Borland®

Migrating Borland® Database Engine Applications to dbExpress

New approach uses provide/resolve architecture

by Bill Todd, President, The Database Group, Inc.
for Borland Software Coporation
September 2002

Contents

Borland® dbExpress–a new vision

Past attempts to create a common API for multiple databases have all had one or more problems. Some approaches have been large, slow, and difficult to deploy because they tried to do too much. Others have offered a least common denominator approach that denied developers access to the specialized features of some databases. Others have suffered from the complexity of writing drivers, making them either limited in functionality, slow, or buggy. Borland® dbExpress overcomes these problems by combining a new approach to providing a common API for many databases with the proven provide/resolve architecture from Borland for managing the data editing and update process. This paper examines the architecture of dbExpress and the provide/resolve mechanism, demonstrates how to create a database application using the dbExpress components and describes the process of converting a database application that uses the Borland® Database Engine to dbExpress.

The dbExpress architecture

dbExpress (formerly dbExpress) was designed to meet the following six goals.

- Minimize size and system resource use.
- Maximize speed.
- Provide cross-platform support.
- Provide easier deployment.
- Make driver development easier.
- Give the developer more control over memory usage and network traffic.

white paper

dbExpress drivers are small and fast because they provide very limited functionality. Each driver is implemented as a single DLL on the Windows® platform and as a single shared object library on the Linux® platform. A dbExpress driver implements five interfaces that support fetching metadata, executing SQL statements and stored procedures, and returning a read only unidirectional cursor to the result set. However, when used with the DataSetProvider and ClientDataSet to implement the provide/resolve data access strategy of Borland, dbExpress gives you a full-featured, high performance, high concurrency system for working with data in SQL databases.

How provide/resolve architecture works

The provide/resolve architecture uses four components to provide data access and editing. The first is the SQLConnection component that provides a connection to the dbExpress driver for the database you are using. Next is one of the dbExpress dataset components that provides data by executing a `SQL SELECT` statement or calling a stored procedure. The third component is the DataSetProvider, and the fourth is the ClientDataSet. When you open the ClientDataSet, it requests data from the DataSetProvider. The DataSetProvider opens the query or stored procedure component, retrieves the records, closes the query or stored procedure component, and supplies the records, with any required metadata, to the ClientDataSet.

The ClientDataSet holds the records in memory while they are viewed and modified. As records are added, deleted, or updated, either in code or via the user interface, the ClientDataSet logs all changes in memory. To update the database, you call the ClientDataSet ApplyUpdates method. ApplyUpdates transmits the change log to the DataSetProvider. The provider starts a transaction, then creates and executes SQL statements to apply the changes to the database. If all changes are applied successfully, the provider commits the transaction; if not, it rolls the transaction back. Database updates may fail if, for example, a change violates a business rule enforced by a trigger or if another user has changed a record you are trying to update since you read the record. If an error occurs, the transaction is rolled back, and the ClientDataSet OnReconcileError event is fired, giving you control of how the error is handled.

Benefits of provide/resolve architecture

Short transaction life

Long transactions force the database server to hold locks, which reduces concurrency and consumes database server resources. With provide/resolve architecture, transactions exist for a moment when updates are applied. This dramatically reduces resource consumption and improves concurrency on a busy database server.

Make any rows editable

Rows returned by multi-table joins, stored procedures, or read-only views cannot be edited directly. By using the ProviderFlags property of the field objects to identify the fields that should be updated and the DataSetProvider OnGetTableName event to supply the name of the table, many read only datasets can be edited easily.

Instantaneous sorting and searching

Since the ClientDataSet holds records in memory, they can be sorted quickly. If an in-memory sort is too slow, you can create indexes on the ClientDataSet data at design time or runtime. These in-memory indexes let you change the viewing order of records or locate records virtually instantaneously without the overhead of maintaining indexes in the database.

Automatic summary information

ClientDataSets will automatically maintain complex summary calculations you define, such as Sum(Price)—Sum(Cost). You can group summary calculations by any field or combination of fields to provide group totals. You can also use the Min, Max, Count and Avg (average) aggregates.

View subsets of data

Filter expressions using SQL WHERE syntax let you easily display a subset of the records in a ClientDataSet without the overhead of executing another query on the database server.

Multiple simultaneous views of data

The ability to clone a ClientDataSet cursor lets you view different subsets of the data in a ClientDataSet simultaneously. You can also view the same data sorted differently.

Calculated fields with no server overhead

You can add calculated fields to a ClientDataSet at design time to make computed fields part of the in-memory dataset. Since the calculations are performed using compiled Borland Delphi™ or C++ language code, they are faster and can be far more complex than computed columns in a SQL statement or the calculations possible in stored procedures, yet they impose no storage or computational burden on the database server.

The limitation that isn't there

Holding records in memory may seem like a limitation on the number of records you can work with. However, consider that traditional client/server application design has always been to select small sets of records to minimize network traffic and database server load. Even if you need to work with an unusually large number of records, remember that 10,000 records containing 100 bytes each requires only one megabyte of memory. In the unlikely event that you need to work with a very large number of records, the ClientDataSet and DataSetProvider include properties and events that let you fetch a portion of the records, edit them, remove them from memory, and then fetch the next group of records.

Easier deployment

An application using dbExpress requires just two DLLs to function. The first is the dbExpress driver, for example `DBEXPINT.DLL` in the case of Borland InterBase,® and the second is `MIDAS.DLL`, the ClientDataSet support library. Together, these two DLLs are less than half a megabyte in size. This minimizes application size and simplifies installation. If you prefer not to distribute these DLLs, you can compile them directly into your EXE. Deployment on Linux is identical, except that the DLLs are replaced by two shared object libraries.

Easier driver creation

dbExpress drivers must implement just five interfaces that are described in the on-line help. Borland also provides the source code for the Microsoft® MySQL™ driver as a model. This makes it easier for database vendors to create robust high-performance drivers. You can even create your own driver if you are working with an unusual or legacy database and no commercial driver is available.

Building a dbExpress application

Before you can convert an existing Borland Database Engine application to dbExpress, you need to be familiar with the dbExpress components and how they are used. This section creates a dbExpress application, step by step, describing each component as it is used. While this example is built using Borland Delphi™ on the Windows platform, the steps are identical when using Borland Kylix™ on the Linux platform.

The sample application uses the InterBase sample EMPLOYEE.GDB database and displays a one-to-many relationship between the Employee table and the Salary History table. The sample application demonstrates the following features of dbExpress.

- Nesting a detail table in a field in the master table.
- Editing data using the SQLQuery, DataSetProvider, and ClientDataSet components.
- Using a SQLQuery as a read-only dataset without a DataSetProvider and ClientDataSet.
- Applying updates contained in the ClientDataSet to the database.
- Handling errors that occur when updates are applied to the database.

The SQLConnection component

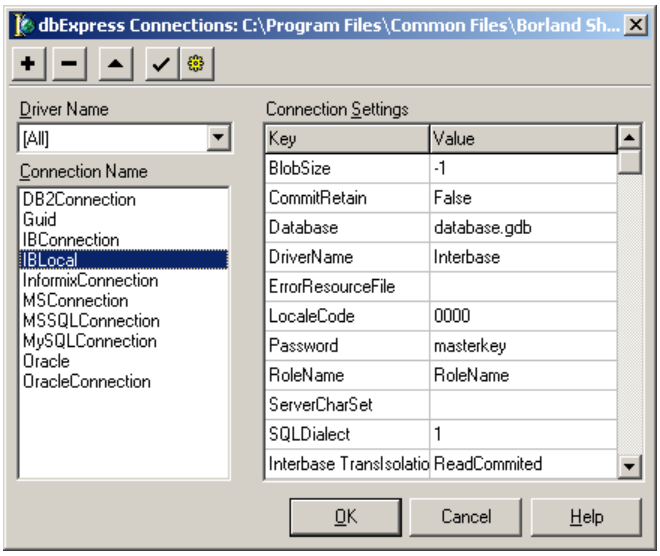
To create a simple dbExpress application, begin with the following steps.

1. Create a **new application** and add a **data module** to it. Name the data module **`MainDm`**.
2. Use the **Project Options** dialog to make sure that the data module is created automatically before the main form is created.
3. Drop a **SQLConnection component** from the dbExpress page of the component palette on **the data module**.
4. Name the SQLConnection component **`EmployeeConnection`** and set its DriverName property to **InterBase**.
5. Open the **property editor** for the Params property and set the **database parameter** to the path to the sample InterBase **`EMPLOYEE.GDB`** database. On most installations, this will be `c:\program files\Borland\interbase\examples\database\employee.gdb`.
6. Change the **UserName** and **Password** parameters if you require different values to connect to your InterBase server.
7. Set the **LoginPrompt** property to **false** so you will not be prompted for a username and password each time you run the program.
8. Set the **Connected** property to **true** to test your connection, and then set it to **false** again.

The SQLConnection component provides a database connection for any number of dataset components. You can use multiple SQLConnection components to connect to many databases simultaneously. There are three ways to define a connection to a database. You can use an existing named connection, create a new named connection or put the connection parameters in the Params property of the SQLConnection component. To use an existing named connection just set the ConnectionName property.

The dbExpress Connection Editor

To create a new named connection, double-click the **SQLConnection component** to open the dbExpress Connection Editor. The Connection Name list box, on the left, shows any connections that have already been defined. The Driver drop-down list lets you **filter** the **Connection Names** to show only the connections for the driver you select. The Connection Settings grid on the right shows the connection settings for the selected connection. All of the connections you create are stored in the `dbxconnections.ini` file.



The Connections Editor

The connections file

After creating a named connection, you can assign it to the ConnectionName property of the SQLConnection. If you use named connections, you will have to distribute a connections file with your application or locate an exiting connections file on the target computer and add your connection to it.

The Params property of the SQLConnection component

An alternative solution is to begin by setting the DriverName property of the SQLConnection component. The drop-down list for the DriverName property lists all of the drivers installed on your system. The driver information is contained in the `dbxdrivers.ini` file. Setting the DriverName property will also set the LibraryName and VendorLib properties using information from the drivers file. LibraryName contains the name of the dbExpress driver DLL, and VendorLib contains the name of the database vendor's client library file.

Enter the connection parameters from the Connections Editor in the Params property of the SQLConnection component as shown in the preceding screen. Using this method, the connection information is contained entirely within your application. If you want application users to be able to change any connection parameters, you can store them in the applications own configuration file and provide a way to edit the file either within your application or with a separate configuration program. This makes each application completely self-contained.

The SQLConnection component also provides the StartTransaction, Commit, and Rollback methods for explicit transaction control. If you need to execute SQL statements that do not return a result set, you can use the Execute or ExecuteDirect methods of the SQLConnection component. No dataset component is required. If you need access to the metadata of the database you are working with, SQLConnection provides the GetTableNames, GetFieldNames, and GetIndexNames methods.

The DataSet components

dbExpress provides four dataset components; SQLDataSet, SQLQuery, SQLStoredProc, and SQLTable. SQLDataSet is the component of choice for any new application you write. By setting its CommandType property, you can use it to execute SQL statements, call a stored procedure or access all of the rows and columns in a table. The other dataset components are designed to resemble their Borland Database Engine counterparts as closely as possible. Using these components makes converting a Borland Database Engine application to dbExpress easier.

The SQLQuery component

The properties and methods of the SQLQuery component are very similar to the Borland Database Engine Query component. Because the SQLQuery returns a read only unidirectional result set, properties and methods relating to the Borland Database Engine or to editing data are missing. You can use a SQLQuery to execute SQL for both DML and DDL statements. For statements that return a cursor to a result set, call the SQLQuery Open method or set its Active property to true. For statements that do not return a cursor, call the ExecSQL method. Continue building the sample application as follows.

1. Drop **three SQLQuery components** on the **data module**.
2. Set the **SQLConnection** property of all three to **`EmployeeConnection`**.
3. Name the first **`EmployeeQry`**, the second **`HistoryQry`**, and the third **`DeptQry`**.
4. Set the SQL property of **EmployeeQry** to:

```
SELECT * FROM EMPLOYEE
WHERE DEPT_NO = :DEPT_NO
ORDER BY LAST_NAME
```

5. Set the SQL property of **HistoryQry** to:

```
SELECT * FROM SALARY_HISTORY
WHERE EMP_NO = :EMP_NO
```

6. Set the SQL property of **DeptQry** to:

```
SELECT DEPT_NO, DEPARTMENT FROM DEPARTMENT
ORDER BY DEPARTMENT
```

7. Drop a **DataSource** on the **data module**, set its name to **`EmpLinkSrc`** and set its DataSet property to **`EmployeeQry`**.
8. Set DataSource property of **HistoryQry** to **`EmpLinkSrc`**.
9. Return to the **EmployeeQry**, open the **Params** property editor and set the value of the `DEPT_NO` parameter to **`XXX`**. Since this is an invalid value, it causes no records to be displayed until the user has entered a valid department number.
10. Double-click **EmployeeQry** to open the **Fields Editor** and add **field objects** for all fields.
11. Select the **`EMP_NO`** field, expand the **`ProviderFlags`** property, and set **pfInKey** to

true. Setting the `pfInKey` flag identifies the EMP_NO field as the primary key. The DataSetProvider (that you will add later) needs this information to construct the SQL statements to apply changes to the database.

12. Select the **`FULL_NAME`** field; expand its **ProfiderFlags** property, and set **pfInUpdate** and **`pfInWhere`** to **false**. FULL_NAME is a computed field, so it should not be updated or included in the WHERE clause of any SQL statements generated by the DataSetProvider.
13. Set the **Active** property of **EmployeeQry** to **true**.
14. Double-click **HistoryQry** to open the **Fields Editor** and add all **field objects**.
15. Select the **EMP_NO** field in the Fields Editor and set **pfInKey** to **true** in the ProviderFlags property. Repeat this for the `CHANGE_DATE` and `UPDATER_ID` fields that are also part of the primary key.
16. The **`NEW_SALARY`** field is also a computed field, so select it and set **pfInUpdate** and **pfInWhere** provider flags to **false**.
17. Set the **Active** property of **EmployeeQry** and **HistoryQry** to **false**.
18. Set the Connected property of **EmployeeConnection** to **false**.

The SQLTable

The SQLTable component resembles the Borland Database Engine Table component. Like its Borland Database Engine counterpart, it returns all of the rows and columns in a table so it is not well suited to client/server applications. Its only value is that it allows you to quickly convert a desktop database application that uses BDE Table components to dbExpress and a database server.

To use a SQLTable, set its **SQLConnection** property to a **SQLConnection component**. Set the **TableName** property to the **name of the table** you want to access. Open the table by setting its **Active** property to **true** or by **calling** its **Open method**.

The SQLStoredProc

Use the SQLStoredProc component to call stored procedures in the same way that you used the BDE StoredProc component in your Borland Database Engine applications. You can also call stored procedures with the SQLDataSet component, however, converting a Borland Database Engine application using the SQLStoredProc component is easier, because it is almost identical to the BDE StoredProc component.

To use a SQLStoredProc, set its **SQLConnection** property to a **SQLConnection component**. Set its **StoredProcName** property to the **name of the stored procedure** you want to execute. If the stored procedure returns a cursor to a set of rows, execute it by setting the **SQLStoredProc Active** property to **true** or by calling its **Open** method. If the stored procedure does not return a cursor to a result set, execute it by calling the ExecProc method.

The SQLDataSet

To use a SQLDataSet, fix its **SQLConnection** property to the **SQLConnection component** you want to use. Next, set the **CommandType** property to ctQuery, ctStoredProc, or ctTable. Most often you will use the default value of ctQuery. The value of the CommandText property depends on the value of CommandType. If CommandType is ctQuery, CommandText contains the SQL statement you want to execute. If CommandType is ctStoredProc, CommandText is the name of the stored procedure. If CommandType is ctTable, CommandText is the name of the table. You use the Params property to supply parameters for a "parameterized" query or a stored procedure, and the DataSource property to link the SQLDataSet to another dataset component in a master/detail relationship. If the SQLDataSet will return a cursor to a set of records, open it by calling its Open method or by setting its Active property to true. If it will not return a result set, call its ExecSQL method.

The SQLDataSet provides a read-only unidirectional cursor only. If this is the only access you need, for example for printing a

report, you can use the SQLDataSet by itself or with a DataSource component, depending on the requirements of your reporting tool. If you need to scroll back and forth through the records or edit the data, either add a DataSetProvider and ClientDataSet or use the SimpleDataSet component as described later in this paper.

If you need more detailed metadata information than the SQLConnection methods provide, use the SetSchemaInfo method of a SQLDataSet component. SetSchemaInfo takes three parameters, SchemaType, SchemaObject, and SchemaPattern. SchemaType can be either stNone, stTables, stSysTables, stProcedures, stColumns, stProcedureParams, or stIndexes. This parameter indicates the type of information that the SQLDataSet will contain when it is opened. The schema type is set to stNone when you are retrieving data from a table using a SQL statement or stored procedure. Each of the other schema types creates a dataset with a structure appropriate for the information being returned. SchemaObject is the name of the stored procedure or table when a stored procedure or table name is required. Schema pattern lets you provide a SQL pattern that will filter the result set. For example, if SchemaType is stTables and SchemaPattern is 'EMP%' the dataset will only contain tables that start with EMP.

The SimpleDataSet

To view and edit a set of records using the Borland Database Engine, all you need to do is drop a TQuery on your data module, set a couple of properties, and you are done. With dbExpress, you have to drop a SQLQuery or SQLDataSet, a DataSetProvider and ClientDataSet on your data module and set the properties to connect them together. There are two ways to avoid the extra time required to add three components and set their properties.

Borland Delphi™ 7 Studio introduces the SimpleDataSet component. SimpleDataSet combines a SQLDataSet, DataSetProvider, and ClientDataSet in a single component. If all you need to do is view and edit records from a single table or stored procedure, **SimpleDataSet** is a quick solution. Just add it to your **data module**; set the **Connection** property to your SQLConnection component; set the **CommandType** and **CommandText** properties of the embedded SQLDataSet component, and you are done. The SimpleDataSet component does have some limitations.

- You cannot use it in a multi-tier application. If you may convert your application to multi-tier in the future, use separate components.
- You cannot link a child dataset to a SimpleDataSet as a nested dataset field.
- None of the events of the internal DataSetProvider are exposed.
- The Options property of the internal DataSetProvider is not exposed. You cannot set the provider options at design time or runtime.
- You cannot instantiate field objects for the internal dataset at design time. This means that you cannot set the ProviderFlags properties of the field objects at design time. You will have to do it in code.
- If you are only using Borland MyBase,™ the internal database of the ClientDataSet component, and you are not connecting to a database server, using a ClientDataSet alone reduces your application's resource requirements.
- The properties and methods of the embedded SQLDataSet do not resemble the properties of the Borland Database Engine Query component as closely as the properties and methods of the SQLQuery do. Therefore, using the SimpleDataSet in a conversion project may require more code changes.

If you need more flexibility than the SimpleDataSet offers, drop a **SQLQuery, DataSetProvider**, and **ClientDataSet** on a **form** or data module. Set the **Connection** property of the SQLQuery. Set the **DataSet** property of the DataSetProvider to connect it to the SQLQuery. Set the **ProviderName** property of the ClientDataSet to connect it to the DataSetProvider. Select **all three components** and choose **Component** | Create **Component Template** from the main menu. Supply a **class**

name and **palette** page for your new template. Now, you can drop three separate components on a data module as easily as you can drop a single component.

SQL Monitor

The last dbExpress component, the Borland SQL Monitor, is provided to help you tune the performance of your application. SQL Monitor checks all of the SQL statements passed between a SQLConnection component and the database server it is connected to. The SQL statements can be logged to a file, displayed in a memo component, or processed in any other way you wish.

The data access components

If you need bi-directional scrolling and the ability to update data, you will need to use the DataSetProvider and ClientDataSet components.

The DataSetProvider

A DataSetProvider is linked to one of the dbExpress dataset components through the DataSetProvider DataSet property. The DataSetProvider supplies data to the ClientDataSet on request and generates the SQL DML statements to update the database, using the change log supplied by the ClientDataSet.

Return the data module for the sample application and perform the following steps.

1. Add a **DataSetProvider** from the Data Access page of the component palette.
2. Set its **DataSet** property to **EmployeeQry** and its **Name** property to **EmployeeProv**.

The DataSetProvider UpdateMode property lets you control how the provider determines if another user has updated a record you are trying to update since you read the record. When the provider generates SQL statements to update the database each `UPDATE` and `DELETE` statement includes a `WHERE` clause to identify the record. If UpdateMode is set to `upWhereAll`, the original value of every non-Blob field in the record is included in the WHERE clause unless the field's pfInWhere provider flag is false. This means that your UPDATE or DELETE will fail if another user has changed any of these fields. Setting UpdateMode to upWhereChanged means that only fields you have changed will be included in the WHERE clause. Setting UpdateMode to upWhereKeyOnly will generate a WHERE clause that contains just the primary key fields.

The provider's Options property includes many flags that let you control the provide/resolve process. If the `poCascadeDeletes` option is true, the provider will not generate SQL statements to delete the detail records for a master record you have deleted. The provider assumes the database server supports cascaded deletes and will delete the detail records automatically. The poCascadeUpdates option provides the same feature for changes to the master table's primary key.

If the SQLQuery that supplies data to the provider has an `ORDER BY` clause in its SQL statement, and you want the records to retain that order in the ClientDataSet, set the `poRetainServerOrder` flag to true. If you want to be able to change the SQL statement in the SQLQuery by changing the CommandText property of the ClientDataSet, set the **`poAllowCommandText`** option to **true**.

If you create a `BeforeUpdateRecord` event handler for the DataSetProvider, and the event handler may change the value of a field in a record before the database is updated, set the **`poPropogateChanges`** option to **true**. The provider will now send any changes back to the ClientDataSet to update the records it holds in memory.

While the DataSetProvider has many useful events, the two most important are OnGetTableName and BeforeUpdateRecord. Rows returned by multi-table joins, stored procedures, or read-only views cannot be edited directly in database applications.

The DataSetProvider gives you three tools to handle these situations. If the records include fields from a single table, for example records returned by a stored procedure, the only

problem is that the DataSetProvider has no way to discover the name of the table it should update. The solution is to create an OnGetTableName event handler for the DataSetProvider that returns the name of the table.

A second possibility is a multi-table join where fields from a single table must be updated. First, set the **ProviderFlags** property of the individual fields to **identify the fields** that should be updated. Then create an **OnGetTableName** event handler to return the **table name**, and the provider will generate the SQL statements automatically. If you need to update multiple tables for each record, add a **BeforeUpdateRecord** event handler to the **DataSetProvider**. In the event handler, you can generate the SQL statements for each table and execute them.

The BeforeUpdateRecord event handler also gives you a place to examine each record, before it is updated, and change any field values. You can even block the update entirely by raising an exception. This makes BeforeUpdateRecord a good place to enforce business rules.

The ClientDataSet component

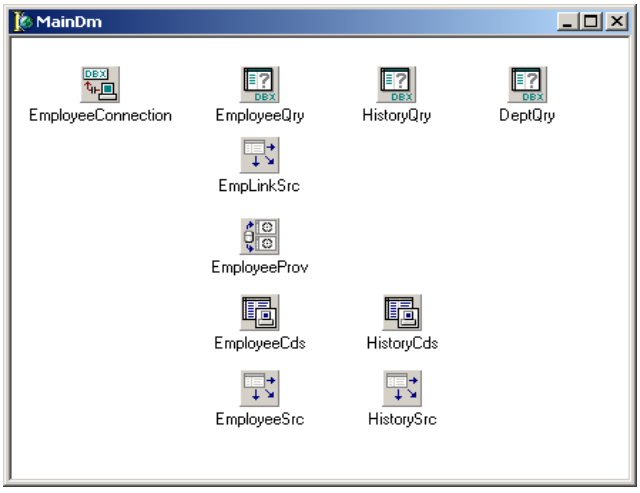
The ClientDataSet is connected to a DataSetProvider via its ProviderName property. It receives data from its DataSetProvider, buffers the data in memory, logs all changes to the data, and sends the change log to the DataSetProvider when the ClientDataSet ApplyUpdates method is called.

Continue with the sample application as follows.

Drop two **ClientDataSets** on the **data module** in the sample application.

2. Set the **ProviderName** property of the first to **`EmployeeProv`** and set the Name property to **`EmployeeCds`**.
3. Double-click **EmployeeCds** to open the **Fields Editor** and add **all field objects**. The last field object in the Fields Editor is named HistoryQry. This is a nested dataset field and contains the salary history records for each employee record. The records returned by the HistoryQry component appear as a nested dataset within the EmployeQry result set. This happens because the HistoryQry of the SQLClientDataSets DataSource property is set to EmpLinkSrc, which is connected to EmployeeQry, and because it gets the parameter value for its SQL statement from the EmployeeQry record.
4. Right-click **EmployeeCds** and choose **Fetch Params** from the context menu so the ClientDataSet will update its parameter list to match the EmployeeQry component.
5. Name the second **ClientDataSet** as **`HistoryCds`** and set its **DataSetField** property to **`EmployeeCdsHistoryQry`** so it will get its data from the nested dataset field in EmployeeCds.
6. Double-click **HistoryCds** and add **all fields**.
7. Drop two **DataSource components** on the **data module** and name them **EmployeeSrc** and **HistorySrc**.
8. Set their **DataSet** properties to **EmployeeCds** and **HistoryCds** respectively.

You should now be able to set the EmployeeCds and HistoryCds components Active properties to true. The data module should look like this.



The data module should look like this

Nested detail datasets are very powerful because they avoid confusion about the order in which updates should be applied for the master and detail datasets. For example, if a new master record with one or more detail records is added, the master record `INSERT` must be sent to the database server before the `INSERT` statements for the detail records. However, if you delete the detail records for a master and then delete the master record, the `DELETE` statements for the detail records must be processed before the delete for the master. With a nested dataset, the DataSetProvider ensures that the updates are processed in the right order to avoid errors on the database server.

The ClientDataSet has several useful properties. The Aggregates property lets you define maintained aggregate fields that automatically compute the sum, minimum, maximum, count, or average of any numeric field in the dataset. The aggregates can be grouped by any index. The AggregatesActive property lets you turn the aggregate calculations on and off at runtime. Assign a semicolon-delimited list of field names to the IndexFieldNames property to sort the records in the ClientDataSet in ascending order by those fields. To sort in descending order or for faster performance on very large datasets, create an index on the fields you want to sort by and assign the index name to the IndexName property of the ClientDataSet.
Use the CommandText property to change the SQL statement in the component that supplies data to the DataSetProvider. Just close the **ClientDataSet**, assign a **new SQL statement** to **CommandText**, and open the **ClientDataSet** to see the records returned by the new query. You can also assign new values to the parameters in the source dataset's SQL statement, using the Params property of the ClientDataSet. Why should you use the CommandText and Params properties of the ClientDataSet instead of accessing the properties of the source dataset directly? The advantage is that you will not have to change your code if you convert the program to a multi-tier Borland DataSnap™ application in the future.

The PacketRecords property lets your control the number of records that the provider fetches from the server at one time. The default value of –1 tells the provider to fetch all records returned by the source dataset's SQL statement. Normally, this is fine, but if you need to process a very large number of records, you may need to retrieve them in small groups.

One of the biggest differences between the Borland Database Engine and the provide/resolve architecture of dbExpress is that you must call the ApplyUpdates methods of the ClientDataSet to apply the changes in the change log to the database. Use the ChangeCount property to tell if there are unapplied changes in the change log. ApplyUpdates takes a single parameter that specifies the number of errors allowed before the update process is stopped, and the transaction is rolled back. Typically, you will pass zero, so the update process will end as soon as an error occurs. Passing –1 tells the DataSetProvider to try to apply all of the changes in the change log regardless of the number of errors that occur.

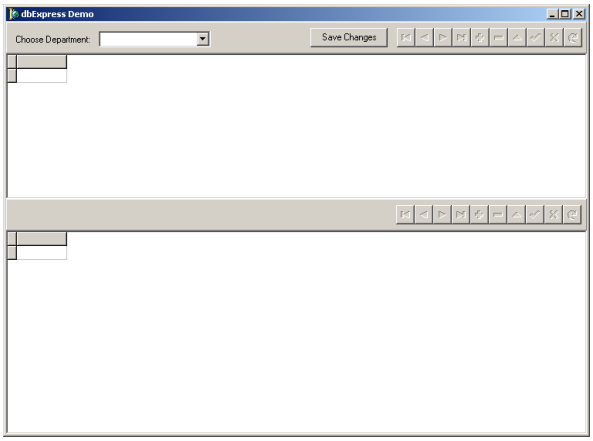
If an update error occurs, the OnReconcileError event of the ClientDataSet will fire. To handle update errors in the demo application, perform the following steps.

1. Add a **Reconcile Error Dialog** to the project from the **Dialogs** page of the Object Repository.
2. Add the **Reconcile Error Dialog** unit to the data module **uses clause**.
3. Make sure the **Reconcile Error Dialog** is not automatically created.
4. Create an **OnReconcile** error event handler for the EmployeeCds and add the following line of code.

```
Action := HandleReconcileError(
      DataSet, UpdateKind, E);
```

If an error occurs when a user applies updates, the Reconcile Error Dialog will show the error message, the record causing the error, and a set of buttons that let the user choose how to handle the error. To finish the sample application, follow these steps.

1. Add two **Panels**, two **DBGrids**, and two **DBNavigators** to the main **form** arranged as shown in the following figure.



2. Add the **data module** unit to the form unit **uses clause**.
3. Set the **DataSource** property of the top DBGrid and DBNavigator to the **EmployeeSrc DataSource component**.
4. Set the **DataSource** property of the bottom DBGrid and DBNavigator to the **HistorySrc DataSource component**.
5. Add a **Label** component and a **ComboBox** component to the top Panel.
6. Create an **OnChange** event handler for the ComboBox with the following code.

```
procedure TMainForm.DeptComboChange( Sender:
TObject);
begin
  with MainDm.EmployeeCds do
  begin
    if Active then CheckBrowseMode;
    Close;
    Params.ParamByName('DEPT_NO').AsString :=
Copy(DeptCombo.Text, 1, 3);
    Open;
  end; //with
end;
```

7. Add a **Button** component to the top **Panel** and set its caption to **Save Changes**.
8. Add the following code to the **Button OnClick** event handler.

```
procedure TMainForm.SaveBtnClick(Sender:
TObject);
begin
  with MainDm do
  begin
    if EmployeeCds.ChangeCount > 0 then
    begin
      if HistoryCds.Active then
HistoryCds.CheckBrowseMode;
      if EmployeeCds.Active then
EmployeeCds.CheckBrowseMode;
      EmployeeCds.ApplyUpdates(0);
      EmployeeCds.Refresh;
    end; //if
  end; //with
end;
```

9. Add a **method** that will fill the department ComboBox.

```
procedure TMainForm.LoadDeptCombo;
begin
  with MainDm do
  begin
    DeptQry.Open;
    while not DeptQry.Eof do
    begin

DeptCombo.Items.Add(DeptQryDept_No.AsString +
' ' +

DeptQryDepartment.AsString);
      DeptQry.Next;
    end; //while
    DeptQry.Close;
  end; //with
end;
```

10. Add the following **OnCreate** event handler for the form.

```
procedure TMainForm.FormCreate(Sender:
TObject);
begin
  MainDm.EmployeeCds.Open;
  LoadDeptCombo;
end;
```

Borland® Database Engine versus dbExpress

The following table compares the Borland Database Engine and dbExpress in several key areas to give you an overview of the differences. The second table shows the equivalent dbExpress component for each Borland Database Engine component.

Feature	Borland® Database Engine	dbExpress
Record buffering	Borland Database Engine determines how many records are buffered in local memory.	You control how many records are buffered locally.
Network traffic control	Borland Database Engine determines how many records are fetched from the server.	You control how many records are fetched from the server and when.
Transaction control	Both manual and automatic transaction control are available. Transactions are active while user is editing data.	Both automatic and manual transaction control are available. Transactions active very briefly while updates are being applied.
Deployment	Deployment is large (approximately 18 megabytes). Installation and configuration are complex and require registry changes.	Deployment requires two DLL's totaling less than half a megabyte which can be compiled into your EXE. No registry changes are required unless you deploy the dbxconnections.ini and dbxdrivers.ini files.
Cross-platform	Windows® only.	Windows and Linux®
Third part drivers	Difficult to develop. Few available.	Easy to develop. Many available. For information on third party drivers see the Borland Developer Network web site at bdn.Borland.com.

The following table shows the dbExpress components that correspond to each of the Borland Database Engine components. Note that dbExpress has no counterpart to the Borland Database Engine BatchMove component.

Borland® Database Engine	dbExpress
Tdatabase	TSQLConnection
TQuery	TSQLQuery
TStoredProc	TSQLStoredProc
TTable	TSQLTable
No equivalent	TSQLDataSet
TBatchMove	No equivalent
SQL Monitor Utility	TSQLMonitor
Session	N/A
UpdateSQL	N/A
NestedDataSet	N/A
BDEClientDataSet	SimpleDataSet

The concept of a session does not exist in dbExpress, so no analog to the Borland Database Engine session component is required. The Borland Database Engine UpdateSQL component is only used with the cached update feature of the Borland Database Engine Query component. In dbExpress, the features of the ClientDataSet component replace cached updates. The ability to handle nested datasets is built into the DataSetProvider and ClientDataSet components in dbExpress.

Migrating a SQL Links application for Borland® Database Engine to dbExpress

Converting an application that uses the Borland Database Engine and a SQL Links driver to dbExpress requires the following steps. Most client/server applications do not use the Borland Database Engine Table component. Since desktop database applications use the Borland Database Engine Table component extensively, it is discussed in the section on "Migrating desktop database applications to dbExpress" later in this paper.

- Replace the **Borland Database Engine database component** with a **SQLConnection component**.
- Replace **Borland Database Engine Query** and **StoredProc components** with **SQLQuery** and **SQLStoredProc components**.
- Add a **DataSetProvider** and **ClientDataSet** for each SQLQuery and SQLStoredProc component where bi-directional scrolling or editing is required.

Replace Borland Database Engine database with SQLConnection

Each Borland Database Engine database component must be replaced with a SQLConnection component.

Setting connection parameters

If you use a Borland Database Engine alias to let you to change your database connection without changing your application, you will need to use a dbExpress named connection, or you will need to store your connection parameters in an INI file. The most flexible and robust method is to use an application specific INI file to hold your connection parameters. If you choose this method, you will have to add code to your application to read the INI file and set the SQLConnection component Params property at startup. You may also want to add code to your application to allow users to edit the connection parameters in the INI file.

Controlling transactions

One of the key decisions you must make before you begin converting your application is how to handle transaction control. If your Borland Database Engine application lets the Borland Database Engine handle transaction control automatically, the only thing you have to do is add calls to ApplyUpdates wherever you want to save changes to your data. dbExpress will handle transaction control for you just as the Boralnd Databas Engine did. If the user is going to continue working with the same set of records after applying updates you should call the ClientDataSet Refresh method. This will re-execute the source query and load the new result set into the ClientDataSet. This lets the user see any changes made by others.

If your Borland Database Engine application makes explicit calls to StartTransaction, Commit, and Rollback, the choice is more complex. One option is to remove all of the transaction related code and replace each call to Commit with a call to the ApplyUpdates method of the appropriate ClientDataSets. This lets dbExpress handle transaction control for you and simplifies your code. The only time this will not work is if you must combine the updates of two or more ClientDataSets in a single transaction. Explicit calls to Rollback should be replaced with a call to the ClientDataSet CancelUpdates method. CancelUpdates undoes the pending changes in the ClientDataset change log.

The other option is to retain your explicit transaction control statements. Surprisingly, this is the most difficult option for three reasons.

- You must add a call to ApplyUpdates before each commit.
- You must change the location of every call to StartTransaction if you wish to take advantage of the short transaction lifetime provided by the dbExpress provide/resolve architecture.
- You must change every call to StartTransaction, Commit, and Rollback to pass a TTransactionDesc parameter.

As you can see, it is less work to remove your explicit transaction code than it is to retain it. Let's look at these three steps in detail. After you start a transaction and before each call to Commit, you must call the ApplyUpdates method of each ClientDataSet that is participating in the transaction so the changes in the ClientDataSet change log will be applied to the database. Until the call to ApplyUpdates, the database has not been changed so there is nothing to commit.
In a Borland Database Engine client/server application the transaction control process looks like this.
Start a **transaction**.

2. Let the user **edit** the data.
3. Commit or Rollback the **transaction**.

In this model, the transaction is active for the entire time the user is editing the data. One of the major advantages of the dbExpress provide/resolve architecture is short transaction lifetime. To realize this benefit, the transaction control process must follow these steps so the transaction is not active while the user is making changes.

1. Let the user edit the data in a ClientDataset.
2. Start a transaction.
3. Call ApplyUpdates.
4. Commit or Rollback the transaction.

To change the transaction control process in your program to the dbExpress model, you must **move** each call to **StartTransaction** so it immediately precedes the corresponding call to **Commit**. Then, add a call to **ApplyUpdates** between the call to StartTransaction and the call to Commit.

Both the Borland Database Engine Database component and the dbExpress SQLConnection component have StartTransaction, Commit, and Rollback methods. Unlike the Borland Database Engine, dbExpress lets you have multiple transactions active at the same time, if your database supports doing so. To support multiple transactions, the SQLConnection StartTransaction, Commit, and Rollback methods take a parameter of type TTransactionDesc. TTransactionDesc is declared as:

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

For each simultaneous transaction, you must declare a variable of type TTransactionDesc and set the TransactionId to a number that is unique among all transactions that are active at the same time. The GlobalId field is used only with Oracle® database transactions. IsolationLevel must be set to `xilDIRTYREAD, xilREADCOMMITTED`, or `xilREPEATABLEREAD`. The CustomIsolation field is not currently used.

Since the Borland Database Engine does not support multiple transactions active at the same time, all you need to do is declare a single variable of type TTransactionDesc in the interface section of a unit that is in the uses clause of every unit that contains transaction control statements. In your application's startup code, set the **TransactionId** field to **`one`** and set the **IsolationLevel** to **`xilREADCOMMITTED`** or **`xilREPEATABLEREAD`**. Finally, change every **call** to **StartTransaction, Commit,** and **Rollback** to pass the TTransactionDesc variable as a parameter.

Replace all DataSet components

Replacing all of your dataset components is the biggest task in the conversion process. You must replace each Borland Database Engine Query component with a SQLQuery component and each Borland Database Engine StoredProc component with a SQLStoredProc component. The process is complicated by the fact that Borland Database Engine components have properties and events that their dbExpress counterparts do not have. Any references in your code to the missing properties will have to be removed.

For each SQLQuery and SQLStoredProc component that requires bi-directional access or the ability to edit records, you will need to add a DataSetProvider and a ClientDataSet. Connect the event handlers for events that are missing from SQLQuery and SQLStoredProc to the corresponding events of the ClientDataSet. To minimize changes to existing code, consider giving the ClientDataSet components the same name as the Bolrand Database Engine Query or Borland Database Engine StoredProc component you are replacing.
Return to the **SQLQuery** and **SQLStoredProc components** and **instantiate** their field objects using the Fields Editor. After the field objects have been instantiated, set the **pfInKey** provider flag to **true** for each field in the primary key. If there are any computed fields that should not be updated, set their **pfInUpdate** and **pfInWhere** provider flags to **false**.

There may be cases where you do not need to add a DataSetProvider and ClientDataSet. For example, suppose you have a query whose result set you read once from the first row to the last to add one of the field values to the Items property of a combo box or list box. In this case, since you are scanning the records in one direction only and require nothing more than read only access, you can use the SQLQuery or SQLStoredProc component directly.

If you are using cached updates in your Borland Database Engine application, remove the **UpdateSQL components** as you convert your **Borland Database Engine Query components** to **SQLQuery components**. Cached updates are not needed with dbExpress since the ClientDataSet and DataSetProvider provide the same features and benefits.

Finally, remember to add calls to the **ClientDataSet ApplyUpdates** methods, as discussed in the preceding section on transaction control, so changes made to data in the ClientDataSet local buffer will be written to the database.

Data type mapping

dbExpress uses two new data types that you have not encountered when working with the Borland Database Engine. All numeric values that will not fit in a double precision floating-point value will be returned to your application in a TFMTBCDField object. The TFMTBCDField object stores the value as type TBCD which is a true BCD value with a maximum precision of 32 digits. To perform mathematical operations on the value in a TFMTBCDField, use the **AsVariant** property to store the **value** in a **variant variable.**

dbExpress uses the TSQLTimeStampField field object type to return date-time values. The TSQLTimeStampField stores the date-time information in a variable of type TSQLTimeStamp. TSQLTimeStamp is a Delphi record with separate fields for the year, month, day, hour, minute, second, and fraction. Fraction is the number of milliseconds. TSQLTimeStamp allows datae-time values to be stored with no loss of precision. TSQLTimeStampField has As… properties to convert the date-time value to other data types.

Replacing the BatchMove component

dbExpress has no analog to the Borland Database Engine BatchMove component. If you are using BatchMove components in your application, you will have to replace them with code.

Migrating a desktop database application to dbExpress

If you are migrating an application that uses a desktop database, such as Paradox® or dBase,® to dbExpress, you must deal with several additional problems.

Data conversion

Since dbExpress does not support Paradox or dBase tables, you will have to change to a SQL database server supported by dbExpress and move your data to the new database. There are three ways to do this.

- Use the Borland Database Engine Data Pump utility that comes with current versions of Delphi Enterprise and Borland C++Builder.™
- Use a third party data conversion utility. Some databases include data import or conversion utilities. You may also find a conversion utility that meets your needs on the Internet at little or no cost.
- Write your own data conversion program.

The SQL database you are converting to may not support all of the data types in your desktop database. For example, Paradox has a Boolean data type but not all SQL database servers do. If you are going to use a conversion utility be sure to identify any data types that are not directly supported by your new database and determine how the conversion tool handles these types.

Most SQL database servers support both the CHAR and VARCHAR data types, while desktop databases have a single string type. Make sure that the conversion tool you choose lets you convert your string fields to the appropriate data type in your database server.

You may want to change data types for other reasons. For example, Paradox tables store all real numbers in double precision floating-point format. Most SQL database servers

support a fixed-point format, such as NUMERIC or DECIMAL. Unlike floating-point, fixed-point formats can represent all decimal fractions exactly and are, therefore, a much better choice for storing currency values or any data where accurate fractional values are important. Once again, you need to find out if the conversion tool you are considering will perform this type of conversion.

Server security

You cannot open a database on a SQL database server until you enter a valid username and password. You can have all users of your application use the same username and password on the database server or each user can have their own account. However you manage security, you will have to add code to your application.

Most desktop databases allow you to encrypt the database file to prevent unauthorized access. Many database servers do not. Instead, they rely on the security features of the operating system to protect the database file or files from unauthorized access. Normally the only users that require any access at all to the physical database files are the database administrator and the account used by the database server service. If your application will be installed on stand-alone systems, for example, notebook computers that sales or service representatives take on the road, make sure the database you choose provides the security you need.

Set architecture

Typical desktop database applications let users open and browse entire tables using the Borland Database Engine Table component. This approach can lead to poor performance when working with SQL database servers, because the entire table must be fetched across the network from the database server to each workstation. Table components also use a lot of network and server bandwidth fetching metadata so they can construct the SQL statements to select and update records on the server.

Holding records in memory in a ClientDataSet is not a problem in a properly designed client/server application, because these applications fetch only a few records at a time from the server. However, converting a desktop database application that lets users browse large tables can consume a lot of memory on the client since the entire table is held in memory by the ClientDataSet.

When you convert an application that uses Borland Database Engine Table components extensively, you have three choices.

1. Replace each Borland Database Engine component with a SQLTable, DataSetProvider, and ClientDataSet.
2. Replace each Borland Database Engine component with a SQLQuery, DataSetProvider, and ClientDataSet. Select all columns and all rows from the table.
3. Replace each Borland Database Engine component with a SQLQuery, DataSetProvider, and ClientDataSet and convert the application to client/server set orientation.

Option 1 is the easiest since the properties of the SQLTable component most closely match the properties of the Borland Database Engine Table component. If your tables are not too large, performance may be adequate. The disadvantage of this option is that if you want to convert some or all of the datasets to select smaller sets of records at some time in the future, you will have to replace the SQLTable components with SQLQuery components.

The second option is a better choice. Even though you will still be selecting all columns and all rows, your application will now be query based. All you have to do to the data access components to change to a more set-oriented design in the future is to change the SQL statement.

Option three will likely require substantial code changes. Applications that are designed for a client/server environment typically force the user to enter some selection criteria before displaying any data. The selection criteria are used in the WHERE clause of a query to fetch a small number of records for the user to work with. Once the user has finished with one

set of records, the user enters another set of selection criteria to get the next set of records. If your application lets the user browse entire tables, you will have to add the code and forms to let the user enter selection criteria and change the WHERE clause to use the values supplied by the user.

Summary

Migrating your applications from the Borland Database Engine to dbExpress provides many benefits.

- Shorter transactions.
- More control over network traffic and system resource use.
- Reduction in size and system resource use.
- Improved performance.
- Easier deployment.
- A smaller deployment package.
- Cross-platform support.

dbExpress minimizes the conversion effort because it includes a suite of dataset components designed to mimic the properties, methods and events of the Borland Database Engine dataset components as closely as possible. This lets you move up to a superior technology with minimum changes to your existing code.

About the author

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books and over 90 articles, and is a member of Team Borland, providing technical support on the Borland Internet newsgroups. He has presented over two dozen papers at Borland Developer Conferences in the U.S. and Europe. Bill is also a nationally known trainer and has taught Delphi and InterBase programming classes across the country and overseas. Bill can be reached at bill@dbginc.com.

Borland®

100 Enterprise Way
Scotts Valley, CA 95066-3249
www.borland.com | 831-431-1000