Comment faire un chargement en blocs avec OLEDB

Optimisation des accès Base de données, IVème Partie

par Franck SORIANO

Date de publication : 09/05/2009

Dernière mise à jour : 09/05/2009

OLEDB expose l'interface **IRowsetFastLoad** qui permet d'effectuer des chargements massifs de données dans une table **SQL Server**.
Cet article explique comment utiliser facilement cette API pour insérer des données en base de données à la vitesse de **DTS**, **SSIS** et autres **bcp**.

Commentez cet article :

I - Introduction

Lorsqu'on travaille avec une base de données, on doit fréquemment réaliser des imports massifs de données. Il peut s'agir d'importer des fichiers textes générés par une autre application, ou même simplement de convertir une autre base de données.

Lorsqu'on travaille avec **SQL Server**, on peut utiliser différentes solutions pour charger efficacement ces données : Il y a tout d'abord la commande **bcp**. On peut également utiliser **DTS**, **SSIS**, ou encore une requête **BULK INSERT**.

Cependant, toutes ces solutions sont difficilement exploitables lorsqu'on veut importer une base de données un peu exotique ou en tout cas que **DTS** et **SSIS** ne savent pas gérer.
Par exemple, si on veut importer une base de données Paradox. Microsoft fournit certes un pilote OLEDB capable de lire les tables Paradox 5.0. Cependant ce dernier a beaucoup de mal avec les caractères accentués, les champs binaires et memo. Au final, le résultat de l'import n'est pas toujours satisfaisant.

De plus, lorsqu'on effectue ces tâches d'import, on souhaite souvent transformer les données avant de les injecter dans la base.

Ne serait-il pas intéressant de pouvoir injecter les données dans une table, directement depuis nos applications Delphi, sans être obligé de passer par un fichier intermédiaire ? C'est justement ce que permet de faire OLEDB grâce à l'interface **IRowsetFastLoad** et la copie en bloc.

Dans ce tutoriel, nous allons voir comment injecter massivement des données dans une table grâce à OLEDB. Il s'inscrit directement dans la suite logique de **Utiliser OLEDB en Delphi** qu'il est indispensable d'avoir lu pour aborder ce tutoriel.

*Cette fois ci, tout ce que nous allons voir concerne uniquement **SQL Server**. L'interface **IRowsetFastLoad** n'est pas supportée par les autres providers OLEDB que ceux fournits par Microsoft pour **SQL Server**.*

I-A - Télécharger les sources de l'article

Ce tutoriel est la continuité de celui sur OLEDB. Il utilise les mêmes sources et s'appuie sur les mêmes composants :

- **ETW :** Pour tracer les accès au SGBD et bénéficier d'un profiler SQL.
- **TMemoryDataSet :** Pour bénéficier d'un composant de type **TDataSet** afin d'insérer les données en base.

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

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

Les sources ont été légèrement modifiées pour gérer au mieu la copie en bloc.

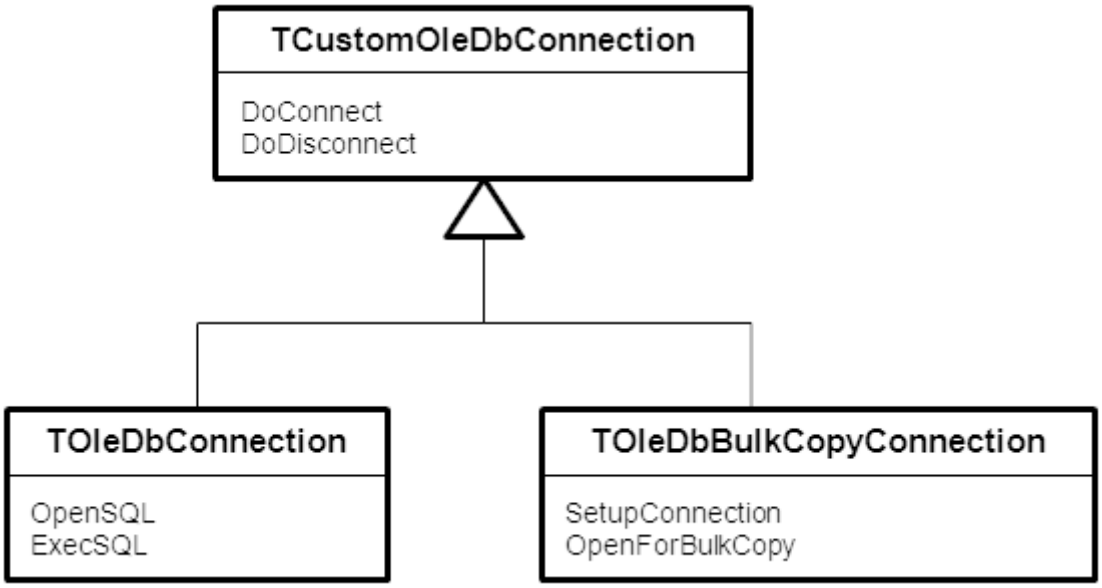
II - Principes de mise en oeuvre de IRowsetFastLoad

II-A - Ouverture de la connexion à la base

Pour pouvoir utiliser **IRowsetFastLoad**, il faut activer la propriété **SSPROP_ENABLEFASTLOAD** à *true* sur la connexion OLEDB avant de créer la session.
Une fois la session créée dans ce mode, il n'est plus possible de s'en servir pour exécuter des requêtes SQL classiques. Bon nombre des interfaces OLEDB ne sont plus accessibles.

C'est pourquoi nous allons définir un composant **Connection** spécifique pour les opérations de copie en bloc. Bien évidemment, il sera basé sur la classe **TOleDbConnection** que nous avions définie pour OLEDB.

Plus exactement, nous aurons la hiérarchie suivante :



L'ouverture de la connexion s'effectue pratiquement à l'identique d'une connexion OLEDB classique. C'est pourquoi, j'ai remonté l'implémentation de **DoConnect** dans **TCustomOleDbConnection** :

```
procedure TCustomOleDbConnection.DoConnect;
var
  Unknown : IUnknown;
  FDataInitialize : IDataInitializeSC;
begin
  // Tout d'abord, la chaîne de connexion doit avoir été définie
  if FConnectionString = ''
  then raise EOleDbException.Create('La chaîne de connexion n''est pas définie !');

  SQLLogger.Trace(EVENT_SQL_INFO, 'Ouverture de la connexion : ' +
    FConnectionString, TRACE_LEVEL_INFORMATION);
  try
    // Enfin, on a besoin d'un accès à IMAlloc pour gérer certaines allocations
    // de mémoire.
    OleCheck(CoGetMalloc(1, FMAlloc));

    // L'initialisation de la connexion est faite indirectement,
    // par l'intermédiaire du service OLEDB gérant les chaînes de connexion.
    // On commence donc par se connecter à ce service, en instanciant un
    // objet CLSID_MSDAINITIALIZE.
    OleCheck(CoCreateInstance(CLSID_MSDAINITIALIZE, nil, CLSCTX_INPROC_SERVER, IID_IDataInitialize,
      FDataInitialize));

    // Dans un deuxième temps, on crée l'objet DataSource oledb à partir
```

```
    // de la chaîne de connexion. L'objet ainsi créé est déjà initialisé.
    // Il ne restera plus qu'à ouvrir la connexion.
    Unknown := nil;
    FDataInitialize.GetDataSource(nil, CLSCTX_INPROC_SERVER,
      PWideChar(FConnectionString), IID_IDBInitialize,  Unknown);
    FDbInitialize := Unknown as IDBInitialize;

    FDataInitialize := nil; // On n'a plus besoin d'accéder au service !
    // On ouvre la connexion sur la source de donnée.
    OleDbCheck(FDbInitialize.Initialize);

    // Avant de créer la session, on peut avoir besoin de modifier les propriétés de la connexion.
    // C'est le rôle de la méthode SetupConnection qui peut être surchargée dans les classes dérivées.
    SetupConnection;

    // Il ne reste plus qu'à créer une session par défaut.
    FSession := nil;
    OleDbCheck((FDbInitialize as IDBCreateSession).CreateSession(nil, IID_IOpenRowset, FSession));

    // On essaie d'obtenir l'interface ITransactionLocal pour la gestion des
    // transaction. Cependant, il se peut que le provider OLEDB ne gère pas les
    // transaction et n'implémente pas l'interface ITransactionLocal.
    if FSession.QueryInterface(IID_ITransactionLocal, unknown) = S_OK
    then FTransaction := unknown as ITransactionLocal
    else FTransaction := nil;

    // La connexion a été effectuée.
    SQLLogger.TraceConnect;
  except
    on e:exception do
    begin
      // En cas d'erreur à la connexion, on trace l'exception.
      SQLLogger.TraceException(e);

      // Puis on nettoie les interfaces.
      FSession := nil; // La connexion a échoué.
      FDbInitialize := nil;
      FDataInitialize := nil;

      // On redéclenche l'exception.
      raise;
    end;
  end;
end;
```

On peut remarquer l'appel à la méthode **SetupConnection** qui n'existait pas dans **TOleDbConnection**. Cette méthode sera surchargée dans **TOleBulkCopyConnection** afin de définir les propriétés spécifiques pour la copie en bloc.

C'est d'ailleurs ce que nous allons faire tout de suite.

II-B - Définition des paramètres de la connexion

Dans **TOleDbBulkCopyConnection** on surcharge **SetupConnection** afin de configurer **SSPROP_ENABLEFASTLOAD** :

```
procedure TOleDbBulkCopyConnection.SetupConnection;
var
  FProperties : TOleDbPropertySet;
  Properties : IDBProperties;
begin
  // Il faut définir la propriété SSPROP_ENABLEFASTLOAD de la connexion à TRUE
  // avant de créer la session. De cette façon, OleDb créera un objet session
  // dédié pour faire des copies en bloc.
  // Dès lors ce dernier ne permettra plus de faire autre chose que des copies en bloc.

  // On va définir une propriété du jeu de propriétés DBPROPSET_SQLSERVERDATASOURCE
  FProperties := TOleDbPropertySet.Create(DBPROPSET_SQLSERVERDATASOURCE);
  try
    // Maintenant, on définit la propriété à modifier ainsi que sa nouvelle valeur
    FProperties.AddOption(SSPROP_ENABLEFASTLOAD, true);

    // Enfin, on utilise l'interface IDBProperties afin de définir la valeur de la propriété.
    Properties := FDbInitialize  as IDBProperties;
    try
      OleCheck(Properties.SetProperties(1, FProperties.AsPropertySet));
    finally
      Properties := nil;
    end;
  finally
    FProperties.Free;
  end;
end;
```

OLEDB est une API générique, indépendante des SGBD sous jacents. On ne peut donc pas connaitre à priori les propriétés nécessaires pour chaque SGBD. En effet, chaque provider possède ses propres spécificités.

C'est pourquoi dans OLEDB, les propriétés de chaque objet ne sont pas définies de façon statique dans les interfaces qui les manipulent. Au lieu de ça, on dispose d'un mécanisme de propriétés étendues.

Pour lire, écrire ou obtenir des informations sur les propriétés supportées par un objet OLEDB, on doit passer par l'interface **IDBProperties**.

Cette dernière possède trois méthodes :

- **GetProperties :** Pour lire la valeur des propriétés.
- **SetProperties :** Pour définir la valeur des propriétés.
- **GetPropertyInfo :** Pour obtenir des informations sur les propriétés supportées.

On remarquera au passage que ces méthodes sont au pluriel. C'est parce que en fait, lors de chaque appel on accède non pas à une propriété unique, mais à un tableau de propriétés.

En fait, ça va même plus loin. Les propriétés OLEDB sont organisées en groupes de propriétés : le **PropertySet**. Chaque **PropertySet** correspond à un domaine fonctionnel particulier et est identifié par l'intermédiaire d'un GUID.

Ensuite chaque propriété à l'intérieur d'un **PropertySet** est identifiée sous la forme d'un numéro unique pour le **PropertySet**.

Lorsqu'on veut lire ou écrire des propriétés avec **IDBProperties**, on doit respecter cette logique.

Ainsi la méthode **IDBProperties.SetProperties** attend deux paramètres :

- **cPropertySets :** Nombre de **PropertySet** à déifnir. C'est-à-dire, le nombre d'éléments présents dans le tableau référencé par **rgPropertySets**.
- **rgPropertySets :** Pointeur sur un tableau de structures **TDBPropSet** décrivant les **PropertySet** à définir.

Donc avant d'appeler **SetProperties**, il faut préalablement avoir renseigné un tableau de **TDBPropSet**.

TDBPropSet est elle-même une structure qui identifie le **PropertySet**, et surtout qui décrit la liste des propriétés concernées. Elle se compose de trois champs :

Champ	Type	Description
rgProperties	PDBPropArray	Pointeur sur le premier élément d'un tableau de **TDBProp**.
cProperties	UINT	Indique le nombre de propriétés. Donc le nombre d'éléments dans **rgProperties**.
guidPropertySet	TGUID	GUID identifiant le groupe de propriétés auquel on veut accéder.

Finalement, c'est dans les **TDBProp** qu'on peut définir le contenu d'une propriété.

TDBProp est une structure définie de la façon suivante :

Champ	Type	Description
dwPropertyID	DBPROPID	Identifiant de la propriété (numéro), relatif au **PropertySet**.
dwOptions	DBPROPOPTIONS	Options associées à la propriété. Concrètement, il s'agit ici d'indiquer si la propriété est optionnelle ou obligatoire. Ce champ indique si une erreur doit être déclenchée en cas d'echec en lecture/ecriture de la propriété.
dwStatus	DBPROPSTATUS	En sorti, ce champ indiquera si la lecture/ecriture de la propriété s'est déroulée correctement.
Colid	DBID	Identifiant de la colonne concernée par la propriété. Ce champ n'a de sens que pour accéder aux propriétés d'une colonne d'une table. Le reste du temps, il faut définir sa valeur à DB_NULLID.
vValue	VARIANT	Valeur associée à la propriété.

Comme on peut le voir, lire ou écrire une propriété OLEDB est un processus un peu lourd, qui nécessite pas mal de code. En particulier, il faut préparer deux tableaux en initialisant pas mal de champs.

Pour simplifier ce travail, je définis la classe **TOleDbPropertySet**. Son rôle est d'encapsuler la définition d'un **PropertySet** et de ses propriétés. Par soucis de simplicité, elle ne gère qu'un seul **Propertyset** à la fois. On indique le GUID du **PropertySet** à définir au constructeur, ou par l'intermédiaire de la propriété **PropertySetGUID**.

Ensuite la méthode **AddOption** permet d'ajouter une propriété au **PropertySet**. Elle se charge alors d'encoder le tableau de **TDBProp** correspondant.

Enfin, au moment d'appeler **IDBProperties.SetProperties**, on utilise la propriété **TOleDbPropertySet.AsPropertySet** pour obtenir la valeur à fournir au paramètre **rgPropertySets**.

De cette façon, on peut définir la propriété **SSPROP_ENABLEFASTLOAD** à *true*, au moment de l'ouverture de la connexion à la base. La connexion est alors prête pour les opérations de copies en bloc.

II-C - Obtention de l'interface IRowSetFastLoad

Nous avons ouvert une connexion à la base et l'avons préparée pour qu'elle accepte les copies en bloc. Il ne reste plus qu'à obtenir une interface **IRowsetFastLoad** afin d'effectuer le traitement.

Pour obtenir l'interface **IRowSetFastLoad**, il suffit de la demander à la session ouverte au moment de la connexion à la base :

```
var
  TableID : DBID;
  ds : TOleDbBulkCopy;
  OpenRowSet : IOpenRowSet;
  unknown : IUnknown;
  Properties : TOleDbPropertySet;
begin
  // Tout d'abord, on demande l'interface IOpenRowSet de la session.
  OpenRowSet := FSession as IOpenRowSet;

  // Ensuite, on initialise une structure DBID pour indiquer le nom de la table à utiliser.
  Fillchar(TableId, sizeof(TableId), 0);
  TableID.eKind := DBKIND_NAME; // L'objet est identifié par son nom.
  TableID.uName.pwszName := PWideChar(TableName); // On indique le pointeur sur le nom de la table.

  // A présent on définit les options pour l'opération Bulk :
  Properties := TOleDbPropertySet.Create(DBPROPSET_SQLSERVERROWSET);
  try
    Properties.AddOption(SSPROP_FASTLOADKEEPIDENTITY, KeepIdentity);
    Properties.AddOption(SSPROP_FASTLOADKEEPNULLS, KeepNulls);
    Properties.AddOption(SSPROP_FASTLOADOPTIONS, 'ROWS_PER_BATCH =' +
      IntToStr(FBulkSize) + ',TABLOCK');
    Properties.AddOption(SSPROP_IRowsetFastLoad, true);

    unknown := nil;
    OleDbCheck(OpenRowSet.OpenRowset(nil, @TableID, nil, IID_IRowsetFastLoad, 1,
      Properties.AsPropertySet, @unknown));
  finally
    Properties.Free;
  end;
```

La copie en bloc s'effectue sur une table de la base de données. En fait, concrètement tout se passe comme si on utilisait un composant **TTable** et qu'on faisait des **Append** en boucle sur ce dernier pour insérer les données.
Par contre, les lignes ne sont pas réellement insérées une par une dans la table. Au lieu de ça, elles sont regroupées en lots de taille paramétrable et écrites bloc par bloc dans la base.

Dans OLEDB, pour effectuer un traitement de copie en bloc, il suffit en fait d'ouvrir une table de la base à partir de l'interface **IOpenRowSet** de la session. En temps normal, on fait cette opération pour obtenir une interface **IRowSet**. On dispose alors d'un curseur bidirectionnel pour manipuler la table comme avec un composant **TTable**. Ici, au lieu de réclamer une interface **IRowSet** on demande l'interface **IRowsetFastLoad**.

La méthode **IOpenRowSet.OpenRowset** attend plusieurs paramètres :

- **pUnkOuter :** interface de l'objet parent, lorsqu'on veut faire de l'agrégation. Ce n'est pas notre préoccupation. Donc on peut le renseigner à **nil**.
- **pTableID :** Il doit s'agir d'un pointeur sur une structure *DBID* permettant d'identifier la table à ouvrir. C'est donc ce paramètre qui nous permet d'indiquer la table sur laquelle on veut travailler. Dans notre cas, il s'agit de la table sur laquelle on effectue la copie en bloc. On lui fournit donc *@TableID*, que l'on a initialisé juste avant.
- **pIndexID :** Il doit s'agir d'un pointeur sur une structure *DBID* référençant un index, lorsqu'on veut ouvrir un index. Ce n'est pas notre cas de figure, donc on passe ce paramètre à **nil**.
- **riid :** Très important, ce paramètre permet d'indiquer le type d'interface que l'on veut utiliser en retour. **IOpenRowSet** va instancier un objet approprié qui implémente l'interface qu'on indique ici. C'est donc grâce à ce paramètre qu'on peut indiquer qu'on veut une interface **IRowsetFastLoad**. On lui fournit la valeur **IID_IRowsetFastLoad**.
- **cPropertySets** et **rgPropertySets :** Ces deux paramètres permettent d'initialiser des jeux de propriétés sur l'objet qui sera retourné. Ils fonctionnent de la même façon que pour **IDBProperties.SetProperties**. **cPropertySets** indique le nombre de **propertySet** à définir, tandis que **rgPropertySets** est un pointeur sur le premier élément du tableau correspondant. Comme dans **SetupConnection**, on utilise la classe **TOleDbPropertySet** pour définir ces valeurs.
- **ppRowset :** Enfin en retour, ce paramètre contiendra l'interface demandée. Comme il s'agit d'un paramètre *var*, on doit obligatoirement indiquer une variable de type **IUnknown**. Cependant, il suffira ensuite de faire un cast de l'interface avec l'opérateur **As** pour obtenir l'interface **IRowsetFastLoad**.

II-D - Propriétés étendues de IRowSetFastLoad

Comme on peut le voir, **OpenRowset** permet de renseigner des jeux de propriétés étendues directement au moment de l'appel. On utilise à nouveau la classe **TOleDbPropertySet** afin de configurer différentes options pour la copie en bloc à réaliser.

Dans l'exemple précédent on a définit un certain nombre de propriétés :

- **SSPROP_IRowsetFastLoad :** Cette propriété est obligatoire si la connexion a été ouverte par l'intermédiaire de **IDataInitialize**. Autrement dit, elle est obligatoire si on s'est servit d'une chaîne de connexion. En revanche, elle n'est pas nécessaire si le provider OLEDB a été instancié directement. On définit simplement sa valeur à *true*.
- **SSPROP_FASTLOADKEEPIDENTITY :** Cette propriété permet d'indiquer comment SQL Server doit se comporter vis-à-vis des champs auto-incrémentés.

Valeur	Description
False	Les auto-incréments sont gérés par SQL Server. Ca signifie qu'il attribuera les valeurs auto-incrémentées lui-même, en ignorant les valeurs qu'on pourrait lui fournir.
True	On désactive la gestion des auto-incréments pour l'opération de copie en bloc. On peut ainsi renseigner la valeur du champ sans que ce dernier ne soit renuméroté. Si on écrit un programme de reprise de données par exemple cette option peut être très utile pour forcer les valeurs des auto-incréments.

- **SSPROP_FASTLOADOPTIONS :** Cette propriété sert à définir différentes options pour optimiser le chargement des données. La liste complète des options utilisables est définie dans la **MSDN**. Dans l'exemple précédent, on définit deux options de chargement :

Valeur	Description
ROWS_PER_BATCH	Ce paramètre permet d'indiquer le nombre de lignes qui seront écrites lors de chaque lot. Lorsqu'on doit insérer un grand nombre de lignes, ces dernières ne sont pas écrites une par une en base. Elles sont regroupées en lots et c'est le lot complet qui est écrit. Cette option permet d'indiquer quelle sera la taille des lots (en nombres de lignes) pour que SQL Server puisse optimiser le chargement. Cependant, il s'agit uniquement d'un paramètre d'optimisation. La taille réelle d'un lot étant le nombre de lignes insérées entre chaque validation. Dans l'exemple précédent, on indique la taille d'une page du dataset.
TABLOCK	Il s'agit d'un indicateur spécifiant à SQL Server qu'il doit utiliser des verrous au niveau de la table pour la session de copie en bloc. Ceci permet d'améliorer les performances de façon significatives puisque les verrous ne sont posés qu'une fois pour toute.

II-E - Utilisation de IRowsetFastLoad

L'interface **IRowsetFastLoad** fournit deux méthodes qu'on utilise pour insérer les données en base. Nous devons déclarer l'interface en Delphi de la façon suivante :

```
IRowsetFastLoad = interface(IUnknown)
  ['{5CF4CA13-EF21-11d0-97E7-00C04FC2AD98}']
  function InsertRow(hAccessor : HACCESSOR; pData : pointer) : HResult; stdcall;
  function Commit(fDone : BOOL) : HResult; stdcall;
end;
```

La méthode **InsertRow** sert à insérer une nouvelle ligne de données dans la table. Elle attend deux paramètres :

- **pData :** Il s'agit d'un pointeur sur un buffer contenant les données de la ligne à écrire.
- **hAccessor :** Désigne l'accesseur OLEDB qui permet d'interpréter le contenu de **pData**. Les données à insérer sont fournies sous la forme d'un seul et unique buffer, quelles que soient les colonnes de la table. Il faut donc indiquer à SQL Server comment interpréter ce buffer pour en extraire les valeurs des champs de la ligne. Dans l'article précédent sur OLEDB, on utilisait un accesseur pour dire à OLEDB comment charger les données retournées par une requête dans les buffers internes de la classe **TMemoryDataSet**. Ici nous utilisons le même accesseur dans le sens inverse.

InsertRow sera ainsi appelée, une fois par ligne à insérer. Lors de chaque appel, la ligne est vérifiée par le provider OLEDB. Si des conversions de type doivent être effectuées, elles sont également préparées par ce dernier. A ce stade, on peut donc rencontrer des erreurs, notamment au niveau de la conversion des champs Date.
Cependant, les lignes ne sont pas encore écrites dans la base.

La méthode **Commit** doit être appelée périodiquement afin de valider le lot en cours d'écriture et ainsi écrire physiquement les lignes dans la table. C'est à ce moment que les lignes insérées avec **InsertRow** seront réellement envoyées au serveur pour être écrites en base.

Le paramètre **fDone** sert à indiquer à SQL Server si la copie en bloc est terminée ou si on va continuer à insérer d'autres lignes ensuite.

III - Mise en oeuvre en Delphi

Nous venons de voir les principes d'utilisation de **IRowsetFastLoad**. Lorsqu'on a bien compris le principe de fonctionnement de OLEDB, **IRowsetFastLoad** est très simple d'utilisation.

Cependant, insérer des lignes en manipulant directement cette interface pose un petit problème pratique : Il faut encoder les buffers des lignes !

Pour se simplifier la vie, l'idéal serait de passer par un **TDataSet**. Par exemple, il serait bien d'ouvrir un composant du genre **TTable**, identifié à partir du nom de la table à charger. Puis d'insérer les données en faisant de bêtes **Append**.

C'est ce que nous allons réaliser avec les classes **TOleDbBulkCopyConnection** et **TOleDbBulkCopy**.

III-A - Ouverture de la connexion, la classe TOleDbBulkCopyConnection

La classe **TOleDbBulkCopyConnection** spécialise la classe **TCustomOleDbConnection** afin de gérer la connexion pour le chargement en bloc.

Nous allons lui définir une fonction **OpenForBulkCopy** qui attendra en entrée tous les paramètres nécéssaires à l'initialisation d'une copie en bloc sur une table et qui renverra en sortie un objet **TDataSet** spécialisé permettant d'insérer les données en base :

```
TOleDbBulkCopyConnection = class(TCustomOleDbConnection)
private
  FBulkSize : cardinal;
protected
  procedure SetupConnection; override;
public
  constructor Create(AOwner : TComponent); override;

  function OpenForBulkCopy(const TableName : widestring;
    KeepIdentity : boolean = false;
    KeepNulls : boolean = true) : TDataset; virtual;

  property BulkSize : cardinal read FBulkSize write FBulkSize;
end;
```

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

```
function TOleDbBulkCopyConnection.OpenForBulkCopy(
  const TableName: widestring; KeepIdentity : boolean; KeepNulls : boolean): TDataset;
var
  TableID : DBID;
  ds : TOleDbBulkCopy;
  OpenRowSet : IOpenRowSet;
  unknown : IUnknown;
  Properties : TOleDbPropertySet;
begin
  CheckConnected; // On s'assure que la connexion est bien ouverte !
  try
    // Tout d'abord, on demande l'interface IOpenRowSet de la session.
    OpenRowSet := FSession as IOpenRowSet;

    // Ensuite, on initialise une structure DBID pour indiquer le nom de la table à utiliser.
    Fillchar(TableId, sizeof(TableId), 0);
    TableID.eKind := DBKIND_NAME; // L'objet est identifié par son nom.
    TableID.uName.pwszName := PWideChar(TableName); // On indique le pointeur sur le nom de la table.

    // A présent on définit les options pour l'opération Bulk :
    Properties := TOleDbPropertySet.Create(DBPROPSET_SQLSERVERROWSET);
    try
      Properties.AddOption(SSPROP_IRowsetFastLoad, true);
```

```
      Properties.AddOption(SSPROP_FASTLOADKEEPIDENTITY, KeepIdentity);
      Properties.AddOption(SSPROP_FASTLOADKEEPNULLS, KeepNulls);
      Properties.AddOption(SSPROP_FASTLOADOPTIONS, 'ROWS_PER_BATCH =' +
        IntToStr(FBulkSize) + ',TABLOCK');

      unknown := nil;
      OleDbCheck(OpenRowSet.OpenRowset(nil, @TableID, nil, IID_IRowsetFastLoad, 1,
        Properties.AsPropertySet, @unknown));
    finally
      Properties.Free;
    end;

    // unknown contient la référence à l'interface IRowsetFastLoad pour le chargement en bloc.
    // Il ne reste plus qu'à créer un objet TOleDbBulkCopy pour manipuler facilement cette
    // interface.
    ds := TOleDbBulkCopy.Create(self);
    ds.initialize(TableName, self, unknown as IRowsetFastLoad);
    ds.FBulkSize := BulkSize;
    RegisterClient(ds); // On enregistre le dataset dans la liste des datasets actifs de la connexion.

  // de cette façon, la fermeture de la connexion fermera automatiquement tous les
                       // dataset associés qui sont encore ouverts.
    result := ds;
  except
    on e:exception do
    begin
      // En cas d'erreur, on écrit l'exception dans la trace, et on la redéclenche pour qu'elle soit
      // traitée par l'appelant.
      SQLLogger.TraceException(e);
      raise;
    end;
  end;
end;
```

On a déjà tout vu jusqu'à l'appel à **OpenRowset**.

La seule nouveauté se trouve dans le traitement qui est effectué une fois l'interface obtenue : On crée un objet **TOleDbBulkCopy** et on l'initialise en lui renseignant le nom de la table en cours de chargement, ainsi que l'interface **IRowsetFastLoad** à utiliser.

BulkSize est une propriété de la classe **TOleDbBulkCopyConnection** qui permet d'indiquer la taille des lots de chargement. Elle est transmise au dataset au moment de son initialisation.

C'est cet objet qui est retourné à l'utilisateur.

III-B - La classe TOleDbBulkCopy

La classe **TOleDbBulkCopy** est un dataset spécialisé qui va nous permettre de manipuler l'interface **IRowsetFastLoad**. Nous allons réutiliser la classe **TMemoryDataSet** pour disposer d'un dataset en mémoire, ainsi que la classe **TCustomOleDbDataSet** développée dans l'article sur OLEDB pour gérer les spécificités OLEDB. De cette façon l'implémentation de **TOleDbBulkCopy** sera très aisée.

III-B-1 - Initialisation

L'initialisation de la classe est effectuée par la méthode **Initialize**. Cette dernière est appelée par **TOleDbBulkCopyConnection.OpenForBulkCopy**.

```
procedure TOleDbBulkCopy.initialize(ATableName: widestring;
  AConnection: TOleDbBulkCopyConnection; AFastLoad: IRowsetFastLoad);
begin
  SQLLogger.TraceSQLBegin('Initialisation BULK : ' + ATableName, t0);
  FTableName := ATableName;
  FOleDbBulkConnection := AConnection;
  FFastLoad := AFastLoad;

  // On définit la structure des champs en fonction de la table.
  Describe(FFastLoad as IColumnsInfo, FOleDbBulkConnection);

  // Lorsqu'on fait un Bulk insert, seul l'insertion de nouvelles lignes est
  // autorisée. On passe le dataset en mode Unidirectionnel pour désactiver
  // toutes les autres fonctionnalités.
  SetUniDirectional(true);

  // On demande l'ouverture du dataset. Cette opération va initialiser les buffers
  // de stockage interne et créer les champs.
  Open;

  // Il ne reste plus qu'à créer l'accesseur qui nous permettra d'écrire les
  // données dans la table.
  CreateAccessor;

  FNbLineDone := 0;
  FBulkSize := 1000;
end;
```

Durant cette étape, on doit effectuer pas mal de chose. Heureusement, le plus gros du travail a déjà été fait dans **TMemoryDataset** et **TCustomOleDbDataset**.

Tout d'abord, on commence par mémoriser les paramètres d'appel : Nom de la table à charger, interface **IRowsetFastLoad** et connexion parente.

Ensuite, nous devons initialiser l'ancêtre **TMemoryDataset** de façon à mettre en place la structure de la table à charger : On doit créer les composants **TFieldDef** qui décrivent les colonnes de la table. On ne connait pas à priori la structure de la table pour laquelle on a demandée l'interface **IRowsetFastLoad**. Cependant, on peut utiliser cette dernière pour obtenir une interface **IColumnsInfo** qui décrit la structure de la table. De cette façon, on se trouve dans le même cas que celui du traitement du résultat d'une requête SQL.

On appelle donc la méthode **Describe** héritée de la classe **TCustomOleDbDataset**. Cette dernière va se charger de lire l'interface **IColumnsInfo** et d'initialiser la classe pour créer les champs.

L'étape suivante consiste à basculer le dataset en mode unidirectionnel. Dans ce mode, la plupart des fonctions de navigation ne sont plus disponibles. Le dataset est alors uniquement capable de passer à l'enregistrement suivant. C'est idéal pour un dataset sur lequel la seule opération qui sera autorisée est l'insertion d'un nouvel enregistrement.

A présent, on peut ouvrir le dataset en appelant sa méthode **Open**. Les comportements hérités de **TMemoryDataset** vont alors créer les composants **TField**.

Il ne reste plus qu'à créer l'accesseur OLEDB qui permettra d'appeler **InsertRow**. La méthode **CreateAccessor** est définie de la façon suivante :

```
procedure TOleDbBulkCopy.CreateAccessor;
begin
  InitializeBindings;

  // Ensuite, on crée un accesseur basé sur ce binding.
  FOleDbBulkConnection.OleDbCheck((FFastLoad as IAccessor).CreateAccessor(
      DBACCESSOR_ROWDATA + DBACCESSOR_OPTIMIZED,
      FieldDefs.Count,
      @Bindings[0],
      RecordSize,
      FAccessor,
      nil));
end;
```

On appelle la méthode **InitializeBindings** héritée depuis **TCustomOleDbDataset**. Cette dernière initialise le tableau *Bindings* à partir des **TFieldDefs** du dataset. Il ne reste alors plus qu'à créer l'accesseur avec **IAccessor.CreateAccessor**.

De cette façon, on dispose d'un composant **TDataset** spécialisé, ouvert sur la table à charger et qui possède les composants **TField** correspondants aux colonnes de la table.

III-B-2 - Insérer une nouvelle ligne

On va pouvoir utiliser la méthode **TDataSet.Insert** ou **TDataSet.Append** pour passer le dataset en mode insertion, utiliser les **TField** pour définir les valeurs des champs de la ligne et ainsi encoder un buffer pour **IRowsetFastLoad.InsertRow**.

Il ne reste plus qu'à traiter le **Post** pour que les lignes soient écrites en base. En fait, il suffit de surcharger la méthode **InternalPost** du dataset :

```
procedure TOleDbBulkCopy.InternalPost;
var
  Buffer : Pointer;
  i : integer;
  FieldData : PFieldData;
  FBlobIdx : cardinal;
  Size : cardinal;
  LineMin : integer;
begin
  // On commence par vérifier l'état du dataset. Pour un chargement en bloc, seules les insertions
  // sont autorisées. Le Dataset ne doit donc jamais se retrouver en édition. Si ce cas se produit
  // un déclenche une erreur.
  if state = dsEdit
  then begin
    Raise Exception.Create('Seul les Insertions sont autorisées !');
  end
  else begin
    // Avant d'écrire la ligne, il faut initialiser la longueur de chaque champ
    Buffer := GetActiveBuffer;
    for i := 0 to Fields.Count -1 do
    begin
      FieldData := GetPFieldData(Buffer, FFieldInfo[i].Offset);
      if FieldData.NullStatus = 0
      then begin
        FieldData.NullStatus := DBSTATUS_S_OK; // La valeur du champ est définie
        size := FFieldInfo[i].Size;
```

```
        // Maintenant, on calcule la longueur des champs
        case Fields[i].DataType of
        ftString:
          size := strlen(PAnsiChar(@FieldData.Data[0]));
        ftWideString:
          size := WStrLen(@FieldData.Data[0])*2;
        ftBlob, ftMemo, ftWideMemo:
          begin
            // Pour les champs Blob, il faut remplacer le numéro du Blob par son pointeur direct sur
            // ses données.
            FBlobIdx := cardinal(FieldData.Data);
            pointer(FieldData.Data) := FBlobList[FBlobIdx];
            size := FIeldData.LengthValue;
          end;
        end;
        FieldData.LengthValue := size;
      end
      else FieldData.NullStatus := DBSTATUS_S_ISNULL; // Le champ est à NULL
    end;

    // Il reste à insérer la ligne en base avec InsertRow.
    try
      FOleDbBulkConnection.OleDbCheck(FFastLoad.InsertRow(Accessor, GetActiveBuffer));
    except
      on e:exception do
      begin
        // En cas d'erreur on trace l'exception et on la redéclenche pour qu'elle soit traitée par
        // l'appelant.
        SQLLogger.Trace(EVENT_SQL_ERROR, 'Erreur lors de l''insertion de la ligne n°' +
            IntToStr(FNbLineDone) + ' : ' + e.Message);
        raise;
      end;
    end;

    inc(FNbLineDone); // On compte les lignes insérées
    FPending := true; // Une ligne a été insérée et devra être validée !


  ClearBlobs; // La ligne vient d'être insérée. Elle n'est plus lisible. On libère la mémoire utilisée
              // pour les Blobs.

    if FNbLineDone mod FBulkSize = 0 // Lorsqu'on a atteind la taille d'un lot, il faut valider les
    then begin                       // données en base.
      try

   // On effectue la validation dans la base. On part du principe que les insertions vont continuer
        // tant que le dataset n'a pas été fermée.
        FOleDbBulkConnection.OleDbCheck(FFastLoad.Commit(false));
        FPending := false; // Les lignes insérées ont été validées. Il n'y a plus de lignes en attente.
        SQLLogger.Trace('BULK: ' + TableName + ' Commit'); // On trace la validation.
      except
        on e:exception do
        begin
          // En cas d'erreur on trace l'exception et on la redéclenche pour qu'elle soit traitée par
          // l'appelant.
          LineMin := FNbLineDone - FBulkSize+1;
          if (LineMin<0)
          then LineMin := 0;

          SQLLogger.Trace(EVENT_SQL_ERROR,
            Format('Erreur lors de la validation du lot %d-%d : %s', [LineMin, FNbLineDone,
  e.Message]));
          raise;
        end;
      end;
    end;
   end;
  end;
```

Le buffer d'une ligne nécessite une préparation avant d'être inséré en base. D'abord, il faut définir précisément le statut de chaque champ pour OLEDB. Cette opération est très simple, il suffit d'indiquer si le champ est à **null** (*DBSTATUS_S_ISNULL*) ou si sa valeur est définie (*DBSTATUS_S_OK*). Les autres statuts OLEDB n'ont pas de sens dans ce contexte.

Ensuite, on doit initialiser la longueur effective des champs de taille variable. Cette information n'est pas nécessaire au fonctionnement du dataset. C'est pourquoi la longueur du champ n'est pas définie en même temps que sa valeur. Mais comme cette information est nécessaire pour OLEDB, on doit calculer la longueur des champs de type **ftString** et **ftWideString** (**varchar** et **nvarchar** dans SQL Server).

Enfin, les champs de type **Blob** ont besoin d'un traitement spécial. Pour OLEDB, on a définit le binding en disant qu'il s'agit de champs de type référence. C'est-à-dire que la valeur présente dans le buffer de ligne est un pointeur vers les données réelles du champ.

Or dans le fonctionnement de la classe **TMemoryDataset**, les valeurs des Blobs sont en réalité un numéro de Blob qui permet de retrouver son contenu. Avant de donner le buffer à OLEDB, on doit donc remplacer ce numéro de Blob par la valeur du pointeur correspondant.

Une fois le buffer préparé, il ne reste plus qu'à faire l'insertion de la ligne en appelant **IRowsetFastLoad.InsertRow**.

On a décidé d'encapsuler l'interface **IRowsetFastLoad** à l'aide d'un dataset unidirectionnel. Une fois la ligne écrite (le Post terminé), il faut considérer que la ligne qui vient d'être insérée n'est plus valide dans le dataset et ne doit plus être lue (de toute façon, elle ne pourra pas être modifiée). Il n'est pas possible de faire de navigation arrière. On peut donc en profiter pour libérer la mémoire utilisée par les Blobs. On appelle pour cela la méthode **ClearBlobs**.

Enfin, on compte les lignes insérées avec **InsertRow**. Chaque fois qu'on atteind la taille d'un lot (le nombre de lignes indiqué par **BulkSize**), on doit valider les données en base en appelant **IRowsetFastLoad.Commit**.

III-B-3 - Fermeture du dataset

Au moment de la fermeture du dataset, il faut penser à valider les lignes insérées qui ne l'ont pas encore été.

Ensuite on peut libérer les ressources associées :

```
procedure TOleDbBulkCopy.InternalClose;
begin
  // On regarde s'il reste des données à valider
  if FPending
  then begin
    // et on appelle Commit pour effectuer la validation.
    FOleDbBulkConnection.OleDbCheck(FFastLoad.Commit(true));
    FPending := false;
    SQLLogger.Trace('BULK: ' + TableName + ' Commit');
  end;

  // On n'a plus besoin de l'accesseur.
  (FFastLoad as IAccessor).ReleaseAccessor(FAccessor, nil);

  FFastLoad := nil;

  FOleDbBulkConnection.UnRegisterClient(self); // Le dataset n'est plus connecté. On le supprime des
                                               // dataset actifs de la connexion.

  inherited internalClose;
  SQLLogger.TraceSQLEnd('Fermeture du BULK : ' + TableName, t0);
end;
```

IV - Exemples et conseils d'utilisation

IV-A - Exemple d'utilisation

Pour essayer ce chargement par lot, nous allons réaliser un programme de test. Ce dernier va insérer automatiquement 1 000 000 de lignes dans la table **ErrorLog** de la base de données **AdventureWorksLT**.

Je rappelle que la structure de cette table est la suivante :

```
CREATE TABLE [dbo].[ErrorLog](
 [ErrorLogID] [int] IDENTITY(1,1) NOT NULL,
 [ErrorTime] [datetime] NOT NULL CONSTRAINT [DF_ErrorLog_ErrorTime]  DEFAULT (getdate()),
 [UserName] [sysname] NOT NULL,
 [ErrorNumber] [int] NOT NULL,
 [ErrorSeverity] [int] NULL,
 [ErrorState] [int] NULL,
 [ErrorProcedure] [nvarchar](126) NULL,
 [ErrorLine] [int] NULL,
 [ErrorMessage] [nvarchar](4000) NOT NULL,
 CONSTRAINT [PK_ErrorLog_ErrorLogID] PRIMARY KEY CLUSTERED
(
 [ErrorLogID] ASC
)
)
```

Elle comprend donc un champ auto-incrémenté **ErrorLogID**, un champ de type date **ErrorTime**, des entiers, des chaînes de caractères. Elle contient en particulier le champ **ErrorMessage** qui est définit avec une longueur maximale de 4000 caractères.
Or dans le composant que nous avons réalisé, les chaînes de longueur maximale supérieure à 4000 sont traitées comme des mémos.

On va remplir la table avec le programme suivant :

```
var
  ds : TDataset;
  i : integer;
  fldID : TField;
  fldUser : TField;
  flderrorTime : TField;
  FldErrNr : TField;
  FldErrMsg : TField;
  BulkCnt : TOleDbBulkCopyConnection;
begin
  // Premièrement, il faut créer et initialiser une connexion à la base de données en vue d'effectuer
  // une copie en blocs. On instancie un objet TOleDbBulkCopyConnection
  BulkCnt := TOleDbBulkCopyConnection.Create(nil);
  try

    // On configure les paramètres de connexion avec une chaîne de connexion assez standard :
    BulkCnt.ConnectionString := 'Provider=SQLNCLI.1;Integrated Security=SSPI;' +
      'Persist Security Info=False;Initial Catalog=AdventureWorksLT;Data Source=.\SQLExpress';
    BulkCnt.Connected := true; // Enfin on ouvre la connexion

    // Une fois la connexion ouverte, il ne reste plus qu'à préparer la copie en blocs pour la table
    // ErrorLog.
    ds := BulkCnt.OpenForBulkCopy('ErrorLog', true);
    try
      // Le dataset est maintenant prêt pour effectuer le chargement. On commence par rechercher les

  // composants TField correspondants aux champs qui vont être initialisés de façon à ne pas appeler

  // FieldByName en boucle lors de l'insertion des lignes. En effet, on perdrait alors le bénéfice de
      // la copie en blocs du fait des recherches des TField.
      fldID := ds.FieldByName('ErrorLogid');
```

```
      fldErrorTime := ds.FieldByName('ErrorTime');
      fldUser := ds.FieldByName('UserName');
      FldErrNr  := ds.FieldByName('ErrorNumber');
      fldErrMsg := ds.FieldByName('ErrorMEssage');
      for i := 1 to 1000000 do // On boucle pour insérer 1 000 000 de ligne dans la table !
      begin
        // Enfin on peut insérer les lignes comme avec un TTable :
        ds.Append; // On passe le dataset en insertion (Rq: Insert fait exactement la même chose).

        // On initialise les valeurs des champs.
        fldID.AsInteger := i;
        fldErrorTime.AsDateTime := now;
        fldUser.AsString := 'Erreur';
        fldErrNr.AsInteger := i;
        FldErrMsg.AsString := 'Message d''erreur n°' + IntToStr(i);

        // Enfin, on demande l'insertion de la ligne.
        ds.Post;
      end;
    finally
      ds.Free; // En fin de traitement, on détruit le Dataset.
    end;
  finally
    BulkCnt.Free; // Enfin, on ferme la connexion.
  end;
end;
```

Comme on peut le voir, le chargement en bloc s'effectue aussi simplement qu'en insérant des lignes dans un **TTable**. Par contre, on a une différence de taille avec le **TTable** : **Les performances !**.

*Si on exécute le programme, **les 1 000 000 de lignes sont écrites en base en ... 12 secondes !***

***OpenForBulkCopy** est appelée ici avec ses paramètres par défaut. Ca veut notamment dire que les champs auto-incrémentés seront gérés par SQL Server. Pourtant, on affecte malgré tout une valeur au champ **ErrorLogid**. C'est nécessaire à cause de OLEDB. En effet, si on n'affecte pas de valeur au champ, ce dernier aura la valeur NULL. Or le champ en question n'autorise pas les NULL. Le provider OLEDB (côté client) vérifie que tous les champs NOT NULL sont bien affectés au moment de l'appel à **InsertRow**. Donc si on n'initialise pas le champ, **InsertRow** retourne une erreur ! En affectant une valeur au champ, on passe le contrôle côté client. Ensuite SQL Server (côté serveur donc) ignorera purement et simplement la valeur renseignée pour lui affecter celle du champ **Identity**.*

IV-B - Gestion des erreurs

Lorsqu'on effectue une copie en bloc, la gestion des erreurs est un peu problématique. On peut rencontrer des erreurs à deux niveaux :

Au moment de l'appel à **InsertRow**, les données fournies vont être contrôlées par l'accesseur et on peut rencontrer des erreurs de conversion sur les types de données (le plus souvent, pour des dates invalides), ou sur les champs NOT NULL non renseignés.
Le problème, c'est qu'on obtient souvent une erreur générique. On sait alors que la ligne est invalide, mais il est souvent difficile d'identifier précisément le champ qui pose problème

Deuxièmement, au moment du **Commit** du lot, SQL Server va déclencher ses contraintes d'intégrités sur les données insérées. Si la table possède des triggers, ces derniers vont également être exécutés. On peut alors obtenir une erreur, avec un beau message de violation de contrainte par rapport au lot, sans pouvoir identifier précisément la ligne en erreur dans le lot.

Par exemple, si dans l'exemple précédent on force les valeurs auto-incrémentées et qu'on lance le chargement sans avoir vidé la table au préalable, on obtient toujours le même message d'erreur :

```
Erreur lors de la validation du lot 1-1000 : MSG-2627, Line 1, Violation de la contrainte PRIMARY KEY
 'PK_ErrorLog_ErrorLogID'. Impossible d'insérer une clé en double dans l'objet 'dbo.ErrorLog'.
```

Le numéro de ligne indiqué dans le message est toujours 1, quel que soit la ligne en erreur dans le lot.

IV-C - Conseils d'utilisation

Le chargement par copie en bloc est particulièrement performant pour insérer massivement de gros volumes de données en base. Il se prête bien pour les traitements de type import de fichiers, migration de données (par exemple, migrer une base Paradox dans une base SQL Server), et tous les traitements de type ETL d'une façon générale.

L'intérêt de cette technologie se trouve bien évidemment dans ses performances exceptionnelles. Cependant, pour en profiter pleinement, il faut prendre quelques précautions :

- Pré-allouer la taille de la base de données avant d'effectuer son chargement. En effet, on part bien souvent d'une base vide, pour lui injecter plusieurs giga-octets de données. Si la base est configurée en croissance automatique avec un incrément de 1 Mo, on va passer notre temps à l'agrandir plutôt qu'à charger les données (sans parler de la fragmentation qui va en résulter). Prévoyez donc une base suffisamment grande dès le départ, et réduisez sa taille ensuite en cas de besoins.
- Chaque fois que c'est possible, supprimer les indexes et les contraintes avant de charger une table et recréés les une fois le chargement terminé. Les performances seront bien meilleures. De toute façon, si on part d'une table vide qu'on remplit avec beaucoup de données ligne par ligne, il y a de grandes chances pour que les indexes ne soient pas correctement équilibrés à la fin. Donc autant les supprimer et les recréer à la fin.
- Faites très attention à ce que le code qui va effectuer les insertions de lignes soit lui-même écrit pour être très efficace. La copie en bloc est vraiment très performante. Tellement performante que dans la pratique, le temps d'exécution de **InsertRow** et **Commit** est souvent presque négligeable devant le temps qu'il a fallu pour lire/préparer/calculer les données à insérer.
- N'utilisez jamais **FieldByName** dans les boucles d'insertion des lignes. Vous passeriez alors votre temps à rechercher les champs plutôt qu'à les initialiser et à insérer les lignes. A la place, préparer le traitement en recherchant les **TField** avant la boucle et travaillez directement sur le **TField** ensuite. Au pire, mémorisez au moins l'index du champ et référencer le par index et non pas par nom.
- On peut multi-threader les traitements pour charger plusieurs tables simultanément. Pour ce faire, instanciez un composant **TOleDbBulkCopyConnection** (et donc une connexion) par thread.

Dans l'exemple qui a été présenté, on a pu insérer 1 000 000 de lignes en 12 secondes, soit pratiquement 100 000 lignes par secondes. Cependant bien évidemment, ces performances dépendent beaucoup des données à charger. Une table avec peu de champs et des lignes assez courtes sera chargée beaucoup plus rapidement qu'une table avec 300 champs !

V - Conclusion

Dans ce tutoriel, nous avons vu comment utiliser l'interface **IRowsetFastLoad** de OLEDB afin d'insérer efficacement de gros volumes de données dans une base de données SQL Server. On a pu ainsi insérer 1 000 000 d'enregistrements dans une table, en près de 12 secondes.

Nous avons vu comment réutiliser les composants qui avaient été développés pour OLEDB afin d'utiliser facilement **IRowsetFastLoad** en Delphi.

Ce tutoriel conclu la série "Optimisation des accès Base de données".

VI - Références

Le Tracing avec Event Tracing for Windows (ETW) :
http://fsoriano.developpez.com/articles/etw/delphi/

Comparatif des architectures des API d'accès aux données :
http://fsoriano.developpez.com/articles/db/comparatifapi/

Développer un DataSet en mémoire :
http://fsoriano.developpez.com/articles/db/dataset/delphi/

Utiliser OLEDB en Delphi :
http://fsoriano.developpez.com/articles/db/oledb/delphi/

La documentation OLEDB sur MSDN :
http://msdn.microsoft.com/en-us/library/ms722784(VS.85).aspx

La documentation IRowsetFastLoad sur MSDN :
http://msdn.microsoft.com/en-us/library/ms131708.aspx

La base de données d'exemple AdventureWorksLT :
http://www.codeplex.com/MSFTDBProdSamples/Release/ProjectReleases.aspx?ReleaseId=4004

VII - Remerciements