

StupidTempTable Theory

Resumen

Esta "Teoria" fue propuesta por Troy Sorzano hace varios años. Primero transcribo una traduccion del planteo original, luego veremos como simplificarla para el uso diario.

Para que necesitamos esta teoria?

- Para obtener y manipular facilmente campos calculados por el server. (esto se puede aplicar a funciones como sum, avg, max, etc como tambien a la llamada de Stored procedures)
- Para no tener que establecer relaciones en el diccionario por cada JOIN que se nos ocurra usar.
- Para no necesitar definir una clave por cada ordenamiento que necesitemos de las tablas, independientemente de las relaciones entre las mismas.

Como funciona esta teoria?

En el server existe una tabla de validacion (StupidTempTable) con un campo por cada tipo de dato que usemos en nuestra aplicacion.

Esta tabla no va a existir en nuestro diccionario de Clarion, sera usada para validar a las Local TempTables definidas en la data section de cada procedimiento.

Al definir la tabla local en el procedimiento se le asigna el atributo **NAME** tanto de la tabla como de cada uno de los campos que se desea retornar. El Driver de Clarion no verifica el orden, o el tamaño, solo que el nombre del campo exista en el SQL y tenga el mismo tipo de datos.

Se pueden definir y abrir tantas tablas locales temporales como se necesiten, cada una de ellas haciendo referencia a la misma StupidTempTable, obviamente todas ellas tendran diferente nombre, pero el mismo atributo NAME. (Asegurese de cerrarlas una vez que las use)

Se pueden tener tantos campos en la tabla temporal como se necesiten con el mismo atributo NAME , obviamente sin repetir sus nombres.

Implementando la teoria.

Crear la tabla de validacion en el server.

Solo es necesario una.
 StupidTempTable
 X_INT INT
 X_VARCHAR VARCHAR(255)
 X_CHAR CHAR(255)
 X_MONEY MONEY
 X_TEXT TEXT
 Etc.

Crear la tabla local

En el embed data section de su procedimiento.
 TempTable FILE, DRIVER('ODBC'),
 NAME('StupidTempTable'),
 OWNER(Glo:ConnectionString), PRE(Stt)
 Record RECORD
 Provider_ID LIKE(Pro:Provider_ID), NAME('X_INT')
 Pro_Name LIKE(Pro:Name), NAME('X_VARCHAR')
 Account_ID LIKE(Acc:Account_ID), NAME('X_INT')
 Acc_Name LIKE(Acc:Name), NAME('X_VARCHAR')
 Directions LIKE(Pro:Directions), NAME('X_TEXT')
 Distance LONG, LIKE('X_INT') !for Server Calculated
 field
 END
 END

Obtener la informacion

Y asignarla en cada campo

```
TempTable{Prop:Sql} = 'select p.provider_id,' &|
'p.name,a.account_id,a.name,p.directions,' &|
'round(sqrt(power(p.Zip_XCoord - a.Zip_XCoord,2) + '&|
'power(p.Zip_YCoord - a.Zip_YCoord ,2)),0) DISTANCE
from '&|
'providers p, accounts a, acctproviders ap where '&|
'a.state = <39>PA<39> and ap.account_id = a.account_id
'&|
'and p.provider_id = ap.provider_id order by a.name,
'&|
'distance'
```

Luego de esto procesar la tabla como lo haríamos con cualquier tabla normal.

Simplifiquemos un poco

Si bien es muy poderosa, el problema que tiene esta teoria es que resulta bastante engorroso declarar las tablas temporales en cada procedimiento.

Teoria simplificada

Una opcion mas sencilla es crear en el diccionario una tabla con campos Cstring. Yo generalmente la llamo simplemente SQL.

```
SQL FILE, DRIVER('MSSQL'), OWNER(GLO:Conexion),
NAME('dbo.SQL'), PRE(SQL), BINDABLE, THREAD
```

```
Record                                RECORD, PRE()
C1                                   CSTRING(255)
C2                                   CSTRING(255)
C3                                   CSTRING(255)
C4                                   CSTRING(255)
C5                                   CSTRING(255)
C6                                   CSTRING(255)
C7                                   CSTRING(255)
C8                                   CSTRING(255)
C9                                   CSTRING(255)
C10                                  CSTRING(255)
C11                                  CSTRING(255)
C12                                  CSTRING(255)
                                     END
END
```

Cuantos Campos?

Le creamos tantos campos Cstring como necesitemos para las consultas de nuestra aplicacion.

Obviamente la limitacion es que si creamos una tabla con 10 campos y necesitamos una consulta con 11 campos, tenemos que modificar la tabla. Lo mismo sucede si necesitaramos un campo de texto mayor a 255.

No hay ningun inconveniente en crear varias tablas temporales en el Diccionario para que se ajusten a nuestros requerimientos. Por ejemplo es mas rapido tener una tabla temporal solo con un campo LONG para realizar sumatorias, maximos o traer el @@identity desde la base.

Por que CString?

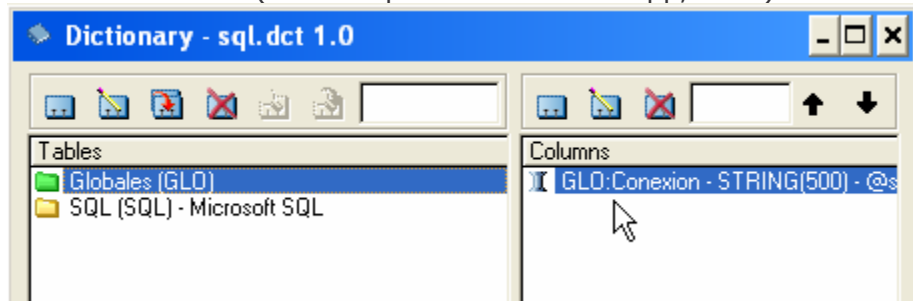
Bueno, el tipo CString se **mapea automaticamente** contra cualquier tipo de datos, con lo cual cualquier cosa que pongamos en el Select se va a asignar en el mismo orden a los campos Cstring, sin importar su tipo.

Los datos nos van a llegar formateados como Alfanumericos, en algunos casos es necesarios aplicarles un DEFORMAT para poder manipularlos mejor. Especialmente las fechas.

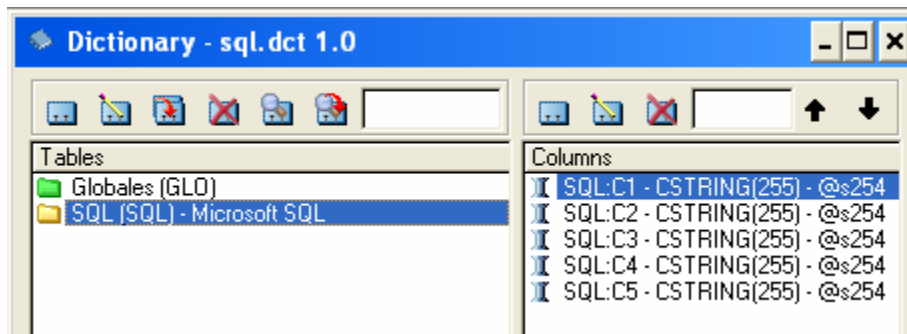
Ejemplo: Creando un Editor de SQL.

Apliquemos lo anterior para crear un editor de SQL sencillo.

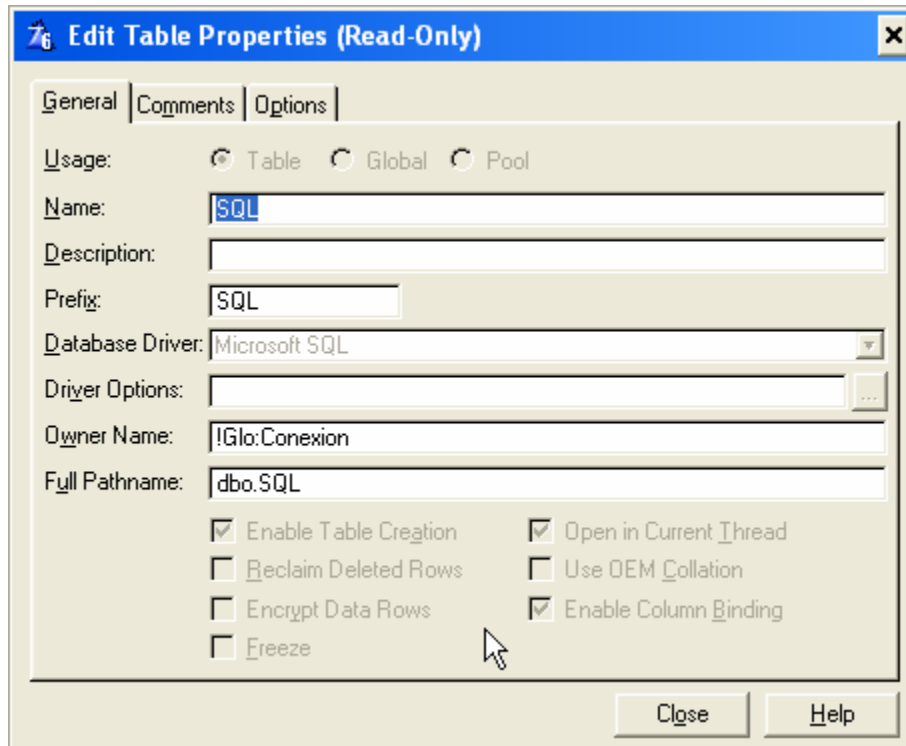
Primero necesitamos crear un Diccionario con la variable Global de conexión.(tambien podria estar en el app, claro)



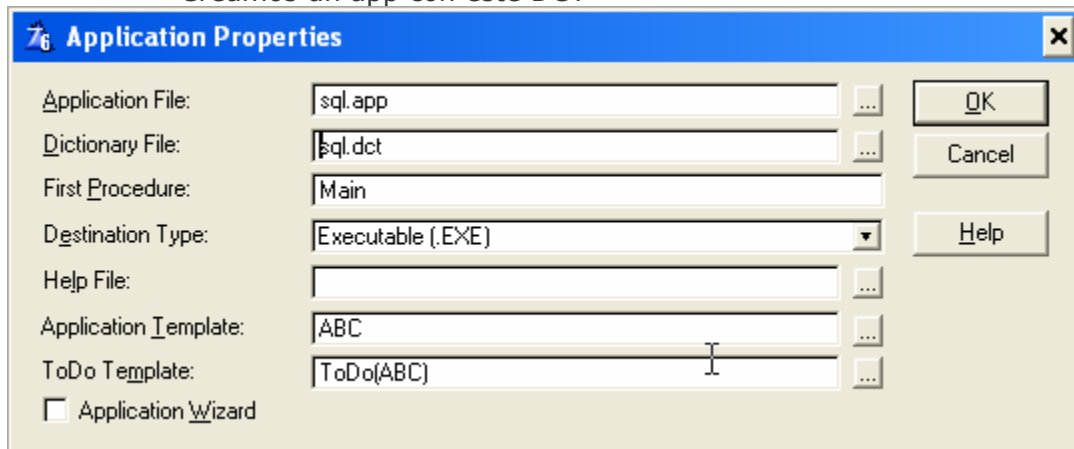
Y la tabla SQL. En este caso crearemos solo 5 campos...



Recordemos que siempre se debe usar un String de conexión variable en el Owner de las tablas.

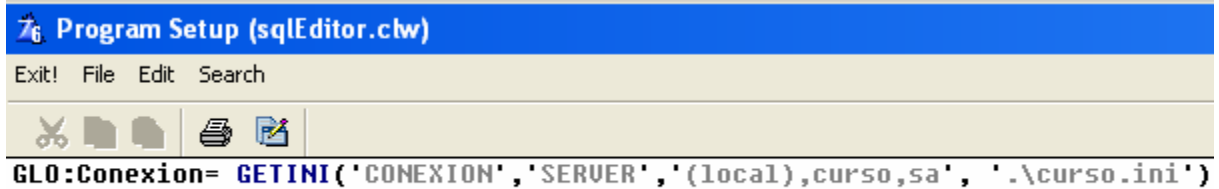


Creamos un app con este DCT



En global Embeds – Program setup inicializamos el String de Conexión.

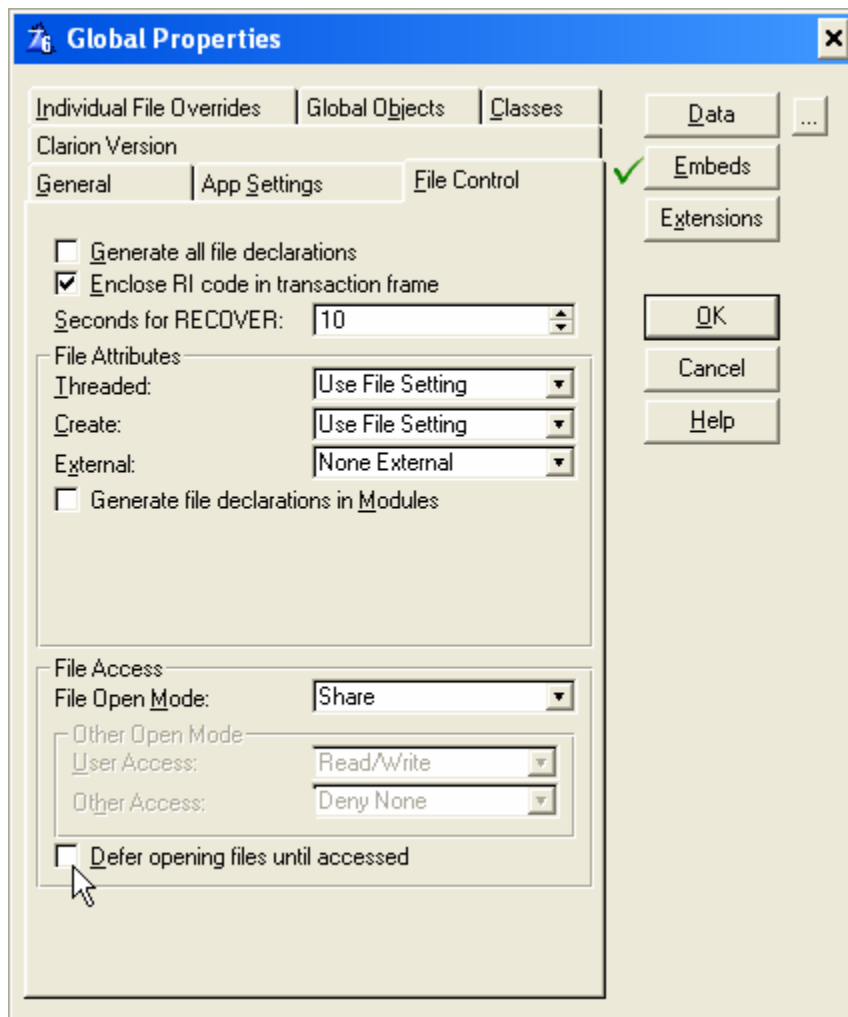
En este caso desde un INI, pero tambien se podria llamar a una simple pantalla de conexión que nos pida los datos de server, base, usuario y password.



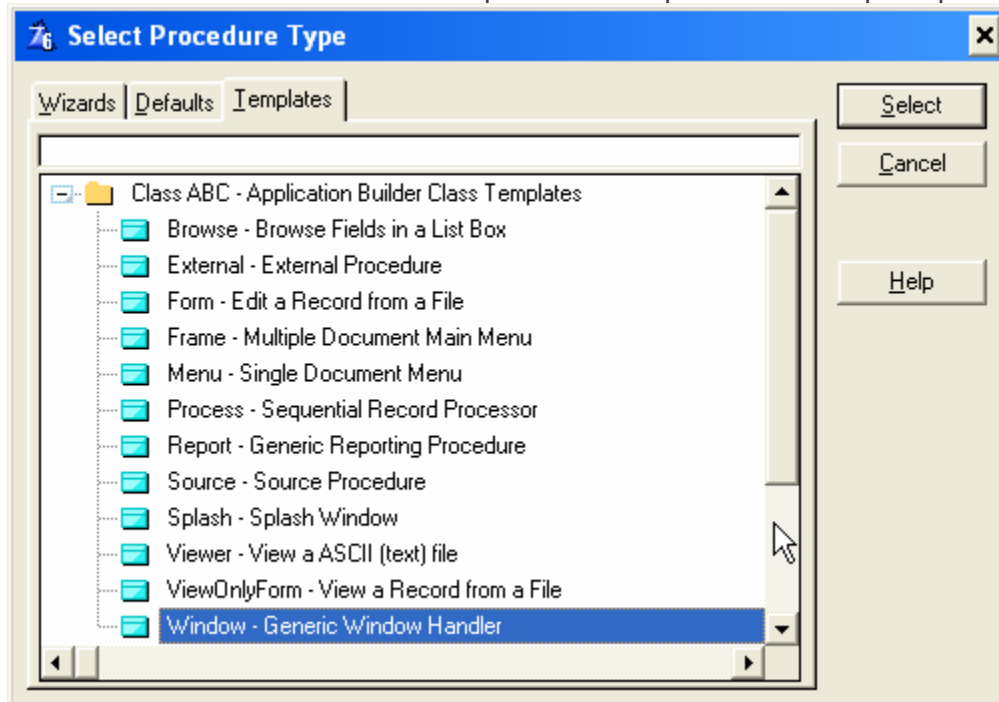
Para simplificar el ejemplo usaremos codigo basico CLARION, no ABC. Por lo tanto deshabilitamos el Defer Open Files.

Si no lo hicieramos, la tabla SQL podria no estar abierta cuando quisieramos recorrerla con NEXT.

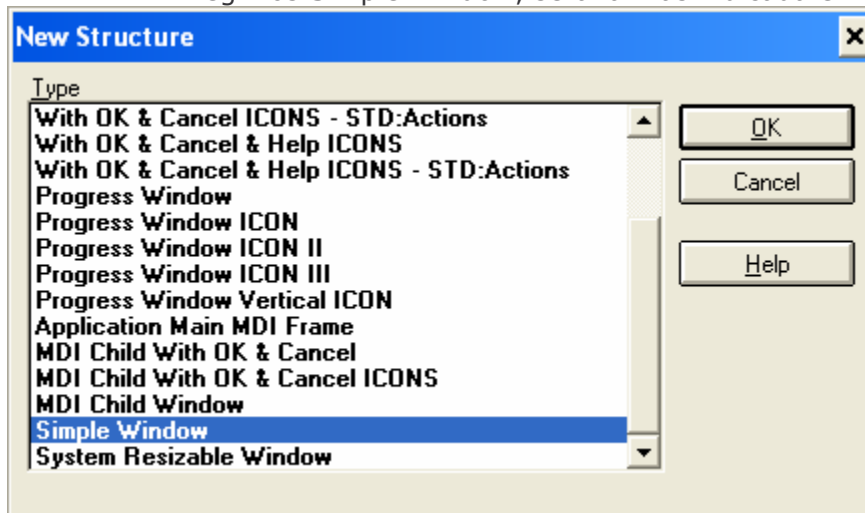
(En general yo "siempre" deshabilito ese check.)



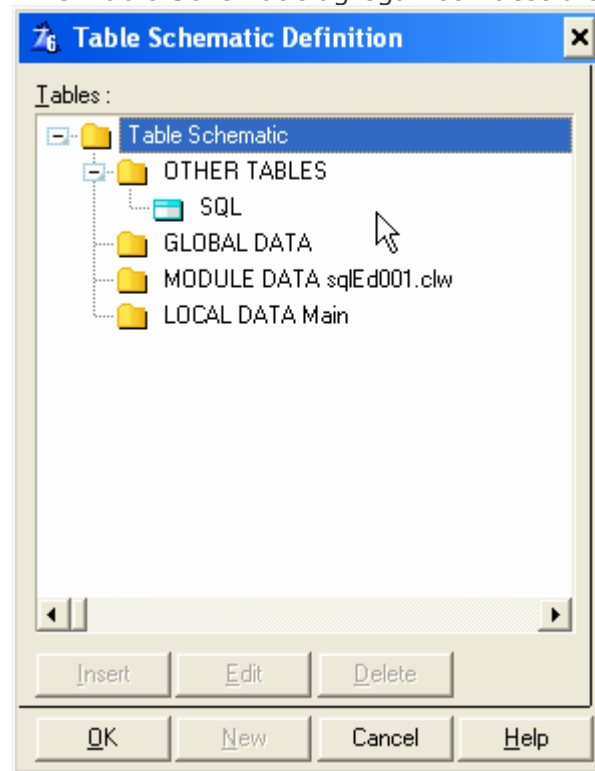
Creamos una ventana para nuestro procedimiento principal.



Elegimos Simple Window, sera la mas indicada en este caso.

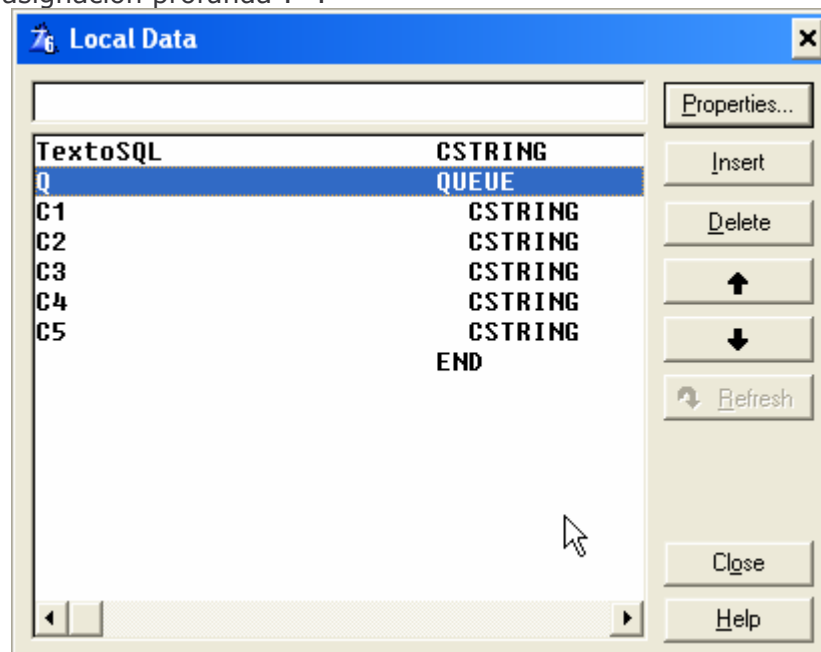


En el Table Schematic agregamos nuestra tabla SQL

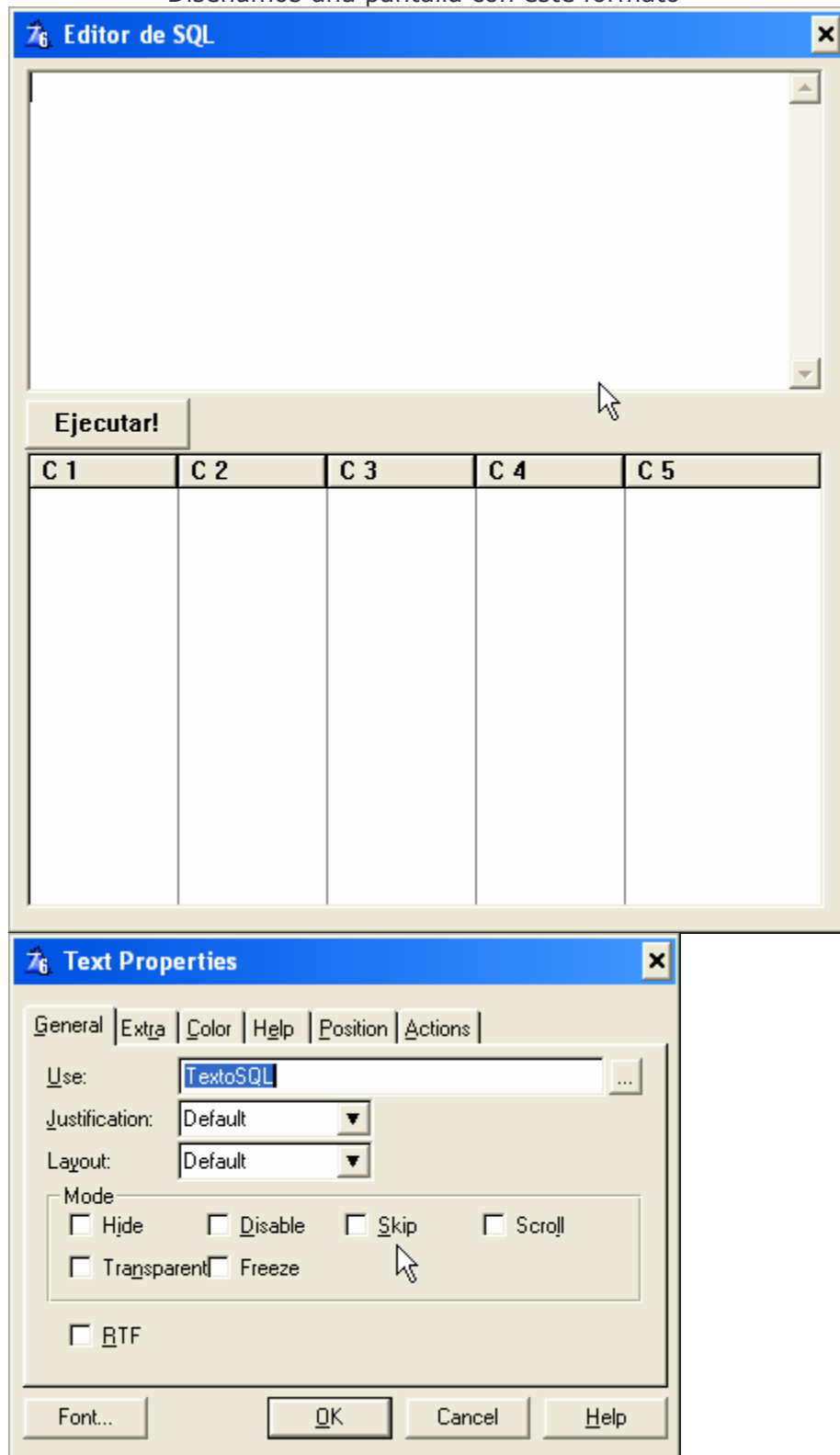


En Local Data definiremos una variable TextoSQL donde poder escribir las sentencias SQL a ejecutar y una cola de memoria donde vamos a guardar los resultados.

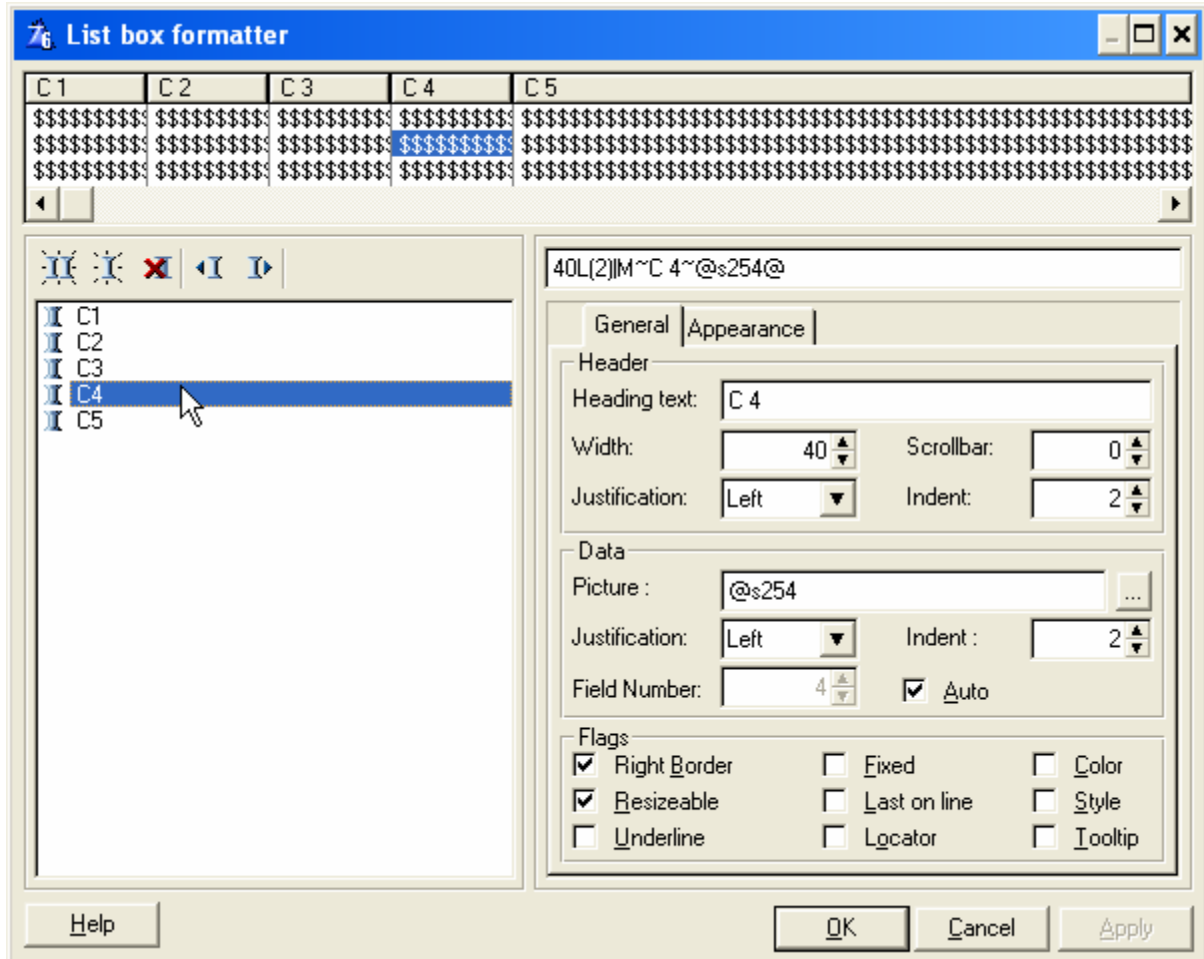
A los campos de la cola de memoria le ponemos los mismos nombres que a la tabla SQL para poder hacer directamente una asignacion profunda :=:



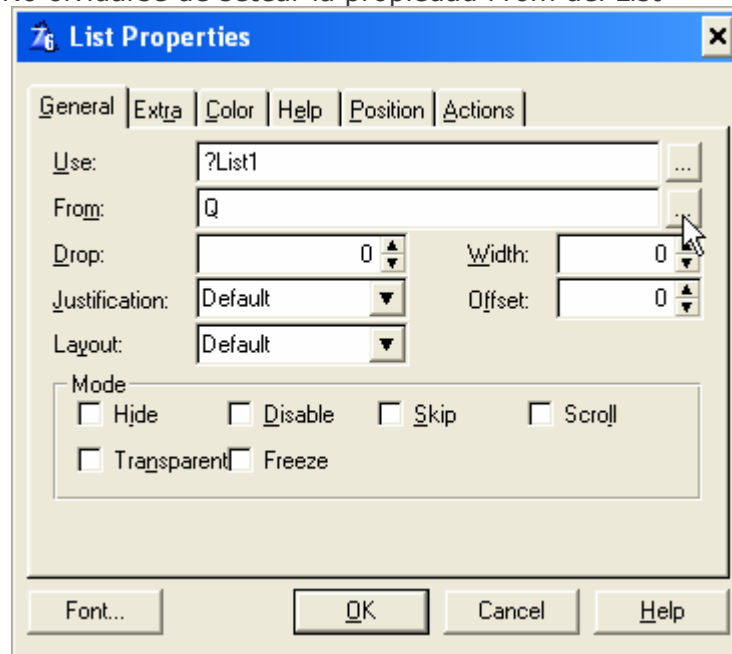
Diseñamos una pantalla con este formato



Creamos un control listbox, NO un template browse.
Elegimos los campos de la cola de memoria **en el mismo orden**.
(Tambien podriamos usar el atributo Field Number si quisieramos
cambiar el orden de los campos en el list, pero en este caso no
tiene sentido)



No olvidarse de setear la propiedad From del List



Este es todo el código que va en el Boton.

En SQL usamos siempre **FileError()** en vez de Error().

FileError() devuelve el error detallado que reporta el motor SQL.

```

Control Events. ?Button1.Accepted (Main)
Exit!  File  Edit  Search

SQL{PROP:SQL}= TextoSQL
IF ERRORCODE() THEN STOP (FILEERROR()).
FREE(Q)
LOOP
    NEXT(SQL)
    IF ERRORCODE() THEN BREAK.
    Q := SQL:RECORD
    ADD(Q)
END

DISPLAY
  
```

Ya tenemos nuestro propio Editor SQL!