MANEJO DE INDICES CDX EN CLIPPER 5.3

En clipper 5.3 el RDD para manejo de índices CDX tiene varias ventajas que vamos a analizar y a reforzar con ejemplos.

A modo de introducción se puede decir que:

- Los archivos índices que se crean son pequeños, puesto que están comprimidos, siendo por esto más rápidos.
- Se pueden crear varias claves de indexación (teóricamente hasta 49) que se pueden guardan en un sólo archivo físico (.CDX) en vez de en multitud de (.NTX) y por lo tanto el código fuente y el trabajo para abrirlos es menor, así como el consumo de files del sistema.
- Se puede usar un archivo índice con el mismo nombre que la tabla a la que pertenece; al abrirse la tabla, automáticamente se abrirá el índice. Esto reduce el riesgo de corrupción.
- Crea y mantiene índices condicionales (FOR / WHILE / REST / NEXT).
- Tecnología que optimiza el uso de filtros utilizando índices o SCOPES, mucho más rápido.
- Ahora, los archivos memo (.FPT) pueden almacenar hasta 4.2 Gb, siendo más pequeños y eficientes que los del formato DBFNTX (.DBT)
- Crear ficheros memo con un tamaño mínimo del bloque de 1 byte.
- El espacio vacío de los ficheros memo se reutiliza de forma automática.
- Almacenar todos los tipos de datos de CA-Clipper (menos bloque de código) en ficheros memo, incluso imágenes o sonidos.
- Utilizar las extensiones de Clipper (funciones BLOB) para la gestión de ficheros y campos de ficheros memo.
- Sistema de información sobre campos, registros y Tablas desde los mismos índices.

USO DEL DBFCDX

El controlador DBFCDX viene en dos ficheros: DBFCDX.LIB y _DBFCDX.LIB. Que se encuentran en el subdirectorio \LIB del clipper 5.3.

Para usar el Driver CDX se puede hacer de dos maneras:

1.- se deben agregar las siguientes líneas al programa principal para establecer el driver CDX por defecto

```
#INCLUDE "Ord.ch"       //si se van a utilizar SCOPES
REQUEST DBFCDX
RDDSETDEFAULT ( "DBFCDX" )
```

...y linkear las librerías: DBFCDX y _DBFCDX

2.- Use la cláusula VIA "DBFCDX" del mandato USE o bien especifique "DBFCDX" en el argumento <*cControlador*> si abre la base de datos con la función DBUseArea()

...y linkear las librerías: DBFCDX y _DBFCDX

CREACION DE INDICES CDX

En principio existen 3 funciones para construir índices: **OrdConSet**(), **OrdCreate**() y **DbCreateIndex**(), pero no se recomienda trabajar directamente con ellas. Por el contrario, usemos **INDEX ON** (que en el fondo se transforma en funciones por medio del preprocesador) por su claridad y flexibilidad.

Analicemos cada una de las partes de este comando.

```
INDEX ON  ;
      TAG  TO  ;
      FOR  ;
      WHILE  ;
      NEXT  | RECORD  | REST | ALL ;
      UNIQUE ;
      ASCENDING | DESCENDING ;
      EVAL  EVERY  ;
      USECURRENT ;
      ADDITIVE ;
      CUSTOM ;
      NOOPTIMIZE
```

- **FOR**: Con FOR creamos índices condicionales, es decir que contengan solo los registros que cumplan con una condición lógica. Ejemplo:

```
INDEX ON MiBase->edad TAG joven ;
      FOR edad <= 18       // el TAG "joven" contiene personas con edades menores o iguales a 18 años
```

Cuando usamos FOR, la condición lógica queda guardada en el índice y el sistema usa la utiliza cada vez que actualizamos la tabla o se hace un REINDEX. Es decir, creamos el TAG al comienzo del programa y después nos olvidamos, ya que el sistema lo mantiene.

Si olvidamos la expresión condicional contenida en un TAG:

```
OrdFor("joven")                  // Devuelve: MiBase->edad <=18
```

- **WHILE**: Es parecido al comando anterior. Ponemos una condición que limita el número de registros a incluir en el TAG. Ejemplo:

```
INDEX ON MiBase->edad TAG joven ;
      WHILE edad <= 18
```

WHILE sólo procesa los datos al momento de crear el TAG, pero la condición no queda integrada al BAG, por lo tanto el sistema no mantiene este índice. La garantía es que el TAG se crea con mayor rapidez que con un FOR.

- **NEXT**: Especifica cuantos registros se van a procesar desde la posición actual del puntero. Ejemplo:

```
INDEX ON MiBase->edad TAG joven ;
      WHILE edad <= 18 ;
      NEXT 40
```

- **RECORD** : especifica un registro individual a procesar.

- **REST**: Especifica el procesamiento de todos los registros desde la posición actual del puntero, hasta el EOF.

- **ALL**: Se especifican todos los registros del área de trabajo. Si no se especifica WHILE, NEXT, RECORD, ni REST, se asume ALL por defecto.

- **UNIQUE**: Evita que claves duplicadas se agreguen al TAG. Hay que ser cuidadosos con esta cláusula, porque si en la tabla hay 2 registros con la misma clave, solo el primero se agrega. Se recomienda usar el comando OrdSkipUnique() (lo veremos más adelante).

- **ASCENDING | DESCENDING**: La clave del TAG se ordena ascendente o en forma descendente. Por defecto se asume ASCENDING.

- **EVAL**: Evalúa un bloque de código cada cierto intervalo especificado en la cláusula **EVERY**. EVERY por defecto es 1. Veamos un ejemplo en donde se muestra el progreso de la indexación de una tabla:

```
USE MiTabla New
INDEX ON Mitabla->Codigo TAG Cod;
       EVAL { Avance() } EVERY 10

PROC Avance()

      local nRegProc := 0

      nRegProc + = 10  // incremento
      ? ( nRegProc/Mitabla->(Reccount()) ) * 100

RETURN
```

- **USECURRENT**: En el TAG se incluirán sólo los registros establecidos por otra condición creada antes, por ejemplo con un filtro.

- **ADDITIVE**: Le dice al sistema que mantenga abierto índices que lo estaban antes de crear el suyo. Si no se especifica, los BAGs abiertos se cierran antes de crear el nuevo, pero ¡OJO!, el BAG estructural NUNCA se cierra, a menos que antes de abrirlo ejecute SET AUTOPEN OFF.

- **CUSTOM**: Este es un índice especial cuya utilidad reside en que se crea un TAG vacío que llenaremos según nuestro propio criterio. M s Adelante nos detendremos para explicar como funciona.

- **NOOPTIMIZE**: El driver DBFCDX posee una tecnología especial de optimización de filtros; en las versiones anteriores a 5.3 no se recomendaba el uso de filtros porque hacia m s lenta la ejecución del programa. Ahora, si no se específica NOOPTIMIZE, la cláusula FOR será optimizada automáticamente.

DESDE LOS NTX A LOS CDX

El RDD DBFCDX crea índices individuales 70% más pequeños y más rápidos que los antiguos NTX. Veamos un Ejemplo de creación:

```
USE MiBase NEW
INDEX ON MiBase->nombre1   TO nom1
INDEX ON MiBase->nombre2   TO nom2
INDEX ON MiBase->apellido1 TO ape1
INDEX ON MiBase->apellido2 TO ape2
SET INDEX TO nom1, ...etc. Se usan igual que los NTX.
```

Se han generado 4 índices individuales: nom1.cdx, nom2.cdx, ape1.cdx y ape2.cdx
Al igual que los NTX, sobreescriben cualquier índice anterior con el mismo nombre. Se utilizan igual que los NTX.

CONVERSION DE UN FICHERO ANTERIOR DBFNTX CON MEMOS A DBFCDX

Si hemos venido trabajando hasta ahora con ficheros de tipo NTX queremos pasar a trabajar con CDX no hay ningún problema, se borran los índices antigüos y se crean los nuevos, pero en el caso de que estos ficheros contengan campos memos, esto es, ficheros DBT asociados al DBF no se convierten automáticamente a los nuevos FPT.
Para solucionarlo debemos crear un simple programa que no use por defecto CDX y hacer un duplicado de los archivos actuales DBF+DBT a los nuevos DBF+FPT.

```
USE ARCHIVO1   // sin hacer request del cdx ni nada se abrirá como dbfntx
COPY TO ARCHIVO2 VIA "DBFCDX"
```

DBFCDX:INDICES COMPUESTOS (BAGs.)

La mayor potencia del RDD BBFCDX, la encontraremos usando los índices compuestos.

El Driver CDX crea índices compuestos, los que en inglés se llaman BAGs. La traducción de BAG es "bolsa" o "paquete", porque al igual que verdaderos sacos, dentro de ellos se van a almacenar las claves de indexación. (Antes se ubicaban en archivos individuales con extensiones NTX o CDX). Estas claves ahora las llamaremos "TAGs", que en ingles se traduce como "etiqueta".

Entonces podremos trabajar con un sólo archivo (el BAG) que va a contener varias etiquetas con los nombres de nuestras claves de indexación, los TAGs. Veamos ejemplos de creación de varios TAGs por separado.

```
USE MiBase NEW
INDEX ON MiBase->nombre1   TAG nom1 TO lista
INDEX ON MiBase->nombre2   TAG nom2 TO lista
INDEX ON MiBase->apellido1 TAG ape1 TO lista
INDEX ON MiBase->apellido2 TAG ape2 TO lista
```

Se crea una "bolsa de índices" o BAG con el nombre LISTA.CDX, el que contiene 4 TAGs: nom1, nom2, ape1, ape2.

APERTURA DEL BAG

En general no se recomienda SET INDEX ni Dbsetindex() para abrir BAGs; se prefiere OrdListAdd() que abre el Bag pero no cierra los otros índices que pudiesen estar abiertos.

Por defecto el TAG de control será el primero.

ORDLISTADD([,])

Si se utiliza SET INDEX en la apertura, por defecto se pone el primer TAG de la lista dentro del BAG, como el maestro, pero no se podrán utilizar los útiles comandos SET AUTOPEN, ni SET AUTORDER que veremos m s adelante.

Veamos un ejemplo:

```
USE MiBase New
MiBase->(OrdListAdd("lista"))          // se abre un archivo índice llamado lista.cdx
```

BAGs ESTRUCTURALES

Un índice estructural no es mas que un BAG que tiene como particularidad tener el mismo nombre que la tabla. ¿En que nos beneficia esto? Es que al abrirse la tabla al que está asociado, automáticamente se abre también con economía de comandos. Usualmente se crea automáticamente con:

```
USE MiBase New
INDEX ON  TAG
```

```
INDEX ON  TAG  to Nombre_archivo_igual_a_Tabla
```

```
INDEX ON  TO Nombre_archivo_igual_a_Tabla
```

En este último caso, un nuevo .CDX se creará sobreescribiendo cualquier archivo existente con el mismo nombre. No se recomienda hacer esto con un BAG estructural abierto.

El TAG creado por defecto tomará el mismo nombre que el BAG. Si existe un TAG con el mismo nombre, se borra y el orden de los TAGs en el BAG cambiar , a menos que el nuevo TAG sea el último en el índice. Esto causa un incremento no deseable del tamaño del BAG.

Veamos un ejemplo de creación de un BAG estructural:

```
USE MiBase NEW
INDEX ON MiBase->nombre1   TAG nom1
INDEX ON MiBase->nombre2   TAG nom2
INDEX ON MiBase->apellido1 TAG ape1
INDEX ON MiBase->apellido2 TAG ape2
```

Este código le dice al driver RDDCDX que se quiere poner los Tags en un archivo BAG que se crea con el mismo nombre de la Tabla, pero con diferente extensión: "MiBase.CDX". Este BAG contiene 4 TAGs: nom1, nom2, ape1, ape2.

Posteriormente al abrir la tabla ocurre que...

```
USE MiBase New          // automáticamente se abrió el archivo índice MIBASE.CDX
```

Nota: El TAG "nom1", por defecto será el maestro, aunque podemos escoger otro con la función OrdSetFocus().

ORDSETFOCUS([,]) Establece cual TAG de la lista contenida dentro del BAG, será el que controle el ordenamiento de nuestra tabla, tanto por el nombre del ATG como por su número de orden (el orden 0 desabilita todos los TAG y deja la tabla sin orden por ningún índice). Ejemplo:

```
Use MiBase New
MiBase->( OrdSetFocus("ape1") ) // se establece "ape1" como el TAG de control
```
O también
```
MiBase->( OrdSetFocus(4) ) // se establece "ape2" como el TAG de control
```

SET AUTOPEN ON | OFF

Puede darse, en casos muy particulares, que la apertura automática cause problemas, como por ejemplo, que un CDX ajeno se asocie con una tabla recién creada o si tratamos de borrar un índice estructural auto-abierto. Estos problemas deben ser controlados por el programador, ya que la única manera de cerrar un BAG estructural abierto, es cerrando primero la Tabla asociada.

En todo caso, la apertura automática puede anularse con SET AUTOPEN OFF. No se abrirá automáticamente el índice asociado. Por defecto SET AUTOPEN esta en ON.

SET AUTORDER TO

Con SET AUTORDER TO N, el TAG "N" ser el índice maestro al abrirse el índice en forma automática. Por defecto SET AUTORDER esta en OFF, y la tabla se abre en orden natural, es decir en el orden en que fueron creados los TAGs. Ejemplo:

```
SET AUTORDER TO 3
USE MiBase New          // automáticamente se abrió el archivo índice !
                        // El orden de control por defecto ahora es "ape1"
```

RECONSTRUCCION DE TAGs en BAGs ESTRUCTURALES

Si el BAG estructural está abierto, se puede utilizar el comando REINDEX para reconstruir los TAGs. Esto hará un PACK y recuperará espacio ocupado por TAGs que han sido borrados. Pero, no se recomienda este

comando, porque no permite recuperar el índice si la cabecera del archivo índice est dañada, no existe el BAG en el directorio o simplemente est corrupto. Se prefiere INDEX ON.

Ahora, el comando INDEX ON, a pesar de que funciona si no está el BAG, porque genera los TAGs a partir de la tabla, los va agregando al archivo, no sobreescribe por lo tanto, se va almacenando información y el BAG va creciendo con datos inútiles que al largo plazo lentificar n el sistema y ocasionarán problemas.

Algunas formas de salvar esta situación podrían ser:

1. Al comienzo de nuestro programa:

```
SET AUTOPEN OFF            // No se abre automáticamente el índice
Use MiTabla new
FERASE( "MiTabla.cdx" )   // Ferase() no produce error si no existe el archivo
INDEX ON MiBase->nombre1 TAG nom1
INDEX ON ...etc, etc

SET AUTOPEN ON             // Restaura el valor por defecto
```

2. Si AUTOPEN está en ON y la tabla ya está abierta en nuestra aplicación:

```
CLOSE SELECT ( "MiBase" )  // Cerramos la tabla
FERASE( "MiBase.CDX" )     // Borramos el índice
USE MiBase NEW             // Reabrimos la tabla
INDEX ON MiBase->nombre1 TAG nom1
INDEX ON ...etc, etc
```

También se puede usar los siguiente:

```
WHILE !EMPTY( ( cTag := OrdName( 1 ) ) )
      OrdDestroy( cTag )
END

INDEX ON MiBase->nombre1  TAG nom1
INDEX ON ...etc, etc
```

Con la función OrdDestroy() borramos todos los TAGs. Si se destruye uno solo no se recupera el espacio que este usaba, pero borrándolos todos, se cierra y se borra el archivo BAG.

3. Si AUTOPEN est OFF y la tabla está abierta en nuestra aplicación:

```
DbClearIndex()
INDEX ON MiBase->nombre1  TAG nom1
INDEX ON ...etc, etc
```

La función DbClearIndex() cierra todos los índices abiertos y escribe al disco todas las actualizaciones pendientes.

GESTION DE LOS TAGs

ORDBAGNAME( )
Para conocer a que BAG pertenece un TAG. Ejemplo:

```
? OrdBagName( "nom1" )     // Devuelve: MiBase
? OrdBagName( "ape2" )     // Devuelve: MiBase
```

ORDNAME([,])
Para saber que TAGs están disponibles dentro de los BAGs.

```
? MiBase->ORDNAME(1)       // Devuelve: "nom1"
? MiBase->ORDNAME(3)       // Devuelve: "ape1"
```

ORDBAGEXT()
Si no sabemos la extensión del RDD usado.

```
USE MiBase New
? OrdBagExt()                // Devuelve .cdx
```

ORDKEY([,])
Si no estamos seguros de la expresión clave que originó un TAG tenemos:

```
? OrDkey("ape2")             // Devuelve: MiBase->apellido2
```

ORDESTROY([,])
Elimina el TAG del archivo índice. Ejemplo:

```
USE MiBase NEW
MIBase->OrdDestroy( "nom1")          // Borra TAG "nom1" de BAG "MiBase"
MIBase->OrdDestroy( "otro","Clientes" )  // Borra TAG "otro" en BAG "Clientes"
```

ORDKEYCOUNT( )
Permite determinar cuantas claves de indexación contiene un TAG. El número de claves puede ser diferente al número total de registros, porque se puede haber usado un índice condicional, o construido un índice temporal con la cláusula CUSTOM. Ejemplo:

```
USE TablaMujeres NEW
INDEX ON TablaMujeres->Nombre TAG joven FOR edad < 18

? "Número Total de Mujeres:", TablaMujeres->( Reccount() )
? "Mujeres menores de edad:", tablaMujeres->( OrdKeyCount("joven") )
```

ORDKEYNO( )
Devuelve la posición lógica relativa de un registro en el TAG. Esto nos va a permitir saber si un registro está dentro de un TAG o no. Ejemplo:

```
USE TablaMujeres NEW
INDEX ON TablaMujeres->nombre TAG nombre

TablaMujeres->(DbSeek( "Claudia" ) )
nNumero := TablaMujeres->( OrdKey("nombre") )
nReg:= TablaMujeres->( Recno() )

? "Claudia es la empleada n§:", nNumero
```

NOTA: Las funciones OrdKeyCount() Y OrdKeyNo() no trabajan si OPTIMIZE est en OFF, o si una variable de filtro es un elemento de matriz.

ORDKEYGOTO( )
Saltará a un registro específico dentro del índice.

```
? TablaMujeres->(OrdKeyGoto( nReg ))    // muestra "Claudia"
```

ORDSCOPE( )
Función que permite establecer los límites de comienzo y de final de un archivo índice.

Se usan las constantes TOPSCOPE y BOTTOMSCOPE (agregar Ord.Ch al PRG) para hacer referencia al límite inferior o superior del archivo índice. Ejemplo:

```
USE TablaMujeres NEW
INDEX ON TablaMujeres->Edad TAG edad

//Mostrar las edades entre 18 y 45 años
tablaMujeres->( OrdScope( TOPSCOPE, 18 ) )
```

```
TablaMujeres->( OrdScope( BOTTOMSCOPE, 45 ) )
List TablaMujeres->Edad

// Muestra todas las edades hasta 45 años
TablaMujeres->( OrdScope( TOPSCOPE, NIL ) )
List TablaMujeres->Edad

// muestra todas las edades
TablaMujeres->( OrdScope( BOTTOMSCOPE, NIL ) )
List TablaMujeres->Edad
```

ORDSETRELATION( )
Combina un relación con un filtro. Esto implica que el archivo relacionado es dinámicamente filtrado basado en el valor de la relación establecida con la tabla principal. Esto es muy útil en relaciones uno a varios. Veamos un ejemplo:

```
USE Deuda NEW
INDEX ON NumClien TAG NumCli
Deuda->DbSetFilter( { || deuda->saldo > 1000000 }, "deuda->saldo > 1000000" )

USE Cliente NEW
// Establecemos una relación selectiva entre el cliente y su deuda
OrdSetRelation( "Deuda", {|| cliente->numcli },  "Cliente->numcli" )

cliente->(DbGotop())

Do While cliente->(!Eof() )

     ? cliente->Nombre, cliente->apellido

    // sólo se muestran deudas mayores a $1.000.000

     Do While Deuda->(!eof())

          ? deuda->producto, deuda->fecha

          deuda->(dbskip())

     EndDo

     factura->(dbskip())

EndDo
```

ORDSKIPUNIQUE( )
Permite encontrar registros con valores únicos ya sea hacia adelante o hacia atrás en el índice. Ejemplo:

INDICES "CUSTOM" O DE USUARIO

Dentro de un BAG podemos tener una clase especial de índice los "CUSTOMs". Al usar la cláusula CUSTOM del comando INDEX ON, creamos un índice vacío, sin valores claves, el que podemos llenar según nuestro propia conveniencia, en tiempo de ejecución.

Esta clase de índices no es actualizado por el sistema, por lo que deben ser mantenidos por el programador.

Ejemplo:

```
USE MiBase NEW
INDEX ON MiBase->nombre TAG nombre TO MiBase CUSTOM

For i := 1 to Reccount() Step 50

      MiBase->( OrdKeyAdd() )
      MiBase->( Dbskip() )

Next

For i := 1 to Reccount() Step 100
      MiBase->( OrdKeyDel() )
      MiBase->( Dbskip() )
Next
```

Este ejemplo crea un TAG personalizado agregando uno de cada 50 registros y borrando uno de cada 100 registros

Estas funciones son exclusivas para índices custom

ORDEKYADD()
Función para dar altas registros en el índice Custom.

ORDKEYDEL()
Da de bajas registros del índice CUSTOM.

ORDKEYVAL()
Recupera el valor clave de un registro en un índice CUSTOM

DOS RECOMENDACIONES UTILES

INDICES ESTANDAR VERSUS INDICES TEMPORALES

Toda aplicación tiene índices estándar, que necesitan estar siempre abiertos y actualizados, de acuerdo a los cambios en las tablas. Es buen idea ubicarlos todos juntos en un BAG estructural. Así, al abrir la tabla, se abre el índice y queda abierto y el sistema va actualizando los TAGs durante toda la aplicación.

Algunas veces se necesitan índices adicionales, que se usan en forma esporádica por ejemplo, para reportes con opciones de usuario. Estos deberían ser creados en índices individuales (creados sin la cláusula TAG) por cada usuario. No se recomienda crear índices temporales como TAGs dentro de un BAG estructural.

INDEX ON... FOR VERSUS SET FILTER TO

Los índices condicionales, creados con la cláusula INDEX ON... FOR, se usan cuando se desea mantener índices estándares actualizados. Es cómodo porque se crean una sola vez y el sistema va actualizando los cambios que ocurren en el registro. Además la cláusula FOR es optimizada por lo tanto son índices rápidos.

Ahora, para situaciones puntuales, donde se requieran listados o Tbrowses de uso esporádico o por ejemplo, con alguna opción de filtrado elegida por el usuario, como no se sabe su estructura de antemano, es buena idea construir un SET FILTER con esta condición. Obviamente es más rápido que construir un índice porque solo lee y aprovecha los índices existentes.

El RDD DBFCDX tiene la capacidad de optimizar el comando SET FILTER, y ahora funciona mucho más r pida y eficientemente que su contrapartida en clipper 5.2 pero se deben cumplir algunas condiciones técnicas, las que veremos a continuación.

EL OPTIMIZADOR DE FILTROS

El driver DBFCDX posee una tecnología que optimiza el uso de los filtros. Cuando se establece una condición de filtro , el RDD no accesa toda la tabla para ver que registros la cumplen, sino que revisa en todos los índices disponibles, que son mucho más pequeños que las tablas, aumentando la velocidad de proceso, sobre todos en tablas de gran tamaño.

Si ponemos en nuestro programa la expresión de filtro "SET FILTER TO Edad = 18", y existe en el CDX un TAG creado con INDEX ON edad..., el Rdd buscará en el índice, y solo los registros para esa edad en particular serán accesados.

El optimizador funciona en dos situaciones:

- Cuando se crean índices condicionales usando: INDEX ON FOR (Excepto si se usa WHILE, USECURRENT o NOOPTIMIZE)

- Cuando dentro de la aplicación necesitamos filtrar con: SET FILTER o DbSetFilter(). Este comando funciona igual que en clipper 5.2, pero ahora mucho más eficiente y rápido.

Recordemos que hace DBSETFILTER(): Establece una condición que filtra en forma lógica, los registros que no cumplan la con ella. Estos no serán visibles y no serán procesados.
Atención, el filtro no se activa nunca hasta que no se haga un DBGOTOP().

LIMITACIONES DEL OPTIMIZADOR

Hay 3 limitaciones del optimizador:

- No se pueden usar variables locales en las condiciones de filtrado.
- El optimizador no usará índices condicionales o creados con FOR.
- La expresión de filtro debe coincidir con la expresión del índice.

Los programadores recomiendan construir funciones con variables definidas en forma local o estáticas, por razones de un mejor manejo de la memoria. Pero, si se usa DBSETFILTER() no se podrá hacer referencia a este tipo de variables porque ocurrirá un error en tiempo de ejecución en nuestra aplicación: "variable no definida".

Para salvar la situación podemos usar 3 trucos:

- Cambiar las variables locales y est ticas por públicas o privadas. (No recomendado)
- Usar la cláusula NOOPTIMIZE del comando INDEX ON (No recomendado)
- Convertir el valor de la variable a su equivalente en una constante de texto. (Muy Recomendada.) Ejemplo:

```
Use Midbf New
INDEX ON fecha TAG fecha

Local dFecha := date()

@ 10,10 say "Hasta que fecha se Lista?" get dFecha
read

DdSetFilter ( &( "{ || fecha <= " + dtos(' " + dtoc( dFecha ) + " ') }" ), ;
                "fecha <= " + dtos(' " + dtoc( dFecha ) + " ') }" )

List MiDbf...etc.
```

¿Se ve complejo No?. Pero para alivianar el trabajo se puede utilizar la función data2str() que convierte cualquier tipo de dato a texto, con lo que el ejemplo de arriba quedaría mucho m s simple:

```
DdSetFilter ( &( "{ || fecha <= " + data2str( dFecha ) + "}" ), ;
                "fecha <= " + data2str( dFecha ) + )  )
```

Nota: Por si no la tiene, se lista el código fuente de data2str():

```
Func Data2Str( xVar )
Local cRet := "", cTipo :=VALTYPE( xVar )

      Do Case

        Case ( cTipo == 'C' ) .OR. ( cTipo == 'M' )
            cRet := '"' + STRTRAN( xVar, '"', '" + ["] + "' ) + '"'

        Case ( cTipo == 'D' )
            cRet := "CTOD('" + DTOC( xVar ) + "' ) "

        Case ( cTipo == 'L' )
            cRet := IIF( xVar, ".T.", ".F." )

        Case ( cTipo == 'N' )
            cRet := LTRIM( STR( xVar ) )

        EndCase

RETURN cRet
```

INDICES CONDICIONALES

El optimizador no usará índices temporales o condicionales. Si un índice fue creado con la sentencia FOR, no ser usado para optimizar. Ejemplo:

```
INDEX ON edad TAG FOR edad >= 30
DbSetFilter( {|| edad = 45 }, "edad = 45" )   // NO OPTIMIZABLE
```

EXPRESIONES OPTIMIZABLES

Para sacar el máximo partido al optimizador, la expresión de filtro debe calzar o coincidir exactamente con la expresión contenida en el índice: Ejemplo

```
INDEX ON edad TAG edad
DbSetFilter( {|| edad = 30 }, "edad = 30" )

INDEX ON paterno TAG paterno
DbSetFilter( {|| paterno = 'J' }, "paterno = 'J'" )

INDEX ON Upper( paterno + nombre + escivil ) TAG pme
DbSetFilter( {|| (paterno + nombre) + escivil = 'C'}, ;
             "(paterno + nombre) + escivil = 'C'" )

INDEX ON IIF(casado, 'C' + paterno, 'S' + paterno) TAG SiCasado
DbSetFilter( {|| IIF(casado,'C' + paterno, 'S' + paterno) = 'C'}, ;
             "IIF(casado,'C' + paterno, 'S' + paterno) = 'C'"

INDEX ON paterno + nombre TAG patnom
DbSetFilter( { || paterno = 'Robles' }, "paterno = 'Robles'" )
DbSetFilter( { || paterno + nombre = 'Robles  G' },;
              "paterno + nombre = 'Robles  G'")
```

Si A+B+C son expresiones de carácter, el optimizador lo hará en una expresión de filtro que use A, A+B, A+B+C. El siguiente comando no es optimizable porque "nombre" no está al comienzo de la expresión.

```
INDEX ON paterno + nombre TAG patnom
DbSetFilter( { || nombre = 'Willy' }, "nombre = 'Willy'" )  // NO OPTIMIZABLE


INDEX ON edad TAG edad
DbSetFilter( {|| empty(edad) }, "empty(edad)" )


INDEX ON Dtos(ultimo) TAG fecha
DbSetFilter( { || dtos(ultimo) = '1999' }, "dtos(ultimo) = '1999'" )
DbSetFilter( { || ultimo = ctod( '01/01/99') }, "ultimo = ctod( '01/01/99') " )

INDEX ON ultimo TAG ultimo
DbSetFilter( { || dtos(ultimo) = '1999' }, "dtos(ultimo) = '1999'" )
```

REGISTROS BORRADOS

La función DbSEtFilter() asume que SET DELETE esta OFF. Entonces si hay registros marcados, aunque no sean visibles serán incluidos en la lista filtrada.

ESTABLECER SCOPES CON DBFCDX EN LUGAR DE SET FILTER TO

Lo ideal es usar directamente SCOPES para establecer los límites de ámbito de control para los valores de las claves con la función ORDScope() en vez del SET FILTER TO mucho más lento.

Para establecer que en nuestro fichero de Clientes nos saque solo los registros que tengan el código mayor o igual a 20 y sean inferior a 25 actuaremos del siguiente modo:

```
#define TOPSCOPE          0
#define BOTTOMSCOPE 1
USE Clientes INDEX Clientes NEW VIA "DBFCDX"
Clientes->( OrdSetFocus( 1 ) )                    && Establecer el orden
Clientes->( DBGoTop() )

Clientes->( OrdScope( TOPSCOPE, "0020" ) )
Clientes->( OrdScope( BOTTOMSCOPE, "0025" ) )

WHILE !( Cliente->( Eof() ) )
.
.
.
END DO
```

Se pueden cambiar los límites de forma dinámica y seguir trabajando como si tal cosa:

```
Clientes->( OrdScope( BOTTOMSCOPE, "0050" ) )          &&Límite inferior en 50
Clientes->( DBGoTop() )

WHILE !( Cliente->( Eof() ) )
.......
END DO
```

Si en vez de especificar un valor se especifica NIL, se tomará el límite, superior o inferior, del TAG activo.

TECNICAS DE USO DEL DBFCDX EN REDES

APERTURA DE TABLAS E INDICES

Se recomienda, al trabajar en redes, no abrir los Indices con el comando DbSetIndex(). Aprovechemos los índices estructurales usados con los siguientes cuidados:

```
SET AUTOPEN OFF            // Impide que se abra automáticamente el BAG
                               // Borrar el CDX estando abierto, produce error

USE MiTabla EXCLUSIVE New   // Intentamos abrir en exclusiva para asegurarnos
                              // que nadie use la tabla al regenerar la tabla

// Proceso para regenerar los índices con seguridad
if ( !NETERR() )
      Ferase("MiTabla.cdx")
      INDEX ON MiTabla->nombre1 TAG nom1
      INDEX ON ...etc, etc
else
      ? "Lo Siento.... Imposible Reindexar Ahora"
endif

CLOSE SELECT( "MiTabla" )   // Cerramos el  área
SET AUTOPEN ON             // Restablecemos el comando por defecto
USE MiTabla SHARED New      // Llamamos la Tabla en compartido
```

COMANDO APPEND EN RED

Al hacer un DbAppend() a una Tabla abierta en SHARED (modo compartido), el registro que se agrega LastRec() se bloquea automáticamente, lo que puede ocasionar problemas si no tomamos la siguiente precaución:

```
Use MiTabla SHARED New

MiTabla->( Dbappend() )

// verificamos que no ocurra error si otros usuario están ocupando la tabla
if MiTabla->( neterr() )
      ? "Imposible Agregar Registro en este momento"
      Return
else
      // si no ocurrió error, la tabla está  libre para agregarle un registro
      // en blanco, pero queda bloqueado así que...

      MiTabla->( LastRec( DbUnLock() ) )

      // Con LastRec() nos aseguramos que sea el último registro físico
      // Ahora se pueden reemplazar valores
      replace mitabla->campo1 with xvalor1
      replace mitabla->campo2 with ...etc, etc

Endif
```

COMANDOS PARA PROGRAMAR EN MULTIUSUARIO

SET AUTOSHARE es un comando que permite que clipper analice el medio en que se desempeña la aplicación, para determinar el mejor manejo de la memoria.

- **SET AUTOSHARE TO 0** : El programa está escrito para multiusuario, pero si se utiliza en ambiente monousuario, igual se comportará como si estuviese en una red, siendo su performance más lenta, porque clipper revisa si las tablas están lockeadas antes de hacer una actualización.

- **SET AUTOSHARE TO 1** : El programa está escrito en multiusuario, pero si se utiliza en ambiente monousuario, Clipper determina que los archivos se abrirán en exclusiva, mejorando la rapidez.

- **SET AUTOSHARE TO 2** : Este comando garantiza que los archivos siempre se abrirán monousuario, independiente de si se está en red o no. Este comando nos permite ahorrar programación. Se escribe el programa multiusuario, pero se vende como monousuario (Y funciona como tal si SET AUTOSHARE esta en 2). Pero, si el cliente requiere (Y paga $$), cambiando SET AUTOSHARE TO 1 activamos la aplicación para RED.

Por defecto SET AUTOSHARE es 1, por esto, si está desarrollando una aplicación multiusuario se recomienda poner AUTOSHARE 0, porque si olvidamos poner algún RLOCK() o FLOCK(), el programa no funcionará hasta que se corrijan los bloqueos. Si AUTOSHARE está en 1, el programa correrá bien en nuestra máquina, a pesar de que existen errores en el código, y al probar la aplicación en ambiente multiusuario ¡OH Vergüenza!, se caerá nuestra aplicación y no sabremos por qué.

COMPATIBILIDAD CON FOXPRO

USO DE TABLAS

Los archivos índices CDX son compatibles con VfoxPro. No así los IDX. En términos generales, para poder trabajar con VfoxPro se deben crear las tablas con clipper y leerlas desde Fox, incluyéndolas en un proyecto como tablas libres. Al revés no funciona ya que Vfox crea tablas con encabezado diferente no compatibles con clipper.

APERTURA

Existe un byte de control en la cabeza de la DBF que indica si un BAG con el mismo nombre que la tabla, debe ser abierto automáticamente o no.
Cuando se crea automáticamente un BAG estructural, este byte es puesto al valor CHR(1). Cuando se borra el índice, borrando todos los TAGs con OrdDestroy(), el valor del byte es puesto a CHR(0). Clipper no mantiene este byte, lo que puede producir problemas al usar FoxPro.

Foxpro al abrir la tabla, no abre automáticamente el índice estructural, a menos que el byte esté en CHR(1). Así, si se usa FoxPro para accesar CDX creados con Clipper, debemos manipular el byte manualmente, vía una función de bajo nivel o hacer un SET INDEX desde dentro de la aplicación FoxPro.

Veamos un ejemplo de Función para modificar el byte de control.

```
Func IndexByte( cNomArchivo, lSet )
Local nHandle

    If ( VALTYPE( lSet ) != "L", lSet:= .F. )
        FSEEK( nHandle := FOPEN( cNomArchivo,2 ) ) == -1 )
        FWRITE( nHandle, IIF( lSet, CHR(1), CHR(0) ) )
        FCLOSE( nHandle )
        RETURN .T.
    Endif

RETURN .F.
```

MANEJO DE LOS CAMPOS MEMO

Antiguamente el driver DBFNTX manejaba archivos memo con la extensión .DBT con un tamaño fijo de bloque de 512 bytes y con un tamaño máximo de 64 Kb. Para guardar texto o código ascii . Esto significa que aunque se escriba un solo byte en un campo memo, 64 bytes serán escritos en el archivo .DBT, perdiendo el resto del espacio.

El RDD DBFCDX usa por defecto la extensión .FTP para los campos Memo. Pero aparte de texto, también se pueden almacenar otros tipo de informaciones, como matrices, fechas, etc. y que pueden tener un tamaño mayor de 64 KB. Este formato se denomina BLOBs (Binary Large Objects) u "Objetos Binarios Grandes", en español.

Se recomienda transformar los archivos desde DBT a FPT.

Veamos un ejemplo de como almacenar en un campo memo:

```
USE Mitabla NEW
REPLACE MiTabla->memo WITH "Bitácora de vuelo 123, Fecha espacial..."
REPLACE MiTabla->memo WITH Date()
```

COMANDOS SET PARA EL USO DE MEMOS

SET MEMOBLOCK TO o SET MBLOCKSIZE TO: Pone el tamaño por defecto del bloque usado por los campos memos. nTamaño debe estar entre 1 y 16384. Si se quiere mantener compatibilidad con FoxPro, no se debe especificar un valor menor que 16. Se recomienda evitar bloques memo sobre 2048 bytes.

SET MFILEEXT TO: Permite cambiar la extensión por defecto .FPT a cualquier combinación de tres letras, aunque no se recomienda ni .BDF ni .CDX. Se puede reemplazar .FPT por .DBV

FUNCIONES PARA EL USO DE LOS BLOBs

BLOBGET( ], [], [])-> uBlob
Permite leer el contenido de un campo memo basado en su posición ordinal. (La posición ordinal es el nº de campo que posee en la estructura de la tabla) Cuando el contenido de un campo memo sea texto de más de 64 KB, podemos usar el parámetro y para procesar el campo en partes.

En el siguiente ejemplo, supongamos que el campo Nº 10 es un campo BLOB y se llama "memo":

```
#include "Blob.ch"
USE MITABLA NEW

nComienzo := 1

Do While !( cTexto := Empty( BlobGet(  10, nComienzo, 1000 ) )
      ? cTexto
      nComienzo += 1000      // Se va imprimiendo de 1000 bytes
EndDo
```

Para asignar un valor a un campo memo se puede usar replace, FieldPut(), etc.
Veamos un ejemplo:

```
#include "Blob.ch"
USE MItabla New

MiTabla->( FieldPut( 10, "Este es un mensaje." ) )

Mitabla->( DbSkip() )
```

```
Mitabla->( FieldPut( 10, { "Gómez", "Padilla", "Robles", "Ramos" } )  )

Mitabla->( DbSkip() )

_Field->Memo := date()

Mitabla->( DbSkip() )

REPLACE Memo With 100
```

BLOBIMPORT(,) -> lOK
Permite importar un archivo a un campo BLOB. El contenido del archivo no es importante para el driver, pudiendo ser palabras, imágenes o un .EXE.

```
#include "Blob.ch"
USE MiTabla NEW

BlobImport( 10, "C:\Config.sys" )

Mitabla->( DbSkip() )

BlobImport( 10, "C:\Util\Blinker\Config.blk" )
```

BLOBEXPORT(, ,) ->lOK
Permite escribir el contenido del campo BLOB a un archivo. El parámetro nModo indica que ocurre si existe cTarget:
BLOB_EXPORT_APPEND indica que el BLOB debe ser agregado al final del archivo BLOB_EXPORT_OVERWRITE indica que el archivo debe ser sobreescrito.

```
#include"Blob.ch"
USE MiTabla NEW

BlobExport( 10, c:\memos\registros\"Error.reg", BLOB_EXPORT_APPEND )
```

Este manual es una adaptación hecha de varias fuentes:
- Manuales de DBFCDX y FLEXFILE de Comix y Clipmore (LoadStone)
- Manual de DBFCDX para CAVO de Robert Van Der Hulst (TechniCom)
- Manual de DBFCDX de Claudio Torillo (ClipSupport)
- Manual de DBFCDX de Guillermo Robles Galaz
- Manual de DBFCDX de José A. Suárez Moreno
- Guías Norton del Clipper 5.3a