Developpez.com
Club des développeurs

Développer un DataSet en mémoire

Optimisation des accès Base de données, IIème Partie

par Franck SORIANO (Pages perso)

Date de publication : 06/01/2009

Dernière mise à jour :

Cet article présente en détail le fonctionnement de la classe **TDataSet**. Il explique notamment comment dériver la classe TDataSet pour réaliser un dataset en mémoire : la classe **TMemoryDataSet**.

Les articles suivants utiliseront ce dataset pour s'interfacer avec **OLEDB**.

Commentez cet article :

I - Introduction....3
Télécharger les sources de l'article....3
II - Organisation générale du TDataSet....4
II-A - Gestion des buffers....4
II-B - Ouverture/Fermeture du curseur....4
II-C - Navigation....5
II-D - Modification des données....5
III - La classe TMemoryDataSet....6
III-A - Organisation des données en mémoire....6
III-B - Gestion des buffers....7
III-B-1 - Définition des Champs du TMemoryDataSet....7
III-B-2 - Gestion des champs LOB....7
III-B-3 - Organisation d'un buffer....7
III-B-4 - Implémentation des buffers....9
III-B-4-a - GetRecordSize....9
III-B-4-b - AllocRecordBuffer....10
III-B-4-c - FreeRecordBuffer....11
III-B-4-d - InternalInitRecord....11
III-B-4-e - Accès direct aux champs....11
III-C - Ouverture/Fermeture....12
III-C-1 - InternalOpen....12
III-C-2 - InternalClose....13
III-C-3 - IsCursorOpen....14
III-C-4 - InternalInitFieldDefs....14
III-D - Navigation....15
III-D-1 - GetRecord....15
III-D-2 - GetBookmarkFlag/SetBookmarkFlag....19
III-D-3 - GetBookmarkData/SetBookmarkData....19
III-D-4 - InternalGotoBookmark....20
III-D-5 - BookmarkValid....21
III-D-6 - InternalSetToRecord....22
III-D-7 - InternalFirst....22
III-D-8 - InternalLast....22
III-D-9 - GetRecNo/SetRecNo....23
III-E - Lecture/Ecriture des champs....24
III-E-1 - GetFieldData....24
III-E-2 - SetFieldData....26
III-F - Gestion des LOB....27
III-F-1 - TMemoryBlobStream....28
III-F-2 - CreateBlobStream....32
III-G - Modification des données....32
III-G-1 - GetCanModify....32
III-G-2 - InternalEdit....33
III-G-3 - InternalCancel....34
III-G-4 - InternalPost....35
III-G-5 - InternalDelete....36
Conclusion....36
IV - Quelques cas d'utilisations....37
IV-A - Scénario 1 : Ouverture/Fermeture du DataSet....37
IV-B - Scenario 2 : Affichage et Navigation avec une DBGrid....41
IV-B-1 - First....41
IV-B-2 - Last....42
IV-B-3 - Prior....43
IV-B-4 - Next....43
IV-B-5 - SetRecNo....44
V - Conclusion....45
VI - Remerciements....45
VII - Références....45

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I - Introduction

Un DataSet en mémoire. Encore un ! Ce type de composant fait légion sur le net. Même Delphi intègre le **TClientDataSet** en standard qui peut faire office de DataSet en mémoire.

Alors pourquoi vouloir en développer un de plus ? Tout simplement pour trois raisons majeures :

- Malgré tous les composants existants, je n'en ai jamais trouvé un seul qui ait le niveau de performances que j'attends. Comme je l'ai expliqué dans le **précédent article**, le **TClientDataSet** est très lourd, avec une architecture complexe. On ne dispose d'aucun moyen permettant de le charger efficacement.
- Le **TDataSet** est au coeur de toute application base de données en Delphi. Si on veut écrire des applications performantes, il est important de comprendre les mécanismes mis en oeuvre dans son implémentation. De cette manière, lorsqu'on écrit une ligne de code, on appréhende mieux ses conséquences. Ecrire un DataSet en mémoire est un très bon prétexte pour étudier son fonctionnement.
- Enfin, on aura besoin de surcharger un **TDataSet** dans les prochains articles afin de s'interfacer avec OLEDB.

Nous allons voir les principes de base permettant d'implémenter un **TDataSet** complet. Il sera cependant bridé aux fonctionnalités essentielles :

- Navigation, Curseur bidirectionnel.
- Accès direct à une ligne.
- Les champs LOB (Blob, Memo, ...) seront gérés.
- Les champs calculés ne seront pas gérés.
- Les index et toutes les méthodes associées ne seront pas gérés.
- Les filtres seront partiellement gérés. On ne pourra filtrer le dataset qu'avec l'événement **OnFilterRecord**.

Cependant, si les index et les filtres ne seront pas totalement gérés, je donnerai quelques pistes pour ceux qui voudraient le faire eux même.

Télécharger les sources de l'article

Description	Emplacement
TMemoryDataSet	**ftp://ftp-developpez.com/fsoriano/archives/db/dataset/delphi/fichiers/MemoryDataSet.zip**
ETW	**ftp://ftp-developpez.com/fsoriano/archives/etw/delphi/fichiers/etw-sources.zip**
L'article au format docx	**ftp://ftp-developpez.com/fsoriano/archives/db/dataset/delphi/fichiers/dataset.docx**

L'ensemble des codes sources compile avec **Turbo Delphi Explorer**.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

II - Organisation générale du TDataSet

Tous les composants d'accès aux données permettant de lire un jeu de données dérivent d'un ancêtre commun : La classe **TDataSet**.

Cette dernière définit les opérations et les mécanismes communs à tous les composants. Cependant elle ne s'occupe pas de la source de données elle-même. Elle ne peut pas être instanciée directement car bon nombre de méthodes essentielles sont abstraites.

L'écriture d'un DataSet personnalisé consiste essentiellement à implémenter ces méthodes abstraites.

Elles peuvent être classées en plusieurs catégories :

II-A - Gestion des buffers

La classe **TDataSet** maintient un cache permettant de conserver en mémoire un certain nombre de lignes sans être obligé de déplacer constamment le curseur provenant du SGBD.

Ce cache est composé d'un ensemble de buffers. Chaque buffer permet de stocker une ligne de données complète.

La classe **TDataSet** décide elle-même du nombre de buffers à gérer. En principe, il s'agit plus ou moins du nombre de lignes visibles dans les composants **TDBGrid** liés. Ainsi, les DBGrids bénéficient d'un mécanisme spécial permettant de lire rapidement les données sans être obligé de toucher au curseur de la base de données.

Ces buffers servent également en modification. Ils permettent de stocker temporairement les valeurs de la ligne en cours de modification, avant que cette dernière ne soit envoyée au SGBD. Ils permettent également de mémoriser les valeurs de la ligne avant modification pour pouvoir les annuler si besoin.

Un buffer correspond à une ligne complète. Il contient les valeurs de tous les champs de la ligne. Tous les buffers ont la même taille. Ceci pose un problème pour les champs de taille variable. En effet, en principe ils doivent être stockés à l'intérieur du buffer. Comme ce dernier a une taille fixe, on est obligé de prévoir des buffers suffisamment grands pour stocker des champs de taille maximale. Ainsi si on définit un champ *varchar(4000)* dans la base de données, le champ occupera bel et bien 4000 octets une fois chargé à l'intérieur du DataSet. Et ce quelle que soit la taille réelle du champ, même si ce dernier est à NULL !

Les allocations et destructions de buffers sont gérées automatiquement par la classe **TDataSet**. A notre niveau, on doit simplement calculer la taille d'un buffer et fournir une méthode d'allocation et de libération de la mémoire.

II-B - Ouverture/Fermeture du curseur

Le principe du DataSet est d'encapsuler un curseur sur une source de données. Bien évidemment, la classe **TDataSet** ne peut pas le créer elle-même. On doit donc définir des méthodes pour sa création.

On remarquera au passage, que c'est généralement lors de l'ouverture du curseur base de données que l'on peut connaître la structure du jeu de résultat. C'est donc à ce moment que l'on peut connaître les champs qui font partie du DataSet, calculer la taille des buffers et créer les composants **TField**.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

II-C - Navigation

La classe **TDataSet** définit les fonctions de navigation de base. C'est-à-dire : **First**, **Next**, **Prior** et **Last**. Mais toutes ces opérations peuvent ne pas être disponibles en fonction du type de dataset. Sur un dataset unidirectionnel, on peut uniquement faire un **Next**.

Lorsqu'on définit un dataset personnalisé, il faut implémenter ces méthodes pour se déplacer sur la source de données et charger les buffers avec les données des lignes correspondantes.

Les fonctions de navigation et de recherche plus évoluées (**Locate**, **FindKey**, **GotoKey**...) ne sont pas gérées par la classe **TDataSet**. C'est aux descendants de les définir et de les implémenter si nécessaire.

En revanche, la gestion des bookmarks fait partie du **TDataSet** et ces derniers doivent être implémentés dans les classes dérivées (sauf pour les datasets unidirectionnels).

II-D - Modification des données

La classe **TDataSet** gère automatiquement les méthodes **Insert**, **Append**, **Edit**, **Cancel** et **Post**. Elle assure la gestion des buffers et maintient la définition de l'état du dataset. En revanche, les classes dérivées doivent s'occuper de répercuter ces modifications sur la source de données si elles sont validées.

De plus, la classe **TDataSet** ne définit pas le mode de stockage des champs à l'intérieur du buffer. Chaque descendant est libre d'organiser la structure interne des buffers comme bon lui semble. Aussi, lorsque les **TField** essaient de lire/écrire la valeur d'un champ dans le buffer, il faut implémenter les méthodes d'extraction/écriture de cette valeur.

Si on implémente toutes les méthodes dans chacune de ces catégories, on obtient un DataSet bidirectionnel, complètement opérationnel et pouvant être affiché dans une grille.

Si on connecte ce dataset à un tableau en mémoire en guise de source de données, on obtient un dataset en mémoire.

C'est ce que nous allons mettre en pratique immédiatement en développant la classe **TMemoryDataSet**.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III - La classe TMemoryDataSet

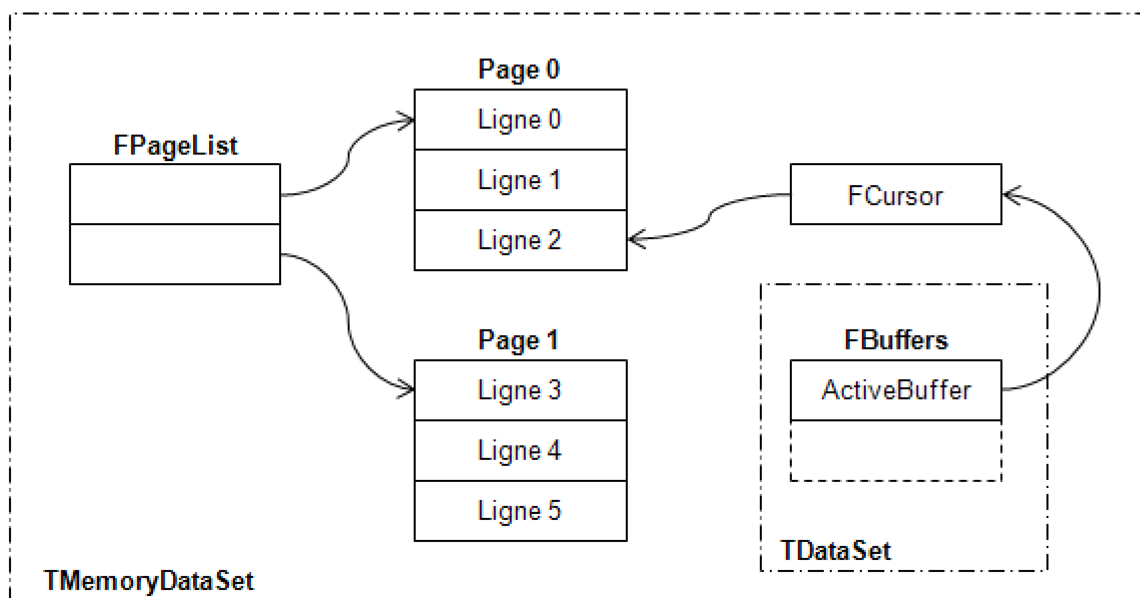
III-A - Organisation des données en mémoire

Le **TMemoryDataSet** va devoir stocker les lignes qui seront lues par le DataSet. Pour cela on définit simplement un tableau de buffers. Par contre, par la suite il faudra que les lignes occupent des espaces mémoires contigus.

C'est pourquoi ces dernières seront regroupées par Pages. Chaque page correspondra à un bloc mémoire pouvant contenir un certain nombre de lignes (propriété **PageSize**). Ainsi, les allocations mémoires s'effectueront page par page et non pas ligne par ligne. On évitera ainsi de faire de l'allocation mémoire à outrance.

Lorsqu'une page est pleine, on en allouera une deuxième, puis une troisième...

Enfin, on va maintenir une liste des pages.



Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-B - Gestion des buffers

III-B-1 - Définition des Champs du TMemoryDataSet

La classe **TDataSet** prévoit qu'on utilise une collection **TFieldDefs** pour décrire les champs du DataSet. Inutile de réinventer la roue, c'est ce que nous allons faire.

En principe, on définit les **TFieldDef** pour indiquer les champs que l'on souhaite gérer dans le DataSet. Cette définition doit être effectuée avant son ouverture.

Au moment de l'ouverture du **TMemoryDataSet**, ce dernier créera automatiquement les **TField** nécessaires pour accéder aux données.

Avec les composants base de données de Delphi, il est possible de définir les **TField** en design, alors que le dataset est fermé. Cela permet de configurer les paramètres d'affichage des **TField** depuis l'IDE. On peut alors les réordonner, définir les *DisplayFormat*, *DisplayLabel*...

Cependant, je ne conseille pas cette façon de travailler. En effet, si elle permet de développer rapidement une maquette fonctionnelle, cette approche pose ensuite de gros problèmes de maintenance. Si on ajoute un champ dans la base, il faut repasser sur tous les datasets qui ont été ainsi configurés, sinon les nouveaux champs ne sont pas visibles. Si un champ change de taille, on obtient une erreur à l'ouverture du DataSet...

C'est pourquoi, la classe **TMemoryDataSet** ignorera purement et simplement les **TField** créés en design. Ils seront systématiquement détruits et recréés à l'ouverture du DataSet. D'ailleurs, on ne prévoira même pas la possibilité d'installer le composant dans l'IDE.

III-B-2 - Gestion des champs LOB

Les champs LOB (Memo, Binaire, ...) posent un problème particulier. Ils peuvent avoir une taille importante, voir très importante. Leur taille n'est pas limitée, tandis qu'au contraire les buffers des lignes ont une taille fixe.

Cette simple particularité interdit purement et simplement de les stocker directement dans les buffers.

On va devoir les stocker séparément. Une technique simple consiste à ne stocker à l'intérieur du buffer qu'un pointeur sur un autre bloc mémoire qui lui contient réellement les données du LOB.

Avec cette technique, il faut faire attention à ne pas créer de fuite mémoire. Pour s'en assurer, on va travailler légèrement différemment. Les champs LOB alloués sont stockés dans une liste indépendante à l'intérieur de **TMemoryDataSet**. Dans le buffer d'une ligne, au lieu de stocker directement le pointeur sur le contenu du LOB, on va stocker le numéro de LOB dans la liste des LOB. De cette façon, on conserve toujours les références sur les LOB qui ont été alloués.

Au pire, la destruction du **TMemoryDataSet** libérera les LOBs oubliés.

III-B-3 - Organisation d'un buffer

La première chose à faire est de définir la structure d'un buffer. Ce dernier doit pouvoir contenir les données de tous les champs du DataSet.

Sur le principe, il va s'agir de la juxtaposition des valeurs de chaque champ. Cependant, la valeur d'un champ n'est pas suffisante. Pour chaque champ on doit également savoir si le champ est à *NULL* ou s'il possède une valeur. Pour les champs de taille variable, il faut également que l'on sache quelle est la longueur courante du champ.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Aussi, pour chaque champ, nous allons définir la structure suivante :

```
type
  // La structure TFieldData représente le format de stockage d'un champ
  // à l'intérieur d'un buffer du DataSet.
  PFieldData = ^TFieldData;
  TFieldData = packed record
    NullStatus : integer;       // Indique si le champ est à NULL.
    LengthValue : integer;      // Indique la longueur actuelle du champ.
                                // LengthValue n'a de sens que pour les champs de taille variable.
    Data : array[0..3] of byte; // Début des données du champ.
  end;
```

On utilise un *record* et non pas une classe car ces structures vont ensuite être alignées à l'intérieur de chaque buffer. Si on avait définit une classe, Delphi n'aurait pas stocké la structure à l'intérieur du buffer, mais un pointeur sur l'instance.

J'ai utilisé un *integer* pour indiquer si un champ est à NULL. Ca peut paraître comme un gaspillage mémoire (et c'en est bien un) car un simple *boolean* aurait pu faire l'affaire. On aurait également pu optimiser la mémoire en définissant un tableau de bits à l'intérieur du buffer.

Cependant si toutes ces solutions optimisent la mémoire, elles n'optimisent pas les performances. A terme, on utilisera la classe **TMemoryDataSet** pour charger les données retournées par OLEDB. Or pour obtenir les meilleures performances avec OLEDB, il vaut mieux que ces indicateurs soient des entiers sur 32 bits.

Le champ **Data** indique simplement un tableau d'octets. Sa taille réelle dépendra du type du champ stocké.

Outre les données relatives aux champs, un buffer doit également contenir trois informations supplémentaires nécessaires à la gestion du Dataset :

Les BookmarkFlags : Il s'agit d'une information utilisée par **TDataSet** pour mémoriser l'état du buffer par rapport à l'ensemble de données. Elle indique au dataset, s'il s'agit de la première ligne (*bfBOF*), de la dernière ligne (*bfEOF*), d'une ligne ordinaire (*bfCurrent*) ou d'une ligne en cours d'insertion (*bfInserted*).

Identifiant du buffer : Les buffers doivent posséder un identifiant unique. Il peut s'agir d'un simple entier. Cet identifiant permet au dataset de positionner la source de données sur un buffer en particulier.

Bookmark : Le bookmark est un identifiant indiquant précisément la ligne sur laquelle on se trouve. C'est cette valeur qui est retournée lorsqu'on fait un **GetBookmark**. Lorsqu'on veut effectuer un **GotoBookmark**, il faut rechercher la ligne qui possède le bookmark voulu. Cette information est un peu redondante avec l'identifiant du buffer. C'est pourquoi il est fréquent d'utiliser le même identifiant comme identifiant de buffer et valeur de bookmark.

Ainsi la structure d'un buffer est la suivante :

```
type
  // TLineBuffer désigne la structure d'un buffer.
  PLineBuffer = ^TLineBuffer;
  TLineBuffer = packed record
    BkFlags : TBookmarkFlag;    // Bookmark flag
    Bookmark : cardinal;        // Valeur du bookmark/identifiant du buffer
    Data : array[0..0] of byte; // Début de la zone des données des champs
  end;
```

Data est en réalité en tableau de structures **TFieldData**. Cependant on ne peut pas le déclarer comme tel car chaque élément possède une taille variable.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://fsoriano.developpez.com/articles/db/dataset/delphi/

III-B-4 - Implémentation des buffers

III-B-4-a - GetRecordSize

La méthode **GetRecordSize** doit retourner la taille d'un buffer en octets. Cette dernière dépend du mode d'organisation choisi pour les buffers, mais aussi des champs présents dans le dataset. La taille d'un buffer peut donc être calculée au moment de l'ouverture du dataset et mémorisée jusqu'à sa fermeture :

```
// Calcule la taille d'un buffer, en fonction des champs définis dans FieldDefs.
// Une fois calculé, le résultat est mémorisé dans FRecordSize.
procedure TMemoryDataSet.CalcRecordSize;
var
  i : integer;
  currentOffset : cardinal;
begin
  SetLength(FFieldInfo, FieldDefs.Count);

  // Initialisation de la taille initiale du buffer.
  currentOffset := sizeof(TLineBuffer) - 1; // On doit soustraire la taille de TLineBuffer.Data

  // Maintenant, il faut ajouter la taille de chaque champ. On en profite pour indexer les champs
  // à l'intérieur du buffer et initialiser le tableau FFieldInfo.
  for i := 0 to FieldDefs.Count -1 do
  begin
    FFieldInfo[i].Offset := currentOffset; // On mémorise la position du champ dans un buffer.
    FFieldInfo[i].Size := GetFieldSize(FieldDefs[i]);
    inc(currentOffset, sizeof(TFieldData) - 4 + FFieldInfo[i].Size);
  end;
  FRecordSize := currentOffset;      // On mémorise la taille du buffer qui vient d'être calculée.
  BookmarkSize := sizeof(cardinal); // On mémorise la taille d'un bookmark.

{$IFDEF TRACE}
  // On vient de calculer la taille des buffers. On trace cette information sous
  // la forme d'un message d'information au niveau TRACE_LEVEL_VERBOSE
  GenericLogger.Trace(EVENT_INFO, 'CalcRecordSize=' +
    IntToStr(FRecordSize), TRACE_LEVEL_VERBOSE);
{$ENDIF}
end;
```

FFieldInfo est un tableau mémorisant les informations nécessaires pour accéder directement à un champ à l'intérieur d'un buffer. Il est déclaré de la façon suivante :

```
    FFieldInfo : array of TFieldInfo;
```

Avec :

```
type
  TFieldInfo = record
    Offset : cardinal; // Offset d'un champ dans un buffer, par rapport au début du buffer.
    Size : cardinal;   // Taille de la zone des données du champ à l'intérieur du buffer.
  end;
```

La taille du buffer est mémorisée dans **FRecordSize**. **GetRecordSize** n'a qu'à renvoyer cette valeur :

```
// Retourne la taille d'un buffer.
function TMemoryDataSet.GetRecordSize: word;
begin
  result := FRecordSize; // On renvoie la valeur calculée lors de l'ouverture du DataSet.
end;
```

Pour effectuer le calcul, on fait appel à une méthode **GetFieldSize**. Cette dernière est chargée de calculer l'espace nécessaire pour stocker un champ en fonction de son type et du mode de stockage choisi :

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
// Retourne la taille en octets nécessaire pour stocker le champ décrit dans <FldDef>.
function TMemoryDataSet.GetFieldSize(FldDef: TFieldDef): cardinal;
begin
  case FldDef.DataType of
  ftGuid, ftString, ftFixedChar, ftBytes, ftVarBytes :
    result := FldDef.Size+1; // La taille correspond à la taille du champ plus un 0 terminal.
  ftSmallint, ftWord: // Entier court
    result := 2;
  ftAutoInc, ftInteger: // Entier long
    result := 4;
  ftBoolean: // Les boolean sont stockés comme des entiers courts.
    result := 2;
  ftFloat: // Les floats sont stockés dans des double.
    result := sizeof(double);
  ftCurrency: // Le TCurrencyField veut lire un double dans le dataset
    result := sizeof(double);
  ftBCD: // Le TBCDField lit un currency depuis le dataset.
    result := sizeof(currency);
  ftFMTBcd : // Le TFMTBcdField lit un TBCD.
    result := sizeof(TBCD);
  ftDateTime, ftDate, ftTime : // Dans les trois cas, on lit un TDateTime.
    result := sizeof(TDateTime);
  ftTimeStamp:
    result := sizeof(TSQLTimeStamp);
  ftWideMemo, ftOraBlob, ftOraClob, ftBlob,
  ftMemo, ftGraphic, ftFmtMemo: // Les LOB font l'objet d'une gestion spéciale
    result := 4;
  ftFixedWideChar, ftWideString: // Chaînes unicode.
    result := FldDef.Size*2+2;
  ftLargeint: // Entier long
    result := sizeof(int64);
  ftVariant: // Le variant est stocké dans un variant...
    result := sizeof(variant);
  else raise MemoryDataSetNotSupported.Create(FldDef.Name, FldDef.DataType);
  end;
end;
```

III-B-4-b - AllocRecordBuffer

Cette méthode doit être implémentée pour allouer un nouveau buffer. On peut se contenter d'effectuer une simple allocation mémoire. On initialise également les champs à l'intérieur du buffer à la valeur NULL :

```
// Cette méthode est appelée par TDataSet pour allouer un nouveau buffer.
function TMemoryDataSet.AllocRecordBuffer: PChar;
begin
{$IFDEF TRACE}
  GenericLogger.Trace(EVENT_INFO, 'AllocRecordBuffer',
    TRACE_LEVEL_VERBOSE);
{$ENDIF}
  // On alloue simplement un bloc mémoire de la taille d'un buffer.
  GetMem(Result, GetRecordSize);

  // Ensuite on initialise le buffer pour définir tous les champs à NULL.
  InternalInitRecord(Result);
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-B-4-c - FreeRecordBuffer

FreeRecordBuffer est le symétrique de **AllocRecordBuffer**. Cette méthode doit être implémentée pour libérer les buffers alloués avec **AllocRecordBuffer**. Comme nos buffers sont de simples allocations mémoire, il suffit de faire le free correspondant :

```
// Libère la mémoire utilisée par le buffer <Buffer>. <Buffer> doit contenir
// un buffer préallablement alloué avec AllocRecordBuffer.
procedure TMemoryDataSet.FreeRecordBuffer(var Buffer: PChar);
begin
{$IFDEF TRACE}
  GenericLogger.Trace(EVENT_INFO, 'FreeRecordBuffer',
    TRACE_LEVEL_VERBOSE);
{$ENDIF}

  // Libération de la mémoire.
  FreeMem(Buffer);
end;
```

III-B-4-d - InternalInitRecord

La méthode **InternalInitRecord** peut être surchargée pour initialiser un buffer avec une valeur spécifique. La méthode de base dans **TDataSet** ne fait rien.

Nous la surchargeons pour forcer tous les champs à NULL dans les nouveaux buffers :

```
// Permet d'initialiser une nouveau <Buffer>. Le <Buffer est initialisé en
// définissant tous les champs à NULL.
procedure TMemoryDataSet.InternalInitRecord(Buffer: PChar);
var i : integer;
begin
  // On trace l'appel à InternalInitRecord au niveau TRACE_LEVEL_VERBOSE.
  // On pourra ainsi suivre les allocations/libération de buffer effectuées par
  // le TDataSet et analyser son fonctionnement en détails si besoin.
  SQLLogger.Trace(EVENT_TRACE_TYPE_INFO, 'InternalInitRecord',
    TRACE_LEVEL_VERBOSE);

  // On définit tous les champs à NULL un par un.
  for i := 0 to high(FFieldInfo) do
  begin
    ClearField(Buffer, FFieldInfo[i].Offset);
  end;
end;
```

ClearField est une procédure utilitaire **inline** pour forcer le statut d'un champ à NULL.

III-B-4-e - Accès direct aux champs

Par la suite, nous aurons constamment besoin d'accéder aux champs individuellement. Comme ces derniers sont alignés à l'intérieur d'un buffer, nous avons besoin d'une série de fonctions permettant d'accéder directement à la structure d'un champ. Ces dernières doivent être particulièrement efficaces. De plus elles sont très simples.
Aussi pour avoir les meilleures performances, nous allons les définir **inline** :

```
// Renvoie un pointeur sur le champ qui se trouve à l'offset <Offset> dans le buffer <Buffer>.
function GetPFieldData(Buffer : PChar; Offset : cardinal) : PFieldData; inline;
begin
  result := PFieldData(Buffer + Offset);
end;

// Renvoie true si le champ pointé par <Field> vaut NULL.
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
function IsNull(Field : PFieldData) : boolean; inline; overload;
begin
  result := Field.NullStatus <> 0;
end;

// Renvoie true si le champ qui se trouve à l'offset <Offset> du buffer <Buffer> vaut NULL.
function IsNull(Buffer : PChar; Offset : cardinal) : boolean; inline; overload;
begin
  result := IsNull(GetPFieldData(Buffer, Offset));
end;

// Indique que le champ qui se trouve à l'offset <Offset> dans <Buffer> contient la valeur NULL.
procedure ClearField(Buffer : PChar; Offset : cardinal); inline;
begin
  GetPFieldData(Buffer, Offset).NullStatus := DBNULL_VALUE;
end;
```

III-C - Ouverture/Fermeture

Voyons à présent les méthodes à implémenter pour ouvrir le DataSet.

III-C-1 - InternalOpen

InternalOpen est appelée par **TDataSet** lorsque l'utilisateur veut ouvrir le dataset. Cette méthode doit initialiser le curseur sur la source de données et créer les **TField**. C'est le moment pour effectuer toutes les initialisations du DataSet :

```
// Procède à l'ouverture du DataSet.
procedure TMemoryDataSet.InternalOpen;
var
{$IFDEF TRACE}
  currentTime, Time2 : int64;
{$ENDIF}
  i : integer;
begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'InternalOpen', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    FCursor := -1; // On se positionne au début du DataSet.

    // On va ouvrir le DataSet. C'est le moment pour calculer la taille des buffers.
    CalcRecordSize;

    // Si les TField n'ont pas été créés en Design, on détruit les champs
    // existants pour les recréer à partir de leur définition actuelle.
    if DefaultFields
    then begin
      DestroyFields;
      CreateFields;
    end;

    // A présent, il faut faire mapper les TField sur les champs définis
    // dans TFieldDefs. En effet, les TField ont pu être créés en Design. Dans
    // ce cas, l'ordre des champs dans Fields peut ne pas être le même que dans
    // FieldDefs. Or il faut initialiser la propriété FieldNo de chaque champ
    // pour connaître sa position à l'intérieur des buffers.
{$IFDEF TRACE}
    GenericLogger.TraceBegin(EVENT_START, 'BindFields', Time2, TRACE_LEVEL_VERBOSE);
    try
{$ENDIF}
      BindFields(true);
{$IFDEF TRACE}
    finally
      GenericLogger.TraceEnd(EVENT_END, 'BindFields', Time2, TRACE_LEVEL_VERBOSE);
    end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
{$ENDIF}

    // On construit un index des champs LOB du dataset. On en profite également
    // pour rechercher et initialiser les auto-incréments.
    FBlobFields.Clear;
    FAutoIncFieldOffset := -1; // Pas de champ autoincrémenté par défaut.
    for i := 0 to Fields.Count -1 do
    begin
      if Fields[i].IsBlob
      then FBlobFields.Add(pointer(Fields[i].FieldNo));

      if Fields[i].DataType = ftAutoInc
      then FAutoIncFieldOffset := FFieldInfo[Fields[i].FieldNo-1].Offset;
    end;
    FIsCursorOpen := true; // L'ouverture vient d'être faite.
    ClearPages;

{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'InternalOpen', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```

III-C-2 - InternalClose

Lors de la fermeture du DataSet, on a juste besoin de détruire les éléments qui ont été créés au moment de l'ouverture :

```
// Procède à la fermeture du DataSet.
procedure TMemoryDataSet.InternalClose;
{$IFDEF TRACE}
var
  currentTime : int64;
{$ENDIF}
begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'InternalClose', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    // Annule le binding des TField.
    BindFields(False);

    // Si les champs ont été créés à l'ouverture du DataSet, c'est le moment
    // de les détruire.
    if DefaultFields
    then DestroyFields;

    FIsCursorOpen := false; // Le curseur vient d'être fermé.
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'InternalClose', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-C-3 - IsCursorOpen

Cette fonction est utilisée par certaines méthodes du **TDataSet** pour savoir si le curseur de la source de données a déjà été ouvert. Il suffit de retourner la valeur mémorisée dans **FIsCursorOpen**.

```
// Indique si le curseur a été ouvert.
function TMemoryDataSet.IsCursorOpen: Boolean;
begin
  result := FIsCursorOpen;
end;
```

III-C-4 - InternalInitFieldDefs

Cette méthode est appelée par **TDataSet** lorsque la classe de base veut récupérer la définition des champs **TFieldDefs** de la table, sans procéder pour autant à son ouverture.

Comme dans l'implémentation du **TMemoryDataSet** les **TFieldDef** doivent être créés manuellement en premier lieu, on n'a pas besoin de faire quoi que ce soit dans cette méthode.

On va simplement se contenter de l'instrumenter pour qu'on puisse par la suite étudier le fonctionnement du **TDataSet**.

```
// Doit être surchargée pour initialiser la liste des FieldDefs, sans ouvrir
// le curseur sur la source de données.
procedure TMemoryDataSet.InternalInitFieldDefs;
begin
{$IFDEF TRACE}
  GenericLogger.Trace(EVENT_INFO, 'InternalInitFieldDefs', TRACE_LEVEL_VERBOSE);
{$ENDIF}

  // Comme la structure du dataset est définie manuellement par l'intermédiaire
  // des FieldDefs, on n'a rien à faire. Ils ont déjà été définis.
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-D - Navigation

III-D-1 - GetRecord

On arrive à présent au coeur du DataSet. La méthode **GetRecord** est certainement la plus importante et doit faire l'objet de la plus grande attention.

Elle joue un double rôle : Elle procède à la lecture des buffers sur la source de données et elle effectue la plupart des fonctions de navigation :

```
function TMemoryDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var
  TempBuf : PChar;
{$IFDEF TRACE}
  currentTime : int64;
{$ENDIF}

begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'GetRecord', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    // Premièrement, il faut rechercher l'enregistrement en fonction du mode de déplacement demandé.
    case GetMode of
    gmCurrent: // On veut l'enregistrement courant.
      begin
        dec(FCursor);    // on remonte d'un enregistrement
        FindNextRecord; // et on recherche l'enregistrement suivant
      end;
    gmPrior: // On veut l'enregistrement précédent.
      FindPriorRecord;
    gmNext: // On veut l'enregistrement suivant.
      FindNextRecord;
    end;

    // A présent, il faut vérifier si on se trouve au début ou à la fin du DataSet
    if (FCursor <0) // On se trouve au début, il faut renvoyer BOF.
    then result := grBOF
    else begin
      if FCursor>=integer(FLineIndex.Count) // On se trouve à la fin, on retourne EOF.
      then result := grEOF
      else begin // L'enregistrement existe, on lit le buffer.
        TempBuf := FLineIndex[FCursor];
        move(TempBuf^, Buffer^, RecordSize);
        PLineBuffer(Buffer).BkFlags := bfCurrent; // L'enregistrement est valide
        GetCalcFields(Buffer); // C'est le moment de calculer les champs calculés.
        result := grOK;
      end;
    end;
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'GetRecord', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

GetRecord est appelée pour chaque buffer que **TDataSet** veut lire :

- Le paramètre *Buffer* est un pointeur sur le buffer à remplir.
- Le paramètre *DoCheck* indique s'il faut tester les erreurs lors des déplacements sur la source de données. Comme tout est en mémoire, on ne peut pas avoir d'erreurs. On n'a pas besoin de s'en préoccuper.
- Le paramètre *GetMode* indique le type de navigation demandée, c'est-à-dire le sens de lecture des buffers.

Les valeurs possibles pour *GetMode* sont :

- **gmCurrent :** Il faut relire l'enregistrement en cours, sans effectuer de déplacement.
- **gmPrior :** Il faut lire l'enregistrement précédent et se déplacer sur cet enregistrement.
- **gmNext :** Il faut lire l'enregistrement suivant et se déplacer sur cet enregistrement.

La méthode doit retourner :

- **grOK :** Le buffer a été lu correctement.
- **grBOF :** Le buffer n'a pas pu être lu car on a atteint le début du jeu de données.
- **grEOF :** Le buffer n'a pas pu être lu car on a atteint la fin du jeu de données.

Ce sont les codes de retour de **GetRecord** qui permettent au **TDataSet** de positionner les indicateurs **EOF** et **BOF**.

Lorsqu'un buffer est lu correctement, il faut lui définir le bookmarkflag *bfCurrent* pour indiquer qu'il s'agit d'un buffer ordinaire. Ce dernier sera peut-être modifié par la suite si **TDataSet** se rend compte qu'il s'agit en fait du premier enregistrement ou du dernier.

La classe **TMemoryDataSet** ne gère pas les index. Par conséquent la lecture d'un buffer est directe sur la source de données. Si on voulait gérer les index, **GetRecord** serait la première méthode concernée. En effet, les index permettent entre autre de trier les enregistrements dans un ordre particulier. Cependant, il n'est pas nécessaire de trier réellement les enregistrements en mémoire. On peut se contenter de gérer une liste indiquant l'ordre des enregistrements dans l'index. Le tri se résume alors à trier la liste, sans déplacer les enregistrements, ce qui est beaucoup plus efficace.

Dans ce cas, il suffirait que **GetRecord** lise les enregistrements dans l'ordre de l'index.

GetRecord est également le meilleur endroit où implémenter les filtres. En effet, le principe d'un filtre, c'est que les enregistrements qui ne respectent pas les critères du filtre ne sont pas visibles. Lorsque le filtre est défini, on pourrait calculer un index de tous les enregistrements qui respectent ce filtre, puis maintenir cet index chaque fois qu'un enregistrement est modifié. Cependant ce mode de gestion serait complexe à implémenter. Il faudrait tester tous les enregistrements au moment de la définition du filtre, et rien ne garantit que le filtre soit déterministe.

En revanche, on peut tester le filtre au moment de la lecture d'un enregistrement par **GetRecord**. De cette façon, on ne teste que l'enregistrement courant. Si ce dernier est filtré, on l'ignore sur la source de données et on recherche l'enregistrement suivant.

C'est déjà ce qui a été fait pour **GetRecord**. Les méthodes **FindNextRecord** et **FindPriorRecord** sont codées pour tenir compte de la présence d'un filtre. Je rappelle qu'ici, seuls les filtres définis avec **OnFilterRecord** sont gérés :

```
// Recherche l'enregistrement suivant sur la source de données.
function TMemoryDataSet.FindNextRecord: boolean;
var
  Accept : boolean;
  Buffer : PChar;
{$IFDEF TRACE}
  currentTime : int64;
{$ENDIF}

begin
{$IFDEF TRACE}
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  GenericLogger.TraceBegin(EVENT_START, 'FindNextRecord', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    // On recherche le premier enregistrement suivant qui n'est pas filtré.
    repeat
      // On se déplace sur l'enregistrement suivant.
      inc(FCursor);
      if (FCursor<FLineIndex.Count) // Il faut vérifier si on a atteint la fin.
      then begin
        // L'enregistrement pointé par FCursor existe. Il faut vérifier s'il n'est pas filtré.
        Buffer := FLineIndex[FCursor];
        Accept := FilterRecord(Buffer);   // On regarde s'il n'est pas filtré
      end
      else begin
        Accept := false;
        FCursor := FLineIndex.Count;
      end;
    until Accept or (FCursor >=integer(FLineIndex.Count));
    result := Accept;
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'FindNextRecord', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;

// Recherche l'enregistrement précédent sur la source de données.
function TMemoryDataSet.FindPriorRecord: boolean;
var
  Accept : boolean;
  Buffer : PChar;
{$IFDEF TRACE}
  currentTime : int64;
{$ENDIF}

begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'FindPriorRecord', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    // On recherche le premier enregistrement précédent qui n'est pas filtré.
    repeat
      // On se déplace sur l'enregistrement précédent.
      dec(FCursor);
      if (FCursor>=0) // Il faut vérifier si on a atteint la fin.
      then begin
        // L'enregistrement pointé par FCursor existe. Il faut vérifier s'il n'est pas filtré.
        Buffer := FLineIndex[FCursor];
        Accept := FilterRecord(Buffer);   // On regarde s'il n'est pas filtré
      end
      else Accept := false;
    until Accept or (FCursor<0);
    result := Accept;
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'FindPriorRecord', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```

Les deux méthodes font appel à **FilterRecord** pour tester si un buffer doit être conservé ou filtré.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

L'implémentation de **FilterRecord** est la suivante :

```
// Teste si l'enregistrement <Buffer> est filtré ou doit être conservé.
// La méthode renvoie true si l'enregistrement doit faire partie du jeu
// de résultat.
function TMemoryDataSet.FilterRecord(Buffer: PChar): boolean;
var
  Accept : boolean;
  OldState : TDataSetState;
{$IFDEF TRACE}
  currentTime : int64;
{$ENDIF}

begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'FilterRecord', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    // Par défaut, l'enregistrement est conservé.
    Accept := true;

    if Filtered and Assigned(OnFilterRecord)
    then begin
      // On doit basculer le Dataset dans l'état dsFilter. Dans ce mode, les TField ne lisent plus
      // la ligne courante du DataSet, mais la ligne mémorisée dans FilterBuffer. De cette façon,
      // l'événement OnFilterRecord pourra lire les champs de la ligne à filtrer au lieu de ceux
      // de la ligne courante.
      OldState := SetTempState(dsFilter);
      try
        FFilterBuffer := Buffer;

        // Si les filtres sont activés et qu'un gestionnaire d'événement
        // OnFilterRecord existe, il faut l'appeler.
        OnFilterRecord(Self, Accept);

      finally
        RestoreState(OldState);
        FFilterBuffer := nil;
      end;
    end;
    result := Accept;
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'FilterRecord', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```

Si on veut gérer d'autres façons de filtrer (par exemple gérer la propriété *Filter*), il suffit de modifier l'implémentation de **FilterRecord**.

Une fois la méthode **GetRecord** écrite, nous avons terminé 80% de **TMemoryDataSet**.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-D-2 - GetBookmarkFlag/SetBookmarkFlag

Ces deux méthodes ne sont que des accesseurs sur le champ du buffer qui contient les *bookmarkflag*.

Leur implémentation est très simple :

```
// Lit la valeur du BookmarkFlag du buffer.
function TMemoryDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
  result := PLineBuffer(Buffer).BkFlags
end;

// Ecrit la valeur du BookmarkFlag du buffer.
procedure TMemoryDataSet.SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
begin
  PLineBuffer(Buffer).BkFlags := Value;
end;
```

III-D-3 - GetBookmarkData/SetBookmarkData

Comme pour les bookmarkFlag, **GetBookmarkData** et **SetBookmarkData** ne sont que des accesseurs sur la valeur du bookmark d'un buffer :

```
// Lit la valeur du bookmark de <buffer> et la retourne dans la zone pointée
// par <Data>.
procedure TMemoryDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  cardinal(Data^) := PLineBuffer(Buffer).Bookmark;
end;

// Ecrit la valeur du bookmark <Data> à l'intérieur du buffer <Buffer>
procedure TMemoryDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PLineBuffer(Buffer).Bookmark := Cardinal(Data^);
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-D-4 - InternalGotoBookmark

La méthode **InternalGotoBookmark** est appelée par **TDataSet** pour se positionner sur un bookmark donné.

La gestion des bookmarks dans **TMemoryDataSet** est très simple : Les enregistrements sont numérotés avec un compteur interne. Lorsqu'un enregistrement est ajouté au jeu de données, on lui attribut un nouveau numéro unique. On maintient également un index des bookmarks.

Lorsqu'un enregistrement est supprimé, la ligne est supprimée de l'index des bookmarks.

La recherche d'un bookmark peut donc être assez rapide, on fait une simple recherche dichotomique dans l'index.

InternalGotoBookmark doit repositionner le curseur de la source de données sur le bookmark cherché. **TDataSet** s'occupe du reste des traitements, c'est-à-dire de relire les buffers une fois le curseur repositionné.

```
// Recherche la ligne qui possède le bookmark <Bookmark>.
procedure TMemoryDataSet.InternalGotoBookmark(Bookmark: Pointer);
var
  TestCursor : integer;
{$IFDEF TRACE}
  currentTime : int64;
{$ENDIF}

begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'InternalGotoBookmark', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    TestCursor := FindBookmark(Bookmark);

    if TestCursor <0
    then raise MemoryDataSetException.Create('Enregistrement non trouvé !');

    FCursor := TestCursor;
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'InternalGotoBookmark', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Avec pour **FindBookmark** :

```
// Recherche le numéro de la ligne qui possède le bookmark <Bookmark>.
// Si la ligne n'existe pas, la fonction renvoie -1
function TMemoryDataSet.FindBookmark(Bookmark: pointer): integer;
var
  bk : cardinal;
  TestCursor : integer;
  bkTest : cardinal;
  borneMin, borneMax : cardinal;
begin
  bk := Cardinal(Bookmark^);

  // Les lignes sont triées dans l'ordre des bookmark dans FBookmarkIndex.
  // On effectue donc une recherche dichotomique dans la liste.

  // On commence par utiliser la valeur du bookmark comme point de départ pour
  // la recherche.
  borneMin := 0;
  borneMax := FBookmarkIndex.Count -1;

  while (borneMax-borneMin)>3 do
  begin
    TestCursor := (borneMin + borneMax) div 2;
    bkTest := cardinal(FBookmarkIndex[TestCursor]);
    if bkTest>=bk
    then borneMax := TestCursor
    else borneMin := TestCursor;
  end;

  // A présent, on a restreint l'intervale de recherche, on termine par une
  // recherche séquentielle.
  result := -1; // Par défaut, le bookmark n'a pas été trouvé.
  for TestCursor := borneMin to borneMax do
  begin
    if cardinal(FBookmarkIndex[TestCursor]) = bk
    then begin // Le bookmark a été trouvé.
      result := TestCursor;
      break;
    end;
  end;
end;
```

III-D-5 - BookmarkValid

BookmarkValid permet de savoir si un bookmark est toujours valide et utilisable. La méthode permet de tester si un **Gotobookmark** va réussir ou provoquer une exception.

Une façon simple de l'implémenter consiste tout simplement à regarder si on trouve l'enregistrement référencé dans le jeu de données :

```
// Indique si le bookmark <Bookmark> référence un enregistrement du jeu de
// données.
function TMemoryDataSet.BookmarkValid(Bookmark: TBookmark): Boolean;
begin
  result := FindBookmark(Bookmark)<>-1;
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-D-6 - InternalSetToRecord

InternalSetToRecord fait la même chose que **InternalGotobookark**. La seule différence c'est que cette méthode reçoit un buffer en paramètre au lieu d'un bookmark. Cependant, il suffit de lire le bookmark du buffer pour l'implémenter avec **InternalGotoBookmark** :

```
// Positionne le jeu de données sur l'enregistrement <Buffer>.
procedure TMemoryDataSet.InternalSetToRecord(Buffer: PChar);
{$IFDEF TRACE}
var
  currentTime : int64;
{$ENDIF}
begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'InternalSetToRecord', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    InternalGotoBookmark(@(PLineBuffer(Buffer).Bookmark));
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'InternalSetToRecord', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```

III-D-7 - InternalFirst

InternalFirst est appelée par **TDataSet** pour se positionner sur le premier enregistrement, lorsque la méthode **First** est appelée. Tout ce qu'on doit faire, c'est positionner le curseur au début. **TDataSet** s'occupe ensuite du reste, c'est-à-dire relire les buffers par rapport à la nouvelle position.

```
// Positionne le curseur sur le premier enregistrement du jeu de données.
procedure TMemoryDataSet.InternalFirst;
begin
{$IFDEF TRACE}
  GenericLogger.Trace(EVENT_INFO, 'InternalFirst', TRACE_LEVEL_VERBOSE);
{$ENDIF}
  FCursor := -1; // On se positionne au début.
end;
```

III-D-8 - InternalLast

InternalLast fonctionne de la même façon que **InternalFirst**. Cette fois, on doit se positionner sur le dernier enregistrement :

```
// Positionne le curseur sur le dernier enregistrement du jeu de données.
procedure TMemoryDataSet.InternalLast;
begin
{$IFDEF TRACE}
  GenericLogger.Trace(EVENT_INFO, 'InternalLast', TRACE_LEVEL_VERBOSE);
{$ENDIF}
  FCursor := FLineIndex.Count; // On se positionne à la fin.
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-D-9 - GetRecNo/SetRecNo

On arrive à présent au dernier mode de déplacement dans le **TDataSet** : La propriété **RecNo**. Cette dernière permet de manipuler le DataSet comme un tableau, en numérotant les lignes et en se positionnant directement sur une ligne donnée.

GetRecNo est très simple à implémenter. Il suffit de retourner la position du curseur.

En revanche **SetRecNo** est un peu plus complexe. En effet, toutes les méthodes de navigation précédentes étaient des méthodes bas niveau, appelée par d'autres plus évoluées de **TDataSet**. Ces dernières s'occupaient de générer les événements liés au déplacement du curseur. Elles s'occupaient également de relire les buffers du DataSet.

Cette fois pour **SetRecNo**, nous allons devoir tout faire nous même :

```
// positionne le DataSet sur l'enregistrement <Value>.
procedure TMemoryDataSet.SetRecNo(Value: Integer);
{$IFDEF TRACE}
var
  currentTime : int64;
{$ENDIF}
begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'SetRecNo', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    CheckBrowseMode; // On s'assure que le dataset est ouvert et déplaçable.
    dec(Value); // Le premier enregistrement porte le numéro 1
    if Value<>FCursor // On ne fait le traitement que si on se déplace vraiment
    then begin
      // Déclanche l'événement BeforeScroll.
      DoBeforeScroll;

      // Positionne le curseur à la position demandée.
      FCursor := Value;

      // Provoque la relecture des buffers du DataSet.
      Resync([rmCenter]);

      // Déclenche l'événement AfterScroll.
      DoAfterScroll;
    end;
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'SetRecNo', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```

On remarque l'appel à la méthode **Resync**. Cette dernière effectue la relecture des buffers du DataSet et permet de redéfinir les flags *EOF* et *BOF*.

Elle doit être appelée chaque fois que le curseur du jeu de données est déplacé par une méthode de haut niveau. Par exemple, si on voulait implémenter les fonctions de recherche telles que **Locate**, **FindKey** et **GotoKey** il faudrait appeler **Resync** après chaque recherche pour que le DataSet se resynchronise avec la nouvelle position du curseur.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-E - Lecture/Ecriture des champs

III-E-1 - GetFieldData

GetFieldData est appelée chaque fois qu'un composant **TField** veut lire les données d'un champ.

Elle existe avec trois *overload* :

```
function GetFieldData(Field: TField;
   Buffer: Pointer): Boolean; overload; override;

function GetFieldData(FieldNo: Integer;
  Buffer: Pointer): Boolean; overload; override;

function GetFieldData(Field: TField;
  Buffer: Pointer; NativeFormat: Boolean): Boolean; overload; override;
```

A chaque fois, le but est de remplir la zone pointée par *Buffer* avec la valeur d'un champ.

La fonction doit renvoyer **true** si le champ a été lu correctement. C'est-à-dire si le champ ne vaut pas NULL, et **false** si le champ est à NULL.

Il se peut que **GetFieldData** soit appelée uniquement pour savoir si le champ est à NULL. Dans ce cas, *Buffer* vaut **nil**. Il faut alors renvoyer *true* ou *false* en fonction du champ et bien sûr, on ne peut pas renvoyer de valeur dans *buffer*.

Pour **TMemoryDataSet**, nous utilisons la propriété **FieldNo** du **TField** pour identifier le champ.

La troisième version de **GetFieldData** possède un paramètre supplémentaire : **NativeFormat**. Les valeurs des champs peuvent être stockées à l'intérieur du buffer d'une ligne dans un format différent de celui attendu par le **TField** pour le lire. Par exemple, pour les champs **Date**, le **TDateTimeField** retourne toujours un **TDateTime**. Cependant, si on a un champ **ftDate** ou **ftTime**, le BDE stocke le champ sous la forme d'un entier.

La propriété **NativeFormat** indique s'il faut effectuer la conversion entre le type natif et le type de stockage.

La classe **TDataSet** possède déjà une implémentation de la troisième version, basée sur les deux autres.

On peut donc se contenter de surcharger les deux premières :

```
// Retourne dans Buffer les données du champ <FieldNo>. Si le champ est à
// NULL, la fonction retourne false.
function TMemoryDataSet.GetFieldData(FieldNo: Integer;
  Buffer: Pointer): Boolean;
var
  LineBuf : PChar;
  FieldOffset : cardinal;
  FieldSize : cardinal;
begin
  CheckActive; // Le DataSet doit être ouvert.

  LineBuf := GetActiveBuffer; // On recherche le buffer de la ligne à lire.

  FieldOffset := FFieldInfo[FieldNo-1].Offset;
  FieldSize := FFieldInfo[FieldNo-1].Size;

  // On commence par regarder si la valeur du champ est définie
  result := Assigned(LineBuf) and not IsNull(LineBuf, FieldOffset);

  // Si Buffer est défini, il faut également lire la valeur
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  if result and Assigned(Buffer)
  then move(GetPFieldData(LineBuf, FieldOffset).Data, Buffer^, FieldSize);

  case FieldDefs[FieldNo-1].DataType of
  ftTimeStamp:
    TSQLTimeStamp(Buffer^).Fractions := TSQLTimeStamp(Buffer^).Fractions div 1000000;
  end;
end;
```

GetFieldData fait appel à **GetActiveBuffer** afin de sélectionner le buffer à lire en fonction de l'état dans lequel se trouve le DataSet :

```
// Retourne le Buffer à Lire/Ecrire en fonction de l'état du DataSet.
function TMemoryDataSet.GetActiveBuffer: PChar;
begin
  case State of
    dsBlockRead, dsSetKey, dsCalcFields: // Non géré.
      result := nil;
    dsBrowse: // Cas normal, on renvoie ActiveBuffer sauf si le dataset est vide
      if IsEmpty
      then result := nil
      else result := ActiveBuffer;
    dsEdit, dsInsert: // En Edit ou Insert on utilise toujours ActiveBuffer
      result := ActiveBuffer;
    dsFilter: // On est en train de filtrer les enregistrements, il faut
              // utiliser l'enregistrement FFilterBuffer.
      result := FFilterBuffer;
    dsNewValue: // On veut lire/ecrire la nouvelle valeur d'une ligne en cours
                // de modification. On retourne ActiveBuffer.
      result := ActiveBuffer;
    dsOldValue: // On veut lire l'ancienne valeure d'une ligne en cours de
                // modification. On retourne FOldBuffer.
      result := FOldBuffer;
  else
    result := nil;
  end;
end;
```

On remarquera au passage que **GetFieldData** ne tient pas compte des champs LOB qui sont gérés différemment. C'est tout simplement parce que les champs LOB sont toujours lus par l'intermédiaire de **CreateBlobStream**.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-E-2 - SetFieldData

SetFieldData est légèrement différente de **GetFieldData** dans la mesure où il faut vérifier que le DataSet est modifiable et qu'il se trouve dans un état autorisant les modifications :

```
// Définit la valeur d'un champ.
procedure TMemoryDataSet.SetFieldData(Field: TField; Buffer: Pointer);
var
  LineBuf : PCHar;
  FieldOffset : cardinal;
  FieldSize : cardinal;
  FieldData : PFieldData;
begin
  CheckActive; // On vérifie que le DataSet est ouvert.

  // On vérifie que le dataset se trouve dans un état autorisant les écritures.
  if not (State in dsWriteModes)
  then DatabaseError(SNotEditing, Self);

  // On rechercher le buffer de la ligne à modifier.
  LineBuf := GetActiveBuffer;

  if Field.FieldNo > 0 // Les champs calculés ne sont pas supportés !
  then begin
    // Si le champ est en lecture seule, on ne peut pas le modifier
    if Field.ReadOnly
    then DatabaseErrorFmt(SFieldReadOnly, [Field.FieldName]);

    // Déclenche l'événement on Validate du TField.
    Field.Validate(Buffer);

    FieldOffset := FFieldInfo[Field.FieldNo-1].Offset;
    FieldSize := FFieldInfo[Field.FieldNo-1].Size;

    if Assigned(Buffer) // On définit une valeur pour le champ
    then begin
      // On commence par obtenir le pointeur sur les données du champ
      FieldData := GetPFieldData(LineBuf, FieldOffset);

      // Le champ ne vaut pas NULL.
      FieldData.NullStatus := DBNOTNULL_VALUE;

      move(Buffer^, FieldData.Data, FieldSize);
      case Field.DataType of
      ftTimeStamp:
        TSQLTimeStamp((@FieldData.Data[0])^).Fractions :=
                TSQLTimeStamp((@FieldData.Data[0])^).Fractions * 1000000;
      end;
    end
    else ClearField(LineBuf, FieldOffset);

    // Enfin, il faut déclencher les événements
    if not (State in [dsCalcFields, dsFilter, dsNewValue])
    then DataEvent(deFieldChange, Longint(Field));
  end;
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-F - Gestion des LOB

Les champs LOB sont toujours manipulés avec la fonction **CreateBlobStream**. Même lorsqu'on veut lire directement la valeur d'un LOB avec **TField.AsString** par exemple, en interne la VCL appelle implicitement **CreateBlobStream** pour accéder au champ LOB.

Nous devons créer une classe dérivée de **TStream** permettant de manipuler les champs LOB. Ce sera la classe **TMemoryBlobStream**.

Rappelons tout d'abord que les LOB sont gérés indirectement : **TMemoryDataSet** conserve la liste de tous les LOB alloués. Les champs LOB ne contiennent pas directement les données, mais seulement un numéro de LOB.

Ainsi, **TMemoryDataSet** doit également gérer une liste des LOB. C'est ce qui est fait avec les méthodes **AllocBlob** et **ReleaseBlob** :

```
// Réserve une position pour la valeur d'un LOB et retourne l'emplacement réservé.
function TMemoryDataSet.AllocBlob: cardinal;
begin
  // On commence par regarder dans FEmptyBlob pour voir si un LOB préalablement
  // défini ne peut pas être recyclé.
  if FEmptyBlob.Count > 0
  then begin
    // Il existe un élément recyclable. On renvoie la dernière position libérée.
    result := cardinal(FEmptyBlob[FEmptyBlob.Count-1]);
    FEmptyBlob.Delete(FEmptyBlob.Count-1);
  end
  else begin
    // Il n'existe pas d'élément recyclable. On crée un nouvel emplacement.
    result := FBlobList.Add(nil);
  end;
end;
```

FBlobList est un simple **TList** qui contient tous les pointeurs sur les LOB alloués. Lorsqu'on crée un nouveau LOB il suffit d'ajouter le pointeur sur ces données dans **FBlobList**.

Cependant, lorsque le LOB est détruit, on ne peut pas supprimer le pointeur de la liste. Ca décalerait les autres LOB. Or les champs LOB contiennent justement la position du LOB dans **FBlobList**. Lorsqu'on libère un LOB, il faut conserver l'emplacement vide dans **FBlobList**.

Cet emplacement pourra être recyclé lors de l'allocation suivante. Pour retrouver rapidement ces emplacements, **TMemoryDataSet** maintient une liste des LOB à recycler dans **FEmptyBlob**.

Ainsi pour allouer un LOB, on commence par chercher un emplacement à recycler. S'il n'y en a pas, on ajoute un élément à **FBlobList**. Sinon on recycle un emplacement existant.

Une fois l'emplacement réservé, ce sera l'écriture du LOB qui allouera le bloc de données et définira le pointeur.

Lorsqu'on libère un LOB, il faut libérer la mémoire utilisée et ajouter son emplacement à la liste des LOB à recycler :

```
// Libère un LOB préalablement alloué avec AllocBlob
procedure TMemoryDataSet.ReleaseBlob(idx: cardinal);
begin
  if integer(idx) < FBlobList.Count
  then begin
    Freemem(FBlobList[idx]); // Libère la mémoire utilisée par le LOB
    FBlobList[idx] := nil;  // Le pointeur ne référence plus rien.
    FEmptyBlob.Add(pointer(idx)); // Ajoute l'emplacement aux LOB à recycler.
  end;
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-F-1 - TMemoryBlobStream

La classe **TMemoryBlobStream** sert à lire et écrire le contenu d'un champ LOB. Le plus souvent, les champs LOB servent à manipuler des données de petite taille. En effet, pour de gros volumes, on préfère souvent utiliser des fichiers externes à la base.

De plus, un DataSet en mémoire doit stocker l'intégralité des données en mémoire vive. Il n'est pas adapté à la gestion de LOB très important.

Aussi, nous allons restreindre la gestion des LOB aux champs de taille suffisamment petite pour qu'ils puissent être lus et écrit en une seule fois.

Autrement dit, la classe **TMemoryBlobStream** ne pourra être utilisée que pour lire/écrire le LOB avec un seul appel à **Read**/**Write**. La méthode **Seek** ne sera pas supportée.

TMemoryBlobStream sera déclarée privée pour éviter que l'utilisateur ne soit tenté de l'instancier directement. En effet, elle est plutôt prévue pour que le LOB soit manipulé automatiquement de façon implicite par **TMemoryDataSet** et les **TField**.

Sa déclaration est la suivante :

```
// TMemoryBlobStream permet de manipuler les données d'un champ LOB comme
// s'il s'agissait d'un TStream.
TMemoryBlobStream = class(TStream)
private
  FDataSet : TMemoryDataSet; // DataSet à lire/écrite
  FBlobData : pointer;       // pointeur sur les données du LOB à lire.
  FFieldData : pointer;      // pointeur sur le champ LOB manipulé
  FieldOffset : cardinal;    // offset du champ à l'intérieur d'un Buffer.
  FBlobIdx : cardinal;       // Numéro du LOB à l'intérieur du dataset.
  FSize : cardinal;          // Longueur du LOB.
  FMode : TBlobStreamMode;   // Mode d'accès demandé.
  FLineBuf : PChar;          // Buffer de la ligne contenant le LOB.
public
  // Crée et initialise une nouvelle instance de la classe TMemoryBlobStream
  constructor Create(Field : TField; Mode: TBlobStreamMode);

  // Detruit l'instance et libère la mémoire utilisée.
  destructor Destroy; override;

  // Lit les données depuis le LOB.
  function Read(var Buffer; Count: Longint): Longint; override;

  // Ecrit les données dans le LOB.
  function Write(const Buffer; Count: Longint): Longint; override;

  // Déplace le curseur à l'intérieur du LOB.
  function Seek(Offset: Longint; Origin: Word): Longint; override;
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Le constructeur effectue la plupart des initialisations. Il s'occupe en particulier d'obtenir un nouveau LOB pour accéder au champ en écriture :

```
// Crée et initialise une nouvelle instance de la classe TMemoryBlobStream
constructor TMemoryBlobStream.Create(Field: TField; Mode: TBlobStreamMode);
begin
  // On commence par mémoriser les paramètres de création.
  FDataSet := TMemoryDataSet(Field.DataSet);
  FMode := Mode;

  // On lit les informations du champ pour lequel on veut accéder au LOB.
  FLineBuf := FDataSet.GetActiveBuffer;
  FieldOffset := FDataSet.FFieldInfo[Field.FieldNo-1].Offset;
  FFieldData := GetPFieldData(FLineBuf, FieldOffset);
  FBlobIdx := cardinal(PFieldData(FFieldData).Data[0]);

  // Maintenant, le mode d'accès demandé determine la suite des traitements
  case Mode of
  bmRead: // Lecture seul.
    begin
      // On doit juste retourner un flux en lecture seule. Dans ce cas, il suffit
      // d'initialiser les paramètres de lecture.
      if IsNull(PFieldData(FFieldData))
      then FSize := 0 // Pour les champs NULL, on doit retourner un flux de
                      // longueur 0.
      else begin
        // Le champ existe, on initialise la taille et le buffer à lire.
        // Les LOB sont stockés indirectement. Le champ contient le numéro du
        // tampon dans FBlobList.
        FBlobData := FDataSet.FBlobList[FBlobIdx];

        // La taille du LOB est donnée par la longueur courante du champ.
        FSize := PFieldData(FFieldData).LengthValue;
      end;
    end;
  bmWrite, bmReadWrite: // Le mode bmReadWrite n'est pas supporté. Il est traité
    begin               // Comme un write et tronque le champ.
      // On doit pouvoir écrire le champ. Ca signifie que si le champ est à NULL
      // il faut créer un nouveau tampon de données.
      if IsNull(PFieldData(FFieldData))
      then begin
        FBlobData := nil; // Pour l'instant le buffer n'est pas défini.
        FBlobIdx := FDataSet.AllocBlob; // On demande au dataset de réserver
                                        // un emplacement pour le LOB.

        // On définit la valeur du champ sur le nouveau LOB.
        PFieldData(FFieldData).NullStatus := DBNOTNULL_VALUE;
        PFieldData(FFieldData).LengthValue := 0;
        move(FBlobIdx, PFieldData(FFieldData).Data[0], sizeof(FBlobIdx));
      end
      else begin
        // Le champ existe, on initialise la taille et le buffer à lire.
        // Les LOB sont stockés indirectement. Le champ contient le numéro du
        // tampon dans FBlobList.

        // On vide le LOB qui va être réécrit.
        Freemem(FDataSet.FBlobList[FBlobIdx]);
        FDataSet.FBlobList[FBlobIdx] := nil;
        FBlobData := nil;
        FSize := 0;
        PFieldData(FFieldData).LengthValue := 0;
      end;
    end;
  end;
end;
```

En lecture, tout est relativement simple. **TMemoryBlobStream** se contente d'initialiser la taille du LOB et de retrouver le pointeur sur les données. Si le champ est à NULL, il faut considérer que le flux possède une longueur de 0.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

En écriture on peut avoir deux cas de figure :

- Si le champ était à NULL, il faut allouer un nouveau LOB auprès du Dataset. Le pointeur sur les données sera défini au moment de l'écriture.
- Si le champ était déjà alloué, il faut le tronquer. Dans ce cas, on libère la mémoire qui était utilisée. Le bloc mémoire sera réalloué au moment de l'écriture.

La lecture du LOB est alors relativement simple :

```
// Lit les données depuis le LOB.
function TMemoryBlobStream.Read(var Buffer; Count: Integer): Longint;
begin
  // On limite la lecture aux données disponibles
  if Count > integer(FSize)
  then Count := FSize;

  if (FBlobData<>nil)
  then begin
    move(FBlobData^, Buffer, Count);
    result := count;
  end
  else result := 0;
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

L'écriture est un petit peu plus complexe :

```
// Ecrit les données dans le LOB.
function TMemoryBlobStream.Write(const Buffer; Count: Integer): Longint;
begin
  // On alloue un bloc mémoire pour contenir les données à écrire.
  GetMem(FBlobData, Count);
  FDataSet.FBlobList[FBlobIdx] := FBlobData;

  move(Buffer, FBlobData^, Count); // recopie des données du LOB

  // On définit la nouvelle taille du champ.
  FSize := Count;
  PFieldData(FFieldData).LengthValue := FSize;

  result := count;
end;
```

Enfin, lorsqu'on ferme le flux, il faut regarder si une valeur a été définie. Si ce n'est pas le cas, on marque le champ à NULL :

```
// Detruit l'instance et libère la mémoire utilisée.
destructor TMemoryBlobStream.Destroy;
begin
  // Si on a définit un LOB de longueur 0, le champ doit passer à NULL.
  if (FMode = bmWrite) or (FMode = bmReadWrite)
  then begin
    if FSize = 0
    then begin
      ClearField(FLineBuf, FieldOffset);
      FDataSet.ReleaseBlob(FBlobIdx);
    end;
  end;
  inherited Destroy;
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-F-2 - CreateBlobStream

TMemoryBlobStream est privée. Elle ne peut pas être instanciée directement. Pour accéder à un LOB, il faut passer par la méthode **CreateBlobStream**. C'est elle qui crée l'instance de **TMemoryBlobStream**.

Comme le code pour manipuler le champ se trouve dans **TMemoryBlobStream**, l'implémentation de **CreateBlobStream** est très simple :

```
function TMemoryDataSet.CreateBlobStream(Field: TField;
  Mode: TBlobStreamMode): TStream;
begin
  result := TMemoryBlobStream.Create(Field, Mode);
end;
```

III-G - Modification des données

III-G-1 - GetCanModify

Cette méthode doit être surchargée pour indiquer si le dataset peut être modifié. Par défaut, la classe **TDataSet** considère que l'instance est modifiable.

On va l'implémenter simplement en indiquant que **TMemoryDataSet** est modifiable lorsque le dataset est ouvert :

```
// Indique si le Dataset peut être modifié.
function TMemoryDataSet.GetCanModify: boolean;
begin
  result := FIsCursorOpen;
end;
```

Si on voulait ajouter une propriété **ReadOnly**, il faudrait modifier cette méthode pour interdire les modifications.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-G-2 - InternalEdit

InternalEdit est appelée par **TDataSet** lorsque ce dernier doit basculer en mode modification.

Lorsqu'on passe en édition, nous devons définir le pointeur **FOldBuffer** pour que l'on puisse lire la valeur des champs avant modification avec *Field.OldValue*.

De plus, si l'utilisateur modifie un champ LOB, il va d'une part modifier le buffer en cours, et d'autre part modifier les LOB directement dans le dataset. On risque alors de perdre l'ancienne valeur des LOB.

Pour éviter ce problème, nous allons dupliquer les LOB dans le buffer d'édition. De cette façon, si on modifie un LOB, c'est la copie qui sera modifiée et non pas l'original.

Au moment de la validation des données, on pourra détruire les LOB originaux.

```
// Réagit au passage en mode Edition du Dataset.
procedure TMemoryDataSet.InternalEdit;
begin
{$IFDEF TRACE}
  GenericLogger.Trace(EVENT_INFO, 'InternalEdit', TRACE_LEVEL_VERBOSE);
{$ENDIF}
  // On garde une référence sur la ligne en cours de modification.
  FOldBuffer := FLineIndex[FCursor];

  // Ensuite, on effectue une copie des LOB de la ligne en cours. De cette façon
  // en cas de modification, on ne touche pas au LOB avant modification,
  // seule la copie est modifiée.
  CopyBufferBlob(GetActiveBuffer);

  inherited InternalEdit;
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Avec :

```
// Remplace les LOB du buffer <Buffer> par une copie de leur valeur.
procedure TMemoryDataSet.CopyBufferBlob(Buffer: PChar);
var
  FieldNo : cardinal;
  Offset : Cardinal;
  Data : PFieldData;
  Valeur, ValeurDest,
  size : cardinal;
  ptr : pointer;
  i : integer;
begin
  for i := 0 to FBLobList.Count -1 do
  begin
    FieldNo := cardinal(FBLobList[i]);
    Offset := FFieldInfo[FieldNo-1].Offset;
    Data := GetPFieldData(Buffer, Offset);
    if not IsNull(Data)
    then begin
      // On lit la valeur et la taille du LOB à copier.
      Valeur := cardinal(Data.Data[0]);
      size := Data.LengthValue;

      // On alloue un nouveau LOB pour la copie.
      ValeurDest := AllocBlob;

      // Copie de la valeur.
      GetMem(ptr, Size);
      move(FBlobList[Valeur]^, ptr^, size);
      FBlobList[ValeurDest] := ptr;

      // On affecte la nouvelle valeur au champ destination.
      move(ValeurDest, Data.Data[0], sizeof(ValeurDest));
    end;
  end;
end;
```

III-G-3 - InternalCancel

Lorsque l'utilisateur annule les modifications en cours, **TDataSet** va automatiquement relire le buffer depuis la source de données et ainsi annuler les modifications.

Cependant, comme on a dupliqué les LOB au moment de l'édition nous devons détruire les copies des LOB qui ont été créées. Dans la pratique, on détruit les LOB du buffer d'édition.

```
// Réagit à l'annulation des modifications.
procedure TMemoryDataSet.InternalCancel;
begin
{$IFDEF TRACE}
  GenericLogger.Trace(EVENT_INFO, 'InternalCancel', TRACE_LEVEL_VERBOSE);
{$ENDIF}
  // Le TDataSet traite automatiquement l'annulation des modifications :
  // Les buffers vont être relus sur la source de données.
  // Cependant, comme les LOB ont été dupliqués au moment du passage en mode
  // edition, il faut les détruire.
  ReleaseBufferBlob(GetActiveBuffer);
  FOldBuffer := nil;
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-G-4 - InternalPost

InternalPost est appelée par **TDataSet** pour traiter le **Post**. La méthode peut être appelée dans deux cas de figures :

- Après un **Edit** : Dans ce cas la ligne qui a été modifiée existe déjà dans la source de données. Il suffit de recopier le buffer.
- Après un **Insert** ou un **Append** : En fait, l'*insert* et l'*append* sont traités de la même façon. On ajoute la ligne à la fin de la source de données. Là où les choses se compliquent c'est si la ligne possède un champ auto incrémenté. En effet pour le dataset en mémoire, on doit gérer l'autoincrément et lui affecter une nouvelle valeur.

Le code de la méthode **InternalPost** est le suivant :

```
// Valide les modifications de l'enregistrement.
procedure TMemoryDataSet.InternalPost;
var
{$IFDEF TRACE}
  currentTime : int64;
{$ENDIF}
  Buffer : Pointer;
  FieldData : PFieldData;
begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'InternalPost', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    Buffer := GetActiveBuffer;
    if State = dsEdit
    then begin
      // Commençons par le cas simple. La ligne courante est en cours de modification. Il faut
      // écrire le buffer dans la source de données. Cela signifie que la ligne avant modification
      // va être écrasée. Si elle contenait des LOB, il faut également détruire ces LOB.
      ReleaseBufferBlob(FOldBuffer);
      move(Buffer^, FLineIndex[FCursor]^, RecordSize);
    end
    else begin
      // On est en insertion (dsInsert), il faut ajouter la ligne à la source de données.
      // Si le dataset contient des champs auto incrémentés, il faut gérer l'autoincrément.
      if FAutoIncFieldOffset<>-1
      then begin
        // On a un champ auto incrémenté, il faut définir sa valeur.
        FieldData := GetPFieldData(Buffer, FAutoIncFieldOffset);
        inc(FAutoIncValue);
        FieldData.NullStatus := DBNOTNULL_VALUE;
        move(FAutoIncValue, FieldData.Data[0], sizeof(FAutoIncValue));
      end;
      FCursor := AddLine(Buffer);
    end;
    FOldBuffer := nil; // Le buffer n'est plus valide.
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'InternalPost', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-G-5 - InternalDelete

La méthode est appelée pour supprimer la ligne active, en réponse à un appel à **Delete**.

On doit donc supprimer la ligne en cours de la source de données. Comme on gère une liste des lignes en plus des tableaux de données, la suppression d'une ligne peut être assez simple. En effet, il suffit de supprimer la ligne des listes qui la référence. Il n'est pas nécessaire de décaler également les lignes dans les pages de stockage. On mémorise juste l'emplacement de la ligne qui a été supprimée :

```
// Supprime la ligne en cours.
procedure TMemoryDataSet.InternalDelete;
{$IFDEF TRACE}
var
  currentTime : int64;
{$ENDIF}
begin
{$IFDEF TRACE}
  GenericLogger.TraceBegin(EVENT_START, 'InternalDelete', currentTime, TRACE_LEVEL_VERBOSE);
  try
{$ENDIF}
    // On commence par détruire les LOB de la ligne à supprimer.
    ReleaseBufferBlob(FLineIndex[FCursor]);

    // La ligne va être supprimée, on ajoute l'emplacement à la liste des lignes à recycler.
    FEmptyLine.Add(FLineIndexCursor[FCursor]);

    FLineIndex.Delete(FCursor); // La ligne n'existe plus.
    FLineIndexCursor.Delete(FCursor); // La ligne n'existe plus.
    FBookmarkIndex.Delete(FCursor); // La ligne n'existe plus.
{$IFDEF TRACE}
  finally
    GenericLogger.TraceEnd(EVENT_END, 'InternalDelete', currentTime, TRACE_LEVEL_VERBOSE);
  end;
{$ENDIF}
end;
```

Conclusion

A présent, nous venons d'obtenir une classe **TMemoryDataSet** qui se comporte comme n'importe quel DataSet et qui travaille uniquement en mémoire, sans connexion avec une base de données.

Il ne reste plus qu'à la tester et à évaluer ses performances.

La classe a été instrumentée avec ETW. Chaque opération importante génère des traces au niveau **VERBOSE** uniquement. Nous utiliserons ces traces pour examiner le comportement du **TDataSet** lorsqu'on le manipule. En revanche on trace l'exécution de méthodes très courtes et rapide. Même si la gestion de la trace est elle-même très rapide, les méthodes tracées le sont tout autant. Au final, la trace peut diviser les performances par deux.

De plus, cette trace ne présente pas d'autres intérêts que d'analyser le fonctionnement du **TDataSet**. Elle est trop riche pour être exploitable dans un contexte de production. C'est pourquoi, l'instrumentation du code a été placée sous directive de compilation.

Par la suite, **TMemoryDataSet** servira à s'interfacer avec OLEDB. Nous définirons alors d'autres traces avec **SQLLogger** qui auront un rôle plus fonctionnel.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

IV - Quelques cas d'utilisations

IV-A - Scénario 1 : Ouverture/Fermeture du DataSet

Voyons à présent comment se déroule l'ouverture du **TMemoryDataSet**. La classe a été instrumentée pour générer une trace chaque fois qu'une méthode importante est appelée.

Nous allons donc examiner cette trace pour voir ce qui se passe. On exécute le code suivant :

```
FMemoryDataSet := TMemoryDataSet.Create(nil);
FMemoryDataSet.FieldDefs.Add('ftString', ftString, 20);
FMemoryDataSet.FieldDefs.Add('ftSmallint', ftSmallint);
FMemoryDataSet.FieldDefs.Add('ftInteger', ftInteger);
FMemoryDataSet.FieldDefs.Add('ftWord', ftWord);
FMemoryDataSet.FieldDefs.Add('ftBoolean', ftBoolean);
FMemoryDataSet.FieldDefs.Add('ftFloat', ftFloat);
FMemoryDataSet.FieldDefs.Add('ftCurrency', ftCurrency);
FMemoryDataSet.FieldDefs.Add('ftBCD', ftBCD, 2);
FMemoryDataSet.FieldDefs.Add('ftDate', ftDate);
FMemoryDataSet.FieldDefs.Add('ftTime', ftTime);
FMemoryDataSet.FieldDefs.Add('ftDateTime', ftDateTime);
FMemoryDataSet.FieldDefs.Add('ftAutoInc', ftAutoInc);
FMemoryDataSet.FieldDefs.Add('ftFixedChar', ftFixedChar, 10);
FMemoryDataSet.FieldDefs.Add('ftWideString', ftWideString, 20);
FMemoryDataSet.FieldDefs.Add('ftLargeint', ftLargeint);
FMemoryDataSet.FieldDefs.Add('ftGuid', ftGuid, 38);
FMemoryDataSet.FieldDefs.Add('ftFMTBcd', ftFMTBcd, 4);
FMemoryDataSet.FieldDefs.Add('ftBytes', ftBytes, 40);
FMemoryDataSet.FieldDefs.Add('ftVarBytes', ftVarBytes, 40);
FMemoryDataSet.FieldDefs.Add('ftOraBlob', ftOraBlob);
FMemoryDataSet.FieldDefs.Add('ftOraClob', ftOraClob);
FMemoryDataSet.FieldDefs.Add('ftBlob', ftBlob);
FMemoryDataSet.FieldDefs.Add('ftMemo', ftMemo);
FMemoryDataSet.FieldDefs.Add('ftGraphic', ftGraphic);
FMemoryDataSet.FieldDefs.Add('ftFmtMemo', ftFmtMemo);
FMemoryDataSet.FieldDefs.Add('ftWideMemo', ftWideMemo);

FMemoryDataSet.Open;
FMemoryDataSet.Close;
```

On se contente donc de définir un champ par type de données, puis on ouvre le DataSet et on le ferme immédiatement.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

La trace générée lors de l'ouverture et la fermeture est la suivante :

Delta	Type	User
0,0000	START	InternalOpen
0,0073	INFO	CalcRecordSize=566
0,0013	START	DestroyFields
0,0031	END	DestroyFields
0,0011	**START**	**CreateFields**
1,2985	**END**	**CreateFields**
0,0014	**START**	**BindFields**
2,1251	**END**	**BindFields**
0,0043	END	InternalOpen
0,0021	INFO	AllocRecordBuffer
0,0036	INFO	InternalInitRecord
0,0011	INFO	AllocRecordBuffer
0,0009	INFO	InternalInitRecord
0,0015	START	GetRecord
0,0016	START	FindNextRecord
0,0021	END	FindNextRecord
0,0017	END	GetRecord
0,0013	START	GetRecord
0,0015	START	FindPriorRecord
0,0021	END	FindPriorRecord
0,0017	END	GetRecord
0,0121	INFO	FreeRecordBuffer
0,0010	INFO	FreeRecordBuffer
0,0014	START	InternalClose
0,0023	START	DestroyFields
0,0522	END	DestroyFields
0,0019	END	InternalClose

L'ouverture commence par appeler **InternalOpen**. On calcule alors la taille du buffer pour une seule ligne : 566 octets.

Puis vient la destruction des **TField** déjà existant. Ici bien évidement il n'y a pas grand-chose à faire puisque le DataSet vient d'être créé. La durée du traitement est très courte, de l'ordre de 3 microsecondes.

On arrive à l'appel à **CreateFields**. **CreateFields** est la méthode héritée du **TDataSet**, qui s'occupe d'instancier les composants **TField** correspondants aux champs qui ont été définis avec les **FieldDefs**. Après **CreateFields**, on enchaîne avec **BindFields**. Encore une fois, **BindFields** est la méthode directement héritée du **TDataSet**. Son rôle est de faire le mapping entre les **TField** tels qu'ils ont été définis, et la structure des champs dans la source de données. Concrètement pour **TMemoryDataSet**, la seule chose qui nous intéresse, c'est l'initialisation de la propriété **FieldNo** des **TField**.

Si on regarde les temps d'exécution, on se rend compte que c'est une catastrophe ! En effet, à elles deux, ces méthodes prennent **2,1+1,3 = 3,4 ms**. Cela signifie que le DataSet met au moins 3,4 ms pour s'ouvrir ! On travaille uniquement en mémoire, pas d'accès à une base de données. Rien ne justifie un tel temps d'ouverture. Pire encore, il s'agit des méthodes standards du **TDataSet**. Il s'agit donc des méthodes qui sont exécutées avec n'importe quel composant qui dérive du **TDataSet**, donc n'importe quel composant d'accès aux données, quel que soit l'API utilisée (dbGO, dbExpress...).

Lorsqu'on travaille avec une base de données, cela signifierait que quel que soit la requête (enfin pas tout à fait, ce phénomène dépend du nombre de champs retournés), la base de données, l'API et le contexte d'utilisation, il n'est

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

pas possible d'exécuter une requête en moins de 3ms ! J'ai déjà vu des projets PHP qui exécutent une requête, lisent le résultat et ferment le statement en moins d'une milliseconde !

Si on veut pouvoir obtenir de bonnes performances, il faut impérativement corriger ce problème.

La question maintenant c'est comment ? Un petit coup d'oeil dans les sources de la VCL va rapidement apporter une réponse : Par défaut, la propriété **ObjectView** est à **false**. Dans ce mode, **CreateField** se base sur **FieldDefList** au lieu de **FieldDefs** pour connaître la liste des champs à créer. **FieldDefList** s'initialise automatiquement à partir de **FieldDef**. Cependant, chaque fois que **CreateField** parcourt un élément de **FieldDefList**, la collection tente de se mettre à jour et de refaire son initialisation à partir de **FieldDef**. Ces nombreuses mises à jour de la liste dégradent les performances sans raisons en ce qui nous concerne.

En revanche, lorsqu'on définit **ObjectView** à **true**, **CreateField** se base directement sur **FieldDefs** et les performances sont meilleures. D'ailleurs, selon l'aide en ligne de Delphi **ObjectView** vaut **false** par défaut, pour les composants **BDE** et **true** dans tous les autres cas. Comme on dérive directement de **TDataSet**, on doit initialiser nous même **ObjectView** à **true**.

Pour **BindFields** c'est un peu plus compliqué. En effet, dans la pratique **BindFields** doit simplement initialiser la propriété **FieldNo** des **TField** pour indiquer l'index de chaque champ dans la source de données. En temps normal, on peut définir les **TField** en design et les mettre dans un ordre différent de celui de la source de données. Aussi, **BindFields** recherche la position de chaque champ un à un dans **FieldDefList**. Comme pour **CreateField**, **FieldDefList** est constamment recalculée avant d'effectuer la recherche. Au final, la méthode **BindFields** est assez lente.

Pourtant, il nous suffirait d'effectuer le traitement suivant :

```
for i := 0 to Fields.count-1 do
begin
  Fields[i].FieldNo := i+1;
end;
```

Seulement voilà, **FieldNo** est en lecture seule, l'attribut correspondant **FFieldNo** est privé et on ne dispose d'aucun moyen pour faire cette initialisation nous même.

Pourtant, il est malgré tout indispensable de revoir le traitement de **BindFields** si on veut obtenir de bonnes performances.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

On va bidouiller un peu afin d'écrire malgré tout l'attribut **FFieldNo** qui est privé. Pour cela, il suffit de déclarer un autre composant qui possèdent les mêmes attributs que la classe **TField**, puis d'effectuer un cast violent afin d'écrire **FieldNo**. Attention cependant, cette technique risque de ne plus fonctionner si la VCL évolue...

```
  TPrivateField = class(TComponent)
  public
    FAutoGenerateValue: TAutoRefreshFlag;
    FDataSet: TDataSet;
    FFieldName: WideString;
    FFields: TFields;
    FDataType: TFieldType;
    FReadOnly: Boolean;
    FFieldKind: TFieldKind;
    FAlignment: TAlignment;
    FVisible: Boolean;
    FRequired: Boolean;
    FValidating: Boolean;
    FSize: Integer;
    FOffset: Integer;
    FFieldNo: Integer;
    FDisplayWidth: Integer;
  end;

procedure TMemoryDataSet.BindFields(Binding: boolean);
var
  i : integer;
  Field : TPrivateField;
begin
  if Binding
  then begin
    for i := 0 to Fields.Count -1 do
    begin
      Field := TPrivateField(Fields[i]);
      Field.FFieldNo := i+1;
      TProtectedField(Field).Bind(Binding);
    end;
  end
  else begin
    for i := 0 to Fields.Count -1 do
    begin
      Field := TPrivateField(Fields[i]);
      Field.FFieldNo := 0;
      TProtectedField(Field).Bind(Binding);
    end;
  end;
end;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

A présent, si on refait le test, on obtient le résultat suivant :

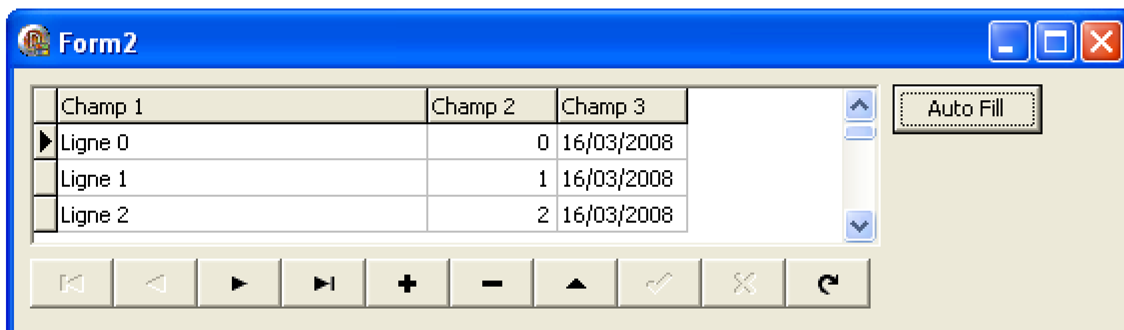
Delta	Type	User
0,0000	START	InternalOpen
0,0068	INFO	CalcRecordSize=566
0,0012	START	DestroyFields
0,0028	END	DestroyFields
0,0011	START	CreateFields
0,1271	END	CreateFields
0,0013	START	BindFields
0,0029	END	BindFields
0,0038	END	InternalOpen

Comme on peut le constater, les performances sont bien meilleures. L'**InternalOpen** ne prend plus que 0,147ms alors qu'il durait 3,4421ms. L'ouverture du **TMemoryDataSet** est ainsi plus de 20 fois plus rapide !

IV-B - Scenario 2 : Affichage et Navigation avec une DBGrid

L'objectif avec le **TMemoryDataSet** est d'obtenir un dataset bidirectionnel, pouvant être utilisé par un composant DB, en commençant par une **TDBGrid**.

Nous devons donc vérifier son fonctionnement avec une grille. Pour le test suivant, nous allons réaliser une fiche de test, qui affiche un **TMemoryDataSet** par l'intermédiaire d'une **TDBGrid**. La grille sera configurée pour autoriser les saisies de données et sera couplée à un **TDBNativator** :



Le bouton *AutoFill* insère des données dans le composant **TMemoryDataSet**.

IV-B-1 - First

Lorsqu'on effectue un **First** à l'aide du **TDBNavigator** on obtient la trace suivante :

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Delta	Type	User
0,0000	START	InternalFirst
0,0042	START	GetRecord
0,0026	START	FindNextRecord
0,0023	START	FilterRecord
0,0021	END	FilterRecord
0,0031	END	FindNextRecord
0,0030	END	GetRecord
0,0022	START	GetRecord
0,0018	START	FindNextRecord
0,0018	START	FilterRecord
0,0018	END	FilterRecord
0,0027	END	FindNextRecord
0,0025	END	GetRecord
0,0017	START	GetRecord
0,0020	START	FindNextRecord
0,0017	START	FilterRecord
0,0019	END	FilterRecord
0,0028	END	FindNextRecord
0,0028	END	GetRecord

On voit que le **First** se traduit par un appel à **InternalFirst**, suivi de trois appels à **GetRecord**. Ces appels correspondent certainement à la grille qui refait la lecture des lignes visibles.

IV-B-2 - Last

Lorsqu'on se déplace sur le dernier enregistrement, on constate le même type de trace :

Delta	Type	User
0,0000	START	InternalLast
0,0026	START	GetRecord
0,0019	START	FindPriorRecord
0,0016	START	FilterRecord
0,0013	END	FilterRecord
0,0024	END	FindPriorRecord
0,0022	END	GetRecord
0,0012	START	GetRecord
0,0013	START	FindPriorRecord
0,0012	START	FilterRecord
0,0013	END	FilterRecord
0,0019	END	FindPriorRecord
0,0018	END	GetRecord
0,0012	START	GetRecord
0,0013	START	FindPriorRecord
0,0012	START	FilterRecord
0,0013	END	FilterRecord
0,0019	END	FindPriorRecord
0,0018	END	GetRecord

InternalOpen est remplacé par **InternalLast**. Il s'en suit à nouveau de trois appels à **GetRecord**. Cependant, on remarque que cette fois, la lecture des lignes s'effectue avec **FindPriorRecord** et non plus **FindNextRecord**. Ceci nous indique que la relecture des lignes s'effectue cette fois en remontant les enregistrements !

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

IV-B-3 - Prior

Si on se déplace sur les enregistrements précédents, on constate deux cas de figure :

- Si on passe d'une ligne visible dans la grille, à une autre ligne visible dans la grille, la navigation ne génère aucune trace ! Ceci montre que les lignes affichées par la grille sont gardées en cache et qu'il n'est pas nécessaire de les relire sur le DataSet. En fait c'est le DataSet lui-même qui maintient ce cache pour nous.
- Lorsqu'on remonte au dessus de la première ligne, on remarque un simple appel à GetRecord. La grille a juste besoin de se repositionner sur l'enregistrement précédent.

IV-B-4 - Next

Sur le Next, on constate les mêmes comportements. Lorsqu'on descend sur les lignes visibles, on n'obtient aucun message dans la trace. Lorsqu'on descend pour afficher les lignes non visibles, **GetRecord** est appelé pour lire les enregistrements suivants.

Signalons cependant une particularité : Si on se déplace sur les enregistrements précédents jusqu'à ce que la grille lise un nouvel enregistrement avec **GetRecord**, puis qu'ensuite, on retourne sur les enregistrements suivant, on obtient une trace un peu différente lors de la lecture du premier enregistrement suivant :

Delta	Type	User
0,0000	START	InternalSetToRecord
0,0027	START	InternalGotoBookmark
0,0031	END	InternalGotoBookmark
0,0016	END	InternalSetToRecord
0,0018	START	GetRecord
0,0014	START	FindNextRecord
0,0015	START	FilterRecord
0,0020	END	FilterRecord
0,0017	END	FindNextRecord
0,0022	END	GetRecord

Avant de lire l'enregistrement suivant avec **GetRecord**, on constate un appel à **InternalSetToRecord**. Ceci est simplement dû au fait que la navigation sur les enregistrements précédents a placé le curseur du DataSet sur la première ligne de la grille.

Ensuite, les déplacements sur les enregistrements suivant n'ont provoqué aucun déplacement du curseur, puisque les lignes étaient déjà en cache. Par contre, pour lire l'enregistrement suivant, il faut d'abord repositionner le curseur du DataSet sur la dernière ligne du cache. C'est ce qui se passe avec l'appel à **InternalSetToRecord**.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

IV-B-5 - SetRecNo

A présent, voyons ce qui se passe si on utilise l'ascenseur de la grille pour se positionner directement dans l'ensemble de données :

Delta	Type	User
0,0000	START	SetRecNo
0,0059	START	GetRecord
0,0019	START	FindNextRecord
0,0016	START	FilterRecord
0,0026	END	FilterRecord
0,0017	END	FindNextRecord
0,0022	END	GetRecord
0,0016	START	GetRecord
0,0014	START	FindPriorRecord
0,0013	START	FilterRecord
0,0026	END	FilterRecord
0,0016	END	FindPriorRecord
0,0017	END	GetRecord
0,0015	START	InternalSetToRecord
0,0016	START	InternalGotoBookmark
0,0023	END	InternalGotoBookmark
0,0016	END	InternalSetToRecord
0,0011	START	GetRecord
0,0013	START	FindNextRecord
0,0012	START	FilterRecord
0,0019	END	FilterRecord
0,0016	END	FindNextRecord
0,0018	END	GetRecord
4,1322	END	SetRecNo

Cette fois, la grille utilise la propriété **RecNo** du DataSet pour se positionner directement sur l'enregistrement demandé.

On remarque à nouveau les trois appels à **GetRecord** caractérisant la lecture des lignes affichées par la grille. De plus, une fois le dernier enregistrement relu, on peut remarquer qu'il s'écoule encore 4 ms avant que **SetRecNo** ne rende la main. Il s'agit probablement du temps passer dans la gestion des événements, et donc également du temps mis par la grille pour se redessiner.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

V - Conclusion

Dans cet article, nous avons vu comment dériver la classe **TDataSet** afin d'écrire notre propre classe personnalisée. Nous avons ainsi réalisé un dataset en mémoire.

Nous avons également utilisé ETW pour effectuer quelques mesures de performances et observer le fonctionnement général du dataset. Ca a été l'occasion d'effectuer des optimisations importantes.

A présent, la classe est prête pour servir d'élément de base à la mise en oeuvre de OLEDB. Ce sera l'objet du prochain article...

VI - Remerciements

Je remercie particulièrement **Pedro** pour la relecture de l'article.

VII - Références

Le tracing avec Event Tracing for Windows (ETW)
Ière partie, Comparatif des architectures des API d'accès aux données

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2009 - Franck SORIANO. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.