SYBASE®
An SAP® Company

WHITE PAPER

Getting Started: Advantage Database Server with Xbase++

By Roger Donnay*

*With Adsclass++ section written by Marcus Herz

TABLE OF CONTENTS

INTRODUCTION

A little history is provided here because many Xbase++ programs were originally developed and compiled under CA-Clipper, a DOS-based, 16-bit language. Advantage Database Server (ADS) provided the CA-Clipper community true client/server technology due to its compatibility with Clipper/Xbase++ databases such as .DBF/.DBT/.NTX and .DBF/.FPT/.CDX. Although Advantage Server includes robust support for SQL, no knowledge of SQL is required to enable client/server technology in your Xbase++ applications. For years, many Xbase programmers had erroneously equated "client/server" to SQL and assumed that they needed to move to a different platform to get this capability, this is not required. SQL is a client/server technology that is "set-based". Advantage supports SQL but also supports client/server technology that is ISAM or "record-based". Using SQL with the ADSBDE is covered later in this document.

Sybase developed a technology which allows Xbase++ programmers to get the advantages of client/server without changing any of their legacy code or requiring a new database structure. In CA-Clipper, this was accomplished by creating a new RDD named DBFNTXAX that was linked into the Clipper (Client) program by linking DBFAXS.LIB (RDD) and AXSCOMM.LIB (communications layer). When USEing a file, it was opened with the DBFNTXAX driver. To the user and application everything looks the same, but there is more activity being accomplished by the SERVER that would normally be accomplished by the CLIENT (the Clipper program).

For example, an INDEX ON command would send a request to the server to create the index rather than creating the index on the work station and writing the index back to the file server. This speeds up index creation and prevents index corruption. In addition, all files are opened under 1 handle therefore you never get a DOS ERROR 4 (out of file handles). Because ADS does much of its work on the SERVER there is reduced network traffic thereby improving performance on heavily used networks.

As Advantage Database Server technology became more popular over the years and more and more CA-Clipper programmers were converting their applications to VO, VB and Delphi, Sybase created a 32-bit client system that works with the same server software that was such a workhorse for CA-Clipper. This consists of two dlls ACE32.DLL (Advantage Client Engine, 32-bit) and AXCWS32.DLL (Communications Layer, 32-bit). This same set of client and communications .DLLs are used with all 32-bit languages, including Xbase++. The ACE32.DLL contains a client API which was used by Alaska Software to develop the ADSDBE. The ADSDBE is a database engine (DBE) that communicates with the ACE32.DLL. This is unlike other DBE's such as DBFNTX and DBFCDX which access databases directly. The ADSDBE accesses databases by making calls to functions in the ACE32.DLL. To you, the programmer, this is all transparent because your code does not need to change. Your USE, SKIP, GO TOP, SEEK, INDEX etc. commands and dbUseArea(), dbSkip(), dbGoTop(), dbSeek(), OrdSetFocus(), etc. functions behave as they did before.

INSTALLATION

The server software can be installed on Windows operating systems (XP and above), Linux operating systems (Kernel 2.4 and above) and Netware (5 and above). For best performance the Advantage Database Server should exist on the same server that is also the file server for all the databases. If the server is not running, an error will occur when trying to load the ADSDBE with the DbeLoad() function.

Another alternative for developers is to download the free ADSLOC32.DLL from the Advantage Developer Zone (DevZone.AdvantageDatabase.com) website. This is referred to as "Advantage Local Server" and is included with every client kit, for example the Advantage Client Engine API. It emulates the server software that is normally installed on the file server and allows you to run applications that use ADSDBE on a local drive. This is great for development when you are not attached to a network such as when you are travelling with a notebook computer.

Advantage extends the capabilities of Xbase++ with features like transaction processing, data encryption, SQL, scoping, filter optimization, etc. You can get access to these extended features by using the wrapper functions that start with ADS*() or AX_*() in ADS.PRG. Once you commit to using these extended functions then your application can no longer run without the ADSDBE and/or ACE32.DLL, this is why the ADSLOC32.DLL is a valuable tool for developers who need to continue development when they do not have access to a remote server.

Xbase++ and Advantage Database Server work very well together on Windows or Novell networks and they do not require IPX protocol as was necessary for CA-Clipper. They are designed to use either the IPX or IP protocol, therefore, they are quite compatible with wide-area networks. Because Advantage uses record-based databases (ie DBF/NTX) it is not recommended that you connect your client to your data over the Internet if your application requires a lot of browsing. If however, your application is designed to access only one record at a time, this can be a good Internet solution. It requires that you use the Internet connection type since it provides a secure connection between the client and the remote server. There are many Advantage Database applications, written in a variety of languages, that use the Internet client to attach to a remote database and they are quite impressive, however these applications were written entirely from scratch and were not migrated applications. They take into consideration the relatively low bandwidth of the Internet in the concept of the design and refrain from heavy use of browsing.

Software Required

Available from Alaska Software, Inc.

- Xbase++ version 1.7 or later for simple database usage.
- Xbase++ version 1.9 or later for advanced features like SQL queries.
- ADSDBE for Xbase++.

Available from Sybase, Inc. (DevZone.AdvantageDatabase.com)

- Advantage Client Engine which includes ACE32.DLL, AXWCS32.DLL and ADSLOC32.DLL

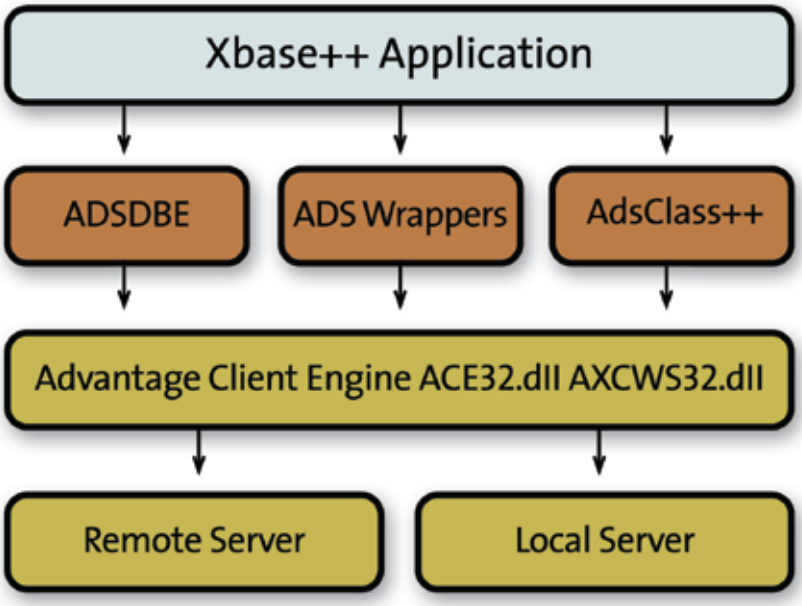
Server vs Client Compatibility

Generally the version of the Advantage client must be the same or older than the Advantage Database Server. When using older client DLLs new features introduced with the version of the ADS higher than the Client are not available. Sybase recommends that you use the same version of both the client and server for best performance and compatibility.

The ADSDBE installation includes versions of the Advantage client DLLs that probably will not match the most current version of Advantage server, therefore, it is necessary to always install the client DLLs matching the version of the Advantage Database Server. Also updating the client DLL at all Xbase++ application deployment sites is strongly recommend when updating the Advantage Database Server to a higher version.

Using Advantage from Xbase++

The below diagram shows how the Xbase++ program connects to legacy .DBF databases. The simplest method is to use the ADSDBE because it requires the fewest changes to legacy code, however it is also possible to manage data via a set of ADS wrappers which bypass the database engine and talk directly to the ACE client engine via direct DLL calls. When deploying applications, it is suggested that the Advantage Server software be installed on a remote server which is available to all Xbase++ client programs via a local area network. This is the best performance model for client/server applications. During testing, however, the ADSLOC32.DLL may be used to “emulate” the remote server or even to replace the remote server in installations where some users of the application may not need full client/server capability but the developer wishes to manage development costs by providing a single application binary that will work in all environments.



The ADSDBE

The ADSDBE is provided by Alaska Software[1]. It includes a help file that provides good documentation to aid in understanding client/server concepts and the Database Engine Overview.

The ADSDBE is the simplest method for developing or migrating Xbase++ applications with Advantage Database Server because it does not require knowing anything about the Advantage Client Engine (ACE) or its functions. It only requires small modifications to the code.

The ADSDBE is basically used like any other Xbase++ Database Engine (DBE). It must be loaded with the DbeLoad() function and can be defined with DbeSetDefault() as the default DBE. Once the ADSDBE is loaded, all commands and functions of the DML (Data Manipulation Language) are directed via the ADSDBE to the ADS and executed on the server station. The result of a DML command is then returned to the Xbase++ application, or client station.

Although the ADSDBE usage is quite similar to other Xbase++ DBEs, there are some differences which must be taken into account. The basic difference is that the ADSDBE consists of a data component, built in order component, dictionary component, and session component. Therefore, there is no need to use DbeBuild() to build a DatabaseEngine with order/index management capabilities.

Procedure DbeSys is used in this example, just as in any Xbase++ application, to load the required DBE. The ADSDBE allows access to files via a database server (ADS) which runs on a remote site. This requires the client station to establish a connection to the server before any file can be used. This is accomplished by the DacSession() class whose objects have the functionality to build a server connection and maintain a session while the client station is connected.

It is also a good idea to disconnect ADS from the client program before quitting the application. This should be done in an EXIT PROCEDURE as shown below to insure that the connection is broken even in the event of a runtime error.

[1] Alaska Software, Inc. – www.alaska-software.com

```
PROCEDURE DbeSys
  IF ! DbeLoad( "ADSDBE" )
    Alert( "Unable to load ADSDBE", {"Ok"} )
  ENDIF
  DbeSetDefault( "ADSDBE" )
  RETURN

PROCEDURE Main
  LOCAL cConnect := "DBE=ADSDBE;SERVER=\\ALASKA\VOL1"

  // store to public variable for use in other threads
  PUBLIC poAdsSession := DacSession():New( cConnect )

  IF .NOT. M->poAdsSession:isConnected()
    Alert( "Unable to establish connection to ADS", {"Quit"} )
    QUIT
  ENDIF

  StartUpProgram()

  RETURN

EXIT PROCEDURE DisconnectSession()

  M->poSession:disconnect()

  RETURN
```

The only difference in using the ADSDBE compared to the DBFNTX DBE, is that the ADSDBE requires the DacSession object to establish a connection to the ADS and to disconnect from the server. The code for database and index handling, or data manipulation, is the same.

In Xbase++ multi-threaded applications it is necessary to take the following precaution to prevent a runtime error. When a new thread is started, the :setDefault() method of the DacSession() object must be called before opening any databases, therefore, it is a good idea to store a pointer to the DacSession() object to a public variable as shown above. Also, make sure to close all databases before terminating a thread.

Example:

```
FUNCTION RunInNewThread( bBlock )

  M->poAdsSession:setDefault()

  Eval( bBlock )

  dbCommitAll()
  dbCloseAll()

  RETURN nil
```

It is important to note that very little application code needs to be changed when converting an application to using the ADSDBE. Basically, the ADSDBE must be loaded at the start of the application, and a DacSession() must be created to connect the ADSDBE to Advantage Server. File and index opening routines do not need to be touched. Multi-threaded applications also require a small modification at the start of each thread to insure that the work area inherits the DacSession() object, but that's all there is to it.

Supported ISAM functions

Before starting a migration project, it is important to understand that the 32-bit version of Advantage has some limitations over the 16-bit Clipper version, mostly in the way index keys are handled.

Advantage Database Server imposes some limitations on database and index file handling which do not exist in Xbase++. However, if an Xbase++ application uses the ADSDBE and instructs the ADS to create files on the server, the limitations must receive attention.

When a database command or function includes an expression that must be evaluated by the ADS, only a limited number of functions can be used in the expression. This applies to index creation, filters, relations and all database commands supporting a FOR clause. The following table list all supported functions[2]:

AllTrim()	IIf()	Round()	At()	Left()	RTrim()
Chr()	Len()	Space()	CtoD()	Lower()	Str()
Date()	LTrim()	StrZero()	Day()	Month()	SubStr()
Deleted()	Pad()	Time()	Descend()	PadC()	Today()*
DtoC()	PadL()	Transform()	DtoS()	PadR()	Trim()
Empty()	Rat()	Upper()	I2Bin()	Recno()	Val()
If()	Right()	Year()			

* Use Date() in Xbase++

The ADS expression engine cannot evaluate expressions containing other Xbase++ functions or user-defined functions. Also, only field variables or literal constants may be included in an expression evaluated on the server side of an application. If the ADS expression engine cannot evaluate an expression, a runtime error is raised. An exception is the definition of a filter expression. If a filter is used that cannot be evaluated by the Advantage Database Server, it will be evaluated locally on the client machine by the ADSDBE.

The above limitations are not considered problematic when designing new Xbase++ applications that use ADS because there are other design concepts that can be utilized that are often more efficient and more sustainable than using filters or user-defined functions, however they can cause some consternation when migrating existing applications. The functions supported by ADS can handle most requirements of an index. If your legacy indexes use functions not supported by ADS, you will probably find that there is an equivalent function or expression that will suffice as a replacement.

Database Concurrency

Advantage allows and manages "concurrent" database access by more than one application. In many of the legacy Xbase applications that are converted to Advantage, it is a requirement during the development and testing stages that the Xbase++ program which is accessing the databases via Advantage Database Server also work concurrently with old legacy Clipper programs that are accessing the same "shared" databases directly. This is a huge benefit to avoid deploying the new xbase++ application in a "big bang" to users, which often times leads to an excessive amount of support issues. Deploying the new application a little at a time allows for natural migration without user productivity dropping to learn a new application all at once. Concurrent access, is possible by making a simple

[2] See "Advantage Expression Engine" in the help file for the most current list of supported expression engine functions.

function call to AX_AXSlocking() to enable a compatible locking mode for shared use. This is a huge benefit,although this is possible, it is not recommended in the long term because it defeats the main purpose for using Advantage in the first place which is to increase performance and reliability. If old legacy programs are still accessing the databases directly, they can still cause file corruption. You achieve the maximum benefit from Advantage by enabling the proprietary locking scheme with a simple function call to AX_AXSLocking(.T.).

ADSDBE Configuration

The Xbase++ function DbeInfo() is used to retrieve and/or configure the current DatabaseEngine set with DbeSetDefault() . For example, the file extension for database files can be specified, or the locking scheme can be set. The most important settings when getting started with ADSDBE are the "Table Mode" and "Locking Scheme".

The table mode can be NTX, CDX or ADT. Legacy applications will most likely use index files with either the NTX or CDX extension, so DbeInfo() would be used as follows:

```
#include ADSDBE.CH

DbeSetDefault('ADSDBE')

DbeInfo( COMPONENT_DATA , ADSDBE_TBL_MODE, ADSDBE_NTX )
DbeInfo( COMPONENT_ORDER, ADSDBE_TBL_MODE, ADSDBE_NTX )

or

DbeInfo( COMPONENT_DATA , ADSDBE_TBL_MODE, ADSDBE_CDX )
DbeInfo( COMPONENT_ORDER, ADSDBE_TBL_MODE, ADSDBE_CDX )
```

The locking scheme can be set with AX_AXSLocking() or with DbeInfo() as follows:

```
DbeInfo( COMPONENT_DATA, ADSDBE_LOCK_MODE, ADSDBE_COMPATIBLE_LOCKING  )

or

DbeInfo( COMPONENT_DATA, ADSDBE_LOCK_MODE, ADSDBE_PROPRIETARY_LOCKING )
```

Additional configuration settings are also available with DbeInfo() such as the Index Extension, Memo File Extension, Memo Block Size, etc. See the ADSDBE help file from Alaska Software for more information.

ADS Wrappers

The full set of ADS*() functions included in ACE32.DLL can be called from within the Xbase++ program by creating a set of wrappers that use the DLLFUNCTION command of Xbase++. These functions are mostly not needed unless it is necessary to use features of Advantage which are not supported by the ADSDBE, such as data dictionary manipulation, SQL statement creation, or ADS Management.

Examples:

```
#include "DLL.CH"

DLLFUNCTION AdsSeek( nIndex, @cKey, cLength, nDataType, nSeekType, @lFound )
DLLFUNCTION AdsSetScope( nIndex, nScopeOption, cScope, nScopeLen, nDataType )
DLLFUNCTION AdsSkip( nTableHandle, nRecords )
```

Note: For faster performance, particularly when using functions in a loop, it is suggested that the ACEFUNCTION command be used in place of DLLFUNCTION. This keeps the ACE32.DLL always loaded.

```
#command  ACEFUNCTION <Func>([<x,...>]) ;
       => ;
FUNCTION <Func>([<x>]);;
STATIC scHCall ;;
IF scHCall == nil ;;
  IF snHdll == nil ;;
    snHDll := DllLoad('ACE32.DLL') ;;
  ENDIF ;;
  scHCall := DllPrepareCall(snHDll,DLL_STDCALL,<(Func)>) ;;
ENDIF ;;
RETURN DllExecuteCall(scHCall,<x>)

ACEFUNCTION AdsSeek( nIndex, @cKey, cLength, nDataType, nSeekType, @lFound )
ACEFUNCTION AdsSetScope( nIndex,  nScopeOption, cScope, nScopeLen, nDataType )
ACEFUNCTION AdsSkip( nTableHandle, nRecords )
```

A full set of Ads wrappers is available in a file named ADS.PRG. This file also contains a set of AX_*() functions for migration from CA-Clipper and a set of miscellaneous functions to make it easier to use ADS with Xbase++.

Additional AX_*() functions

The ADSDBE also includes a set of functions which may be called in your Xbase++ application that will utilize additional features of ADS.

- AX_AXSLocking() - Sets record locking mode
- AX_DBFEncrypt() - Encrypts an entire database
- AX_DBFDecrypt() - Decrypts an entire database
- AX_Encrypt() - Encrypts a string
- AX_Decrypt() - Decrypts a string
- AX_IDType() - Determine if current record is encrypted
- AX_RightsCheck() - Get or set security method to open files
- AX_SetPass() - Set password for encryption / decryption
- AX_TableType() - Determines with current database in encrypted

ADVANTAGE DATA DICTIONARIES

A data dictionary is the key to opening up an Xbase++ application to other applications and for providing SQL in the simplest possible form to Xbase++ programmers.

1. It combines the Xbase++ Work Area concept with the SQL SELECT statement.
2. It encapsulates the data and indexes of an application to improve performance when using SQL operations.
3. It provides a means for other applications, written in any language, i.e. ASP.NET, to access the Xbase++ application data via SQL statements.
4. It opens the application for use with third-party reporting tools such as Crystal Reports.
5. It provides an interface, via the ADS ODBC driver, to Microsoft applications such as Excel.

An Advantage data dictionary is simply a file that contains information about every database and index table in an application. The data dictionary can be created using the Advantage Architect (ARC32.EXE) or in Xbase++ code as follows:

```
cDictName := 'ABC.ADD'
cDictDesc := 'My Test Application'

nError := AdsDDCreate( cDictName, .F., cDictDesc), @nHandle )
IF nError == 0
```

```
  nError := AdsConnect60( cDictName, ;
                ADS_REMOTE_SERVER + ADS_LOCAL_SERVER, ;
                'AdsSys', ;
                '', ;
                ADS_DEFAULT, ;
                @nHandle )
  IF nError == 0
    aDirectory := Directory('*.DBF')
    FOR i := 1 TO Len(aDirectory)
      cTableName := Strtran(Upper(aDirectory[i,1]),'.DBF','')
        nError := AdsDDAddTable( nHandle, cTableName, cTableName+'.DBF', ;
                                  ADS_CDX, ADS_OEM, '', '' )
    NEXT
  ENDIF
ENDIF
```

The data dictionary is key to getting the most from Advantage SQL and the ADSDBE because dbUseArea() currently will not work with DacSession() objects that are FREE connections.

The above code example demonstrates creation of a data dictionary that references DBF/CDX databases. A data dictionary can also contain references to DBF/NTX databases however, there is a bit more code necessary to add the NTX files to the dictionary and associate them with the correct DBF files. For simplicity, the examples in this session will use DBF/CDX databases.

Viewing the Tables in a Data Dictionary

Having a browse or a tree-view of the tables in a data-dictionary is very convenient to both the application user and the programmer. If you do not want to give your users access to the data-dictionary via Advantage Architect, then here is a routine that will display all the tables in a tree-view and allow the user to double-click the item to browse the table. Simply substitute your own GUI browse for the Browse() function. You may also add your own security to prevent editing or viewing without proper permissions.

```
FUNCTION TableTree()

LOCAL aTables, nDictHandle, GetList[0], GetOptions, oDlg, oTree, ;
      nEvent, mp1, mp2, oXbp

nDictHandle := AdsSession():getConnectionHandle()

aTables := _BuildObjectArray( ADS_DD_TABLE_OBJECT, nDictHandle, .t. )
ASort(aTables,,,{|a,b|a[1]<b[1]})

oDlg := XbpDialog():new(AppDeskTop(),,{0,0},{400,600}):create()

oTree := XbpTreeView():new(oDlg:drawingArea,,{0,0},oDlg:drawingArea:currentSize())
oTree:haslines := .t.
oTree:hasButtons := .t.
oTree:alwaysShowSelection := .t.
oTree:setFontCompoundName('9.Lucida Console')
oTree:itemSelected := {|a,b,o|_TableBrowse(o)}
oTree:create()
```

```
_BuildTree(oTree,aTables)

nEvent := 0
DO WHILE nEvent # xbeP_Close
  nEvent := AppEvent(@mp1,@mp2,@oXbp,1)
  IF nEvent > 0 .AND. Valtype(oXbp) == 'O'
    oXbp:handleEvent(nEvent,mp1,mp2)
  ENDIF
ENDDO

RETURN nil

* --------------

STATIC FUNCTION _TableBrowse( oTree )

LOCAL cTableName

cTableName := Substr(oTree:getData():caption,1,10)
USE (cTableName)
DC_CrtRun({||Browse()})

RETURN nil

* --------------

STATIC FUNCTION _BuildTree( oTree, aTables )

LOCAL i, cString, oItem

FOR i := 1 TO Len(aTables)
  cString := Pad(aTables[i,1],10) + ' : ' + Alltrim(aTables[i,2])
  oItem := oTree:rootItem:addItem( cString )
  oItem:cargo := aTables[i,1]
NEXT

RETURN nil

* --------------

STATIC FUNCTION _TableDesc( cTableName, nDictHandle )

LOCAL nProperty, xValue, nLen, nError

nProperty := ADS_DD_COMMENT
xValue := Space(200)
nLen := 200

nError := AdsDDGetTableProperty( nDictHandle, ;
                                 cTableName, ;
```

```
                              nProperty, ;
                              @xValue, ;
                              @nLen )

RETURN Alltrim(xValue)

* --------------

STATIC FUNCTION _BuildObjectArray( nObject, nDictHandle )

LOCAL i, cData, nLen, nHandle, nError, aObject[0]

cData := Space(ADS_DD_MAX_OBJECT_NAME_LEN)
nLen := ADS_DD_MAX_OBJECT_NAME_LEN
nHandle := 0
nError := AdsDDFindFirstObject( nDictHandle, ;
                        nObject, ;
                        '', ;
                        @cData, ;
                        @nLen, ;
                        @nHandle )

IF nError == 0
  cData := Strtran(cData,Chr(0),'')
  AAdd( aObject, {Alltrim(cData),_TableDesc(Alltrim(cData),nDictHandle)})
ENDIF

DO WHILE nError == 0 .AND. nHandle > 0

  cData := Space(ADS_DD_MAX_OBJECT_NAME_LEN)
  nLen := ADS_DD_MAX_OBJECT_NAME_LEN
  nError := AdsDDFindNextObject( nDictHandle, ;
                        nHandle, ;
                        @cData, ;
                        @nLen )

  IF nError == 0 .AND. !Empty(cData)
    cData := Alltrim(Strtran(cData,Chr(0),''))
    AAdd( aObject, { Alltrim(cData), _TableDesc(Alltrim(cData),nDictHandle)})
  ENDIF

ENDDO
AdsDDFindClose(nHandle)

RETURN aObject
```

XBASE++ AND SQL

Advantage includes a SQL system that can be used to manipulate ISAM (Foxpro and Clipper DBF) databases using standard SQL techniques. This powerful feature of Advantage Server was mainly responsible for opening up the Advantage product line to a new world of possibilities.

It is well known that, over the years, SQL databases have become the choice of developers for industrial strength applications. This is mostly due to the fact that popular SQL databases are also client/server databases and the client/server model affords the advantages of security, performance and low risk of corruption. SQL was not an option in the early days of Advantage server because the language concept of SQL was mostly incompatible with ISAM databases and Xbase language programmers became quite skilled at managing their data via techniques available in the Xbase language. SQL is also a very different approach to data access than ISAM because it deals in data sets rather than providing the capability of browsing an entire database. Many Xbase developers found this to be a very limiting factor in the design of an application and were convinced that their customers would reject a redesign that took away the power and performance that made Xbase applications desirable. With the development of more powerful ISAM indexing techniques, scoping, and browse navigation methods, Xbase++ applications proved to be much better performers than SQL-based applications when browsing data.

On the other hand, SQL provides a much simpler programming interface for complex data operations such as queries and updates. It is also extremely fast at providing small datasets as opposed to the common "filtering" technique. It works well across multiple tables, provides a robust set of options and functions, does not require opening of databases and indexes, and does not require record locking. It is nearly impossible to corrupt data tables when using SQL.

As bridges started to be built between the Xbase world and the SQL world, Xbase and Delphi programmers found that a hybrid world was a much better place in which to live. The developers at Sybase found an opportunity to give our community of programmers the best of both worlds by developing both an ISAM and SQL Engine within ADS.

To understand how to use Advantage SQL with Xbase++, it must be understood that ADSDBE accepts cursor handles with the DbUseArea() function. What this means is that you can now browse or otherwise manipulate data without changing your code or coding technique.

Here is a simple example:

```
oSession - A pointer to the AdsSession created by DacSession()
nCursor - A pointer to the SQL Cursor created by AdsExecuteSQLDirect().

DbUseArea( nil, oSession, '<CURSOR>' + L2Bin(nCursor) + '</CURSOR' )

Browse()
```

In order to use this new feature, you must first connect to an Advantage Data Dictionary. This is accomplished in the same way that you would normally make an ADSDBE connection except that you pass the name of the dictionary file in the connect string rather than the drive letter.

```
DbeSetDefault( "ADSDBE" )

cDictServer := Curpath() + '\XWOLF.ADD'

cSession := "DBE=ADSDBE;SERVER=" + cDictServer

oSession := DacSession():new(cSession)
```

Creating a workarea in this manner combines the best of the ISAM and SQL worlds by allowing existing indexes to be used with a SQL dataset, thereby allowing OrdSetFocus() to instantly change the order of records in the dataset.

SQL and the USE command

SQLUse() is a function that uses the AdsStatement() class to create a SQL cursor from a statement and then applies it to a work area.

Creating a SQL statement is accomplished by using the TEXT INTO command of Xbase++. This allows for SQL statements to appear as inline code in an Xbase++ program.

AdsStatement:init() is used to create an instance of the AdsStatement() class. It receives the SQLStatement as a parameter and then calls the AdsStatement:open() method. The open() method uses ADS API functions to verify that the SQL statement is valid and has no syntax errors. The AdsStatement:execute() method uses ADS API functions to convert the statement to a cursor handle which is then used by the Xbase++ dbUseArea() function to apply the SELECT statement to a work area.

The work area can then be navigated using standard Xbase++ navigation functions.

Example:

```
FUNCTION Test

LOCAL oStatement, cStatement

TEXT INTO cStatement WRAP

SELECT
invoice.invnmbr,
invoice.balance,
customer.custnmbr,
customer.billname,
customer.billstrt,
customer.billcity,
FROM customer, invoice
WHERE invoice.custnmbr = customer.custnmbr

ENDTEXT

oStatement := SQLUse( cStatement, 'INVOICES' )

Browse()

oStatement:close()

RETURN nil

*-----------

FUNCTION SQLUse( cStatement, cAlias )

LOCAL oStatement

oStatement := AdsStatement():New(cStatement,AdsSession())
IF oStatement:LastError > 0
  RETURN .f.
```

```
ENDIF

oStatement:Execute(cAlias)

RETURN oStatement

* ---------

CLASS AdsStatement

EXPORTED:

VAR Handle, Statement, Alias, Session, Cursor, LastError

INLINE METHOD GetLastError()
RETURN(::LastError)

* -------------

INLINE METHOD Init(cStatement, oSession)

IF(ValType(oSession)!="O")
  MsgBox( 'Parameter Type error : oSession' + Chr(13) + ;
          '(passed to AdsStatement:Init())' )
  ::LastError := 3
  RETURN Self
ENDIF
IF(!oSession:IsDerivedFrom("DacSession"))
  MsgBox( 'Parameter passed is not a DacSession : oSession' + chr(13) + ;
          '(passed to AdsStatement:Init())' )
  ::LastError := 4
  RETURN Self
ENDIF
::Session := oSession

RETURN ::Open(cStatement)

* ------------

INLINE METHOD Close()

IF(::HANDLE==NIL)
  RETURN(.F.)
ENDIF

IF (Used(::Alias))
  (::Alias)->(DbCloseArea())
ENDIF

::LastError := AdsCloseSQLStatement( ::HANDLE)
```

```
::Statement := NIL
::HANDLE    := NIL
::Alias     := NIL

RETURN .t.

* -------------

INLINE METHOD Open(cStatement)

LOCAL nH, nError, nErrorLen, cErrorString

IF ValType(cStatement)!="C"
  MsgBox( 'Parameter Type Invalid : Statement' + Chr(13) + ;
               '(passed to AdsStatement:Open())' )
  ::LastError := 1
  RETURN self
ENDIF
IF(Upper(Left(cStatement,Len(KEYWORD_SELECT)))!=KEYWORD_SELECT)
  MsgBox( 'Unsupported SQL statement' + Chr(13) + ;
          '(passed to AdsStatement:Open())' )
  ::LastError := 2
  RETURN self
ENDIF
::Statement := cStatement

nH := 0x0
::LastError := AdsCreateSQLStatement( ::Session:getConnectionHandle(), @nH )
::HANDLE := nH
IF ::LastError > 0
  cErrorString := _AdsGetLastError()
  MsgBox(cErrorString)
ELSE
  ::LastError := AdsVerifySQL( nH, cStatement )
  IF ::LastError > 0
    cErrorString := _AdsGetLastError()
    MsgBox(cErrorString)
  ENDIF
ENDIF

RETURN self

* --------------

INLINE METHOD Execute( cAlias )

LOCAL rc := 0x0, nCursor := 0x0, cErrorString, nErrorLen, nError

::LastError := AdsExecuteSQLDirect( ::HANDLE , ::Statement , @nCursor )
IF ::LastError > 0
```

```
    cErrorString := _AdsGetLastError()
    MsgBox(cErrorString)
    RETURN ''
  ENDIF

  DbUseArea( ,::Session, "<CURSOR>"+L2Bin(nCursor)+"</CURSOR>",cAlias)

  IF (Used())
    ::Alias := Alias()
    ::Cursor := L2Bin(nCursor)
  ENDIF

  RETURN (::Alias)

  ENDCLASS

  * --------------

  FUNCTION _AdsGetLastError()

  LOCAL cErrorString, nErrorLen, nError

  cErrorString := Space(500)
  nErrorLen := 500
  nError := 0
  AdsGetLastError(@nError,@cErrorString,@nErrorLen)
  cErrorString := Strtran(Pad(cErrorString,nErrorLen),';',Chr(13))

  RETURN cErrorString
```

The ExecuteSQL() and ApplySQLParams() functions

ExecuteSQL() is a function that is used to execute any SQL statement, however it should be used only for statements that do not return a dataset but instead perform data operations that make updates to data tables, such as updating structures, global replaces, etc. ExecuteSQL() can be used to execute statements embedded in code or to execute *.SQL files.

ApplySQLParams() is a function that is used to pass parameters into the SQL statement.

```
  FUNCTION ExecuteSQL( cStatement, cStatus )

  LOCAL nIndexMode := ADS_CDX, nHandle, nStatementHandle, nVerify, ;
        nLockingMode := ADS_PROPRIETARY, nCursorHandle

  oSession := AdsSession()

  nHandle := oSession:getConnectionHandle()

  nStatementHandle := 0
  AdsCreateSQLStatement( nHandle, @nStatementHandle )

  AdsStmtSetTableType( nStatementHandle, nIndexMode )
```

```
AdsStmtSetTableLockType( nStatementHandle, nLockingMode )

nVerify := AdsVerifySQL( nStatementHandle, cSelect )
IF nVerify > 0
  cStatus := _AdsGetLastError()
  RETURN .f.
ENDIF

nCursorHandle := 0
AdsExecuteSQLDirect( nStatementHandle, cSelect, @nCursorHandle )
IF nCursorHandle = 0
  cStatus := _AdsGetLastError()
  RETURN .f.
ENDIF

RETURN .t.

* -----------

FUNCTION ApplySQLParams( cSqlStatement, aParams )

LOCAL i, cValue, cDate, xValue, nError

FOR i := 1 TO Len(aParams)
   IF aParams[i] == NIL
      LOOP
   ENDIF
   xValue := aParams[i]
   IF Valtype(xValue) == 'N'
      cValue := Alltrim(Str(xValue))
   ELSEIF Valtype(xValue) == 'L'
      IF xValue
         cValue := 'TRUE'
      ELSE
         cValue := 'FALSE'
      ENDIF
   ELSEIF Valtype(xValue) == 'D'
      cDate := DtoS(xValue)
      cDate := Ltrim(Rtrim(SubStr(cDate,1,4) + '-' + ;
         SubStr(cDate,5,2) + '-' + ;
         SubStr(cDate,7,2)))
      cValue := "{ d '" + cDate + "'}"
   ELSEIF Valtype(xValue) == 'C'
      cValue := "'" + xValue + "'"
   ELSE
      cValue := Trim(DC_XtoC(xValue))
   ENDIF
   cSqlStatement := StrTran(cSqlStatement,'?',cValue,,1)
NEXT

RETURN cSqlStatement
```

Using ExecuteSQL() to Modify a Database structure

Here is an example of a SQL file that is used to add a new field named RECORD_ID to all databases.

```
/* Record_ID.SQL */
ALTER TABLE APERAKT ADD COLUMN RECORD_ID Char( 20 )    ;
ALTER TABLE ARDUMP ADD COLUMN RECORD_ID Char( 20 )     ;
ALTER TABLE AUTODATA ADD COLUMN RECORD_ID Char( 20 )   ;
ALTER TABLE AUTOS ADD COLUMN RECORD_ID Char( 20 )      ;
ALTER TABLE BINLABEL ADD COLUMN RECORD_ID Char( 20 )   ;
ALTER TABLE BLDPOOL ADD COLUMN RECORD_ID Char( 20 )    ;
ALTER TABLE CAN_JOB ADD COLUMN RECORD_ID Char( 20 )    ;
.... 80 more tables
```

And here is the code that will execute this SQL file and modify the structure of every database in the data dictionary.

```
cSql := MemoRead('Record_Id.SQL')
cStatus := ExecuteSQL( cSql )
? cStatus
```

Using ExecuteSQL() to change Account numbers

Imagine a scenario in which your customer asked you to write a routine that would change all customer account numbers to the customer phone number. This is a common requirement in modern applications and using SQL is the most effectively way to accomplish this task in a reasonable time-frame. The ripple effect of changing customer account numbers on 100,000 plus customers could be enormous in large applications and could take hours of programming plus hours more to run and test the code. This is because every relational database with a customer account number also needed to be changed to maintain relational integrity. In legacy applications, it is possible that many databases containing millions of records would be affected by this change. Here is an example of how this would be done using SQL:

First, we add a new field to ACCTS.DBF to store the old account number.

```
TEXT INTO cStatement
ALTER TABLE ACCTS ADD COLUMN OLD_ACCT Char( 10 )
ENDTEXT

ExecuteSQL( cStatement )
```

Next, we create a file named ACCTNO-UPDATE.SQL that contains the following SQL code:

```
BEGIN TRANSACTION ;
UPDATE ACCTS     SET old_acct = '%1' WHERE acct_no = '%1'   ;
UPDATE ACCTS     SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE ACCTCROS SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE AUTOS     SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE BLDPOOL   SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE CJPARTS   SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE CORES     SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE CUSTDUMP SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE DELRO1    SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE DELROP    SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE DEPOSITS SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE GPTAGSPD SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE LEADS     SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE LOSTSALE SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE MAILIN    SET acct_no = '%1' WHERE acct_no = '%2'    ;
```

```
UPDATE MAILLST  SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE RO       SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE RMA      SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE RMAPARTS SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE ORDERS   SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE PREMO    SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE PT       SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE PTPARTS  SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE RECOMREP SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE ROPARTS  SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE SALESTAX SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE TLCLIENT SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE TRANSM   SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE WARRANTY SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE COREBANK SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE MO       SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE TRANYOYO SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE SHIPMAST SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE SHIPACCT SET cust_acct = '%1' WHERE cust_acct = '%2' ;
UPDATE CONTACTS SET acct_no = '%1' WHERE acct_no = '%2'    ;
UPDATE CLAIMS   SET acct_no = '%1' WHERE acct_no = '%2'    ;
COMMIT WORK
```

Finally, we created a routine that steps through every record in ACCTS.DBF and executes the above code as follows:

```
cSQL := MemoRead( 'ACCTNO-UPDATE.SQL' )

USE ACCTS
nExceptions := 0
nUpdated := 0

DO WHILE !ACCTS->(Eof())
  cOldAcctNo := ACCTS->acct_no
  cNewAcctNo := Strtran(ACCTS->phone_no,'-','')
  cSQLUpdate := Strtran( cSQL, '%1', cNewAcctno )
  cSQLUpdate := Strtran( cSQLUpdate, '%2', cOldAcctNo )
  ? 'Updating ' + cOldAcctNo + ' to ' + cNewAcctNo
  IF Empty(cNewAcctNo)
    Exception( 'ACCTNO_UPDATE_PHONE', cOldAcctNo, cNewAcctNo,,, ;
               'Phone Number is Empty' )
    nExceptions++
  ELSEIF cNewAcctNo == cOldAcctNo
    // do nothing
  ELSEIF Len(cNewAcctNo) # 10
    Exception( 'ACCTNO_UPDATE_PHONE', cOldAcctNo, cNewAcctNo,,, ;
               'Phone Number is not 10 digits')
    nExceptions++
  ELSEIF ACCTS->(dbSeek(Pad(cNewAcctNo,10)))
    Exception( 'ACCTNO_UPDATE_PHONE', cOldAcctNo, cNewAcctNo,,, ;
               'Account Number Already Exists' )
    nExceptions++
  ELSEIF !ExecuteSQL( cSQLUpdate, @cStatus )
    Exception( 'ACCTNO_UPDATE_PHONE', cOldAcctNo, cNewAcctNo,,, ;
```

```
                cStatus )
    nExceptions++
  ELSE
    nUpdated++
  ENDIF
  ExecuteSQL( cSQLUpdate )
  ACCTS->(dbSkip())
ENDDO
ACCTS->(dbCloseArea())
? 'Accounts Updated = ', nUpdated
? 'Exceptions = ', nExceptions, 'Look at the EXCEPTION table'
```

Using ExecuteSQL() to add a new record to a database

It is suggested that a database be used for log exceptions. Appending a record using SQL requires no record locking. Here we use ExecuteSQL() and ApplySQLParams():

```
FUNCTION Exception( cType, cRef1, cRef2, cRef3, cRef4, cErrorInfo )

LOCAL cStatement, aParams, cStatus

DEFAULT cType := 'UNKNOWN', ;
       cRef1 := '', ;
       cRef2 := '', ;
       cRef3 := '', ;
       cRef4 := '', ;
       cErrorInfo := ''

cRef1 := Strtran(cRef1,"'",'')
cRef2 := Strtran(cRef2,"'",'')
cRef3 := Strtran(cRef3,"'",'')
cRef4 := Strtran(cRef4,"'",'')

cErrorInfo := Strtran(cErrorInfo,"'","''")

TEXT INTO cStatement WRAP
INSERT INTO EXCEPTION ( type, ref1, ref2, ref3, ref4, errorinfo, date, time )
   VALUES ( ?, ?, ?, ?, ?, ?, ?, ? ) ;
ENDTEXT

aParams := { cType, cRef1, cRef2, cRef3, cRef4, cErrorInfo, Date(), Time() }

cStatement := ApplySQLParams( cStatement, aParams )

ExecuteSQL( cStatement, @cStatus )

RETURN nil
```

Using SQL files instead of Code

*.SQL files contain SQL commands that can be executed with the ExecuteSQL() function as follows:

```
cSQL := MemoRead('UPDATE.SQL')
ExecuteSQl( cSQL )
```

Many times it is much more convenient, many times more powerful and also much faster to write SQL code than to try to accomplish the same task in Xbase++ code. In legacy applications it is often required to provide a more "normalized" solution rather than using a single database. In the below example, the master parts file (PARTS) has fields for Department OnHand quantities which need to be placed into a new child database (PARTDEPT). This involves creating a new table named PARTDEPT, adding new index tags to PARTDEPT, transferring the data from the PARTS table to PARTDEPT, then modifying the structure of PARTS by deleting and adding fields, then deleting the unused index tags in PARTS.

Here is the SQL file that will accomplish this task: /* PARTDEPT-CREATE.SQL

This script creates a new child table named PARTDEPT and transfers values in the following fields from PARTS to PARTDEPT:

```
 onhand -> onhand
 minlim -> minlim
 maxlim -> maxlim
 cost -> cost    // renamed this field in PARTS.DBF to std_cost
 onhand_xxx -> onhand
 minlim_xxx -> minlim
 maxlim_xxx -> maxlim
 cost_xxx -> cost

 renamed bin to std_bin */

CREATE TABLE PARTDEPT (
       PART_NO Char( 20 ),
       VENDOR Char( 4 ),
       DEPT_NO Numeric( 3, 0 ),
       ONHAND Numeric( 6 ,0 ),
       MINLIM Numeric( 6 ,0 ),
       MAXLIM Numeric( 6 ,0 ),
       COST Numeric( 11 ,3 ),
       BIN Char( 10 ),
       LSTYR_USED Numeric( 7, 0 ),
       YTD_USED Numeric( 7, 0 )) ;

EXECUTE PROCEDURE sp_CreateIndex(
   'PARTDEPT',
   'PARTDEPT.cdx',
   'PART_NO',
   'PART_NO+VENDOR+Str(DEPT_NO,3)',
   '',
   2,
   512 );

EXECUTE PROCEDURE sp_CreateIndex(
   'PARTDEPT',
   'PARTDEPT.cdx',
   'DEPT',
   'Str(DEPT_NO,3)+PART_NO+VENDOR',
   '',
   2,
   512 );
```

```
EXECUTE PROCEDURE sp_CreateIndex(
    'PARTDEPT',
    'PARTDEPT.cdx',
    'BIN',
    'Str(DEPT_NO,3)+BIN+PART_NO+VENDOR',
    '',
    2,
    512 );

INSERT INTO PARTDEPT
( PART_NO,
  VENDOR,
  DEPT_NO,
  ONHAND,
  MINLIM,
  MAXLIM,
  COST,
  BIN )
SELECT
  PART_NO,
  VENDOR,
  0,
  ONHAND,
  MINLIMIT,
  MAXLIMIT,
  COST,
  BIN
FROM PARTS
WHERE ONHAND > 0 OR MINLIMIT > 0 OR MAXLIMIT > 0 OR BIN IS NOT NULL;

INSERT INTO PARTDEPT
( PART_NO,
  VENDOR,
  DEPT_NO,
  ONHAND,
  MINLIM,
  MAXLIM,
  COST )
SELECT
  PART_NO,
  VENDOR,
  1,
  ONHAND_001,
  MINLIM_001,
  MAXLIM_001,
  COST_001
FROM PARTS
WHERE ONHAND_001 > 0 OR MINLIM_001 > 0  OR MAXLIM_001 > 0 ;

/* Duplicate "INSERT" above for 002 thru 030 */
```

```
DROP INDEX parts.part_oh   ;
DROP INDEX parts.part_bin  ;
DROP INDEX parts.part_bin2 ;
DROP INDEX parts.part_bin3 ;
DROP INDEX parts.part_low  ;
DROP INDEX parts.part_low2 ;
DROP INDEX parts.part_mxl  ;
DROP INDEX parts.pct_low   ;
DROP INDEX parts.pct_low2  ;
DROP INDEX parts.part_sld  ;

ALTER TABLE PARTS
   ALTER COLUMN cost std_cost Numeric( 9, 3)
   ALTER COLUMN bin std_bin Char( 10 )
   ADD COLUMN record_id Char( 20 )
   DROP ONHAND
   DROP ONHAND_001
   /* Duplicate "DROP" above for ONHAND_002 thru ONHAND_030 */
   DROP MINLIMIT
   DROP MINLIM_001
   /* Duplicate "DROP" above for MINLIM_002 thru MINLIM_030 */
   DROP MAXLIMIT
   DROP MAXLIM_001
   /* Duplicate "DROP" above for MAXLIM_002 thru MAXLIM_030 */
   DROP COST_001
   /* Duplicate "DROP" above for COST_002 thru COST_030 */
   DROP BIN2
   DROP BIN3
   DROP LASTYR
   DROP SOLD_YR_TD ;
```

Another advantage to using SQL code is that the code can be run inside the Advantage Architect. The application does not need to be running to write and test the SQL routine. Once the routine is working as required, it can then be called in the application with ExecuteSQL().

CONVERTING EXISTING XBASE++ APPLICATIONS

The ADSDBE is the simplest method for converting an existing application to Advantage Database Server and only requires a few simple steps.

1. Locate the following Alaska Software Compile-time files and make sure they are in your INCLUDE path and LIB path: ADSDBE.CH, ADSUTIL.LIB

2. Locate the following Alaska Software Run-time files and make sure they are in your PATH: ADSDBE.DLL, ADSUTIL.DLL.

3. Locate the following Sybase Run-time files and makes sure they are in your PATH: ACE32.DLL, AXCWS32.DLL, ADSLOC32.DLL.

4. Locate DBESYS.PRG (modified). A ready-to-use version is available in Code Central on DevZone.AdvantageDatabase.com. Add this file to your project file.

5. If you are currently using CDX indexes in your application, nothing else needs to be done to DBESYS.PRG. If you are currently using NTX indexes in your application, make the following change: #define INDEX_TYPE "NTX".

6. If you want ADS to open databases in compatible locking mode, nothing else needs to be done to DBESYS.PRG. If you want ADS to open databases in proprietary locking mode, make the following change: #define COMPATIBLE TRUE.

7. If your existing application is multi-threaded, you will need to add 1 line of code to the start of each thread: AdsSession():setDefault(). AdsSession() is a Get-Set function located in DBESYS.PRG. It returns a pointer to the session object. The setDefault() method insures that the current thread has access to the session object.

8. Locate the following Sybase file: ADS.INI. This file is used to establish the kind of connection to make to Advantage Database Server. Place this file in the same directory as your application .EXE. During testing, it is not necessary to run the remote server. Instead, you can use local server. This can be changed in ADS.INI.

9. Compile and run your application. All databases and indexes should now be opened with ADSDBE.

SAMPLE PROGRAMS AVAILABLE

The above steps are all that is needed for the most rudimentary conversion of an existing Xbase++ application to a client/server application, however there are many more features of Advantage that can improve the performance, functionality and reliability of an Xbase++ application.

A set of sample and utility programs are available in Code Central on DevZone.AdvantageDatabase.com. Some of these programs use only the ADSDBE for access to Advantage and others use ADS wrappers to call ACE functions. Read the README.TXT file for a complete description of the sample programs.

ADSCLASS++

AdsClass++ is provided by DS-Datasoft[3]. It includes code samples, a help file that provides good documentation and the developer version is shipped with the source code.

AdsClass++ is an entirely object oriented solution based only on the Advantage Client Engine ACE32.DLL and the unique ability of Xbase++ to declare classes dynamically at runtime.

These classes enable the usage of all ADS functions as
- all table types: NTX,CDX,VFP,ADT,
- all native ADS field types,
- data dictionary, setting and configuration
- rawkey indices, key expression concatenating different field types,
- NULL values,
- AOF filters,
- easy SQL statement and cursor implementation,
- ADS management,
- high performance due to direct API communication,
- all collations, and so on ...

The basic idea is to create a class for every table or SQL statement declaring ACCESS ASSIGN methods to read from and write to table fields. All Xbase++ workarea commands and functions and all additional ADS specific functions are mapped to class methods. A table is then represented only by this object and not by any workarea. There is no difference between a table dataset and a SQL result set neither in syntax nor functionality.

The object capsulates all handling to this table or SQL. That way the same table can be opened several times in different objects each separated from the others regarding navigation, filtering, indexing, scoping, etc. As AdsClass++ works with handles the same connection can be used in all threads. Another advantage with objects is that they are

[3] DS-Datasoft GmbH&Co.KG, www.ds-datasoft.com

variables (specific data type, can be passed as parameters, visible in all threads, declaration, compiler warnings,...). Working with objects rather than workareas is a quite innovative step as it makes it easier to read and maintain code especially working in a team. This concept can be extended to map any Xbase++ DBE in equivalent classes.

Class overview:

- dsAceSession, connect to free tables (NTX,CDX,VFP,ADT),
- dsAceDD, connect to a data dictionary (ADD), access and configure meta information,
- dsAceServer, class to access a table,
- dsAceSqlServer, class for SQL statements,
- and more (but would exceed this short presentation).

Examples connecting:

```
#include "ACEXBP.CH"

PROCEDURE DbeSys
// no need to load any DBE here
// ACE32.DLL is loaded by INIT PROCEDURE in xclace.dll
RETURN

PROCEDURE MAIN
LOCAL oAdsConnect

// free tables
oAdsConnect := dsAceSession():New()
oAdsConnect:connect("\\ALASKA\VOL1",.F.,;
                    ADS_REMOTE_SERVER+ADS_LOCAL_SERVER)
// some settings
oAdsConnect:TableType := ADS_CDX  // ADS_NTX, ADS_VFP, ADS_ADT
oAdsConnect:Chartype  := ADS_OEM  // ADS_ANSI
oAdsConnect:LockType  := ADS_PROPRIETARY_LOCKING
                                     // ADS_COMPATIBLE_LOCKING
AdsSetDefault(<path>)
AdsSetSearchPath(<path1;path2;...>)

// or ADD
oAdsConnect := dsAceDD():New()
oAdsConnect:connect("\\ALASKA\VOL1\MYSHARE\DEMO.ADD","ADSSYS",;
                    <password>, ADS_DEFAULT,;
                    ADS_REMOTE_SERVER+ADS_LOCAL_SERVER)
//
IF !oAdsConnect:IsConnected()
  Msgbox("Unable to establish connection to ADS: "+;
    var2char(AdsGetLastError())
  QUIT
ENDIF

// more settings
AdsCacheOpenCursor(<number>)
AdsCacheOpenTables(<number>)
oAdsConnect:CacheRecords(<number>)

// declare session for application wide access
```

```
AppDic(oAdsConnect)

...
// end of main
RETURN

// disconnect
EXIT PROCEDURE AdsDisconnect
  AppDic():disconnect()
RETURN
```

Examples working with tables (either free tables or data dictionary tables, any table type):

```
LOCAL dbTable

dbTable := AppDic():openServer("customer.dbf")
// some navigation
dbTable:goTop()
DO WHILE !dbTable:EOF()
  cName := dbTable:NAME           // field access
  IF dbTable:RLock()
    dbTable:NAME := cName         // field write
    dbTable:UnLock()
  ENDIF
  dbTable:Skip()
ENDDO

// rawkey index, key expression with different field types
// enhances performance with SQL execution significantly
// only ADT tables
dbTable:ordCreate(<cIndexname>,<cTagName>,;
  "stringFieldname;dateFieldname;numericFieldname")
// pass different datatypes to seek according to indexkey
dbTable:Seek({"Smith",date(),10.1})

// filter evaluated at client side, function call
dbTable:setFilter({|dbTable | myFilter(dbTable)})
// filter evaluated on server side
dbTable:setFilter("name='SMITH'")
// AOF Filter, only evaluated on server side
dbTable:setAOF("name='SMITH'")
// top and bottom Scope
dbTable:setScope(,"SMITH")

dbTable:Close()
```

Examples using SQL:

```
LOCAL cSql
LOCAL dbSql

// simple select
TEXT INTO cSql WRAP
Select * from customer
```

```
where City like 'B%';
ENDTEXT

dbSql := OpenSqlServer(,cSql, AppDic())
// dbSql is, with no difference, treated like any table object
// including ordcreate on a cursor result set !
// see dbTable examples above

// SQL with parameters
sqlConnection := dsAceSqlServer():New(AppDic())
sqlConnection:prepare("Select * from customer where City=:P1")
sqlConnection:setparam("P1", "Boise" )

dbSql := sqlConnection:Execute2Server()    // get result set

// set new parameter and refresh result
sqlConnection:setparam("P1", "San Francisco" )
dbSql:RefreshSql()                          // new result set


dbSql:close()
sqlConnection:close()

// data manipulation
ExecuteSql(AppDic(),"Update customer set flag=true where ...")
```

Examples using data dictionary:

```
LOCAL aUser  // { computername, login name, ip-adress, ... }

// get array with all connected users
aUser := GetUsernames(AppDic())

// get user with record lock
aUser := GetAdsLockOwner(<cFile>, <nRecno>)

// create new user
AppDic():CreateUser([<cGroup>], <cUser>, <password> )

// disconnect user
AdsKillUser(<cUser>)
```

Examples editing and browsing:

```
LOCAL dbTable

dbTable // browsing
::oBrowse := XbpBrowse():new(::drawingarea,,{0,0},::currentsize())
::oBrowse:create()

// new column
::oBrowse:addColumn({|x| iif(x==NIL, dbTable:NAME,;
                                     dbTable:NAME := x)})
```

```
// navigation codeblock
::oBrowse:skipBlock      := {|n| DbSkipper(n, dbTable)}
::oBrowse:goTopBlock     := {|| dbTable:GoTop()}
::oBrowse:goBottomBlock  := {|| dbTable:GoBottom()}
::oBrowse:phyPosBlock    := {|| dbTable:Recno()}

// codeblock for vertical scrollbar.
::oBrowse:posBlock       := {|| dbTable:Position()*10}
::oBrowse:goPosBlock     := {|n| dbTable:GoPosition(n/10)}
::oBrowse:lastPosBlock   := {|| 1000}
::oBrowse:firstPosBlock  := {|| 0}

//=========================================
FUNCTION DbSkipper( nWantSkip, dbTable)
LOCAL nDidSkip := 0
DO CASE
CASE dbTable:LastRec() == 0  // empty table

CASE nWantSkip == 0           // reread record
  dbTable:Skip(0)

CASE nWantSkip > 0 .AND. !dbTable:Eof()
 // skip next
 DO WHILE nDidSkip < nWantSkip
   dbTable:Skip(1)
   IF dbTable:Eof()
     // ghost record, append mode
     EXIT
   ENDIF
   nDidSkip++
 ENDDO

CASE nWantSkip < 0
  // skip back
  DO WHILE nDidSkip > nWantSkip
    dbTable:Skip(-1)
    IF dbTable:Bof()
      EXIT
    ENDIF
    nDidSkip--
  ENDDO
ENDCASE
RETURN nDidSkip
```

Example converting Xbase++ workarea code to AdsClass++:

```
// Existing workarea code:

LOCAL aOrder:={}

USE customer alias cust
SET INDEX TO custno, custname
```

```
USE order ALIAS order
SET INDEX TO ordno, custno

SELECT cust
SEEK "123"

SELECT order
SET ORDER TO custno
SEEK "123"
DO WHILE ! EOF() .AND. field->custno == "123"
  aadd(aOrder, recno())
  SKIP
ENDDO
USE
SELECT cust
USE

// Converted to AdsClass++ code:

LOCAL dbCust, dbOrder
LOCAL aOrder:={}

dbCust  := AppDic():OpenServer("customer")
dbOrder := AppDic():OpenServer("order")

dbCust:seek("123")

dbOrder:ordSetFocus("custno")
dbOrder:seek("123")

DO WHILE ! dbOrder:eof() .AND. dbOrder:custno == "123"
  aadd(aOrder, dbOrder:recno())
  dbOrder:skip()
ENDDO
dbOrder:close()
dbCust:Close()
```

These examples are a short introduction of AdsClass++. To learn more about details or to test the performance simple download a fully functional evaluation version from www.ds datasoft.com . Combining AdsClass++ with XClass++ for the GUI will also speed up new application development. Due to support of rawkey indices it enables easy migration from DBF to ADT resulting in a significant performance boost.

ADDITIONAL ITEMS

The Advantage Architect

The Advantage Data Architect (ARC) is a valuable resource for maintaining a data dictionary and also for testing SQL SELECT statements and other SQL operations. There are sufficient Ads*() functions in ACE32.DLL to write the Advantage Data Architect entirely in Xbase++ code, and this may be a useful exercise to learn more about ADS, but it is entirely unnecessary because the Advantage Data Architect provides all necessary functionality.

ARC can be used for SQL operations, modifying the data-dictionary, modifying the structure of databases and indexes, browsing/editing data tables, setting permissions, etc. It is important to use this resource when converting an Xbase++ application to run from a data dictionary. ARC can be downloaded for free form the Advantage Developer Zone (DevZone.AdvantageDatabase.com)

Accessing Application Data from Third-Party Apps

A significant advantage to using a data dictionary in your application is the way it opens up your application data to other third party applications, such as ASP.NET, Crystal Reports, Visio, Excel, etc.

When the Advantage ODBC driver is installed, third-party applications that support ODBC will see your application data as an ODBC data source.

CONCLUSION

Xbase++ legacy applications can be given new life by utilizing inexpensive resources available in the ADSDBE from Alaska Software and Advantage Database Server from Sybase. Advantage provides the performance and reliability of client/server technology and the ability to apply the most advanced features of SQL without any redesign of an existing application. Legacy code can be maintained and supported even while adding new features like transaction processing, SQL queries, data replication and data encryption. Legacy databases can be queried and updated using the most advanced SQL techniques or the more familiar ISAM techniques thereby providing a bridge to the future of application development.

ABOUT THE AUTHOR

Roger Donnay is an Idaho-based developer and consultant who specializes in database application development, training, and consulting in the Xbase++ language. He is the author of eXpress++, the #1 selling Xbase++ add-on library designed to improve programmer productivity. He developed award winning products for the Clipper language and specializes in helping application developers in migrating their Clipper applications to Xbase++.

Roger has been a speaker at conferences in the U.S., Europe, Asia, South Africa and Australia.

Contact Information

Roger Donnay
Donnay Software Designs
1486 S. Logger's Pond Place #11
Boise, ID 83706
208-344-0108
208-867-6091 Mobile
email: rogerdonnay@donnay-software.com
home page: http://www.donnay-software.com
support forum: http://bb.donnay-software.com:8080/phpBB3

Sybase, Inc.
Worldwide Headquarters
One Sybase Drive
Dublin, CA 94568-7902
U.S.A
1 800 8 SYBASE

www.AdvantageDatabase.com

SYBASE®
An SAP® Company