

Comparatif des architectures des API d'accès aux données

Optimisation des accès Base de données, 1ère Partie

par Franck SORIANO ([Pages perso](#))

Date de publication :

Dernière mise à jour : 28/09/2008

Cet article est une introduction à une série d'articles sur les performances des accès base de données.

Il compare les architectures des principales API génériques utilisables pour accéder à une base de données (ADO, ADO.NET, dbGO, dbExpress...) en Delphi. Le but de cette comparaison est de montrer l'effet de chacune de ces architectures sur les performances.

I - Introduction.....	3
II - Architecture des différentes API.....	4
II-A - ADO.....	4
II-B - dbGO.....	5
II-C - dbExpress : TSqlDataSet.....	6
II-D - dbExpress : TSqlDataSet avec TClientDataSet.....	8
II-E - ADO.NET : SqlDataReader.....	10
II-F - ADO.NET : SqlDataReader et DataTable.....	11
III - Conclusion.....	11
IV - Remerciements.....	11

I - Introduction

Nous avons tous un jour ou l'autre été confronté au problème suivant : *"Je démarre une nouvelle application. C'est l'occasion d'essayer de nouvelles choses. Quels composants choisir pour accéder à la base de données ?"*

Il y a quelques années sous Delphi, le choix était vite fait. On avait le choix entre utiliser le BDE et utiliser le BDE. Aujourd'hui, la situation a bien changé.

La question a fait l'objet d'un sondage sur developpez.com ("**Quel est le meilleur moyen d'accéder à une base de données ?**" par **aityahia**) où ADO semble avoir la préférence de la communauté. Ce sondage ne précise pas si on parle d'ADO avec dbGO, ou ADO utilisé directement à travers ses objets COM.

En lisant le fil des discussions, il y a une chose qui m'a marquée : On regarde les bugs rencontrés ou ce qu'on a l'habitude de faire. Cependant presque personne n'a l'air de s'intéresser à ce qui pour moi est l'essentiel : **Les performances.**

En effet, toutes les API se valent d'un point de vue fonctionnel. Elles permettent toutes d'exécuter une requête quelconque. Elles permettent toutes de lire le résultat d'une requête, d'effectuer une requête paramétrée... En fait, on n'a besoin de rien d'autre. Si, selon la façon dont l'application est structurée, on a également besoin d'afficher le résultat d'une requête dans une grille.

On sous-estime souvent l'impact de l'API utilisée sur les performances de l'application. Pourtant lorsqu'on exécute une requête assez simple sur un SGBD (genre un SELECT sur une table sans jointure, juste avec une clause WHERE sur des champs indexés), le temps passé par le SGBD pour exécuter la requête est négligeable devant le temps mis par l'API pour lire le résultat.

Or dans une application base de données, surtout une application Delphi, on passe notre temps à faire des requêtes très simples et à lire les résultats. Et c'est bien la lecture des données renvoyées par un SELECT qui fait toute la différence entre une API rapide et une API lente, entre une application molle et une application nerveuse. Les écarts peuvent être très importants. **J'ai déjà vu des requêtes renvoyant 1000 lignes passer de 16 s à 200 ms juste en changeant de composant TQuery.**

Là où je suis surpris par les résultats du sondage, c'est que justement quand on dit ADO, je pense qu'on parle de dbGO. C'est à dire les composants Delphi qui encapsulent les appels aux objets COM d'ADO. Or dbGO n'est pas loin d'être la solution la plus lente.

Au départ, je voulais faire une série de tests sur les performances de chaque API pour comparer et commenter les résultats.

Cependant les contrats de licence de la plupart des API (à commencer par dbExpress) interdisent la publication de tels tests.

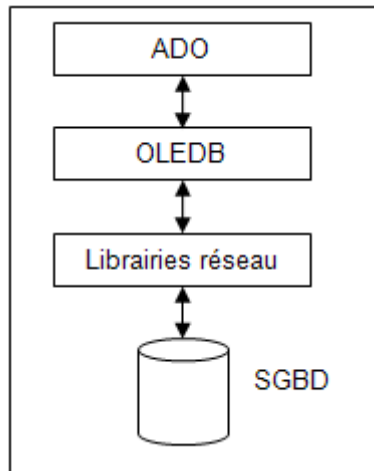
C'est pourquoi, j'ai finalement choisi d'effectuer un comparatif des architectures de chaque API. De cette façon nous pourrions tenter de prévoir les performances de chacune d'entre elle d'après leur architecture. Et j'encourage chacun à effectuer ses propres tests pour les vérifier.

II - Architecture des différentes API

II-A - ADO

ADO signifie "ActiveX Data Object". Comme son nom l'indique, il s'agit d'une librairie d'objets COM représentant des objets de données. Il s'agit d'une API de haut niveau fournissant un niveau d'abstraction élevé pour accéder à une base de données.

ADO est construit par-dessus OLE-DB, selon le schéma suivant :



En fait, c'est la couche OLE-DB qui effectue les accès au SGBD. Elle exécute des requêtes SQL sur le serveur et reçoit les données en retour. On pourrait donc utiliser directement OLE-DB pour accéder au SGBD puisqu'il s'agit d'une API à part entière. Cependant elle est très complexe à mettre en oeuvre.

ADO a été bâti par-dessus OLE-DB pour simplifier son utilisation et la rendre plus accessible au commun des mortels. Son principe de fonctionnement est de fournir aux développeurs un modèle objet de haut niveau masquant la complexité des accès bas niveaux. Il s'agit d'une couche d'adaptation.

ADO définit des objets tels que **Connection**, **RecordSet**, **Command** pour exposer des fonctions permettant de manipuler des données. En soit ADO est similaire à l'architecture db de Delphi. Les objets implémentés sont sensiblement les mêmes que ceux qu'on retrouve en Delphi : **TConnection**, **TTable**, **TQuery**.

Au final, on accède aux données par l'intermédiaire d'un curseur (le RecordSet). Les données sont lues ligne à ligne, champ par champ.

Evidemment, cette couche ADO va avoir un coût. Les données retournées par OLE-DB doivent être remises en forme.

Lorsqu'on lit la valeur d'un champ, il n'est pas possible de lire sa valeur dans son type primitif, on ne peut lire qu'un variant quel que soit le type de données réel du champ.

ADO est donc une API de haut niveau complète. Elle peut être utilisée directement en Delphi, sans aucun composant supplémentaire. Il suffit juste d'importer la bibliothèque de types et d'utiliser les objets COM directement. C'est d'ailleurs de cette façon qu'ADO est utilisé dans VB, en ASP, ou encore dans les langages de script.

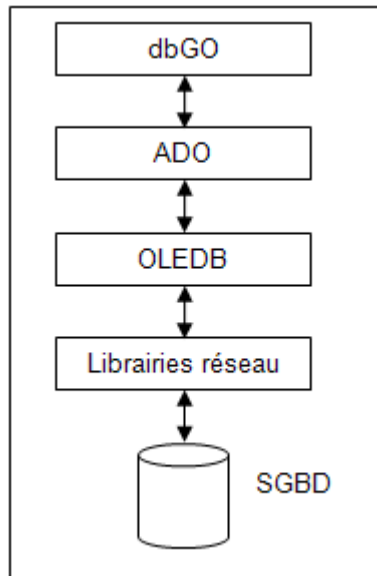
Cependant, on ne peut pas travailler directement avec les composants data-aware de Delphi (DBEdit, DBGrid...). En effet ces derniers ne reconnaissent que les descendants de **TDataSet**.

II-B - dbGO

ADO est déjà une API complète. Cependant, si on veut travailler avec des contrôles orientés données, par exemple une DBGrid, il nous faut un composant qui dérive de la classe **TDataSet**.

C'est ce que fait dbGO. Il s'agit d'une couche de compatibilité fournie par CodeGear pour utiliser ADO. Elle encapsule les objets COM ADO pour que le développeur puisse travailler avec des composants de type DataSet.

L'architecture est alors la suivante :



Ainsi, lorsqu'on ouvre une table **TADOTable**, dbGO s'occupe d'initialiser un objet **RecordSet** et de l'ouvrir pour effectuer la même opération dans ADO.

Pour lire les données, c'est un peu plus compliqué. Le composant **TDataSet** gère des buffers contenant quelques lignes de données. Lorsque le DataSet est utilisé par une DBGrid, ces buffers contiennent au moins autant de lignes que de lignes visibles dans la grille. Dans dbGO, ce buffer ne contient pas une ligne de données, mais mémorise un bookmark sur la ligne correspondante dans l'objet ADO.

Pour lire un champ, le **TDataSet** essaie de lire sa valeur à partir du buffer. La lecture du champ doit s'effectuer en plusieurs temps.

Tout d'abord, il faudra peut-être repositionner le recordset ADO sur la ligne à lire à partir du bookmark. Ce n'est qu'ensuite que le champ pourra être lu.

Le composant effectue une lecture du champ sur l'objet ADO. Il lit alors un variant. Puis ce variant est converti dans le bon type de donnée, en fonction de la façon dont le champ est lu (**AsInteger**, **AsString**, ...).

Cette couche dbGO va ainsi fortement dégrader les performances en lecture et en écriture des champs.

On peut s'attendre à ce que le temps de lecture d'un DataSet complet (lecture complète du résultat d'une requête) soit au minimum doublé par rapport à l'usage direct d'ADO.

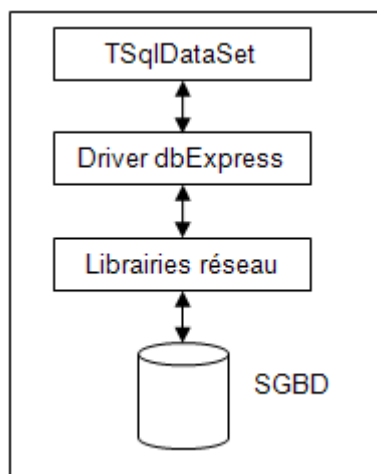
II-C - dbExpress : TSqlDataSet

Comme on vient de le voir, avec ADO et dbGO on empile les couches au dessus d'OLE-DB, ce qui finira par engendrer une dégradation conséquente des performances.

De plus ADO dégrade déjà les performances pour fournir des fonctionnalités de haut niveau qui sont ensuite ré-implémentées dans les composants Delphi.

L'idée de dbExpress est de fournir une API bas niveau, épurée de toute fonctionnalité pouvant être codée à l'extérieur de l'API. De cette façon, on a le choix. On peut profiter des performances de la simplicité. Ou on peut définir des objets de plus haut niveau qui seront alimentés à partir de la couche bas niveau.

Le composant **TSqlDataSet** est là pour nous donner accès à la simplicité de dbExpress :

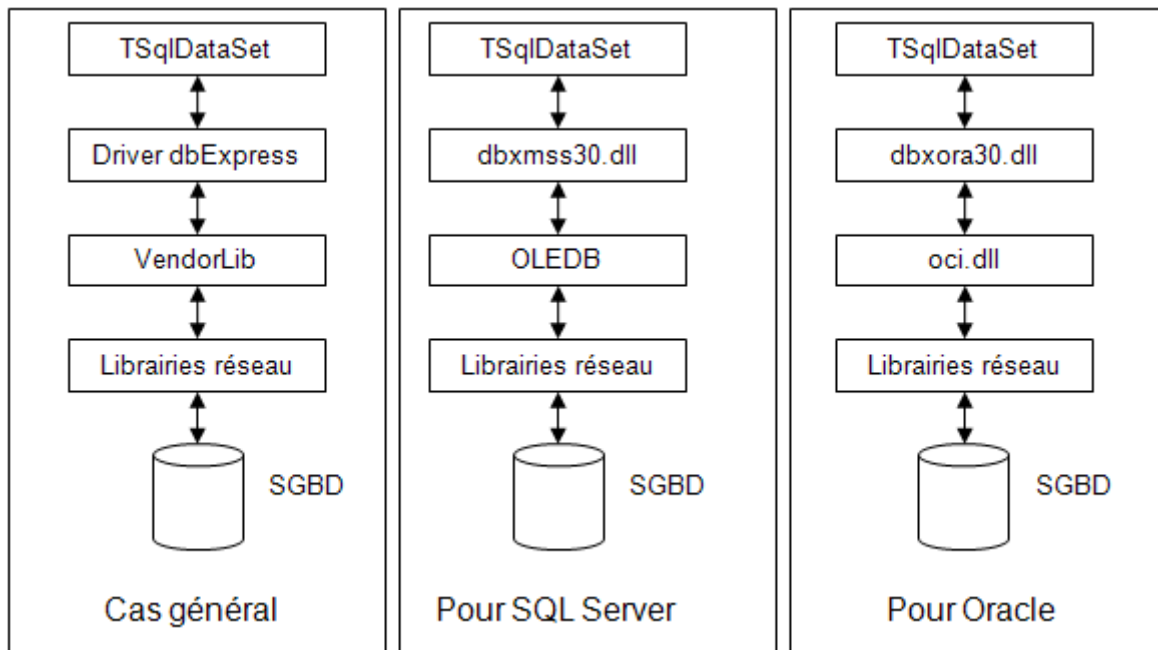


Il encapsule les appels au driver dbExpress. Ce dernier est plus ou moins une boîte noire.

En principe, on peut disposer d'un grand choix de drivers dbExpress, chacun étant alors libre d'implémenter ses services comme il le souhaite.

Cependant, si on examine les drivers fournis en standard par CodeGear, et plus particulièrement la propriété *VendorLib* de l'objet **TSQLConnection**, on se rend compte que ces derniers s'appuient en réalité sur les API fournies par les éditeurs des SGBD.

Ainsi en réalité, l'architecture est plutôt :



Finalement ce n'est pas très différent en nombre de couches successives de l'architecture ADO/dbGO

On peut donc avoir un gain de performances par rapport à ADO/dbGO du fait que les fonctionnalités offertes sont de plus bas niveau. On n'a pas de bookmark à gérer, pas d'ordre de tri, pas de filtre. On n'est pas obligé non plus de passer par des variants pour lire les données.

Par contre le **TSqIDataSet** est unidirectionnel. Ce qui signifie qu'il n'est pas possible de l'utiliser directement pour l'afficher dans une DBGrid.

Chaque couche dégrade nécessairement les performances. Plus on en empile et moins les performances sont bonnes.

De plus, on n'est pas à l'abri d'un défaut dans l'implémentation d'une des couches. Ma propre expérience avec dbExpress m'a montré un grave problème dans la gestion des champs LOB (BLOB, CLOB, ...). Dès qu'une requête retourne un champ dont la longueur n'est pas limitée, les performances de dbExpress s'écroulent à un point tel que dbExpress devient inutilisable.

Ce n'est pas très gênant sur les champs binaires de grande taille, car en principe on n'a besoin que d'une seule valeur à la fois. Donc on lit une seule ligne avec le champ.

Par contre, le problème devient très sérieux si on définit des champs commentaires. Ces derniers sont généralement très courts (< 50 caractères). La plupart du temps ils ne sont pas renseignés. Mais de temps en temps, on a une description longue à saisir. Si on définit un champ memo (*varchar(max)* par exemple sous SQL Server 2005), les performances de dbExpress s'écroulent. Si on définit une taille maximale importante (par exemple *varchar(4000)*), c'est le DataSet lui-même qui s'écroule car les champs de taille variable sont gérés comme des champs de longueur fixe. Donc la taille des lignes augmente en mémoire, le volume des recopies de données aussi. D'où une dégradation des performances.

Bien évidemment, tout ce qu'on peut dire sur dbExpress dépend en fait directement du driver utilisé. Les performances peuvent changer du tout au tout d'un driver à l'autre, même pour le même SGBD.

II-D - dbExpress : TSqlDataSet avec TClientDataSet

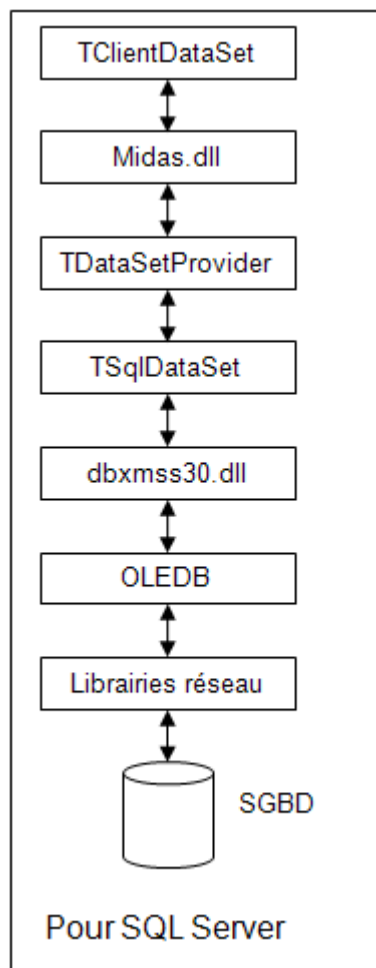
Le gros inconvénient de dbExpress cependant, reste qu'on ne dispose que d'un DataSet unidirectionnel. Ainsi, comme pour ADO natif, il n'est pas possible de travailler avec des composants DB.

Pour remédier à cette "limitation" CodeGear préconise de coupler dbExpress avec un **TClientDataSet**.

Le **TClientDataSet** charge les données en mémoire et nous fournit alors un DataSet bidirectionnel, parfaitement fonctionnel qui offre toutes les fonctionnalités du **TTable** du BDE.

Si les données sont modifiées par l'intermédiaire du **TClientDataSet**, ce dernier se charge de propager les modifications à son provider. Elles seront ainsi automatiquement écrites dans la base en générant les requêtes SQL appropriées.

L'architecture est nettement plus complexe, ici avec le driver CodeGear pour SQL Server :



Le ClientDataSet se manipule comme un composant **TTable**. Pour accéder à une table, on commence donc par faire un **Open** sur le composant.

L'ouverture du **TClientDataSet** va provoquer l'ouverture du **TSqlDataSet** pour exécuter la requête de chargement des données.

Là où les choses se compliquent, c'est le traitement de lecture du résultat en lui-même.

Le **TClientDataSet** ne peut pas lire le **TSqlDataSet** directement. Il passe par l'intermédiaire du composant **TDataSetProvider**. Ce dernier va lire l'intégralité du **TSqlDataSet**, ligne à ligne, champ par champ.

Durant cette première lecture, le **TDataSetProvider** va encoder un **DataPacket**. Concrètement, c'est un gros variant dans lequel sont encodées toutes les données lues par le **TDataSetProvider**. Ce variant est alors transmis au **TClientDataSet**.

Seulement le **TClientDataSet** n'est pas capable de travailler directement sur ce variant. Il va simplement s'en servir pour charger son cache interne. Plus exactement, il appelle un objet COM implémenté dans **midas.dll**. C'est ce dernier qui va décoder le variant et s'en servir pour initialiser un cache interne.

En retour, il fournit un curseur au **TClientDataSet**.

Le **TClientDataSet** va alors pouvoir se servir de ce curseur pour lire les données lignes à lignes. Il ne s'agit en réalité que d'une couche d'adaptation sur cet objet COM.

Au final, chaque fois que notre application aura besoin de lire un champ, ce dernier aura en réalité été lu au moins trois fois. Une fois par le **TDataSetProvider**, une fois par midas, et une fois par nous.

Il va de soit que les performances vont en prendre un sérieux coup.

On peut se demander pourquoi une telle architecture ? Tout simplement parce que le **TClientDataSet** existait bien avant dbExpress et n'a pas été conçu pour cet usage. En effet, le **TClientDataSet** est certes capable de fonctionner comme un DataSet en mémoire. Cependant son rôle premier était de servir de middleware pour le développement d'applications n-tiers. Le composant a simplement été réutilisé pour un usage différent, alors qu'un composant spécifique aurait pu donner de bien meilleures performances.

Au final, si on a besoin d'utiliser des DataSet bidirectionnels (c'est-à-dire probablement la majorité d'entre nous) dbExpress n'est pas vraiment la solution la plus optimale en termes de performances.

II-E - ADO.NET : SqlDataReader

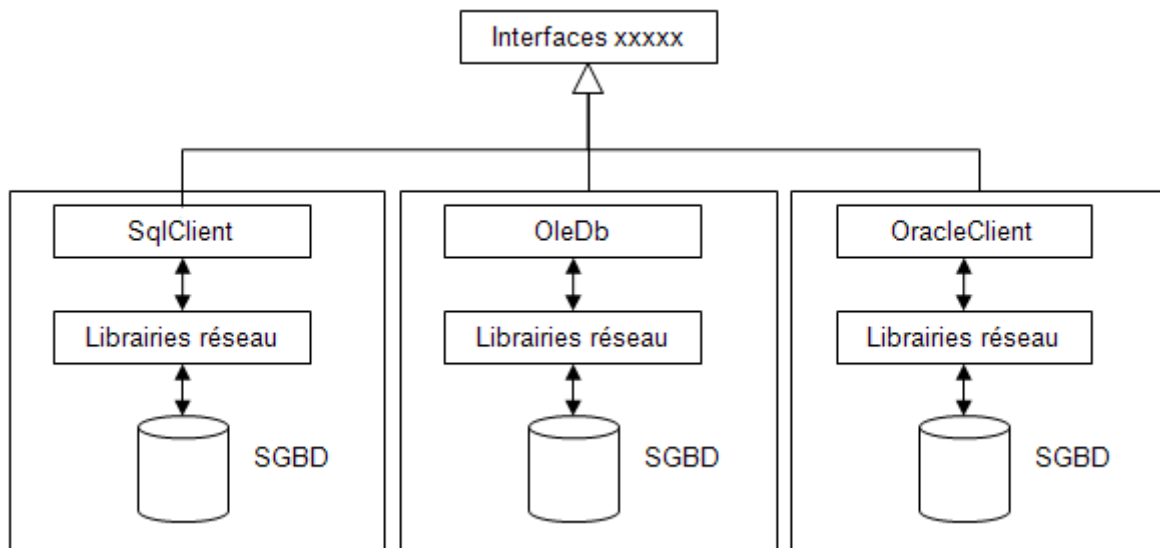
Nous venons d'examiner les API disponibles sous Delphi en Win32. Maintenant voyons ce qu'il en est en .NET.

Sous .NET, il existe un standard qui fait référence : ADO.NET. En fait, ADO.NET n'a rien à voir avec ADO. Sur son principe de fonctionnement, on est beaucoup plus proche de dbExpress ou JDBC que d'ADO.

ADO.NET gère l'indépendance vis-à-vis du SGBD complètement différemment. En fait, chaque classe est spécifique à un fournisseur. Il n'y a plus une couche applicative qui fait appel à des drivers spécifiques par SGBD.

A la place, des interfaces définissent les méthodes que chaque fournisseur doit implémenter. Si le code est écrit avec ces interfaces, l'application ne connaît pas le SGBD sous-jacent. On évite ainsi d'avoir une couche d'abstraction qui dégraderait les performances.

Ainsi l'architecture est la suivante :



Je crois qu'on peut difficilement faire plus simple. Les classes du namespace **SqlClient** (ou celles d'un autre provider) attaquent directement les bibliothèques réseaux pour encoder le protocole réseau. En retour, les données sont lues directement, en minimisant les conversions de données.

Cette architecture a toutes les chances d'offrir des performances optimales, bien meilleures que toutes les API qu'on a pu examiner jusqu'à présent.

Cependant, on a le même problème que pour dbExpress. La classe **SqlDataReader** ne fournit qu'un curseur unidirectionnel. En fait, elle décode le flux TDS au fur et à mesure qu'on avance dans la lecture des lignes.

II-F - ADO.NET : SqlDataReader et DataTable

Sur le même principe que pour dbExpress, si on veut travailler avec des curseurs bidirectionnels, il va falloir charger les données dans un cache en mémoire.

Il peut s'agir soit d'un **DataTable**, soit d'un **DataSet** (qui contiendra en fait un DataTable, donc ça revient au même).

Le chargement du **DataTable** à partir du **SqlDataReader** se fait par l'intermédiaire d'un composant **SqlDataAdapter**.

Sur le principe, l'architecture est donc similaire au **TClientDataSet**. Sauf que cette fois, les composants ont été développés spécifiquement pour cet usage. Le **SqlDataAdapter** va pouvoir charger directement le **DataTable** avec les résultats du **SqlDataReader**.

Au final les performances seront dégradées par rapport au **SqlDataReader**, probablement d'un facteur 2 mais peuvent rester excellentes.

III - Conclusion

Finalement, en Delphi Win32 on est bien mal loti. Les API dotNET ont tout pour donner de bien meilleurs résultats. On a pu constater des écarts très importants en termes d'empilement de couches successives d'une API à une autre. Chaque transition entre couches signifie une recopie et une transformation des données, donc une dégradation des performances.

dbExpress aurait pu être un progrès important. Cependant dans la pratique, l'implémentation n'est pas à la hauteur des promesses théoriques. Si le **TSqlDataSet** peut être aussi rapide qu'ADO natif dans la plupart des cas, l'utilisation du **TClientDataSet** annule complètement tout bénéfice en termes de performances.

Alors devons-nous tous migrer sous dotNET pour bénéficier des performances d'ADO.NET ? Ca peut être une solution mais ce n'est pas indispensable. Dans les prochains articles, nous verrons comment on peut s'approcher des performances d'ADO.NET grâce à un principe tout simple (mais beaucoup plus complexe à implémenter) : S'interfacer directement avec OLE-DB.

IV - Remerciements

Je remercie particulièrement **CI@udius** et **Pedro** pour leur relecture de l'article.