessus doit avoir ComObj dans ses Uses.

L'objet View

La création et la modification d'un objet View se gèrent de la même manière que pour les objets Procedure. C'est pourquoi nous ne nous rééditerons pas ces notions ici.
Il faut cependant savoir que la seule vraie différence existant entre l'Objet Procédure et l'objet View sont que ce dernier n'accepte que des requêtes SELECT sans paramètre.

Conclusion sur les objets View & Procedure

Ces deux objets ne présentent aucune difficulté majeure à programmer si ce n'est de faire attention à toujours affecter les objets Command de façon explicite.

Dans le cas des requêtes renvoyant des enregistrements (View ou Procedure) on peut directement valoriser un recordset avec.

Ne perdez pas de temps à paramétrer l'objet Command lors de la création de requête ou de procédure, seule la propriété CommandText compte.

Voilà vous savez maintenant créer par le code une base de données, nous allons attaquer une partie beaucoup plus complexe, mais néanmoins essentielle, la gestion des utilisateurs.

Gestion des utilisateurs

La gestion des utilisateurs comprend en fait de façon imbriquée deux thèmes que sont les droits d'accès et la propriété. Quasiment tous les SGBD disposent d'un système limitant l'accès au données ainsi qu'à la structure de la base. Avec ADOX on retrouve ces concepts dans les objets User et Group.

Cas particulier d'Access

Les concepts que nous allons voir plus loin fonctionnent pour de nombreux SGBD mais Access gère la sécurité utilisateur par l'utilisation d'un fichier externe. Ceci présente des avantages et des inconvénients mais je ne vais pas démarrer une polémique ici.

Si j'aborde ce point c'est pour une particularité d'Access. Pour garantir l'unicité de chaque utilisateur ou groupe, le fichier de sécurité Access utilise un identifiant Unique nommé SID qui est généré à l'aide du nom de l'utilisateur (ou du groupe) et d'une chaîne nommée PID. Un même nom avec un même PID génère toujours le même SID ce qui permet de reconstruire le fichier de sécurité si besoin est.

Si cette possibilité existe avec DAO, ADO ne permet pas de fournir un PID explicite ce qui rend le fichier impossible à reconstruire en cas de problème. Certes des sauvegardes fréquentes doivent pouvoir éviter ce problème mais c'était une perte de souplesse importante.

De même, normalement un utilisateur hérite implicitement des droits d'accès du groupe, mais cela n'est pas le cas avec ADO 2.7 et inférieur. Il faut alors coder explicitement les droits du groupe et de l'utilisateur.

Comme Access gère sa sécurité par un fichier externe, il est fortement conseillé de spécifier ce fichier dans la connexion du catalogue avec la Property `("Jet OLEDB:System database") et éventuellement la création d'un nouveau fichier avec la Property (Jet OLEDB:Create System Database)`

Si vous spécifiez un fichier de sécurité incorrecte, vous n'aurez pas d'erreur lors de l'ouverture du catalogue. L'erreur surviendra lors de l'appel d'un objet lié à la sécurité. De plus vous obtiendrez une erreur 3251 " L'opération demandée par l'application n'est pas prise en charge par le fournisseur." ce qui ne vous aidera pas beaucoup à voir l'erreur.

Propriétés et droits

Voilà deux notions importantes, corrélatives et pourtant différentes dans leur conception.

Propriétaire

Chaque objet a **toujours** un propriétaire. A l'origine, c'est le créateur de l'objet qui est le propriétaire.

Le propriétaire d'un objet garde le droit d'accès total (modification, suppression, droits) sur l'objet. Le propriétaire d'un objet peut être un utilisateur ou un groupe. La base de données est un objet un peu particulier puisque seul un utilisateur (et non un groupe) peut en être propriétaire.

Le problème de la gestion de la propriété est important à gérer si les utilisateurs ont le droit de créer des objets. Si vous souhaitez qu'ils gardent tous les droits sur les objets qu'ils ont créés, il n'y a rien à faire, mais sinon vous devez gérer le transfert de la propriété.

Faites toujours très attention à cette notion de propriétaire, elle peut permettre des accès que vous ne souhaiteriez pas si vous n'y prenez pas garde.

Administrateur

Habituellement, l'administrateur ou les membres du groupe administrateur sont les seuls à avoir tous les droits sur une base de données, y compris celui de modifier les droits d'accès des autres. De part ce fait, on essaye toujours de limiter le nombre d'administrateurs. Pour ce faire, il faut donc mettre au point une stratégie de droits pour que chaque groupe ait les droits nécessaires et suffisants à un bon fonctionnement. L'avantage dans ce cas d'une approche par programme est la possibilité d'utiliser une connexion ayant les droits administrateurs pour permettre aux utilisateurs de faire des actions (prévues) que leurs droits ne leurs permettraient pas de faire normalement.

Access

Même si cela est transparent, la sécurité d'Access est toujours activée. Il existe un compte "Administrateur" possédant tous les droits et n'ayant pas de mot de passe. Pour activer la sécurité sur Access on procède en général de la manière suivante :

- On donne un mot de passe à l'utilisateur "Administrateur".
- On crée un nouveau compte d'administrateur, avec un autre nom de préférence ("Admin" par exemple) auquel on donne tous les droits et propriétés
- On supprime le compte "Administrateur" du groupe "Administrateurs". Il n'est en effet pas possible de supprimer les Groupes et Utilisateurs par défaut.

La dernière étape n'est pas indispensable, par contre la première l'est, car sans cela, n'importe qui ayant Access sur son poste pourra ouvrir votre base de données.

Utilisateurs et groupes

C'est une notion classique de la sécurité des SGBD. Un groupe représente l'ensemble des utilisateurs possédant les mêmes droits. Un utilisateur représente en générale une personne.

Chaque utilisateur appartient au minimum au groupe des utilisateurs, mais il peut appartenir à plusieurs groupes.

Un utilisateur hérite toujours des droits du(des) groupe(s) dont il dépend, mais peut avoir des droits modifiés par rapport à celui-ci. Attention, en cas de contradiction entre les droits du groupe et ceux de l'utilisateur, c'est toujours la condition la **moins** restrictive qui l'emporte.

Pour simplifier la gestion des droits, on définit habituellement les groupes puis les utilisateurs.

Lorsqu'un groupe est propriétaire, chaque utilisateur du groupe est également propriétaire.

L'héritage des droits d'un groupe est paramétrable.

Héritage des objets

Les droits d'un groupe / utilisateur se définissent normalement pour chaque objet.

Pour une base de données contenant de multiples objets, vous voyez immédiatement le code que cela peut engendrer. Il est possible de propager les droits d'un objet à tous les objets que celui-ci contient. Une solution consiste donc à donner des restrictions standards à l'objet base de données puis à adapter au cas par cas sur les objets. Attention, cette méthode n'est pas réversible, elle n'est donc pas sans risque. Cet héritage n'est jamais transverse, mais il est possible de définir des droits pour une catégorie générique d'objet.

Objet Group

En soit un objet Group n'est pas très complexe. Il ne possède que la collection Users qui représente les utilisateurs membres du groupe, une propriété **Name** qui est son nom et deux méthodes. C'est celles-ci que nous allons voir en détails.

SetPermissions

Attribue les permissions sur un objet pour un groupe ou un utilisateur. De la forme

```
Group.SetPermissions Name, ObjectType, Action, Rights [, Inherit] [, ObjectTypeId]
```

Il s'agit là d'une méthode. Contrairement à la valorisation des propriétés ADOX qui se fait **avant** l'ajout de l'objet à la collection, on applique les méthodes à l'objet **seulement après** son ajout à la collection.

Passons en revue ces paramètres, mais notez bien qu'il s'appliquent aussi bien au groupe qu'à l'utilisateur. Les différences seront vues dans l'étude de l'objet User.

ObjectType

Le paramètre ObjectType peut prendre une des valeurs suivantes :

Constante	Valeur	Description
adPermObjProviderSpecific	-1	Défini un objet spécifique du fournisseur.. Une erreur se produira si le paramètre ObjectTypeId n'est pas fourni
adPermObjTable	1	Objet Table.
adPermObjColumn	2	objet Column
adPermObjDatabase	3	L'objet est la base de données
adPermObjProcedure	4	Objet Procedure.
AdPermObjView	5	Objet View

On retrouve la tous les objets du modèle plus l'objet Database. La valeur adPermObjProviderSpecific permet d'utiliser des objets spécifiques au fournisseur, par exemple pour Access les états ou les formulaires.

Name

Donne le nom de l'objet cible, si vous ne précisez pas ce nom vous devez passer NULL mais vous aurez une erreur si vous omettez le paramètre. Dans ce cas, les droits seront attribués à tous les **nouveaux** objets de même type contenu dans le catalogue (catégorie générique).

Pour l'objet database, vous devez donner une chaîne vide ("") au paramètre Name. Voir dans l'exemple plus loin.

Action

Ce paramètre est combiné avec le paramètre Rights. Il peut prendre les valeurs suivantes :

Constante	Valeur	Description
adAccessGrant	1	Le groupe aura au moins les autorisations demandées.
adAccessSet	2	Le groupe aura exactement les autorisations demandées
adAccessDeny	3	Le groupe n'aura pas les autorisations demandées.
adAccessRevoke	4	Tous les droits d'accès explicites que possèdent le groupe seront annulés.

Détaillons ces valeurs car elles dépendent fortement de la stratégie choisie ainsi que de la gestion de l'existant.

Lorsque l'on considère un utilisateur, on entend par droits (autorisations) explicites, les droits qui appartiennent à celui-ci et par droits implicites les droits de son groupe. Pour un groupe, tous les droits sont explicites.

adAccessRevoke retire donc tous les droits du groupe, il n'est pas utile de renseigner le paramètre Rights (évidemment). Le fait de retirer les droits d'un groupe ne retire pas les droits d'un utilisateur appartenant au groupe lorsqu'ils sont différents de ceux du groupe.

adAccessDeny enlève tous les droits définis dans le paramètre Rights.

adAccessSet est la valeur que l'on utilise en général lors de la création du fichier de sécurité. Les droits définis dans le paramètre Rights seront exactement les droits donnés, ceux existant précédemment seront remplacés ou modifiés.

AdAccessGrant est la valeur pour la modification de droits. Au lieu de redéfinir toute la liste des droits on précise le droit que l'on souhaite modifier, les autres droits ne seront pas redéfinis.

Rights

Valeur de type Long représentant un masque binaire des droits pouvant être une ou plusieurs des valeurs suivantes. Certaines de ces valeurs sont exclusives (adRightFull par exemple) :

Constante	Valeur	Description
adRightCreate	16384 (&H4000)	Le groupe à l'autorisation de créer des objets de ce type
adRightDelete	65536 (&H10000)	Le groupe a l'autorisation de supprimer des données de l'objet
adRightDrop	256 (&H100)	Le groupe a l'autorisation de supprimer l'objet
adRightExclusive	512 (&H200)	Le groupe a l'autorisation d'accéder de façon exclusive à l'objet
adRightExecute	536870912 (&H20000000)	Le groupe a l'autorisation d'exécuter l'objet.(macro Access par exemple)
adRightFull	268435456 (&H10000000)	Le groupe a toutes les autorisations concernant l'objet
adRightInsert	32768 (&H8000)	Le groupe a l'autorisation d'insérer des données dans l'objet.
adRightMaximumAllowed	33554432 (&H2000000)	Le groupe a le maximum d'autorisations spécifiques autorisées par le fournisseur.
adRightNone	0	Le groupe n'a aucune autorisations concernant l'objet.
adRightRead	-2147483648 (&H80000000)	Le groupe a l'autorisation de lire l'objet
adRightReadDesign	1024 (&H400)	Le groupe a l'autorisation de lire la définition de l'objet
adRightReadPermissions	131072 (&H20000)	Le groupe a l'autorisation de lire les autorisations de l'objet
adRightReference	8192 (&H2000)	Le groupe a l'autorisation de référencer l'objet
adRightUpdate	1073741824 (&H40000000)	Le groupe a l'autorisation de modifier des données dans l'objet.
adRightWithGrant	4096 (&H1000)	Le groupe a l'autorisation d'accorder des autorisations concernant l'objet
adRightWriteDesign	2048 (&H800)	Le groupe a l'autorisation de modifier la structure de l'objet
adRightWriteOwner	524288 (&H80000)	Le groupe a l'autorisation de modifier le propriétaire de l'objet
adRightWritePermissions	262144 (&H40000)	Le groupe a l'autorisation de modifier des autorisations spécifiques de l'objet.

Comme il s'agit de masque binaire, on compose les droits avec l'opérateur "OR".

Une erreur fréquente consiste à considérer l'attribution des droits par programmation comme fonctionnant à l'identique de l'assistant sécurité. Celui-ci crée implicitement des droits lorsque l'on attribue certains droits à l'utilisateur (par exemple l'autorisation "modifier les données" donne automatiquement l'autorisation "lire les données"). Il n'en est pas de même par programmation. Vous devez définir de façon cohérente et explicite les droits sous peine de bloquer vos utilisateurs.

Une autre erreur consiste à utiliser un droit qui n'existe pas pour l'objet considéré. Ainsi j'ai déjà vu donner le droit adRightExecute sur des procédures, sûrement parce que l'on dit exécuter une requête, alors que c'est le droit adRightRead qu'il faut utiliser.

Inherit

Paramètre optionnel qui détermine comment les objets contenus dans l'objet cible hériteront des autorisations. Les valeurs peuvent être :

Constante	Valeur	Description

adInheritNone	0	Valeur par défaut. Pas d'héritage.
adInheritObjects	1	Les objets non-conteneurs héritent des autorisations.
adInheritContainers	2	Les objets conteneurs héritent des autorisations.
adInheritBoth	3	Tous les objets héritent des autorisations.
adInheritNoPropagate	4	Les objets n'ayant pas déjà hérités d'autorisations hériteront.

L'héritage a parfois des résultats surprenants avec ADOX, je vous conseille donc de ne pas en abuser. Toutefois on peut toujours déterminer une configuration type par groupe et la donner à la base de données avec un héritage, afin que les objets contenus héritent des autorisations. Je ne vous conseille pas d'utiliser cette technique.

ObjectTypeId

Ce paramètre est obligatoire avec le type adPermObjProviderSpecific. Cela permet d'accéder à certains objets contenus dans le SGBD. Il faut passer comme valeur le GUID de l'objet. Par exemple pour Access les GUIDs suivants permettent de définir des autorisations pour les formulaires, états et macro

Object	GUID
Form	{c49c842e-9dcb-11d1-9f0a-00c04fc2c2e0}
Report	{c49c8430-9dcb-11d1-9f0a-00c04fc2c2e0}
Macro	{c49c842f-9dcb-11d1-9f0a-00c04fc2c2e0}

GetPermissions

Permet de lire les autorisations d'accès d'un objet. Ce n'est pas toujours très simple à interpréter par programmation, aussi je vous conseille plutôt de supprimer puis de recréer les autorisations.

De la forme

ReturnValue = *Group*.**GetPermissions(***Name*, *ObjectType* [, *ObjectTypeId*]**)**

Ou ReturnValue est une valeur de type Long représentant le masque binaire. Pour tester si une autorisations existe dans le masque utiliser l'opérateur AND

La fonction suivante permet d'afficher les droits dans la fenêtre d'exécution.

```
var MaTable: _table;
MonGroup: _Group;
Droits: integer;
begin
     MonCat:=coCatalog.Create;

moncat.Set_ActiveConnection('Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\tutoriel\biblio.mdb;Jet OLEDB:System
database=c:\tutoriel\system.mdw;User Id=Admin; Password=');
     Mongroup:=coGroup.Create;
     mongroup:=moncat.Groups.Item[1];

Droits:=MonGroup.GetPermissions('Authors',adPermObjTable,EmptyParam)
;
     If Bool(Droits And adRightCreate) Then showmessage('Création');
     If Bool(Droits And adRightDelete) Then
showmessage('Effacement');
     If Bool(Droits And adRightDrop) Then
showmessage('Suppression');
     If Bool(Droits And adRightExclusive) Then
showmessage('Exclusif');
     If Bool(Droits And adRightExecute) Then
showmessage('Exécution');
     If Bool(Droits And adRightInsert) Then
showmessage('Insertion');
     If Bool(Droits And adRightRead) Then showmessage('Lecture D');
     If Bool(Droits And adRightReadDesign) Then showmessage('Lecture
S');
```

```
     If Bool(Droits And adRightReadPermissions) Then
showmessage('LirePermission');
     If Bool(Droits And adRightReference) Then
showmessage('Reference');
     If Bool(Droits And adRightUpdate) Then
showmessage('Modification');
     If Bool(Droits And adRightWithGrant) Then
showmessage('ModifPermission');
     If Bool(Droits And adRightWriteDesign) Then showmessage('Modif
Structure');
     If Bool(Droits And adRightWriteOwner) Then showmessage('Modif
Prop');

end;
```

Objet User

Cet objet ressemble fortement à l'objet Group dans sa programmation. Il possède une méthode de plus.

ChangePassword

Permet de modifier le mot de passe d'un utilisateur.
*User***.ChangePassword** *OldPassword***,** *NewPassword*
Là pas besoin d'explication si ce n'est qu'un des paramètres peut être la chaîne vide ""

GetPermissions & SetPermissions

S'utilise comme pour l'objet Group.

Comme je l'ai déjà dit un utilisateur membre d'un groupe peut avoir des autorisations différentes de celles du groupe. Pour des raisons de maintenance, cette situation est toutefois à proscrire. N'oubliez pas que sur un mélange de propriétés utilisateur / groupe c'est toujours l'autorisation la moins restrictive qui l'emporte.

Properties

Les objets User et Group ont une collection properties.

Exemple

Dans l'exemple suivant nous allons sécuriser une base de données, créer deux groupes et deux utilisateurs.

En phase de création de base, commencez toujours par gérer la sécurité comme vous allez le voir c'est beaucoup plus simple.

```
var MonGroup: _Group;
MonUser: _User;
begin
     MonCat:=coCatalog.Create;
     moncat.Create('Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\tutoriel\secu.mdb;Jet OLEDB:System
database=c:\tutoriel\system.mdw;User Id=Admin; Password=');
     Mongroup:=coGroup.Create;
     MonCat.Users.Item['Admin'].ChangePassword('','password');
     Moncat.Groups.Append('Utilisateurs');
     Mongroup:=MonCat.Groups.Item['Utilisateurs'];
     with MonGroup do begin
         SetPermissions(Null, adPermObjTable, adAccessSet,
adRightInsert Or adRightRead Or adRightUpdate,
adInheritNone,EmptyParam);
         SetPermissions(Null, adPermObjProcedure, adAccessSet,
adRightRead Or adRightReadDesign, adInheritNone,EmptyParam);
```

```
          SetPermissions(Null, adPermObjView, adAccessSet,
adRightRead Or adRightReadDesign, adInheritNone,EmptyParam);
     end;
     Moncat.Groups.Append('Consultants');
     Mongroup:=MonCat.Groups.Item['Consultants'];
     with MonGroup do begin
          SetPermissions(Null, adPermObjTable, adAccessSet,
adRightNone, adInheritNone,EmptyParam);
          SetPermissions(Null, adPermObjProcedure, adAccessSet,
adRightNone, adInheritNone,EmptyParam);
          SetPermissions(Null, adPermObjView, adAccessSet,
adRightRead, adInheritNone,EmptyParam);
     end;
     MonCat.Users.Append('Emile','');
     MonCat.Users.Item['Emile'].ChangePassword('','xtf22re');

MonCat.Groups.Item['Utilisateurs'].Users.Append('Emile',unassigned);
     MonUser:= MonCat.Users.Item['Emile'];
     with MonUser do begin
          SetPermissions(Null, adPermObjTable, adAccessSet,
adRightInsert Or adRightRead Or adRightUpdate,
adInheritNone,EmptyParam);
          SetPermissions(Null, adPermObjProcedure, adAccessSet,
adRightRead Or adRightReadDesign, adInheritNone,EmptyParam);
          SetPermissions(Null, adPermObjView, adAccessSet,
adRightRead Or adRightReadDesign, adInheritNone,EmptyParam);
     end;
end;
```

Comme nous le voyons, la première étape après la création est de changer le mot de passe de l'administrateur afin d'activer la sécurité.

Je crée ensuite deux groupes dont je définis les droits puis deux utilisateurs (1 par groupe). Notez que je redéfinis les mêmes droits puisque ADO avec Access ne génère pas l'héritage implicite des droits du groupe.

Dans ce cas, je n'ai encore créé aucun objet dans ma table. Ceci me permet d'utiliser des catégories génériques pour l'attribution des droits (utilisation de Null pour le paramètre Name).

Vous remarquerez enfin que j'utilise SetPermissions après l'ajout de l'objet à sa collection

Techniques de sécurisation

Il est bien évident que gérer la sécurité par le code n'est pas une sinécure, aussi on essaye toujours de passer par les assistants ad hoc des SGBD plutôt que de se lancer dans un code aventureux.

Dans le cas d'Access, il est beaucoup plus parlant de gérer les sécurités depuis Access. Si toutefois vous devez utiliser le code, vous gagnerez en efficacité et en simplicité à suivre le modèle DAO pour gérer la sécurité, du moins tant que Microsoft n'aura pas sorti une version ADO faisant cela efficacement.

Nous allons partir du principe dans cette partie que l'emploi du code avec ADO est indispensable.

Grosso modo il n'existe que deux cas, soit vous allez modifier des groupes, des utilisateurs ou des autorisations existantes soit il vous faut tout créer.

Modification

Si la sécurité est déjà bien gérée, il ne s'agit que d'ajouter des utilisateurs et de bien gérer les droits d'accès ou de propriété. La technique consiste toujours à créer un utilisateur dans un groupe existant, afin de ne pas avoir à gérer tous ses droits objet par objet. Lors de la modification d'autorisations, il est souvent plus simple de détruire les anciennes, puis de ré attribuer les autorisations. Cherchez toujours à donner la propriété des nouveaux objets à l'administrateur ou alors créez un utilisateur fantôme, appelés par exemple "Code", qui possède les autorisations nécessaires à tous vos accès par le code.

Si la sécurité est mal gérée, détruisez tout et repartez du début.

N'essayez pas de prendre la propriété des tables dont le propriétaire est "engine" (information de schéma du moteur Jet) vous déclencheriez une erreur.

Création

Lors de la création de la base, je vous conseille vivement de créer la sécurité en premier. Vous pouvez dès lors utiliser des objets génériques et définir vos droits assez rapidement. Il vaut toujours mieux avoir un groupe de trop que l'on supprime à la fin, plutôt que de se rendre compte qu'il en manque un.

Définissez votre système de sécurité avant de coder. Par le code il vaut mieux multiplier les groupes que les utilisateurs.

Une autre solution : le DDL

A partir d'Access 2000, vous pouvez contourner les faiblesses de ADO pour la création des utilisateurs et des groupes en utilisant le DDL. Sachez que celui vous permet de gérer les PID.

Un objet Command peut être utilisé avec une requête de type :

CREATE USER Nom MotDePasse PID

Mais nous sortons là du cadre de cet article

Conclusion sur la sécurité

Comme nous l'avons vu, gérer la sécurité par le code n'est souvent pas un bon choix. Cela reste possible à faire dans certains cas particulier mais le code est relativement lourd et dur à corriger. Privilégiez toujours la solution des assistants tant que cela est possible.

Bien que je n'ai pas abordé cela, vous aurez une erreur si l'utilisateur qui se connecte n'a pas les droits d'écriture de structure et tente d'ouvrir le catalogue.

Ce catalogue peut être partiel en fonction des droits définis.

CONCLUSION

Vous voilà désormais maîtrisant ADO tel Cochise son tomahawk. Les quelques règles à toujours garder en mémoire sont :

Bien choisir son curseur

Privilégiez une approche Delphi plutôt qu'ADO pur.

Utilisez le SQL quand le fournisseur cible est unique.

Si vous rencontrez des problèmes n'hésitez pas à venir poser vos questions sur

Developpez.com : Delphi et base de données

Bonne programmation.