DBExpress

Descripción general

Presentaré aquí el nuevo 'bebé' de Borland: la tecnología DBExpress. Esta tecnología viene a reemplazar a la vieja y conocida BDE (Borland Database Engine) que acompaña a Delphi desde sus inicios, y que fue responsable de gran parte de su aceptación. Como veremos, los puntos fuertes de la BDE se mantuvieron en el diseño de DBExpress (desde ahora, DBX) mientras que los puntos débiles se modificaron… o al menos esa fue la intención. Si se logró o no, lo sabremos dentro de un tiempo.

Objetivos de DBExpress

Cuando se planteó la creación de este nuevo sistema de acceso a datos, se tuvieron en cuenta los siguientes objetivos:

- Velocidad de respuesta
- Facilidad de implementación (instalación)
- Basado completamente en SQL
- Facilidad para cambiar de sistema de base de datos
- Multiplataforma
- Simpleza para la creación de nuevos controladores
- Permitir la utilización de características particulares de cada servidor

Estos objetivos se han alcanzado haciendo algunas concesiones en el terreno de las características avanzadas. Los controladores DBExpress son veloces, simples de configurar e instalar, y… terriblemente reducidos en características. Únicamente tienen que proveer de una forma de conectarse al servidor, una forma de ejecutar sentencias SQL, una forma de comunicar los errores que se pueden producir en el proceso, y el tipo más básico de cursores que podemos pedir: unidireccionales y no actualizables.

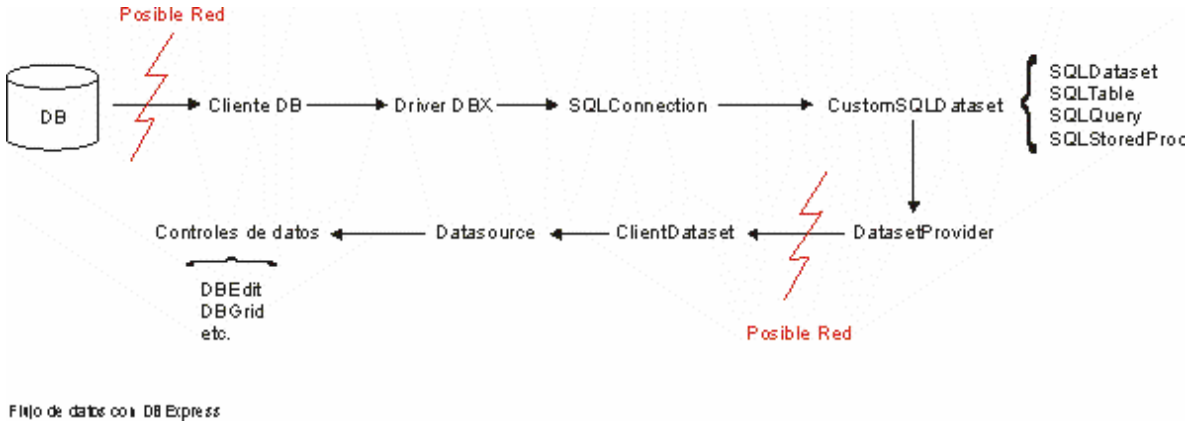
Podemos entender que de un diseño tan minimalista se obtenga velocidad. Los controladores no se molestan en almacenar distintos registros para permitir volver a uno anterior al actual: simplemente, no se puede volver atrás. Tampoco se pueden editar los datos, y por tanto el controlador no necesita ocuparse de la comunicación de los cambios y resolución de conflictos con el servidor. Todas las modificaciones deberán hacerse ejecutando sentencias SQL de acción (Insert, Delete, Update). Esto también significa que se pueden tener características propias de los distintos servidores, siempre que se puedan especificar en SQL.

Pero el hecho que los cursores sean unidireccionales significa que no podemos usar un control como la grilla de datos (DBGrid); además, tenemos que programar la generación de sentencias de acción SQL para cada operación… esto se parece cada vez más a un lenguaje no visual! Bueno, no es para tanto: podemos

programar un sistema de *caché* de registros en un componente, así como un sistema para la generación automática de las sentencias SQL de inserción, modificación y borrado. Aunque preferiría no hacerlo yo, como sin duda estará pensando la mayoría de Uds.

Por suerte Borland ya tiene un par de componentes que realizan esas tareas. Estos componentes aparecieron allá por los tiempos de Delphi 3, si la memoria no me falla; pero en esos momentos de gloria de la BDE era muy poca gente la que los usaba. También tenían una serie de fallos internos que se fueron corrigiendo en las versiones posteriores. En este momento en Delphi 6 / Kylix podemos disfrutar de su madurez y robustez garantizada. Estoy hablando del componente TClientDataset y su compañero del alma, el TDatasetProvider; estos componentes estuvieron desde un principio asociados a la tecnología MIDAS (ahora llamada DataSnap) y por lo tanto sólo disponibles en la versión Enterprise de Delphi. Ahora la cosa ha cambiado, y son una parte integral del esquema de acceso y tratamiento de datos de Borland.

El esquema general de componentes, desde la base de datos hasta los controles visuales de datos (data-aware controls) es el siguiente:



Flujo de datos con DBExpress

Los puntos indicados como ‘posible red’ son los lugares donde podemos separar sin mayores problemas los bloques.

- Entre el servidor de bases de datos y el cliente correspondiente generalmente hay una separación física, con transporte de datos entre el servidor y el cliente a través de una red.
- Entre los componentes DatasetProvider y ClientDataset podemos interponer componentes de conexión DataSnap como TSocketConnection, TDCOMConnection o TSOAPConnection para que se encarguen de la comunicación de paquetes entre el proveedor y el cliente.

El componente ClientDataset nos provee el sistema de almacenamiento de registros en el cliente, que necesitamos para la navegación bidireccional. Cuando los datos se obtienen a través de un descendiente de TDataset (como los componentes de acceso a datos de DBExpress) entonces necesitamos el DatasetProvider para hacer de intermediario entre los componentes. Este componente es capaz de generar las sentencias SQL de acción automáticamente, controladas por propiedades que se pueden configurar en el Inspector de Objetos; recibe los cambios del ClientDataset y los envía al servidor SQL, proveyendo también de algunos eventos que se pueden usar para resolver automáticamente los conflictos que puedan ocurrir.

Resumiendo: la tecnología DBExpress nos brinda acceso eficiente a cursores unidireccionales solamente de lectura. Estos componentes no limitan en modo alguno el acceso a las características avanzadas que

pueda tener cada servidor SQL independiente, ya que únicamente exige esos cursores. Todo lo demás se puede usar a través de SQL o la interface nativa correspondiente.

Sumando a la poción una pareja DatasetProvider – ClientDataset obtenemos lo mismo que teníamos antes con la BDE solo que más simple, rápido y escalable.

¿Y la transparencia de la base de datos final? Cambiar de servidor de base de datos es tan simple como reemplazar una DLL por otra (los controladores) y cambiar los parámetros de la conexión como el nombre del servidor o de la base de datos.

Para la instalación, como veremos en la parte final, solamente hay que copiar en el equipo cliente un par de librerías dinámicas (actualmente son alrededor de 600 Kb). No hay necesidad de registrar ni configurar nada.

Además de todo esto, los mismos componentes están disponibles también en Linux para ser usados con Kylix, y en Solaris –de hecho se escribieron originalmente para esa plataforma, según John Kaster de Borland.

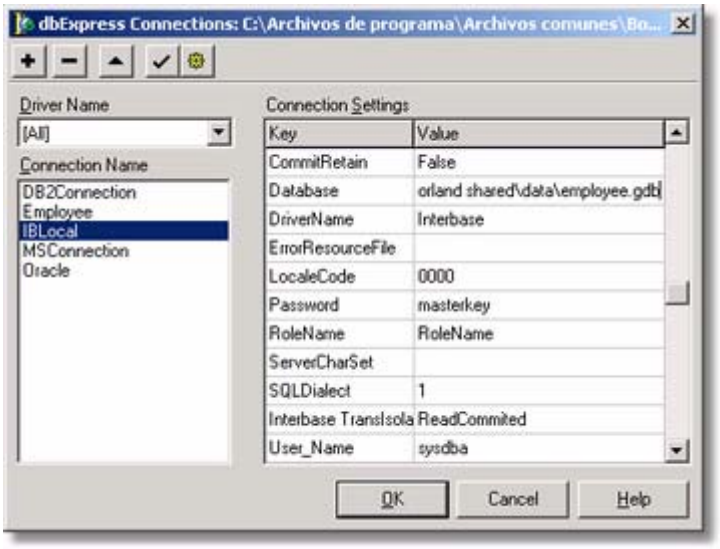
Espero que esto les haya abierto el apetito. Esta tecnología es muy prometedora y fácil de usar y de instalar, por lo que se le augura una larga y próspera vida.

Ejemplo

Comenzaremos en forma poco convencional, con un ejemplo en el cual no usaremos ClientDataset sino que enlazaremos los controles directamente a un SQLTable.

Creamos una nueva aplicación. En la ventana principal –y única- del proyecto colocaremos los siguientes componentes:

- SQLConnection: SQLConnection1
- SQLTable: SQLTable1
- Datasource: Datasource1



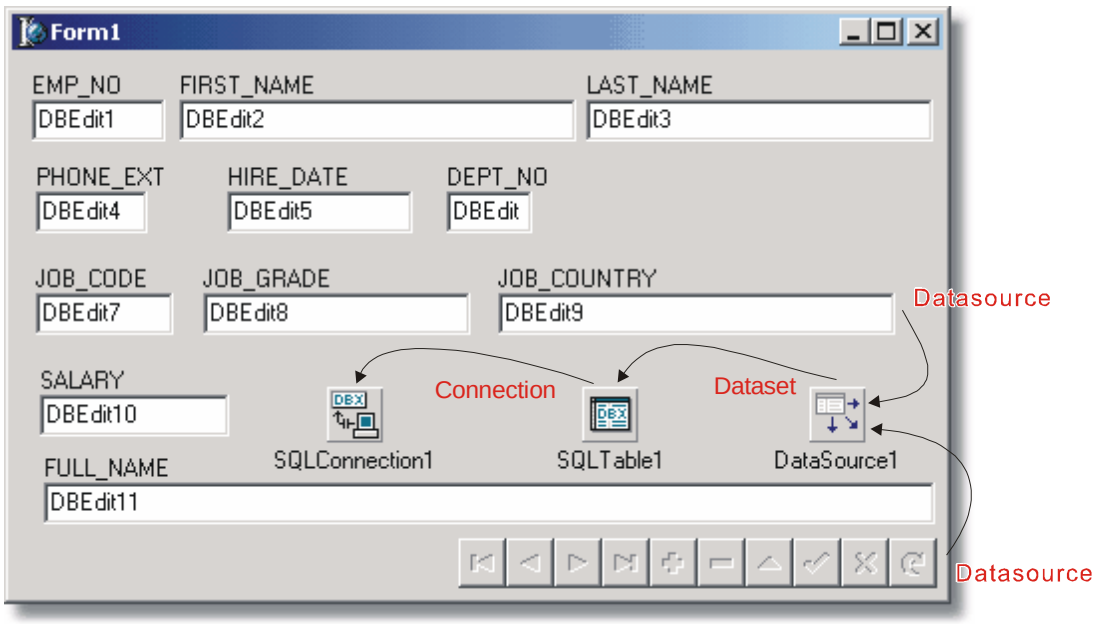
A continuación enlazamos el componente SQLConnection1 con la base de datos haciendo doble clic sobre él y seleccionando ‘IBLocal’ como nombre de la conexión. Debemos colocar el nombre de la base de datos con la que vamos a trabajar; la vieja y querida EMPLOYEE.GDB que se instala con los ejemplos de Delphi. Localice el archivo (la instalación lo pone normalmente en \Archivos de programa\archivos comunes\borland shared\data) y coloque el nombre *con la ruta completa* en el parámetro ‘Database’. Asegúrese también que los parámetros User_Name (nombre de usuario, por defecto ‘sysdba’) y Password (clave de acceso, por defecto ‘masterkey’) contienen datos válidos para acceder a los datos.

Podemos confirmar que hemos puesto bien los valores presionando el botón . Una vez que todo esté bien, aceptamos los cambios presionando el botón OK.

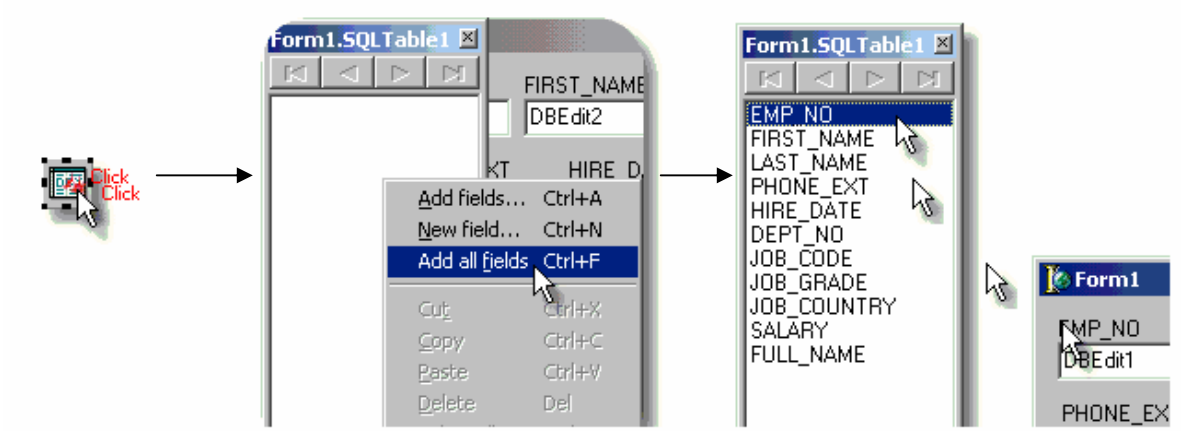
Tenemos la conexión. Ahora nos falta enlazar un componente de tipo Dataset para poder recuperar los datos. Tenemos varias opciones, que veremos luego en forma individual; para este ejemplo colocamos un TSQLTable y referenciamos el SQLConnection1 con la propiedad **Connection**. A continuación seleccionamos el nombre de la tabla EMPLOYEE en la propiedad TableName.

Para poder usar los controles de datos que incluye Delphi, debemos colocar un componente que actúa como intermediario entre el Dataset y los controles: el componente Datasource. Referenciamos el Dataset por medio de la propiedad del mismo nombre.

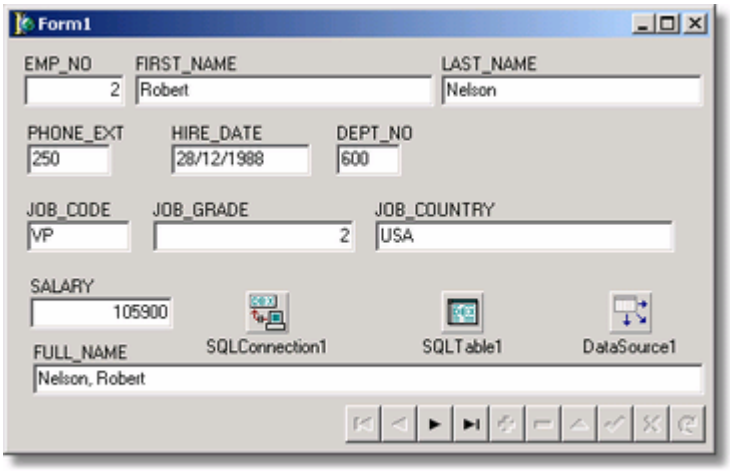
El siguiente esquema muestra las relaciones entre estos tres componentes:



Ahora si podemos colocar los controles de datos. Hacemos doble clic sobre SQLTable1 para mostrar el editor de campos que se ve en la imagen siguiente. En el menú contextual de este editor (invocado como siempre con el botón derecho del ratón) seleccionamos la opción ‘Add all fields’ y veremos que se llena con los *componentes de campo* correspondientes al dataset. Para crear los controles de tipo DBEdit sobre el formulario, arrastramos los componentes de campo y los dejamos caer en el lugar donde deseamos el control; se creará automáticamente un DBEdit y un Label con el nombre del campo.



En este momento ya podemos ver cómo se comportarán los controles de datos, si activamos el dataset poniendo su propiedad Active en True:



Ejecute la aplicación y trate de modificar algún dato; no podrá, así como tampoco podrá volver a un registro anterior. Las únicas operaciones permitidas con un cursor unidireccional de solo lectura como el que nos da el dataset SQLTable1 son la navegación hacia adelante y la vuelta al principio.

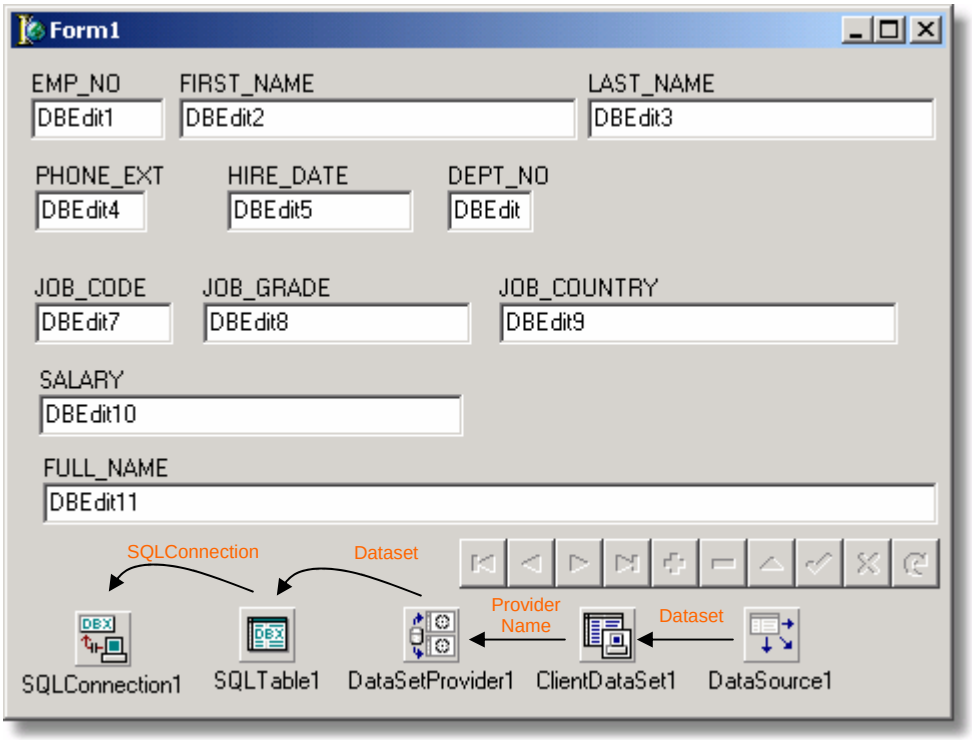
Puede parecer que no se puede hacer mucho con un cursor así; pero en realidad hay muchas operaciones que no necesitan más, por ejemplo la impresión de reportes, y este tipo de cursor es el más rápido que se puede pedir a un servidor SQL. Además ocupa muy pocos recursos, ya que solamente se almacena en memoria un registro, el actual.

Para hacer que el conjunto de datos sea bidireccional y editable necesitamos algún método para mantener un grupo de registros en memoria, y proveer un cursor sobre este grupo. Este método existe desde hace un tiempo, en forma de un componente: TClientDataset.

El componente TClientDataset se estudia en otro lado en relación a las *tablas locales*, o MyBase. Son los mismos componentes que se usan aquí, y básicamente su función es armar una tabla en memoria, con todas las posibilidades de un verdadero descendiente de TDataset. No voy a entrar en detalles sobre el componente TClientDataset, los dejo para la sección dedicada a su utilización con MyBase.

Modificaremos el ejemplo anterior para mostrar la funcionalidad normal del grupo de componentes.

Agregamos a la ventana única del ejemplo anterior un componente TClientDataset, y un TDatasetProvider (ambos en la paleta DataAccess) para conectar al TSQLTable. Las conexiones quedan como sigue:



Ya se puede ver una diferencia al activar el ClientDataset: se activan los botones de edición, inserción y borrado del navegador.

Y ya está, tenemos la posibilidad de movernos libremente y modificar el conjunto de datos. Pero si modifican algo y salen del programa, al volver a ingresar se darán cuenta que las modificaciones no están. ¿Qué pasó?

Los que ya conocen el componente ClientDataset saben lo que hay que hacer. Para los que todavía no lo conocen, digamos solamente que hay que *aplicar* los cambios a la tabla real invocando el método **ApplyUpdates**.

Se pueden aplicar los cambios individualmente a medida que se producen, por ejemplo en el evento AfterPost del ClientDataset:

```
procedure TForm1.ClientDataSet1AfterPost(DataSet: TDataSet);
begin
  ClientDataset1.ApplyUpdates(0);
end;
```

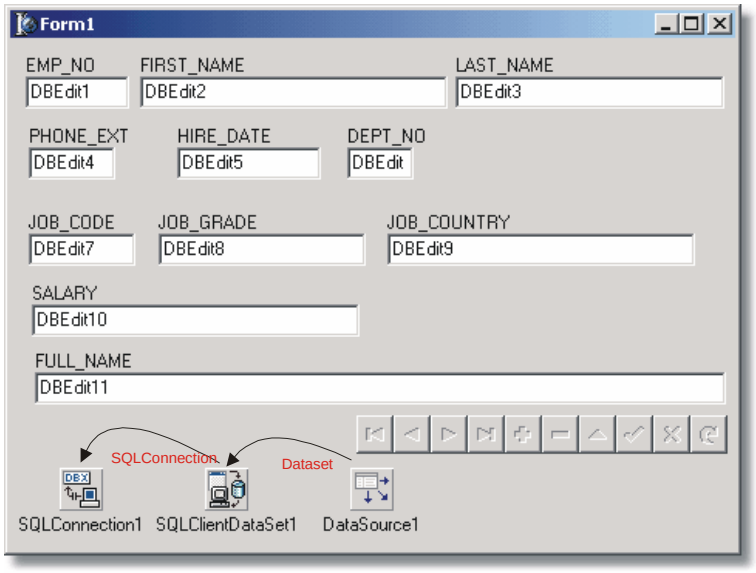
De esta manera conseguimos un funcionamiento similar al que se produciría si los cambios fueran enviados directamente a la tabla física. El parámetro '0' (cero) que usamos con ApplyUpdates indica que no toleraremos ningún fallo: si se produce algún error al grabar en el servidor, se generará una excepción.

También podemos dejar que los cambios se acumulen en la memoria del ClientDataset, y pedirle que los envíe en conjunto al servidor; el método es el mismo (ApplyUpdates), sólo cambia el momento de llamarlo: lo haríamos por ejemplo al cerrar la ventana.

El grupo de registros SQLDataset+DatasetProvider+ClientDataset se utiliza mucho, tanto que Borland decidió crear un nuevo *supercomponente* que sea una mezcla de los tres (en realidad es un ClientDataset

con los otros dos metidos adentro). Este componente se llama SQLClientDataset, y está en la paleta DBExpress.

El mismo ejemplo que antes queda entonces como se ve en la siguiente figura:



Esta última opción es mucho más simple, entonces ¿por qué no usarla siempre? ¿Tiene alguna ventaja el tener los componentes separados? De hecho si, lo de siempre: es más complejo, pero más flexible. Revise el primer gráfico del flujo de datos de DBExpress; sólo es posible la separación en capas si tenemos el DatasetProvider aparte del ClientDataset.

Después de este vistazo a vuelo de pájaro, nos adentraremos en los detalles de los distintos componentes. Tenga siempre en cuenta el esquema del flujo de datos.

Hagamos un resumen de los componentes que nos provee DBExpress:

Icono	Nombre	Función
	SQLConnection	Conexión con la Base de Datos.
	SQLDataset	Acceso a datos. Puede comportarse como un SQLTable, un SQLQuery o un SQLStoredProc según el valor de la propiedad CommandType
	SQLTable	Acceso a una tabla sin escribir SQL.
	SQLQuery	Acceso a datos del servidor a través de una sentencia SQL que puede devolver datos o no.
	SQLStoredProc	Ejecución de un procedimiento almacenado en el servidor. Permite parámetros. Si el procedimiento devuelve una tabla, usar en cambio un componente SQLQuery.

Icono	Nombre	Función
	SQLClientDataset	Componente compuesto por un SQLDataset, un DatasetProvider y un ClientDataset. Provee un cursor bidireccional y totalmente editable.
	SQLMonitor	Permite 'espiar' las sentencias SQL que se envían realmente al servidor.

El último componente no nos da acceso a los datos del servidor, sino que nos permite ver las instrucciones SQL que se ejecutan. Es una ayuda muy valiosa para el aprendizaje y la optimización de consultas.

Conexión

El componente SQLConnection

Lo primero que necesitamos es un medio para conectarnos al servidor. Esta tarea queda en manos del componente TSQLConnection.

Este componente necesita cierta información para poder conectarse; por lo menos, debemos especificar

- El controlador (driver) a usar. Este tiene que ser, por supuesto, compatible con DBExpress; normalmente es una DLL. Se especifica en la propiedad **LibraryName**.

  Con la versión Profesional de Delphi 6 se instalan controladores para Interbase –hasta la versión 6.0- y para MySQL 3.22. Con la versión Enterprise de Delphi se agregan controladores para DB2 y Oracle.

  Después de la salida al mercado de Delphi 6, Borland ha puesto a disposición actualizaciones de los controladores de Interbase (para la versión 6.5) y MySQL (versión 3.23; actualmente está en fase beta un controlador para la versión 4.0). Se ha dicho también que se está trabajando en un controlador para SQLServer y que hay uno disponible para PostgreSQL, pero no lo he comprobado personalmente. Habrá que esperar para estar seguros.

  Los controladores DBX son muy 'livianos', en el sentido que constan de una sola librería compartida (archivo .DLL en Windows) que habrá que distribuir junto con la aplicación compilada.

- La librería de acceso al cliente de base de datos. Es una librería compartida que instala el cliente de la base de datos (por ejemplo, en Interbase se llama gds32.dll). Esta librería es la que permite la conversación con el servidor, y no pertenece a Delphi. Se instala junto con el cliente de la base de datos.

  Se especifica en la propiedad **VendorLib**.

- El nombre de la función de la librería cliente a usar para la conexión. Esta función es exportada por cada cliente particular con un nombre distinto, de ahí la necesidad de especificarlo para la conexión, en la propiedad **GetDriverFunc**.

También tenemos una propiedad que almacena el *nombre* del driver: 'Interbase', 'MySQL', etc. La propiedad se llama **DriverName**, y si le asignamos el nombre de un controlador que esté instalado en el sistema rellenará automáticamente las tres propiedades anteriores.

Además de esos tres, también se pueden especificar otros parámetros para cada conexión; estos nuevos parámetros son característicos de cada sistema de bases de datos, y controlan cosas como el dialecto de lenguaje SQL a usar o el nivel de aislamiento de las transacciones.

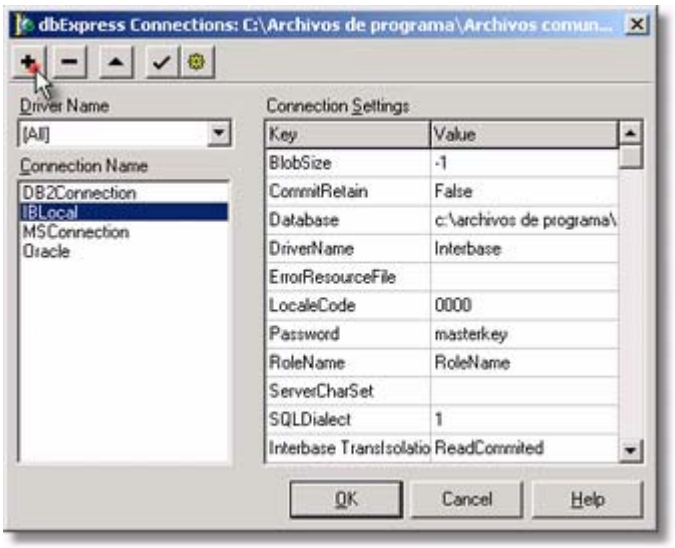
Los valores por defecto para los parámetros de cada servidor se encuentran en el archivo de configuración **dbxdrivers.ini**. Este archivo se copia por defecto en '\Archivos de programa\Archivos comunes\Borland Shared\DBExpress', aunque se puede colocar en cualquier otro lugar mientras lo especifiquemos en la clave de registro 'HKEY_CURRENT_USER\Software\Borland\DBExpress\Driver registry file'.

El componente SQLConnection buscará[1] ese archivo en la carpeta donde reside el ejecutable, y si no lo encuentra ahí leerá la clave del registro para poder ubicarlo. Si esa clave no existe, se creará automáticamente con el valor dado en la clave 'HKEY_LOCAL_MACHINE\Software\Borland\DBExpress\Driver registry file'.

Una conexión creada con los parámetros por defecto no tiene mucho uso, ya que no se especifica la base de datos particular a acceder; podemos configurar este valor en la conexión por defecto para cada base de datos, o crear nuevas conexiones específicas para cada una. Como se imaginarán, seguiremos el segundo camino.

Para cada base de datos podemos crear una conexión personalizada, incluyendo el nombre; luego, logramos el efecto de crear un 'alias' para la base de datos. A partir de su creación, bastará poner el nombre de la conexión particular (el alias) en la propiedad DatabaseName para que entren en vigor todos sus parámetros específicos.

En el primer ejemplo usamos la conexión por defecto para Interbase, llamada IBLocal, modificando su propiedad Database para apuntar al archivo deseado. Ahora modificaremos el ejemplo creando una nueva conexión que llamaremos 'Empleados'.

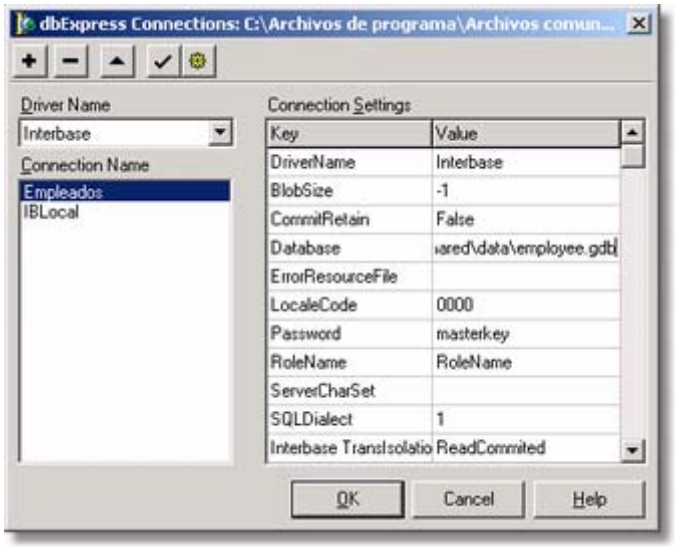


Recupere el proyecto del primer ejemplo, y haga doble clic sobre el componente SQLConnection para abrir el editor de conexiones (también se puede acceder a través de la propiedad DatabaseName).

Ahora presione el botón con el signo '+'. El sistema preguntará por el tipo de driver a usar, y por el nombre de la nueva conexión. Una vez especificados estos valores se copian los parámetros por defecto en la nueva entrada. Ahora colocaremos el nombre y la ruta del archivo Employee.gdb en el parámetro Database.

[1] Si es necesario, porque como veremos pronto la información de conexión se almacena normalmente en el ejecutable.

Cuando aceptamos los datos de la nueva conexión, veremos que algunas propiedades del componente SQLConnection han tomado los valores de los parámetros: las tres principales (DriverName, GetDriverFunc y VendorLib) y la propiedad ConnectionName que muestra el nombre de la conexión recién creada. ¿Y el resto de los parámetros? Fueron a parar a la propiedad correspondiente, es decir, la llamada 'Params'.



Entonces, ¿se guardan todos los parámetros de la conexión? Sí; de hecho el nombre de la conexión es solamente para distinguir este juego de valores de todos los demás. Pero esto tiene una importancia fundamental para la instalación de las aplicaciones desarrolladas con estos componentes: como veremos dentro de poco, los parámetros de la conexión se incluyen en el ejecutable y por lo tanto no se necesitan ni siquiera los archivos .INI. Comparado con la instalación de la BDE, bueno… simplemente no se puede comparar.

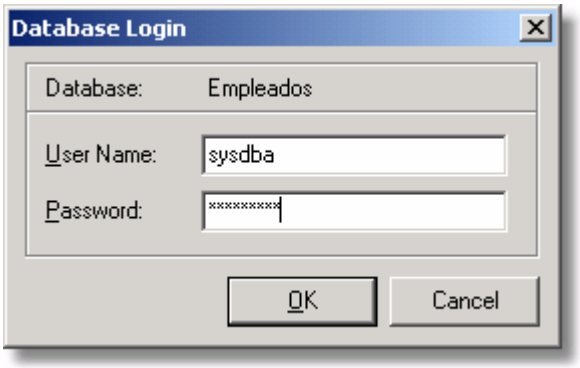
No voy a detallar aquí los valores posibles para cada parámetro de cada tipo de conexión; encontrarán la mayoría en la ayuda. En la página del editor de conexiones (Connection Editor) hay un hiperenlace con el texto 'Driver parameters' casi al final; la página enlazada (que por desgracia no se accede buscando directamente su título) contiene una lista de la mayoría de los parámetros comunes.

Login

La acción de identificarse con el servidor presentando el nombre de usuario y contraseña se denomina en inglés 'Login'. Mantendré aquí este término porque su uso ya está extendido incluso entre los hablantes de castellano.

Entre los parámetros de la conexión hemos visto que existen dos correspondientes al nombre de usuario (**user_name**) y la clave de conexión (**password**). Los valores que se asignen en estos parámetros se enviarán al servidor cuando se intente la conexión.

Aunque rellenemos estos parámetros, si dejamos la propiedad LoginPrompt del componente SQLConnection en True veremos que el sistema nos pregunta por el nombre y la clave de conexión usando un cuadro de diálogo como el siguiente:



Esta ventana está disponible para modificarla en la unit **DBLogDlg**. Si creamos una ventana nueva, debemos mostrarla desde una función con la siguiente definición:

```
function (const ADatabaseName: string; var AUserName,APassword: string): Boolean;
```

Esta función tiene que devolver el nombre de usuario y la clave de conexión en los parámetros AUserName y APassword, respectivamente; el nombre de la Base de Datos le será pasado por Delphi. El valor de retorno de la función indica si se sigue con el proceso de autenticación (True, por ejemplo cuando el usuario presiona 'OK') o no (False).

Esta función debe ser asignada a la variable LoginDialogProc de la unidad DB; es la función que Delphi llama cuando la propiedad LoginPrompt del componente de conexión vale True.

¿Para qué tomarse el trabajo de cambiar el cuadro de diálogo, si el que está cumple bien su función? Bueno, coincido en que no es muy común hacerlo, pero la posibilidad está. Por ejemplo podemos traducir el cuadro de diálogo, o adaptarlo al estilo general de nuestra aplicación.

Hay otra posibilidad para personalizar el proceso de conexión: los eventos.

- Antes de la conexión se produce el evento BeforeConnect. En este momento todavía no se han controlado los parámetros de la conexión, lo que se hace a continuación. Si la propiedad LoadParamsOnConnect es True, se leerán los parámetros de los archivos dbxdrivers.ini y dbxconnections.ini.

  Podemos aprovechar este evento para colocar nuestros propios parámetros de conexión, si tenemos LoadParamsOnConnect en False.

- A continuación se copian los parámetros **User_Name**, **Password** y **Database** en una lista de strings y se comprueba el valor de la propiedad LoginPrompt; si está en True se dispara el evento **OnLogin** enviando la lista de strings creada antes. Si hay un procedimiento de respuesta asociado a este evento, se ejecuta. Si no, se llama a la función apuntada por la variable global *LoginDialogProc* que vimos antes.

  En el evento **OnLogin** podemos especificar nuevos valores para los parámetros de la conexión; si son valores para **User_Name**, **password** o **Database**, los asignamos en la lista de cadenas provista por el evento. Pero también podemos dar valor a otros parámetros, por ejemplo **RoleName** para especificar un rol de conexión, directamente en la propiedad **Params** del componente SQLConnection. Veremos un ejemplo a continuación.

- Ahora se prepara la conexión con los valores de los siguientes parámetros: HostName, RoleName, WaitOnLocks, CommitRetain, AutoCommit, BlockingMode, ServerCharSet, TransIsolation, SqlDialect.
- Se toman los parámetros Database, User_Name y password de la lista de cadenas pasada al evento OnLogin (posiblemente modificada) y se intenta la conexión.
- Si la conexión es exitosa, se dispara el evento AfterConnect.

Veamos un ejemplo de modificación de los parámetros de la conexión en tiempo de ejecución, donde presentaremos al usuario nuestro propio cuadro de diálogo para introducir el nombre de usuario, la clave y el rol de conexión.

Para apreciar el efecto de los roles tendremos que crear un nuevo usuario en la base de datos de empleados, employee.gdb, que se instala con los ejemplos de Delphi. A continuación crearemos dos roles y asignaremos permisos a uno solo de ellos. El usuario recién creado se asociará con los dos roles para probar que funciona la técnica.

Crearemos un usuario llamado 'UsuarioDePrueba' con clave 'prueba'. Puede usar el IBConsole o la utilidad de línea de comandos *gsec*; en este último caso, la secuencia de instrucciones sería

- `C:\archivos de programa\borland\interbase\bin> gsec –user sysdba –password masterkey`
- `GSEC> add UsuarioDePrueba –pw prueba`
- `GSEC> quit`

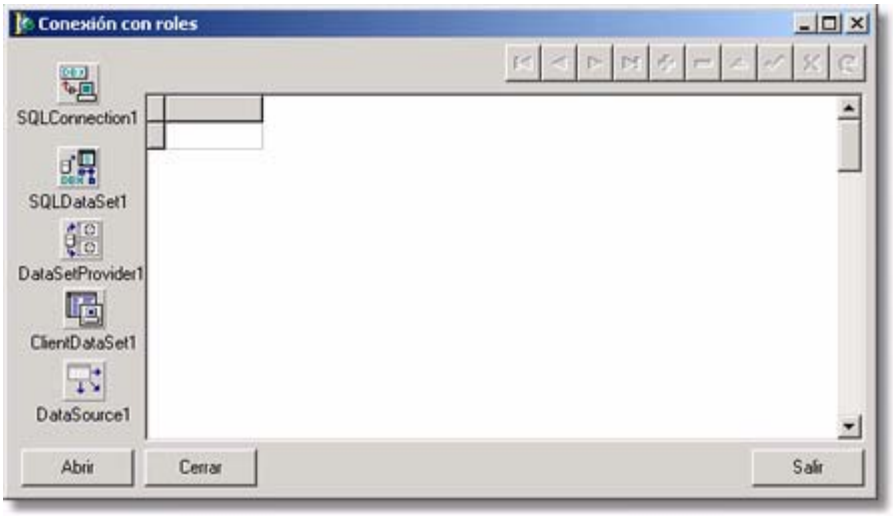
A continuación ejecutamos las siguientes instrucciones:

```
CREATE ROLE ROLDEPRUEBA;
CREATE ROLE ROLSINPERMISO;
GRANT ALL ON EMPLOYEE TO ROLDEPRUEBA;
REVOKE ALL ON EMPLOYEE FROM ROLSINPERMISO;
GRANT ROLDEPRUEBA TO USUARIODEPRUEBA;
GRANT ROLSINPERMISO TO USUARIODEPRUEBA;
```

Tenemos ahora un nuevo usuario ('UsuarioDePrueba') que está habilitado en dos grupos: 'RolDePrueba' con todos los permisos sobre la tabla Employee, y 'RolSinPermiso' con ninguno.

Crearemos una aplicación que permita al usuario ingresar el nombre del grupo (rol) con el que quiere identificarse en la conexión, y trataremos de mostrar el contenido de la tabla Employee en una grilla.

La ventana principal de nuestra aplicación (que he llamado 'FPpal') tiene el siguiente aspecto:



No se preocupe por la proliferación de componentes; simplemente conéctelos entre sí como sigue, para obtener un conjunto de datos navegable y actualizable.

Componente	Propiedad	Valor
`SQLConnection1`	`DriverName`	`Interbase` (se colocan automáticamente las propiedades `GetDriverFunc`, `VendorLib` y `Params`)
	`LoginPrompt`	`True`
`SQLDataset1`	`SQLConnection`	`SQLConnection1`
	`CommandType`	`ctQuery`
	`CommandText`	`select * from employee`
`DatasetProvider1`	`Dataset`	`SQLDataset1`
`ClientDataset1`	`ProviderName`	`DatasetProvider1`
`Datasource1`	`Dataset`	`ClientDataset1`
`DBGrid1`	`Datasource`	`Datasource1`
`DBNavigator1`	`Datasource`	`Datasource1`

El porqué de tantos componentes será explicado más adelante; por ahora, quedémonos solamente con que son necesarios para poder editar los datos en una grilla.

Ahora agregamos una nueva ventana a la aplicación (File | New | Form) y ponemos etiquetas y editores para que quede como se ve en la imagen.

He renombrado los editores para que sea más fácil de seguir el código; de arriba hacia abajo, son '`edNombreUsuario`', '`edClave`' y '`edRol`'. El formulario se llama '`FLogin`'.

Ahora grabamos todo, y vamos al código.

Tomemos primero el componente SQLConnection. Escriba el siguiente código como respuesta al evento **BeforeConnect**:

```
procedure TFPpal.SQLConnection1BeforeConnect(Sender: TObject);
begin
  with SQLConnection1.Params do
  begin
    Values['Database']:= 'Database'; //no puede quedar vacío
    Values['User_Name']:= '';
    Values['password']:= '';
  end; //with
end;
```

Como vemos, no hacemos otra cosa que dar valores a los tres parámetros más importantes de la conexión. Este evento se dispara antes de intentar la conexión.

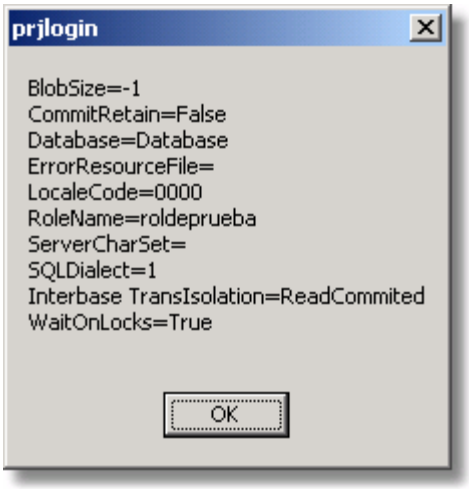
Ahora vamos al evento siguiente en la secuencia: **OnLogin**

```
procedure TFPpal.SQLConnection1Login(Database: TSQLConnection;
  LoginParams: TStrings);
begin
  if FLogin.ShowModal=mrOk then
  begin
    LoginParams.Values['Database']:= 'c:\archivos de programa\archivos
comunes\borland shared\data\employee.gdb';
    LoginParams.Values['User_name']:= FLogin.edNombreUsuario.text;
    LoginParams.Values['password']:= FLogin.edClave.Text;
    SQLConnection1.Params.Values['RoleName']:= FLogin.edRol.text;
  end
  else
    abort;
end;
```

Aquí es donde mostramos la segunda ventana, pidiendo al usuario que se identifique con su nombre, contraseña y rol a asumir. Si se cancela la ventana se aborta la operación; caso contrario, se actualizan los dos parámetros de la lista de cadenas **LoginParams** y el parámetro **RoleName** *en la propiedad Params del componente SQLConnection*.

Si se acepta el diálogo anterior, se intentará finalmente la conexión y en caso de ser exitosa se disparará el evento AfterConnection. Aprovecho en este evento para mostrar la lista completa de parámetros (el contenido de SQLConnection.Params) y comentarles algo curioso.

```
procedure TFPpal.SQLConnection1AfterConnect(Sender: TObject);
begin
  ShowMessage(SQLConnection1.Params.Text);
end;
```



Noten que los parámetros User_Name, password y Database no tienen los valores que les asignamos en el evento OnLogin; el componente en realidad nunca mezcla la lista de cadenas que llenamos en el evento OnLogin con la propiedad Params, que contiene el resto de los parámetros. La única relación entre las dos se da en el momento anterior al disparo de OnLogin, cuando el componente toma los valores de los parámetros Database, User_Name y password como valores iniciales de la lista de cadenas que pasa al evento. Una vez que se efectúa la conexión, esta lista de cadenas temporal se elimina.

La conexión se efectúa cuando se activa el ClientDataset, que recupera los registros de la tabla de empleados y permite su despliegue en la grilla. El código se ejecuta en respuesta al botón 'Abrir':

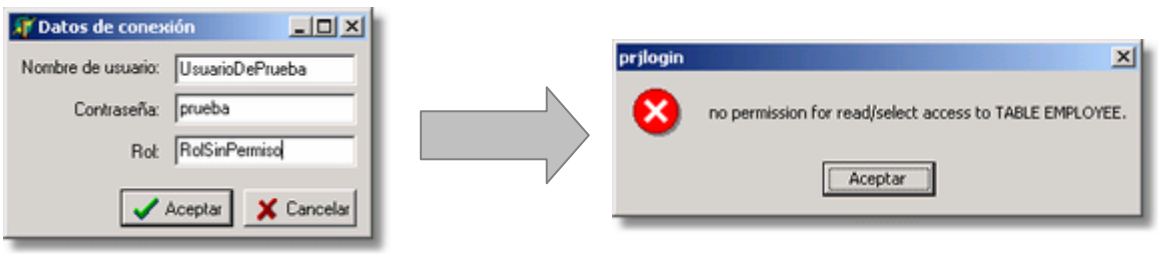
```
procedure TFPpal.Button2Click(Sender: TObject);
begin
  ClientDataset1.open;
end;
```

He previsto otro botón ('Cerrar') para cerrar la tabla y la conexión, para poder probar otra vez sin cerrar la aplicación:

```
procedure TFPpal.Button3Click(Sender: TObject);
begin
  ClientDataset1.Close;
  SQLConnection1.Close;
end;
```

Note que la conexión se abre automáticamente al abrir el conjunto de datos; pero si cerramos la conexión no se cierra el conjunto de datos, hay que hacerlo explícitamente. Todo esto será explicado más adelante.

Y esto es todo en cuanto al código. Pruebe ingresando como 'UsuarioDePrueba' con clave 'prueba' (éste es el único que distingue entre mayúsculas y minúsculas) y cambiando el rol entre 'RolDePrueba' y 'RolSinPermiso'. En el segundo caso les debería mostrar un mensaje de error:



Ejecución de sentencias SQL

El componente SQLConnection nos da la posibilidad de ejecutar sentencias SQL directamente, sin necesidad de otro componente. Tenemos dos métodos disponibles:

- ExecuteDirect

  Este método no podía ser más simple:

  ```
  function ExecuteDirect(const SQL: string): LongWord;
  ```

  el parámetro, como habrá adivinado, es la sentencia SQL a ejecutar. El resultado es 0 si todo fue bien, caso contrario no hay respuesta porque se produce una excepción.

  Como vemos en la declaración, no tenemos la posibilidad de usar parámetros o recibir de vuelta un cursor; esto elimina las sentencias SELECT de las posibilidades de ejecución a través de este método.

  Pero hay otro método más flexible:

- Execute

  Esta función también espera una cadena con una sentencia SQL válida, pero está preparada para trabajar con parámetros y puede devolver un cursor unidireccional.

  ```
  function Execute(const SQL: string; Params: TParams;
    ResultSet:Pointer=nil): Integer;
  ```

  Los parámetros son muy simples: **SQL** es la sentencia en sí, **Params** es una lista de parámetros y **ResultSet** es un puntero a un TCustomSQLDataset donde queremos que el componente deje los datos recuperados, si corresponde.

  El resultado es la cantidad de registros afectados por la sentencia SQL, o cero si se devuelve un cursor.

  Por ejemplo, para insertar un registro podemos usar algo como

  ```
  SQLConnection1.Execute('insert into employee (first_name, last_name,'+
    'hire_date,dept_no,job_code,job_grade,job_country,salary) '+
    'values (''Juan'',''Perez'',''2002-03-01'',''000'',''Doc'',3,'+
    ''''USA'',54000)',nil);
  ```

  La sentencia anterior devuelve un valor de 1, indicando que se modificó un registro.

  Usando un parámetro:

  ```
  var
    ps: TParams;
    t: TParam;
  begin
    ps:= TParams.Create;
    t:= TParam.Create(ps,ptInput);
    t.Name:= 'Fecha';
    t.DataType:= ftDate;
    t.Value:= date;
    SQLConnection1.Execute('insert into employee (first_name, last_name,'+
  ```

```
    'hire_date,dept_no,job_code,job_grade,job_country,salary) '+
    'values (''Juan'',''Perez'',:fecha,''000'',''Doc'',3,'+
    '''USA'',54000)',ps);
  ps.Free;
```

Finalmente, si la sentencia SQL devuelve un cursor, el componente SQLConnection crea una instancia de TCustomSQLDataset y le entrega el cursor. El método Execute espera ahora un *puntero a un TCustomSQLDataset* o descendiente (cualquiera de los componentes de acceso a datosde DBExpress), al que asigna el componente creado al vuelo:

```
var
  r: TSQLDataset;
begin
  SQLConnection1.Execute('select * from job',nil,@r);
  DatasetProvider1.DataSet:= r;
  ClientDataset1.Close;
  ClientDataset1.Open;
```

El número devuelto por Execute es 0 esta vez; si se produce un error, se genera una excepción por lo que no recibimos nada de vuelta[2].

Mantener la conexión abierta

Las conexiones a bases de datos son objetos bastante *costosos*, tanto en tiempo como en recursos del servidor. En cualquier aplicación que se precie de serlo usaremos más de un componente dataset para el acceso a distintas tablas de la misma base de datos. Necesitamos una conexión, que teóricamente podría establecerse en el momento de pedir los datos y cerrarse cuando ya los hemos recuperado.

Pero esto lleva demasiado tiempo. Cada vez que se establece una conexión hay que autenticarse con el servidor, crear estructuras para los procesos tanto en el servidor como en el cliente que se conecta, etc. Todo esto tiene una demora, que es directamente experimentable cuando conectamos a un servidor.

Hay una solución sencilla a este problema: mantener abierta la conexión aunque no haya ningún dataset conectado. Esto es lo que normalmente se hace en Delphi, pero se puede controlar mediante la propiedad *KeepConnection* del componente SQLConnection. Si está en True (el valor por defecto) entonces se mantendrá abierta la conexión aunque se desconecten todos los conjuntos de datos; si le asignamos False, se cerrará la conexión cada vez que detecte que no hay conjuntos de datos activos.

Esto es muy importante cuando se trabaja con conjuntos de datos que se activan, toman los datos y se cierran inmediatamente como es el caso cuando usamos ClientDatasets[3] y en las aplicaciones de servidor Web.

Resumiendo: deje la propiedad KeepConnection en True.

Hay una propiedad más en el componente SQLConnection de la que nos ocuparemos más tarde: TableScope. Pero ahora es más urgente conocer los componentes que nos permitirán acceder a los datos propiamente dichos.

---

[2] En la documentación dice que recibimos un código de error de DBExpress… cosa que no es cierta, como se puede ver en el código fuente.

[3] Con PacketRecords = -1

Transacciones

Las transacciones en DBX se administran desde el mismo componente TSQLConnection, a través de los métodos StartTransaction, Commit y Rollback; también es de utilidad la propiedad lógica InTransaction, que indica si actualmente estamos dentro de una transacción o no, y la propiedad TransactionsSupported que indica si el servidor SQL acepta transacciones (MySQL, por ejemplo, no manejaba transacciones hasta la versión 3.23).

Las transacciones se configuran a través de un registro de tipo TTransactionDesc que tiene la siguiente estructura:

```
type
  TTransactionDesc = packed record
     TransactionID    : LongWord;
     GlobalID         : LongWord;
     IsolationLevel   : TTransIsolationLevel;
     CustomIsolation  : LongWord;
   end;
```

Donde el nivel de aislamiento (IsolationLevel) es uno de los siguientes:

```
type
  TTransIsolationLevel = (xilREADCOMMITTED, xilREPEATABLEREAD,
    xilDIRTYREAD, xilCUSTOM);
```

Estos niveles de aislamiento alcanzan para la mayoría de los servidores; si tenemos alguno nuevo que necesite un valor distinto podemos usar el nivel ‘xilCustom’ en el campo IsolationLevel y el nuevo nivel en el campo CustomIsolation del registro de descripción. Ninguno de los controladores estándar incluidos en Delphi necesita estos campos, pero allí están.

Cuando trabajamos con Interbase podemos tener varias transacciones activas simultáneamente (no anidadas, sino concurrentes); para identificar cada una usamos un entero en el campo TransactionID. Este valor tendrá que ser único para cada transacción concurrente.

El campo GlobalID es, según la documentación, propio de las conexiones con Oracle. Y no hay más información… personalmente creo que se usa en las transacciones distribuidas, donde hay un ID global y uno local. Para los casos ‘normales’ con una sola base de datos, este parámetro puede ser cero.

El esquema de trabajo normal con las transacciones es (veremos algunos ejemplos en la parte de relaciones Maestro/Detalle):

- Llenar un registro TTransactionDesc con los valores apropiados, por ejemplo

```
var
  txn: TTransactionDesc;
begin
  txn.TransactionID:= 2; //El número no importa mucho
  txn.GlobalID:= 0;
  txn.IsolationLevel:= xilREADCOMMITTED;
```

- Antes de comenzar la transacción, debemos saber si el sistema permite transacciones concurrentes (como Interbase). En los sistemas que no las soportan, tenemos que terminar una transacción antes de comenzar otra por lo que únicamente comenzaremos la nueva si la propiedad InTransaction es Falsa:

```
if not SQLConnection1.InTransaction then //Si el sistema soporta
                                         //transacciones anidadas,
                                         //no hace falta la pregunta
  SQLConnection1.StartTransaction(txn);
try
```

- Realizamos todos los cambios que queremos sobre la base de datos…
- Y finalizamos la transacción

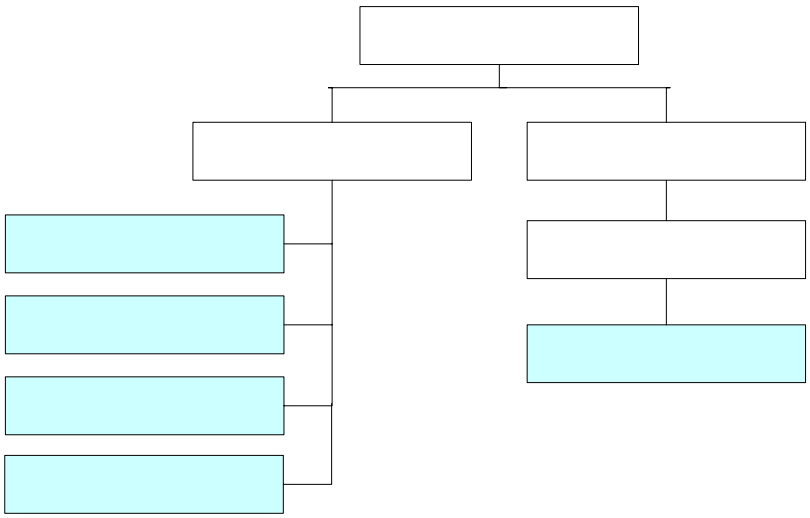
```
  SQLConnection1.Commit(txn);
except
  SQLConnection1.Rollback(txn);
end;
```

Note que hemos encerrado las instrucciones de modificación de la base de datos en un bloque protegido try..except, para poder cancelar todo en caso de error. Todo o nada, esa es la esencia de las transacciones.

Notemos que cuando abrimos un dataset, generalmente se inicia una transacción *implícita* automáticamente… hay que tener esto en cuenta cuando trabajamos con una base de datos distinta de Interbase, para no intentar comenzar otra hasta que no termine la actual.

Acceso a los datos

Para el acceso a los datos disponemos de varios componentes; notemos primero que todos descienden de un antecesor común, llamado TCustomSQLDataset, como se ve en la imagen:



He incluido aquí también el componente TSQLClientDataset con su estirpe; hablaremos de él en breve.

Los componentes sombreados son los que figuran en la paleta de componentes, en la página 'DBExpress', y que podemos usar en nuestras aplicaciones.

La funcionalidad de los componentes descendientes de TCustomSQLDataset está ya presente en la clase base; simplemente se adapta en cada caso para proveer compatibilidad con los 'antiguos' componentes que usábamos con la BDE. El componente TSQLDataset es el que mantiene casi intactas las características de su antecesor, solamente publica las propiedades pertinentes y sobrescribe un par de métodos por lo que mirándolo a él estamos prácticamente viendo la clase base TCustomSQLDataset.

Ninguno de estos componentes es difícil de usar, pero el más simple para empezar, porque genera automáticamente las sentencias SQL necesarias para obtener los datos, es el TSQLTable. De él nos ocuparemos a continuación.

SQLTable

Este componente existe sólo para proveer una manera simple de acceso a todos los campos de una sola tabla. Lo único que necesitamos decirle es la conexión a usar y el nombre de la tabla de la cual obtener los datos. Eso es todo.

¿Un ejemplo usando este componente? Vuelva al principio y lo encontrará. No hay más tela para cortar aquí.

Notemos que internamente el proceso es un poco más complicado; todas las comunicaciones con el servidor se hacen usando SQL, por lo que se debe crear una sentencia válida que recupere todos los campos de la tabla pedida. De hecho, se genera una instrucción del tipo `Select * from tabla`[4].

Aprovecharé que el componente TSQLTable es tan fácil de usar para incorporar un poco más de complejidad con otro componente: el monitor de SQL, TSQLMonitor.

SQLMonitor

Este componente tiene por misión 'escuchar' todo lo que el SQLConnection le dice al servidor SQL, tomar nota y esperar nuevas órdenes. Es una herramienta invaluable para saber con exactitud qué está pasando, con todos los detalles. Armados con este conocimiento podemos optimizar los accesos al servidor o encontrar fallos en el proceso.

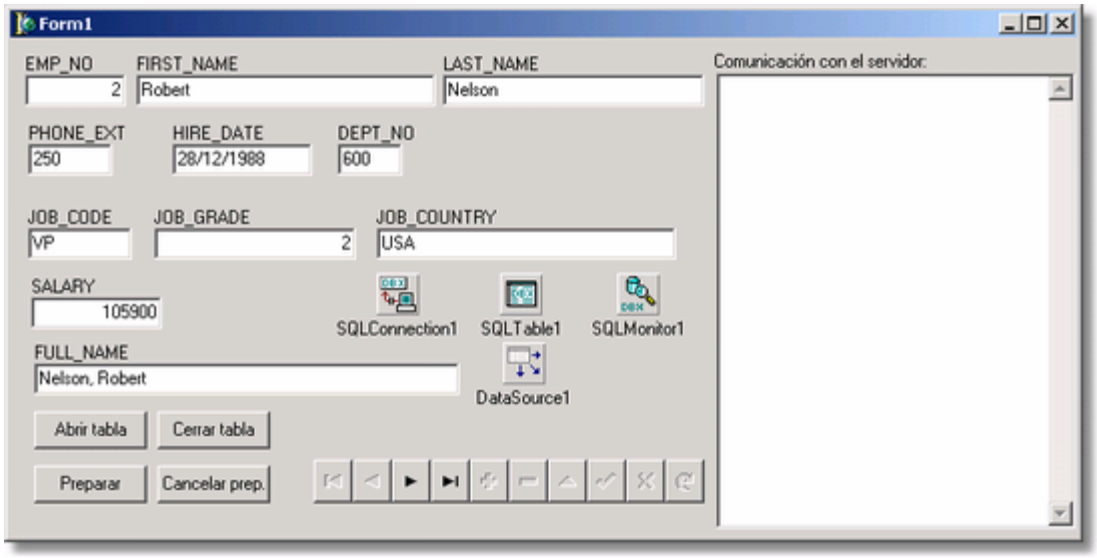
Coloque un TSQLMonitor en algún formulario o módulo de datos desde donde tenga acceso al componente de conexión a monitorear, enlace el monitor a éste por medio de la propiedad **SQLConnection**, y actívelo poniendo la propiedad **Active** en True. Ya tenemos nuestro espía en posición y recibiendo; todos los mensajes que intercepta son almacenados en la propiedad TraceList (de tipo TStringList).

---

[4] El código real tiene que considerar también la posibilidad de que esta tabla esté enlazada como detalle de otra al generar el SQL, ya que en ese caso debe usar parámetros.

Para captar la importancia de esta información es bueno mostrar un ejemplo en este momento. Haremos una variación de la aplicación del primer ejemplo, mostrando en un Memo los comandos interceptados a medida que se envían.

El form principal ha sido modificado agregando algunos botones, un SQLMonitor y un memo para ver los mensajes:



Seleccione SQLConnection1 en la propiedad **SQLConnection** del monitor, SQLMonitor1. A continuación presento el código de los cuatro botones, reconocibles por los nombres:

```
procedure TForm1.bAbrirTablaClick(Sender: TObject);
begin
  SQLTable1.Open;
end;

procedure TForm1.bCerrarTablaClick(Sender: TObject);
begin
  SQLTable1.Close;
end;

procedure TForm1.bPrepararClick(Sender: TObject);
begin
  SQLTable1.Prepared:= true;
end;

procedure TForm1.bCancelarPreparacionClick(Sender: TObject);
begin
  SQLTable1.Prepared:= false;
end;
```

Y para mostrar los mensajes, asociamos el siguiente código al evento OnLogTrace del monitor:

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject;
  CBInfo: pSQLTRACEDesc);
begin
  memo1.Lines.Add(cbinfo.pszTrace);
end;
```

El parámetro **CBInfo** es un puntero a un registro, como se puede ver en la unidad **sqlExpr**:

```
  pSQLTRACEDesc = ^SQLTRACEDesc;
  SQLTRACEDesc = packed record              { trace callback info }
    pszTrace        : array [0..1023] of Char;
    eTraceCat       : TRACECat;
    ClientData      : Integer;
    uTotalMsgLen    : Word;
  end;
```

De este registro extraño nos interesa por ahora el primer campo, donde encontraremos el texto del mensaje enviado al servidor. Como vemos, se agrega directamente como una nueva línea al memo.

Podemos usar esta pequeña aplicación para hacer las pruebas que confirmen lo que sigue.

Es posible indicar al monitor que guarde los mensajes interceptados a un archivo de texto, cuyo nombre se da en la propiedad **Filename**. Si ponemos **AutoSave** en True, la lista *completa* se grabará en ese archivo *por cada comando interceptado*. Esto puede ser una gran carga para el sistema, ya que la lista crece rápidamente; es más eficiente dejar **AutoSave** en False y guardar nosotros la lista en momentos específicos usando el método **SaveToFile** del componente. Se puede llamar a este método sin especificar el nombre del archivo destino, y se tomará el valor de la propiedad **Filename**.

El siguiente es un ejemplo de los mensajes interceptados cuando se abre un SQLTable con TableName='EMPLOYEE', se recupera un registro y se cierra.

```
1.  INTERBASE - isc_attach_database
2.  INTERBASE - isc_dsql_allocate_statement
3.  INTERBASE - isc_start_transaction
4.  select * from EMPLOYEE
5.  INTERBASE - isc_dsql_prepare
6.  INTERBASE - isc_dsql_describe_bind
7.  INTERBASE - isc_dsql_execute
8.  INTERBASE - isc_dsql_allocate_statement
9.  SELECT 0, '', '', A.RDB$RELATION_NAME, A.RDB$INDEX_NAME, B.RDB$FIELD_NAME, B.RDB$FIELD_POSITION, '', 0,
    A.RDB$INDEX_TYPE, '', A.RDB$UNIQUE_FLAG, C.RDB$CONSTRAINT_NAME, C.RDB$CONSTRAINT_TYPE FROM RDB$INDICES
    A, RDB$INDEX_SEGMENTS B FULL OUTER JOIN RDB$RELATION_CONSTRAINTS C ON A.RDB$RELATION_NAME =
    C.RDB$RELATION_NAME AND C.RDB$CONSTRAINT_TYPE = 'PRIMARY KEY' WHERE (A.RDB$SYSTEM_FLAG <> 1 OR
    A.RDB$SYSTEM_FLAG IS NULL) AND (A.RDB$INDEX_NAME = B.RDB$INDEX_NAME) AND (A.RDB$RELATION_NAME =
    UPPER('EMPLOYEE')) ORDER BY RDB$INDEX_NAME
10. INTERBASE - isc_dsql_prepare
11. INTERBASE - isc_sqlcode
12. INTERBASE - isc_dsql_free_statement
13. INTERBASE - isc_dsql_fetch
14. INTERBASE - isc_commit_retaining
15. INTERBASE - isc_dsql_free_statement
```

Un tanto complicado, ¿no? Esta es la forma en que el SQLConnection se comunica con el servidor SQL, en este caso Interbase. He resaltado la línea con la sentencia Select (paso 4) que recupera efectivamente los datos; todo lo demás es la preparación de la transferencia de datos, que se realiza recién en el paso 13: el comando isc_dsql_fetch.

Recuperación de datos extra y preparación de las consultas

En realidad, como podemos ver en las instrucciones anteriores, se están ejecutando dos sentencias Select; la resaltada en el paso 4 trae los datos que nosotros pedimos (todos los campos de la tabla de empleados) mientras que la segunda (paso 9) recupera información de los índices definidos sobre la tabla. Esta

información extra se obtiene para ayudar a otros componentes que puedan necesitarla, como los de DataSnap; se ejecuta una sola vez al abrir el conjunto de datos y puede evitarse poniendo la propiedad **NoMetadata** del dataset en True. Si hacemos esto no veremos cambios en nuestra aplicación, ya que es muy simple y no necesita realmente la información adicional.

Limpiemos entonces un poco el listado anterior eliminando las sentencias extra. En el listado siguiente se ve la salida del monitor cuando la propiedad **NoMetadata** del componente SQLTable es True, con algunos comentarios agregados:

```
Conexión
1.  INTERBASE - isc_attach_database
2.  INTERBASE - isc_dsql_allocate_statement
3.  INTERBASE - isc_start_transaction

Preparación de la consulta
4.  select * from EMPLOYEE
5.  INTERBASE - isc_dsql_prepare
6.  INTERBASE - isc_dsql_describe_bind

Ejecución de la sentencia SQL
7.  INTERBASE - isc_dsql_execute

Recuperación de un registro
8.  INTERBASE - isc_dsql_fetch

Cierre de la tabla
9.  INTERBASE - isc_commit_retaining
10. INTERBASE - isc_dsql_free_statement
```

He resaltado aquí las líneas 4, 5 y 6, que contienen los comandos emitidos *una sola vez* antes de comenzar a tomar datos; le están diciendo al servidor que se *prepare* para la instrucción de la línea 4: que guarde lugar para eventuales parámetros, arme el **plan de ejecución**, etc. Esta operación se llama *preparación de la consulta* y es un paso previo a la ejecución de sentencias SQL.

Una vez que se ha preparado y ejecutado la consulta, cada registro se obtiene con un solo comando: isc_dsql_fetch. Podemos verlo si avanzamos por la tabla en el ejemplo anterior.

La preparación de una consulta se controla mediante la propiedad **prepared**: si colocamos True, se prepara inmediatamente la instrucción; si asignamos False, se eliminan los objetos creados durante la preparación. Cuando esto pasa, el conjunto de datos se cierra[5].

Es conveniente poner `Prepared:= true` antes de abrir el conjunto de datos cuando tenemos que trabajar mucho abriendo y cerrando el conjunto de datos, como cuando tenemos parámetros (que veremos luego); de esta manera, la instrucción se prepara una sola vez y luego solamente se envían los valores de los parámetros y se ejecuta la sentencia. La ganancia en velocidad era muy notable con la BDE, ya que si no lo hacíamos explícitamente cada consulta se preparaba automáticamente antes de ejecutar la sentencia y se 'despreparaba' inmediatamente después. En DBExpress no es igual, ya que el estado de preparación o no se mantiene entre distintas ejecuciones; no obstante, no queda claro si esto es una característica nueva de estos componentes o algo que se le escapó a Borland. De todas maneras, conviene hacerse el buen hábito.

Experimente un poco mirando los comandos que se envían al servidor al cerrar la tabla, volverla a abrir, preparar y cancelar la preparación.

El componente TSQLMonitor tiene dos eventos:

---

[5] Aunque no lo veamos. El dataset permanece activo, los datos se siguen viendo en los controles… pero no podemos hacer nada más que mirarlos, ya que cualquier operación que intentemos es abortada indicando que el conjunto de datos está cerrado. ¿Un bug?

- **`OnTrace`**: ocurre antes de agregar un evento a la lista. Asignando `False` al parámetro `LogTrace` evitaremos que se introduzca el evento en la lista.
- **`OnLogTrace`**: ocurre después que un evento se ha agregado a la lista.

En el ejemplo anterior tomamos el evento *después* de agregado a la lista interna del monitor; no obstante, no habría cambios en el resultado si usamos el evento OnTrace.

El componente SQLMonitor será utilizado constantemente a partir de aquí, para ver qué es lo que sucede en realidad entre nuestra aplicación y el servidor.

SQLQuery

Este componente también nos da acceso a los datos de una tabla, aunque con algunas diferencias respecto al SQLTable que vimos antes.

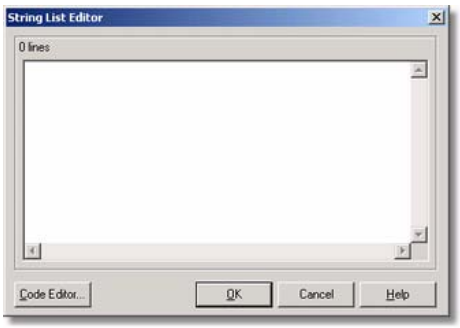
La tabla accedida no necesariamente existe en la Base de Datos; es el resultado de la ejecución de una sentencia SQL en el servidor. Puede contener columnas tomadas de varias tablas reales, relacionadas entre sí. Una vez obtenida la tabla resultado, el servidor nos provee de un cursor a los datos y a partir de aquí se trabaja de la misma manera que con el componente SQLTable.

Modificaremos la aplicación del *primer ejemplo* para que utilice un componente SQLQuery en lugar del SQLTable. Reemplace el componente por el simple trámite de borrar el anterior y colocar el nuevo (también habrá que cambiar las referencias a SQLTable1 en el código fuente por SQLQuery1).

El componente SQLQuery necesita saber con qué conexión debe trabajar; como antes, seleccionamos el componente SQLConnection1 en la propiedad SQLConnection. Ahora tenemos que decirle qué sentencia

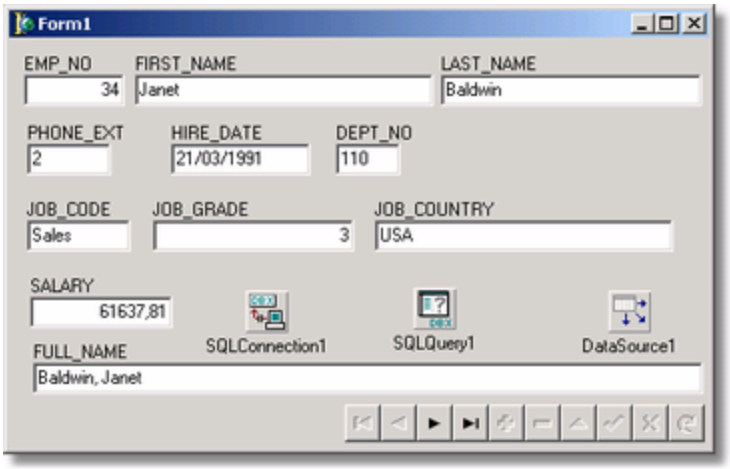
SQL queremos que ejecute, escribiéndola en la propiedad **SQL**.

El editor se muestra en la imagen. Nada del otro mundo, ni siquiera una ayuda para saber cuáles son las tablas que podemos usar, o los campos de cada una. Para esto tendremos que esperar al componente TSQLDataset. No obstante, el editor de la propiedad SQL del TSQLQuery tiene un botón que nos permite editar la sentencia SQL en una página nueva del editor de código; aquí tendremos resalte de sintaxis, posibilidad de copiar y pegar, etc.



En nuestro ejemplo, la sentencia SQL quedaría así (aprovechamos para indicar un orden en los datos):

```
SELECT *
FROM EMPLOYEE
ORDER BY FULL_NAME
```



Si ahora activamos el componente poniendo Active:= True, el servidor recibirá la orden de ejecutar esa instrucción SQL y devolver un cursor sobre la tabla resultado. Para ver los datos tendremos que enlazar el componente DataSource al SQLQuery.

Ejercicio: agregue un monitor a la aplicación anterior, que muestre en un memo los mensajes interceptados. Compare con el ejemplo similar que hicimos antes con un componente SQLTable.

Ejercicio: modifique el programa anterior para que no recupere todos los campos de la tabla. ¿Qué tiene que cambiar para que el programa siga funcionando?

Consultas paramétricas

Las consultas SQL suelen usarse para recuperar un conjunto de registros que cumplan ciertos criterios, mediante la cláusula WHERE de la instrucción SELECT. Por ejemplo, podemos obtener los clientes de Estados Unidos con

```
SELECT *
FROM CUSTOMER
WHERE COUNTRY='USA'
```

Ahora bien, en una aplicación cliente lo normal es dejarle al usuario la posibilidad de definir sus propios criterios –dentro de ciertos límites, claro. Por ejemplo, podría ingresar el nombre del país del que quiere ver los clientes en un editor. Nuestro código debe tomar este valor y reemplazarlo en la sentencia anterior.

Podríamos desactivar el SQLQuery, reemplazar la sentencia SQL incluyendo el nuevo criterio y volver a activarlo; de hecho, a veces conviene hacerlo así y he incluido un ejemplo al final de la sección. Pero ahora vamos a ver otra técnica que nos permite Delphi: el uso de *parámetros* en las expresiones SQL.

Los parámetros son como *variables* para la sentencia SQL; debemos asignarles un valor *antes* de enviar la sentencia al servidor para su evaluación. Lo bueno es que nos podemos desentender de los problemas sintácticos que pueden surgir al usar distintos tipos de datos. Por ejemplo, si el criterio anterior involucrara un valor numérico no se usarían comillas; y ni hablar del caso de las fechas, donde tenemos que tener en cuenta el orden del día, mes y año, la cantidad de dígitos, etc.

Para especificar en la sentencia SQL que una palabra es el nombre de un parámetro la precedemos con el símbolo ‘:’ (dos puntos). Por ejemplo, la instrucción anterior quedaría

```
SELECT *
FROM CUSTOMER
WHERE COUNTRY = :pais
```

Cuando se prepare la sentencia para enviarla al servidor, se reconocerá el parámetro ‘pais’ y al ejecutar se enviará su valor (que tendría que ser asignado previamente) junto con la sentencia SQL. Cuando la propiedad ParamCheck del componente es True, cada vez que cambia la sentencia SQL ésta se analiza en busca de parámetros, creándose una lista –de tipo TParams- con todos los encontrados. Esta lista se encuentra en la propiedad Params.

Estos objetos tienen propiedades como el Nombre, el Tipo de datos y el valor inicial en tiempo de diseño. En la imagen vemos el editor especial de la propiedad Params de SQLQuery, mostrando que el parámetro llamado 'pais' está en la posición 0 de la lista, es de tipo String, de entrada, y tiene un valor inicial igual a 'USA'.

Muchas veces no será necesario, pero siempre es conveniente curarse en salud y definir por lo menos el *tipo de dato* que contendrá cada parámetro en la propiedad **DataType**. De esta manera Delphi nos puede ayudar con algunas conversiones.

Para dar valor a un parámetro podemos usar su ubicación dentro de la colección *Params*, por ejemplo con

```
SQLQuery1.Params[0].Value:= 'USA';
```

O bien usar la función **ParamByName** del componente que nos devuelve un puntero a un parámetro dado su nombre:

```
SQLQuery1.ParamByName('pais').Value:= 'USA';
```

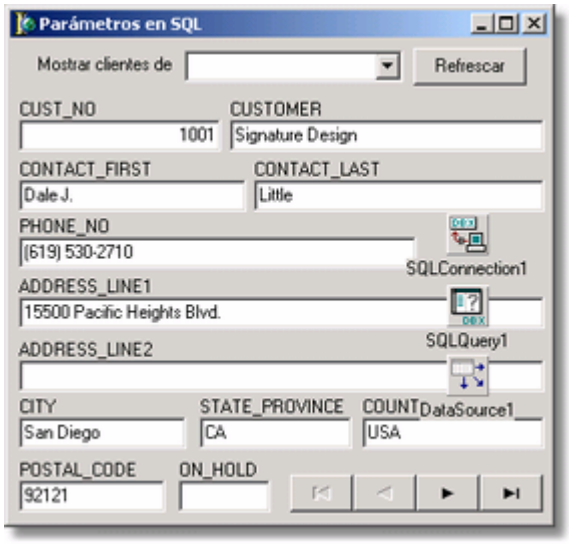
Ahora bien, la propiedad **Value** de `TParam` es de tipo **Variant**; esto significa que podemos asignarle casi cualquier tipo de dato. Pero la clase `TParam` define algunas propiedades que permiten *convertir* datos de un tipo de otro directamente; por ejemplo, **AsBoolean** o **AsString**. En el caso anterior podríamos poner

```
SQLQuery1.ParamByName('pais').AsString:= 'USA';
```

¿Qué sucedería si por ejemplo hacemos la asignación '`SQLQuery1.ParamByName('pais').AsInteger:= 12`'? Pues como ya indicamos que el parámetro es de tipo String, el componente sencillamente convertirá el número 12 a una cadena de caracteres y lo enviará al servidor como tal. Automáticamente. Elegante, ¿no cree?

Ejemplo 1

Haremos un pequeño ejemplo para mostrar la forma de uso normal de los parámetros; justamente lo que comentábamos antes, una interface para que el usuario pueda especificar el país del que quiere ver los clientes. La ventana principal (y única) del proyecto queda como sigue:



En un principio dejaremos el componente SQLQuery1 inactivo, de manera que no traiga ningún registro al arrancar. Únicamente se recuperarán registros después de dar valor al parámetro 'pais', usando el texto escrito en el ComboBox superior. He escrito algunos de los países que figuran en la tabla en la propiedad Ítems del combo, para permitir la selección rápida de los mismos. El código del evento **OnClick** del botón de actualización es el siguiente:

```
procedure TForm1.bRefrescarClick(Sender: TObject);
begin
  SQLQuery1.Close;
  SQLQuery1.ParamByName('pais').AsString:= ComboBox1.Text;
  SQLQuery1.Open;
end;
```

y la sentencia SQL es la misma que mostré antes, la repito aquí por claridad:

```
select *
from CUSTOMER
where COUNTRY = :pais
```

Como puede ver, nada del otro mundo. **NOTE** especialmente que *no es necesario usar las comillas*.

Variación: rescribir el SQL

El mismo efecto se puede conseguir con el siguiente código, que en lugar de usar un parámetro modifica el SQL completo:

```
procedure TForm1.bRefrescarClick(Sender: TObject);
begin
  SQLQuery1.Close;
  SQLQuery1.SQL.Clear;
  SQLQuery1.SQL.Add('SELECT *');
  SQLQuery1.SQL.Add('FROM CUSTOMER');
  SQLQuery1.SQL.Add('WHERE COUNTRY='+QuotedStr(ComboBox1.Text))
  SQLQuery1.Open;
```

```
end;
```

El hecho de escribir la sentencia SQL en varias líneas es completamente irrelevante; lo mismo podría hacerse con tres líneas como sigue:

```
procedure TForm1.bRefrescarClick(Sender: TObject);
begin
  SQLQuery1.Close;
  SQLQuery1.SQL.Text:= 'SELECT * FROM CUSTOMER WHERE COUNTRY=' +
                       QuotedStr(ComboBox1.Text);
  SQLQuery1.Open;
end;
```

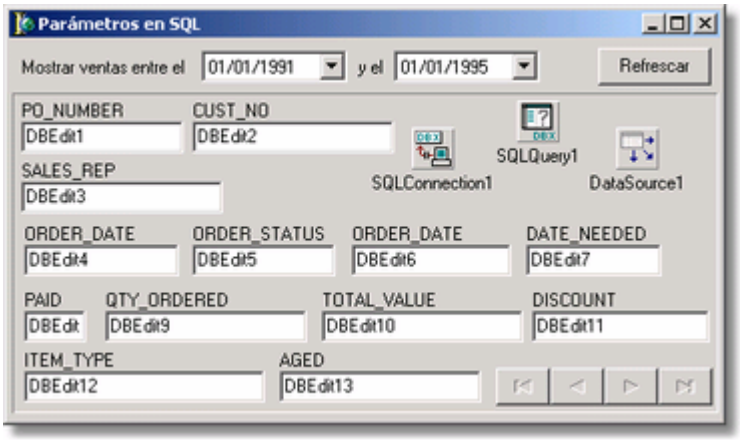
Ahora sí debemos tener cuidado con las comillas; lo mejor es usar la función **QuotedStr**, que toma una cadena como parámetro y devuelve la misma cadena entre comillas simples. Esta función considera y maneja bien los casos en que la cadena original contenga comillas, por ejemplo "Mc Perry's".

Ejercicios:

- Compruebe qué pasa si escribe en el ComboBox un país que no exista en la tabla.
- ¿Qué ocurre si escribe un número en lugar de un nombre de país?

Ejemplo 2: rango de fechas

Este es un caso típico que puede traer grandes dolores de cabeza si no tenemos cuidado. El objetivo es recuperar los registros de ventas de un determinado rango de fechas, a ingresar por el usuario mediante prácticos controles DateTimePicker. La ventana principal es la siguiente:



La sentencia SQL es ahora

```
SELECT *
FROM SALES
WHERE ORDER_DATE BETWEEN :FechaDesde AND :FechaHasta
ORDER BY ORDER_DATE
```

(los datos se traen ordenados por fecha, pero no es necesario). El código del botón de actualización tiene que dar valor a los dos parámetros antes de activar el dataset:

```
procedure TForm1.bRefrescarClick(Sender: TObject);
begin
  with SQLQuery1 do
  begin
    Close;
    ParamByName('FechaDesde').AsDate:= DateTimePicker1.Date;
    ParamByName('FechaHasta').AsDate:= DateTimePicker2.Date;
    Open;
  end;
end;
```

Ejercicios

- Experimente un poco con los tipos de datos. Pruebe a asignar las fechas como strings, usando editores comunes en lugar de los controles DateTimePicker.
- ¿Cómo tendría que escribir la sentencia SQL si indicara las fechas explícitamente (sin parámetros)?

Ejemplo 4: Recuperación de datos desde Vistas y Procedimientos Almacenados

Los Procedimientos Almacenados que devuelven un cursor se ven desde la aplicación cliente como tablas. Esto posibilita consultarlos con una sentencia SELECT, indicando el nombre del procedimiento (con sus parámetros) en la parte FROM.

Ejercicio: en la sección dedicada a IBX hemos escrito un Procedimiento en el servidor (SP_DEMO2) que devuelve un cursor, como ejemplo de la parte de consultas con parámetros. Cree una nueva aplicación donde se vean los datos devueltos por el procedimiento, esta vez usando dbExpress.

Haremos ahora una nueva aplicación que nos muestre datos desde una Vista de la Base de Datos. Una vista es nada más que una sentencia SQL almacenada en el servidor, identificada por un nombre; para nosotros es como si fuera una tabla.

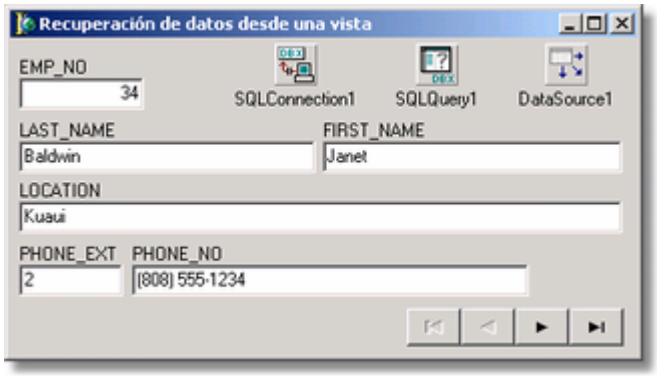
En la base de datos EMPLOYEE hay una vista llamada PHONE_LIST que nos devuelve los números de teléfono con sus extensiones para todos los empleados asignados a un departamento. Esta es la definición de la vista:

```
CREATE VIEW PHONE_LIST (EMP_NO,FIRST_NAME,LAST_NAME,PHONE_EXT,LOCATION,PHONE_NO)
AS
    SELECT
       emp_no, first_name, last_name, phone_ext, location, phone_no
       FROM employee, department
       WHERE employee.dept_no = department.dept_no;
```

Desde nuestro cliente simplemente debemos hacer

```
SELECT *
FROM PHONE_LIST
ORDER BY LAST_NAME, FIRST_NAME
```

La parte de ordenamiento, por supuesto, no es necesaria. A continuación se ve la ventana principal de la aplicación: queda como ejercicio para el lector construirla, que es similar a los ejemplos anteriores.



NOTA: podríamos haber ejecutado la sentencia SELECT de la vista directamente en el SQLQuery1. Entonces, ¿por qué definir una vista en el servidor? ¿No perdemos más tiempo haciendo una consulta de otra consulta?

Bueno, en parte es cierto. Y para este ejemplo muy simple, es un esfuerzo sin sentido. Pero hay un par de razones por las que podemos considerar la creación de vistas:

- Nos independizamos (un poco al menos) de la estructura exacta de las tablas reales. Si el día de mañana cambiamos la estructura de la tabla de Departamentos, por ejemplo, y ponemos los nros. de teléfono en otro lado, solamente tendríamos que cambiar la vista y las aplicaciones clientes seguirían funcionando.
- Hay veces en que el resultado deseado no se puede obtener con una sola sentencia SELECT; podemos usar entonces una vista para hacer un paso intermedio y recuperar los datos finales usando la 'tabla' ficticia generada. También podemos hacer los dos pasos en un Procedimiento Almacenado y devolver el cursor final.

Y hablando de Procedimientos Almacenados, dbExpress nos brinda un componente especializado para acceder a estos objetos de la base de datos: TSQLStoredProc.

SQLStoredProc

Este componente nos permite ejecutar un procedimiento almacenado de una base de datos.

Necesitamos establecer tres elementos antes de poder ejecutar un procedimiento:

- La base de datos a usar. Se especifica indicando la conexión con la propiedad **SQLConnection**, que referencia un componente TSQLConnection.
- El nombre del procedimiento. Una vez seleccionada la conexión, se puede elegir el procedimiento en una lista en la propiedad **StoredProcName**.
- Los parámetros que sean necesarios. Estos dependen de la definición del procedimiento, y se indican en la propiedad **Params**.

Antes de ejecutar el procedimiento debemos dar valor a los parámetros de entrada; al igual que en las consultas paramétricas con el componente SQLQuery, podemos acceder a los parámetros a través de su posición en la lista *Params* (por ejemplo, el primer parámetro se accede con Params[0]) o bien por nombre, usando la función **ParamByName** del componente SQLStoredProc.

Una vez asignados los parámetros, ejecutamos el procedimiento llamando al método **ExecProc** del componente.

Este componente se utiliza únicamente para ejecutar procedimientos almacenados que no devuelven un conjunto de datos; cuando el procedimiento devuelve un conjunto de datos -esto es, un cursor- se trata como una tabla y se puede acceder con cualquier componente de acceso a conjuntos de datos como TSQLQuery o TSQLDataset.

Ejemplo: utilización de parámetros

El siguiente es un procedimiento almacenado que no devuelve registros; toma un valor de entrada y devuelve uno de salida. El objetivo del procedimiento es contar la cantidad de empleados que están asignados a un determinado departamento –cuyo número pasamos como parámetro de entrada. El procedimiento devuelve la cantidad encontrada en un parámetro de salida.

El script para crear el procedimiento almacenado es el siguiente:

```
CONNECT "C:\Archivos de programa\Archivos comunes\Borland Shared\Data\EMPLOYEE.GDB" ;

SET TERM ^ ;

CREATE PROCEDURE SP_DEMO1(NRODPTO CHAR(3)) RETURNS (CANTEMPLEADOS INTEGER)
AS
  declare variable locCantEmpl integer;
begin

  select count(*)
  from employee
  where dept_no = :NroDpto
  into :CantEmpleados;

  if (CantEmpleados is null) then
    CantEmpleados= 0;

end^

COMMIT WORK^

SET TERM ; ^
```

Y la ventana de nuestra aplicación tiene el aspecto que se ve en la imagen. Cuando se presiona el botón ‘Contar!’, la aplicación da valor al parámetro de entrada del SP, lo ejecuta y muestra el parámetro de salida en la etiqueta al efecto.

El código del evento OnClick del botón de cuenta es muy sencillo, y

muestra como pasar parámetros hacia y desde un componente TIBStoredProc:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SQLStoredProc1.ParamByName('NroDpto').AsString:= edit1.Text;
  SQLStoredProc1.ExecProc;
  labResultado.Caption:= format('El departamento %s tiene %d empleados',
    [edit1.text,SQLStoredProc1.ParamByName('CantEmpleados').AsInteger]);
  labResultado.Visible:= true;
end;
```

Claro que no sería práctico usar el programa así como está, ya que habría que acordarse del nro. del departamento… pero no podemos usar un componente que nos ayude como el DBLookupComboBox, porque requiere un cursor bidireccional (en realidad se puede usar, pero si no podemos más que avanzar en la lista tiene muy poca utilidad; le recomiendo que lo pruebe de todas maneras). Cuando lleguemos a la unión de los componentes de DBExpress con el ClientDataset completaremos la interface anterior.

SQLDataset

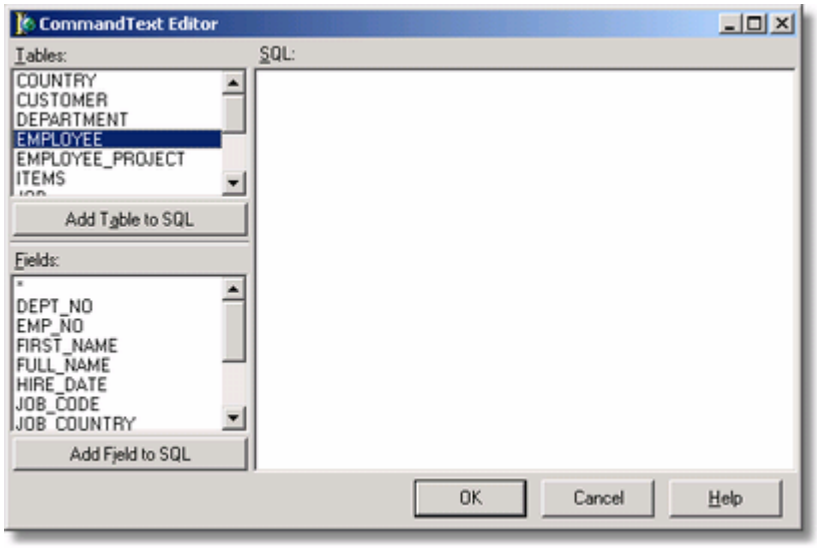
Este componente es una especie de 'comodín': modificando un par de propiedades podemos lograr que se comporte como cualquiera de los que hemos estudiado (SQLTable, SQLQuery, SQLStoredProc). Estas propiedades son **CommandType** y **CommandText**.

- si **CommandType** es igual a `ctTable`, el componente se comportará de la misma manera que un SQLTable. En **CommandText** se mostrará la lista de las tablas disponibles.
- Si **CommandType** es igual a `ctQuery`, el componente se 'convertirá' en un SQLQuery. En **CommandText** escribimos la sentencia SQL a ejecutar.
- Si **CommandType** es igual a `ctStoredProc`, estaremos en presencia de un SQLStoredProc. En **CommandText** elegimos el procedimiento a ejecutar.

El componente SQLDataset es prácticamente igual a su antecesor, TCustomSQLDataset; se limita a cambiar la visibilidad de algunas propiedades y sobrescribir un par de métodos. En los demás, se implementan las propiedades ya existentes desde la BDE… supongo que para hacer menos traumática la transición de una tecnología a otra.

En resumen: TSQLTable, TSQLQuery y TSQLStoredProc son en realidad componentes SQLDataset con agregados para trabajar con un solo tipo de elemento.

La única diferencia visible con los componentes especializados es el editor de sentencias SQL cuando usamos el tipo ctQuery:



En el editor de la derecha podemos escribir directamente la sentencia SQL; pero también tenemos las listas de la izquierda que muestran –y permiten pasar directamente al editor- las tablas disponibles arriba, y los campos de la tabla seleccionada debajo. Es una ayuda interesante para escribir las sentencias de selección, sobre todo si involucran más de una tabla.

Ejercicio

Modifique los ejemplos anteriores reemplazando los distintos componentes (TSQLTable, TSQLQuery, TSQLStoredProc) por TSQLDataset.

Y ahora vamos a la parte más interesante para las aplicaciones: cómo editar datos con dbExpress. Presentaremos brevemente la tecnología de Proveedores y Clientes de Borland, y los componentes que la sustentan –TdatasetProvider y TClientDataset. No entraremos en mucho detalle en esta relación, ya que los mismos se dan en otras secciones:

- Por detalles sobre la utilización de conjuntos de datos clientes (TClientDataset), vea la parte dedicada a *MyBase*
- Los detalles sobre los proveedores, la generación de sentencias de actualización y el pasaje de datos entre éstos y los conjuntos de datos clientes –incluyendo la resolución de conflictos-, vea la parte sobre *Proveedores*.

Edición de datos

Proveedores, clientes…

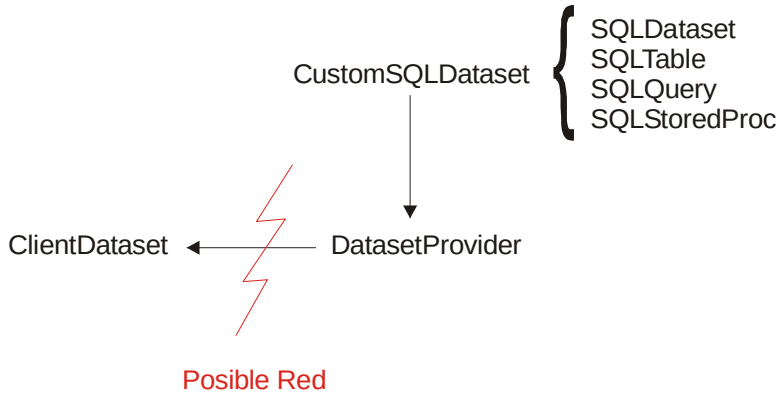
stock, contabilidad… ¡No! No estamos todavía con esos temas. Veremos aquí como hacer para poder editar los datos obtenidos de una tabla, y también conseguir la navegación bidireccional. Esto nos permitirá usar los controles de datos más avanzados, como las grillas y las listas de lookup.

Recapitulemos: los conjuntos de datos de dbExpress nos dan cursores *unidireccionales* y de *sólo lectura*. Para ser capaces de volver atrás en el conjunto de datos (navegación bidireccional), necesitamos mantener una copia en memoria de los registros ya leídos. Pero ya existe un componente que puede hacer justamente eso: el ClientDataset.

Este componente administra un conjunto de datos (una tabla) *en memoria*. Es decir: recibe los registros de algún lado –ya veremos de dónde- y construye una tabla en memoria para mantenerlos. Este accionar tiene algunas consecuencias prácticas que nos interesan ahora:

- Se puede navegar libremente por los datos, ya que todos los registros están en la memoria
- Los cambios no se graban en el servidor; hay que enviarlos explícitamente
- Debemos limitar de alguna manera el conjunto de datos origen, ya que si no podemos experimentar problemas de falta de memoria.

El primer paso es saber de dónde vienen los datos. El componente ClientDataset está preparado para trabajar en conjunto con otro componente llamado TDatasetProvider (proveedor de conjunto de datos) que es el encargado de hacer el enlace con los componentes que acceden realmente al servidor –en nuestro caso, los componentes dbExpress. ¿Recuerda el gráfico inicial con el flujo de los datos en dbExpress? Repito aquí la parte que nos interesa ahora:



El componente TDatasetProvider no solamente se encarga de *proveer* de datos a los conjuntos de datos clientes; también realiza la tarea inversa, generando las sentencias SQL de acción necesarias para *aplicar* los cambios en el servidor. Y entonces completamos el cuadro:

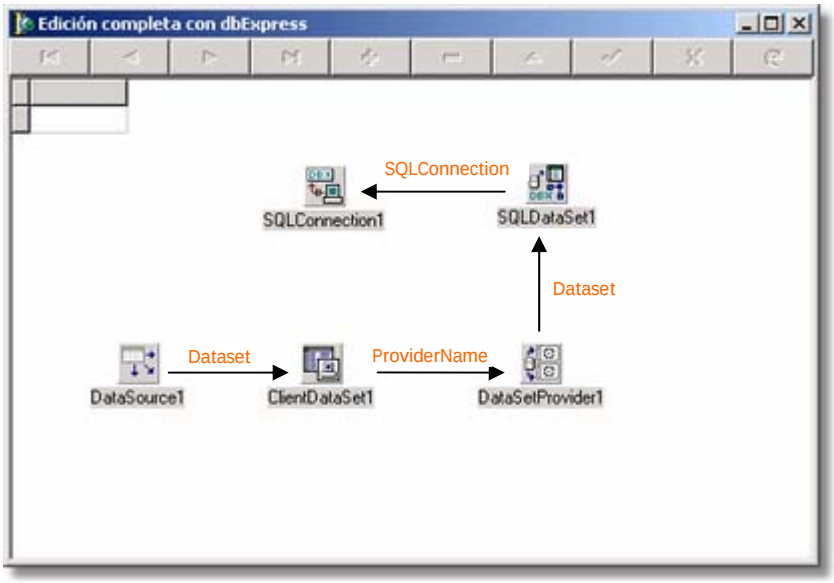
- Los componentes dbExpress conectan con el servidor y consiguen los datos, la ‘materia prima’. Se la entregan al proveedor de conjuntos de datos.
- El componente TDatasetProvider empaqueta los registros y se los envía a los conjuntos de datos clientes.
- El componente TClientDataset crea una imagen de la tabla original en memoria, y permite la edición y navegación de los usuarios utilizando controles de datos.
- Cuando *el usuario* decide enviar los cambios al servidor, el conjunto de datos cliente empaqueta los registros cambiados y los envía junto con algunas indicaciones a su proveedor.
- El proveedor recibe las modificaciones, genera sentencias SQL tales como INSERT, DELETE o UPDATE y las envía al servidor (no al conjunto de datos dbExpress, que es de sólo lectura).

- En caso de producirse algún error en el servidor –por ejemplo, si otro usuario ha bloqueado un registro o si se viola alguna restricción- el error se recibe en el proveedor de conjuntos de datos primero y si no puede resolverse el problema se envía al cliente, junto con todos los datos disponibles, para que sea resuelto ahí. De esta manera, la decisión final en casos que no puedan resolverse automáticamente se puede pedir al usuario de la aplicación a través de cuadros de diálogo.

Veremos los detalles de toda esta danza en la parte de Proveedores; por ahora sólo nos hace falta saber de dónde vienen las cosas y adónde van.

Pero basta de cháchara; pongamos las manos en la masa y montemos una aplicación para probar si realmente funciona todo esto.

La aplicación que realizaremos nos permitirá agregar, modificar y borrar registros en la tabla de empleados. Usaremos una grilla de datos (DBGrid) para que las operaciones sean más visibles. La ventana se ve en la figura siguiente, con las referencias entre los componentes y las propiedades correspondientes.



Los datos se obtendrán desde un SQLDataset de tipo ctQuery, y la sentencia para recuperarlos demuestra que se puede usar toda la potencia de SELECT –orden, grupos, criterios de selección, etc:

```
select *
from EMPLOYEE
order by Last_Name,First_Name
```

Cuando tenemos todo en orden, activamos ClientDataset1. Deberíamos ver todos los registros de la tabla de empleados, ordenados por apellido y nombre.

Esta aplicación ya se puede usar… salvo por un detalle: los cambios no se aplican nunca en el servidor. Recuerde que dijimos que el responsable por iniciar la acción de actualización sobre la base de datos era el usuario; debemos proveerlo de alguna manera de expresar su voluntad de grabar todo en forma permanente. Agregaremos un botón en la parte de arriba, después de achicar un poco el navegador.

En el evento OnClick de este botón ‘Aplicar’ solamente tenemos que escribir lo siguiente:

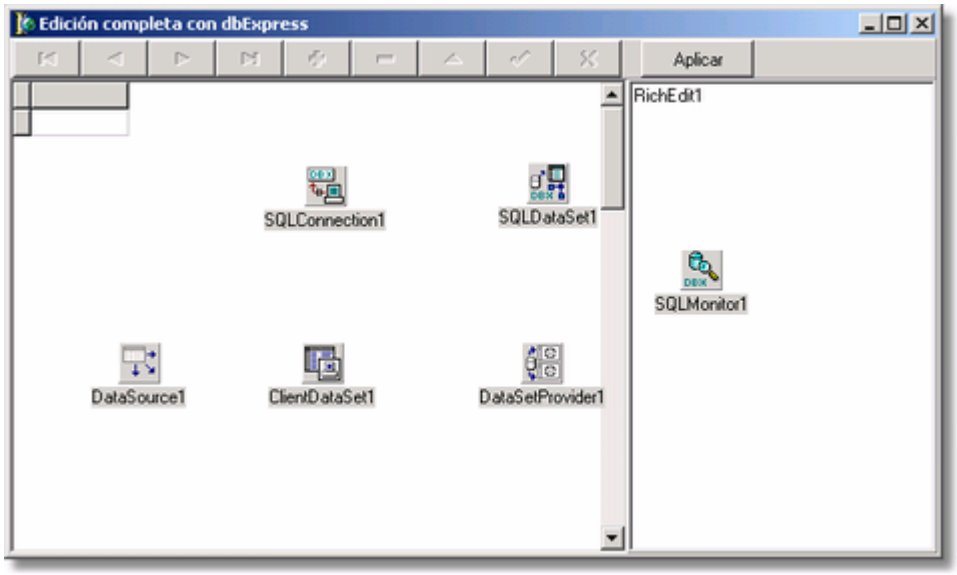
```
procedure TForm1.BitBtn1Click(Sender: TObject);
```

```
begin
  ClientDataset1.ApplyUpdates(0);
end;
```

El parámetro 0 que enviamos al método indica que no toleraremos ningún fallo; recuerde que pueden aplicarse muchos cambios juntos, y cualquiera de ellos puede fallar. Con (0) indicamos al proveedor que cuando falle una sola instrucción se detenga el proceso y nos devuelva el error correspondiente.

Compruebe que la aplicación funciona; modifique datos, inserte y borre registros, cierre y vuelva a abrir la aplicación para estar seguro que sus cambios se guardaron correctamente. Recuerde que cuando quiere que sus cambios se graben realmente debe presionar el botón 'Aplicar'. Para cancelar los cambios, simplemente cierre la aplicación sin presionar ese botón.

Es muy ilustrativo mirar los comandos SQL que se envían realmente al servidor en una aplicación como la anterior; por lo tanto, agregaremos un panel con un componente RichEdit (que tiene más capacidad de texto que el memo común) y un SQLMonitor que envíe el registro de los comandos directamente al RichEdit:



En el evento OnTrace, que se produce *antes* de guardar el comando en la lista interna del monitor, agregamos la línea al editor e indicamos que no la guarde en la lista interna para no duplicar la cantidad de memoria ocupada:

```
procedure TForm1.SQLMonitor1Trace(Sender: TObject; CBInfo: pSQLTRACEDesc;
  var LogTrace: Boolean);
begin
  RichEdit1.Lines.Add(cbinfo.pszTrace);
  LogTrace:= false;
end;
```

Y ahora probamos nuevamente la aplicación. Veamos algunos casos típicos:

1- Edición de un campo

Antes que nada, notemos que cuando hacemos Post *no se envía nada al servidor*. Como ya estarán pensando, recién se envían los comandos SQL cuando se aplican los cambios.

La siguiente es la secuencia de instrucciones que se envían a Interbase cuando se cambia el campo 'Last_Name' del empleado Nro. 107 por el valor 'Cookie':

```
1-  Comienza una transacción
INTERBASE - isc_start_transaction

2-  Prepara la instrucción
INTERBASE - isc_dsql_allocate_statement

update EMPLOYEE  set
 LAST_NAME = ?
where
 EMP_NO = ? and
 FIRST_NAME = ? and
 LAST_NAME = ? and
 PHONE_EXT = ? and
 HIRE_DATE = ? and
 DEPT_NO = ? and
 JOB_CODE = ? and
 JOB_GRADE = ? and
 JOB_COUNTRY = ? and
 SALARY = ? and
 FULL_NAME = ?

INTERBASE - isc_dsql_prepare

3-  Da valor a los parámetros y ejecuta
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute

4-  Libera recursos de la instrucción (cancela la preparación)
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement

5-  Termina la transacción aceptando los cambios
INTERBASE - isc_rollback_transaction
```

Esta secuencia de instrucciones muestra cómo nuestra modificación se traduce en una sentencia UPDATE… con un montón de parámetros; el resto es sólo la preparación obligada de la instrucción y los parámetros, y la transacción que engloba todo para que sea una operación atómica.

Esto es muy importante: el mecanismo de aplicación de cambios de TDatasetProvider verifica si hay una transacción en curso y si no, inicia una nueva como en el caso que vimos. Si ya hay una la utiliza *y no la termina* porque asume que el usuario todavía está usando la transacción en un proceso.

Pero entonces, ¿quiere decir que nosotros no estamos dentro de una transacción cuando editamos un registro? En este caso particular, la verdad es que no estamos en una transacción. Cuando abrimos el conjunto de datos cliente se inició una, se leyeron los datos (todos) y se cerró inmediatamente. Podemos verlo en las primeras instrucciones de la secuencia monitoreada.

Y ahora centremos la atención en la sentencia SQL generada: realmente ¿era necesario usar un criterio de selección tan estricto, incluyendo todos los campos? Bueno, generalmente no; pero el componente DatasetProvider apuesta por defecto al peor caso. Y éste sería que algún otro usuario nos modifique cualquiera de los campos del registro mientras lo editábamos (recordemos que trabajamos sobre una copia en memoria, no hay bloqueos ni nada que se lo impida), y que ese cambio afecte al registro entero.

Generalmente no es este el caso, ni mucho menos; la mayoría de las veces basta con filtrar con la clave primaria solamente. Para conseguir esto debemos modificar una propiedad del proveedor llamada **UpdateMode**.

Esta propiedad puede tomar uno de tres valores:

- **upWhereAll**: valor por defecto, pone todos los campos en el Where

- **upWhereChanged**: pone en el Where solamente los campos que sufrieron modificaciones
- **upWhereKeyOnly**: únicamente pone en el Where los campos de la clave primaria

Probemos entonces las otras opciones. Con **upWhereChanged**, la sentencia generada es la siguiente:

```
update EMPLOYEE  set
 LAST_NAME = ?
where
 LAST_NAME = ?
```

mmm… muy peligroso, considerando que estamos cambiando un apellido. Suponiendo que cambiamos 'Perez' por 'Gonzalez', al reemplazar los parámetros quedaría

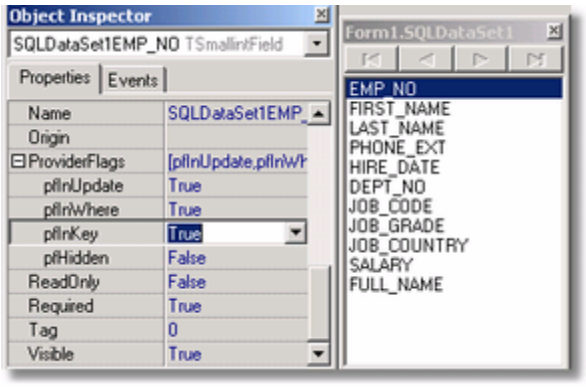
```
update EMPLOYEE  set
 LAST_NAME = 'Gonzalez'
where
 LAST_NAME = 'Perez'
```

¡Catástrofe! No queda ni rastro de los Perez en nuestra tabla! En realidad, lo que quisimos decir con 'incluir los campos modificados en la cláusula Where' era 'además de la clave primaria'. Antes de ver cómo conseguir esto, veamos la instrucción generada con la opción **upWhereKeyOnly**:

```
INTERBASE - isc_start_transaction
INTERBASE - isc_rollback_transaction
```

Ahora sí que hemos logrado la síntesis de la comunicación… por supuesto que la secuencia de instrucciones anteriores no graba absolutamente nada; se limita a iniciar el proceso con la transacción y enseguida la cancela sin realizar ninguna operación. ¿Qué pasó?

El hecho es que dbExpress *no sabe* cuales son los campos de la clave primaria. Se puede averiguar esto investigando la definición de la tabla (el esquema), pero Borland ha preferido pasarnos la pelota y dejar que nosotros definamos qué campos tomar como clave. Y es que puede haber más de una clave válida para la actualización; de hecho cualquier combinación de campos de un índice único serviría para el propósito.



Entonces, debemos especificar los campos que componen la clave de actualización. Esto se hace en la propiedad **ProviderFlags** de los componentes de campo *del SQLDataset*.[6]

Creamos los componentes de campo para el SQLDataset, seleccionamos el correspondiente al campo EMP_NO y agregamos pf_InKey al conjunto de la propiedad **ProviderFlags**, como se ve en la imagen.

Ahora sí, el proveedor puede discernir cuáles son los campos de la clave de actualización de la tabla (en este caso la clave primaria) y armar la instrucción correcta:

[6] No funcionará si coloca la propiedad en los componentes de campo del ClientDataset

- Con upWhereKeyOnly:

```
update EMPLOYEE  set
 LAST_NAME = ?
where
 EMP_NO = ?
```

- Con upWhereChanged:

```
update EMPLOYEE  set
 LAST_NAME = ?
where
 EMP_NO = ? and
 LAST_NAME = ?
```

Ahora si, los Perez están a salvo!

Ejercicios

- Investigue las instrucciones que se envían al servidor cuando se hacen varias modificaciones antes de aplicar los cambios.

- ¿Se modifica en algo la secuencia de instrucciones si *preparamos* la consulta antes de abrirla poniendo SQLDataset1.**Prepared**:= true?

- ¿Se modifica en algo si preparamos la consulta antes de aplicar los cambios?

2- Inserción de un registro

Los comandos enviados al servidor cuando se inserta un registro son los siguientes:

```
INTERBASE - isc_start_transaction

INTERBASE - isc_dsql_allocate_statement

insert into EMPLOYEE
  (EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, HIRE_DATE, DEPT_NO, JOB_CODE, JOB_GRADE, JOB_COUNTRY, SALARY)
values
  (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_commit_transaction
```

Por suerte aquí no tenemos que preocuparnos por claves o campos a incluir en el Where. Solamente una observación:

¿Qué pasó con el campo 'FULL_NAME'?

Convengamos que este campo es calculado en el servidor, y por lo tanto de solo lectura; no debería incluirse nunca en la sentencia INSERT. Parece que el proveedor nos ha sorprendido con una muestra de inteligencia…

…pero no es así. La realidad es que al ingresar el registro simplemente *omití* dar valor al campo FULL_NAME. El proveedor simplemente construye la instrucción INSERT con los campos que fueron modificados. Pruebe a cambiar el valor de la columna en un registro nuevo, y verá que no se da cuenta.

Podemos tomar dos caminos para evitar este problema:

- Impedir la edición de la columna FULL_NAME, por ejemplo poniendo ReadOnly = TRUE en la columna correspondiente de la grilla, o en el componente de campo.
- Indicar al proveedor que no incluya ese campo en la sentencia de inserción, quitando el elemento *pf_InUpdate* de la propiedad **ProviderFlags** del componente del campo FULL_NAME.

Elija la solución que más le guste y pruebe nuevamente para comprobar que funciona.

Es preferible modificar las propiedades del componente de campo, para independizarnos de la interface; si el día de mañana no usamos una grilla sino un DBEdit por ejemplo, la solución seguirá siendo válida.

3- Borrado de un registro

Para borrar un registro también tenemos que poder identificar de alguna manera el registro a borrar; idealmente, con los valores de la clave primaria. Veamos qué pasa cuando eliminamos los componentes de campo del SQLDataset y borramos una fila de la grilla:

```
delete from EMPLOYEE
where
 EMP_NO = ? and
 FIRST_NAME = ? and
 LAST_NAME = ? and
 PHONE_EXT = ? and
 HIRE_DATE = ? and
 DEPT_NO = ? and
 JOB_CODE = ? and
 JOB_GRADE = ? and
 JOB_COUNTRY = ? and
 SALARY = ? and
 FULL_NAME = ?
```

Las demás instrucciones son las mismas de siempre, para preparación y ejecución de la instrucción. Notemos que se han usado todos los campos para ubicar el registro a borrar; Esto es así tanto con **upWhereAll** o **upWhereChanged**, ya que todos los campos cambian.

Cuando usamos UpdateMode=upWhereKeyOnly en el proveedor, tenemos que especificar los campos que componen la clave mediante la propiedad **ProviderFlags** de los componentes de campo correspondientes, como vimos antes.

Ejercicio

Marque el campo EMP_NO como clave en el SQLDataset, y borre otro registro para comprobar la instrucción que se genera.

Cuando estudiemos los proveedores de datos en profundidad volveremos sobre la generación de instrucciones SQL y las maneras de controlarla. Por ahora resumamos la secuencia de operaciones que hemos visto:

- Necesitamos un componente para recuperar el conjunto de datos desde el servidor (unidireccional, sólo lectura): SQLQuery, SQLTable, SQLDataset. Si es necesario escribimos la sentencia SQL correspondiente.
- Debemos crear los componentes de campo de este dataset, y agregar la marca *pfInKey* a la propiedad **ProviderFlags** de los campos que compongan la clave.
- Agregamos ahora un componente DatasetProvider, referenciando el dataset anterior en su propiedad **Dataset**. Fijamos su propiedad **UpdateMode** al modo de actualización deseado.
- Agregamos un componente ClientDataset, referenciando al proveedor anterior en la propiedad **ProviderName**.
- Activamos el ClientDataset. Éste recuperará *todos* los datos de la tabla a través del proveedor, y los almacenará en memoria. El dataset es cerrado luego de extraer todos los registros, por lo que incluso la conexión puede ser cortada[7].
- Cuando estamos listos para enviar las modificaciones al servidor, invocamos el método **ApplyUpdates** del ClientDataset. Los registros modificados se envían al Proveedor, el cual genera las sentencias de acción SQL necesarias para aplicar los cambios en el servidor. Todos los cambios se realizan dentro de una transacción.

El componente SQLClientDataset

El conjunto SQLDataset + DatasetProvider + ClientDataset se utiliza ampliamente en la programación con dbExpress. Tanto es así que se ha creado un 'súper componente' que encapsula los tres: el SQLClientDataset.

Como vimos en el árbol de herencia, este nuevo componente no desciende de TCustomSQLDataset como los Datasets 'normales' de dbExpress, sino de TCustomClientDataset. Es decir que en realidad es un ClientDataset con agregados; estos agregados son precisamente un TDatasetProvider (definido en la clase intermedia TCachedClientDataset) y un TSQLDataset[8].

Este componente reemplaza a otros tres; por lo tanto la lista de propiedades y eventos es bastante larga, aunque *no se publican todas las propiedades y eventos de los tres componentes separados*.

¿Cuál es la razón que impulsa a un programador que por lo demás parece cuerdo, a llenar módulos y módulos con componentes en lugar de usar los fantásticos superpoderosos SQLClientDataset? Bueno, lo de siempre: la facilidad se intercambia con la flexibilidad. Mientras más intermediarios tengamos, podremos incrustar más puntos de control en el proceso. Y no olvidemos la posibilidad de la separación en capas…

Como ejercicio, los animo a que reemplacen el trío de componentes (SQLDataset1, DatasetProvider1, ClientDataset1) de la aplicación de ejemplo anterior por un solo SQLClientDataset y vean nuevamente los comandos SQL enviados al servidor para cada operación. Se encontrarán con una sorpresa cuando usen un UpdateMode distinto de upWhereAll.

---

[7] Estamos en presencia del *Modelo del maletín*, nada menos (Briefcase Model)

[8] En realidad, una instancia de TInternalSQLDataset –que es prácticamente igual a nuestro conocido TSQLDataset.

Debido a la manera en que el generador de sentencias SQL del DatasetProvider (llamado Resolvedor) trabaja, el componente DatasetProvider no toma en cuenta los valores que asignamos a los componentes de campo de un ClientDataset cuando hay un dataset ‘real’ enlazado al proveedor. Y aquí tenemos el problema, ya que aunque en el SQLClientDataset tenemos un dataset ‘real’ éste es interno, y no se le pueden asignar componentes de campo ni propiedades. Como consecuencia, al llegar a la generación de sentencias SQL *no hay ningún campo con la marca pfInKey*.

En el foro de discusion Borland.Public.Delphi.Database.DBExpress se ha lanzado repetidamente esta cuestión. Borland no ha contestado, pero Emil Atanasov creó un nuevo componente con el siguiente 'parche casero':

```
type
  TSQLClientDataSetFix = class(TSQLClientDataSet)
  private
    function GetNoMetadata: Boolean;
    procedure SetNoMetadata(const Value: Boolean);
    procedure DataSetProviderOnUpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
  protected
public
    constructor Create(AOwner: TComponent); override;
  published
    property NoMetadata: Boolean read GetNoMetadata write SetNoMetadata
default True;
  end;

constructor TSQLClientDataSetFix.Create(AOwner: TComponent);
begin
  inherited;
  Provider.OnUpdateData := DataSetProviderOnUpdateData;
  NoMetadata := True;
end;

procedure TSQLClientDataSetFix.DataSetProviderOnUpdateData(Sender: TObject;
  DataSet: TCustomClientDataSet);
var
  i: Integer;
  Field: TField;
begin
  if NoMetadata and (UpdateMode <> upWhereAll) then
    with DataSet.Fields do
      for i := Count - 1 downto 0 do
        with Fields[i] do begin
          Field := Self.FindField(FieldName);
          if Assigned(Field) then ProviderFlags := Field.ProviderFlags
        end
end;

function TSQLClientDataSetFix.GetNoMetadata: Boolean;
begin
  Result := TSQLDataSet(Provider.DataSet).NoMetadata;
end;

procedure TSQLClientDataSetFix.SetNoMetadata(const Value: Boolean);
begin
  TSQLDataSet(Provider.DataSet).NoMetadata := Value
end;
```

Listo el código aquí solamente a título ilustrativo. El problema con este componente es que 'ocupa' el evento OnUpdateData, por lo que no lo tendremos disponible desde nuestras aplicaciones.

Hay otros problemas con la relación entre el conjunto de datos interno y el ClientDataset que vemos en el Inspector de Objetos; por ejemplo, si definen un índice al activar el componente buscará este índice en el conjunto de datos real. Si no está definido en la base de datos, mostrará un mensaje de error.

En resumen: traten de no usar este componente para aplicaciones reales, por lo menos mientras Borland no libere una versión más completa y trabajada a conciencia[9].

Instalación de una aplicación DBExpress

La instalación de una aplicación DBExpress es sumamente sencilla: solamente necesitamos

- el cliente de la base de datos en cuestión
- el archivo MIDAS.DLL si utilizamos ClientDatasets para las actualizaciones
- el driver DBX correspondiente a la base de datos: una librería DLL.

Además de estos archivos, debemos tener en cuenta la manera en que DBX almacena la información sobre la ubicación de la base de datos y la configuración de la conexión.

Esta información se almacena en dos archivos .INI. La ubicación de estos archivos está almacenada en el registro, como se ve a continuación:

HKEY_CURRENT_USER\Software\Borland\DB Express\

Clave	Archivo	Contenido
Connection Registry File	dbxConnections.ini	Alias locales
Driver Registry File	dbxDrivers.ini	Configuración de los drivers instalados

Esta información está repetida en la rama 'HKEY_LOCAL_MACHINE/Software/Borland/DB Express', que se usa como valor por defecto para recrear la otra cuando falta.

El componente SQLConnection lee los datos de configuración de estos dos archivos cuando se crea la conexión; a partir de ese momento, los almacena en las propiedades DriverName, GetDriverFunc, VendorLib (datos del driver) y el resto en la propiedad Params. Todos estos datos se leen en tiempo de diseño cuando se selecciona un alias, y se almacenan en el archivo .dfm del form, integrándose luego al ejecutable.

Los archivos de configuración no son leídos en tiempo de ejecución. Este es el comportamiento por defecto, que puede ser adecuado para muchas aplicaciones. Piense en la facilidad de instalación: nada que

[9] Según John Kaster, de Borland, el componente SQLClientDataset surgió de la necesidad de contar con algo más práctico para mostrar en las presentaciones de dbExpress. Borland recomienda NO usarlo en aplicaciones reales. Y yo también.

configurar, solamente asegurarse que el archivo de la base de datos esté en la carpeta adecuada (para Interbase) o que el servidor y la base de datos se llamen igual en el equipo destino que en el de desarrollo.

NOTA: en el caso de Interbase, que necesita el nombre de un archivo, podemos especificar la ruta en forma *relativa* a la ubicación del ejecutable.

Pero si no podemos –o no queremos– atarnos a unos valores 'escritos en piedra', entonces tenemos que pensar en modificar los datos de la conexión en tiempo de ejecución. Tenemos tres opciones:

1. Podemos poner en True la propiedad *LoadParamsOnConnect* de TSQLConnection, lo que obligará a este componente a descartar la información guardada en diseño y releer los archivos de configuración.

En este caso, la conexión intentará ubicar las claves de registro indicadas anteriormente, para obtener de ahí la ubicación de los archivos de configuración. Si no encuentra las claves, intentará ubicar los archivos en el directorio de la aplicación. Si tampoco puede encontrarlos, generará una excepción y no se conectará.

Esto implica que podemos instalar una copia de los archivos de configuración junto con nuestra aplicación… y este archivo se puede modificar para cada cliente que sea necesario sin recompilar el programa. Claro que también tenemos el riesgo que algún usuario se pregunte 'qué pasaría si cambio esta línea…?' y recibamos las culpas porque nuestra aplicación 'ha dejado de funcionar misteriosamente'.

2. Podemos colocar la información modificando directamente la propiedad *Params* del componente de conexión; podemos hacerlo en el evento BeforeConnect o en el método Loaded (virtual) del form que lo contiene.

Para demostrar esta técnica, tomaremos el primer ejemplo –aquel que mostraba los datos sin ClientDataset. Una vez que asignamos el nombre del alias en la propiedad ConnectionName, los parámetros de la conexión se almacenan en una lista de strings en la propiedad Params. En esta propiedad, borramos el valor de 'Database' y 'Password'. Estos valores serán asignados en tiempo de ejecución, en el evento BeforeConnect:

```
procedure TForm1.SQLConnection1BeforeConnect(Sender: TObject);
begin
  with SQLConnection1.Params do
  begin
    Values['Database']:= '..\data\employee.gdb';
    Values['password']:= 'masterkey';
  end;
end;
```

Es un ejemplo tonto porque no hacemos más que cambiar de lugar los datos, pero siguen estando fijos… pero la idea es mostrarles cómo usar la técnica. En la práctica, los valores se obtendrán de algún archivo de configuración o del usuario mediante un cuadro de diálogo.

3. Podemos finalmente llamar al método *LoadParamsFromIniFile*, que espera el nombre del archivo de configuración a leer. Este archivo por supuesto debe tener el mismo formato que dbxConnections.ini.

El mismo ejemplo anterior sirve para ilustrar este punto; simplemente cambie las líneas que asignan los parámetros por una llamada como

```
SQLConnection1.LoadParamsFromIniFile('Configuracion.ini');
```

La cadena que se pasa como parámetro al método es el nombre del archivo de configuración.

Esta cadena es opcional; si se omite, se utilizará el archivo dbxconnections.ini estándar.

Integrar las DLL dentro del ejecutable

Otra opción para instalar la aplicación y no preocuparse de librerías externas es la de 'incrustar' las DLLs necesarias en el ejecutable. En el caso de DBExpress necesitaremos seguramente MIDAS.DLL y el driver de la base de datos que accedamos.

Lo único que tenemos que hacer es incorporar a la cláusula **uses** de la ventana principal las unidades correspondientes a cada librería:

- Midas.dll: midaslib, crtl
- Driver Interbase: dbexpint
- Driver Oracle: dbexpora
- Driver DB2: dbexpdb2
- Driver MySQL: dbexpmys o dbexpmysql, según la versión.

Notemos que al agregarlas al ejecutable no podremos accederlas desde otros programas; si planea instalar más de una aplicación es una buena idea dejarlas como DLLs para evitar tener que repetirlas con cada ejecutable.