Surviving Client/Server: Delphi 2.0 Potpourri

by Steve Troxell

Last month we began a look into the major new database features of Delphi 2 by exploring cached dataset updates in depth. This month we'll wrap up our look at cached updates, continue on with dataset filtering and conclude with an overview of the new data dictionary facility.

Cached Updates Continued

As promised last month, I'm going to show you how to use `TUpdateSQL` to modify a result set containing columns from multiple tables. I'm going to pick up where I left off with the last issue, so if you missed it, this part will be difficult to understand and you may want to skip to the next section titled *Dataset Filtering*.

If you'll recall from the last issue, we can use the `TUpdateSQL` component in conjunction with cached updates to provide our own SQL statements for the modification of the underlying result set for data-aware controls. The example program shown in Figure 1 illustrates this case. The query shown at the top normally produces a read-only result set because it involves a join between two tables. We are going to allow editing of all three fields in the data-aware grid even though two of them come from the `Employee` table and one comes from the `Department` table.

Note that this is a highly contrived example as it would be far-fetched to provided a grid such as this that allows modification of the name of the department the employee is assigned to. This was the best I could come up with within the limitations of the example InterBase database. However, it does serve to illustrate the point.

For this example, we use two `TUpdateSQL` components: one for the `Employee` table and one for the `Department` table. The SQL statements used for each component's `ModifySQL` property are shown in Listing 1. Note that we included the `Dept_No` field in the result set even though it is not part of our grid display. This is so we would have something with which we can reference the `Department` table in the `UPDATE` statement.

Remember from last month when we had a single `TUpdateSQL` component, we bound it to the dataset component by assigning it to the dataset's `UpdateObject` property. In this case, how can we assign two `TUpdateSQL` components to one `UpdateObject` property? We don't. Instead we attach a handler to the dataset's `OnUpdateRecord` event and employ the `TUpdateSQL` components through code, as shown in Listing 2.

This event handler is ultimately called by `ApplyUpdates` when we post our cached changes. The handler passes in `DataSet`, which refers to the dataset being modified, and `UpdateKind`, an indicator of the type of update (insert, modify, or delete) being performed.

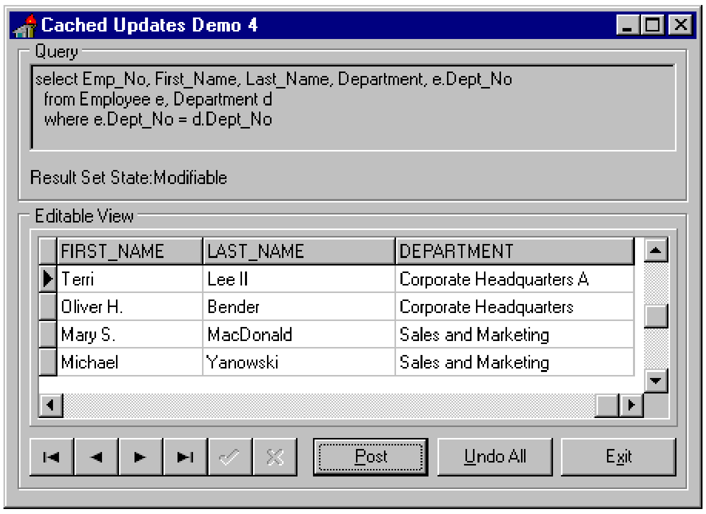
```
update Employee
  set First_name = :First_Name,
      Last_Name = :Last_Name
  where Emp_No = :Old_Emp_No
update Department
  set Department = :Department
  where Dept_No = :Old_Dept_No
```

➤ *Listing 1: TUpdateSQL SQL statements*

```
procedure TfrmMain.qryGetEmployeesUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  UpdateSQL1.DataSet := DataSet;
  UpdateSQL1.Apply(UpdateKind);
  UpdateSQL2.DataSet := DataSet;
  UpdateSQL2.Apply(UpdateKind);
  UpdateAction := uaApplied;
end;
```

➤ *Listing 2: OnUpdateRecord handler*

➤ *Figure 1*

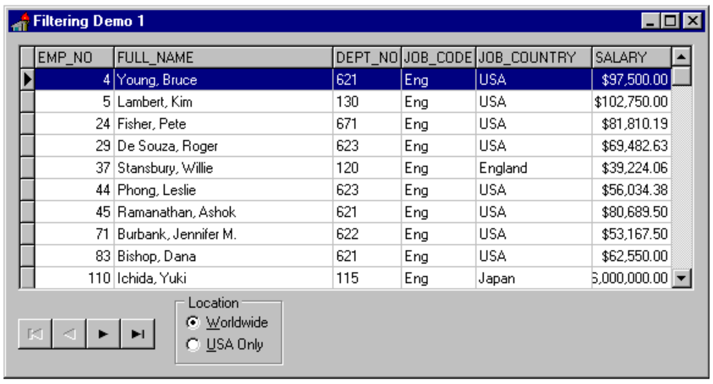


```
select Emp_No, First_Name, Last_Name, Department, e.Dept_No
 from Employee e, Department d
 where e.Dept_No = d.Dept_No
```

Result Set State:Modifiable

FIRST_NAME	LAST_NAME	DEPARTMENT
Terri	Lee II	Corporate Headquarters A
Oliver H.	Bender	Corporate Headquarters
Mary S.	MacDonald	Sales and Marketing
Michael	Yanowski	Sales and Marketing

➤ *Figure 2*



EMP_NO	FULL_NAME	DEPT_NO	JOB_CODE	JOB_COUNTRY	SALARY
4	Young, Bruce	621	Eng	USA	$97,500.00
5	Lambert, Kim	130	Eng	USA	$102,750.00
24	Fisher, Pete	671	Eng	USA	$81,810.19
29	De Souza, Roger	623	Eng	USA	$69,482.63
37	Stansbury, Willie	120	Eng	England	$39,224.06
44	Phong, Leslie	623	Eng	USA	$56,034.38
45	Ramanathan, Ashok	621	Eng	USA	$80,689.50
71	Burbank, Jennifer M.	622	Eng	USA	$53,167.50
83	Bishop, Dana	621	Eng	USA	$62,550.00
110	Ichida, Yuki	115	Eng	Japan	5,000,000.00

Location: Worldwide [✓] / USA Only

```
procedure TForm1.btnWorldwideClick(Sender: TObject);
begin
  qryGetEmployees.Filtered := False;    { Deactivates filtering }
end;
procedure TForm1.btnUSAOnlyClick(Sender: TObject);
begin
  with qryGetEmployees do begin
    { Note the use of two single quotes to achieve
      one embedded single quote }
    Filter := 'Job_Country = ''USA''';
    Filtered := True;                   { Activates filtering }
  end;
end;
```

➤ *Listing 3: Dataset filtering*

Normally, when a TUpdateSQL component is bound to a dataset through the UpdateObject property, it is inherently aware of which dataset it is bound to and executes the SQL statements through that dataset component. When used in the OnUpdateRecord event handler, no direct association has been made between the TUpdateSQL components and the dataset, so we must explicitly bind them by setting the TUpdateSQL.DataSet property.

Then we call the TUpdateSQL.Apply method and pass in the type of update being made so the component knows which SQL statement to execute for this particular update. Finally we set the return parameter UpdateAction to uaApplied to indicate that we successfully applied the changes. UpdateAction defaults to uaFail and the update is aborted unless you explicitly set this parameter otherwise.

That's all there is to it. Since TUpdateSQL gives you direct control over the exact SQL statements being used to modify the result set, you can incorporate behavior that the normal Delphi database handling doesn't allow.

Dataset Filtering

All TDataSet components now have a slick new record filtering capability which allows you to extract various subsets of the result set returned by a TTable, TQuery or TStoredProc. You accomplish this by supplying filtering criteria similar to an SQL WHERE clause in the component's Filter property. Then, by setting the Filtered property to True, the result set is scanned and those rows failing to satisfy the filter are made invisible in the result set. When you set Filtered back to False, all the original records are available again in the result set.

All this is accomplished internally without the need to submit independent queries to the server. The actual result set remains intact, but the visibility of certain rows is affected by filtering. This is the same concept we saw in last month's column when using the UpdateRecordTypes property to show classes of record changes with cached updates.

The program shown in Figure 2 illustrates how this works. This example shows all employees with a job code of Eng. The radio buttons at the bottom allow you to apply a filter to show the whole set or a subset of just those engineers working in the United States. When USA Only is clicked, the entries for Willie Stansbury and Yuki Ichida disappear. The code for the eventhandlers for these buttons is shown in Listing 3.

The filtering is done at the client end, operating on the dataset already returned by the query. Since the filtering in this case involves reducing the set of records already returned by the query, the need to submit a new query with a more restrictive WHERE clause is eliminated. The existing result set can be reduced (and restored) any number of times and any number of ways without querying the database again. Obviously there will still be situations where it will be more efficient to re-query the database, but in those cases where an initial set of records is retrieved from the database and the user is allowed to reduce that set, then this technique is more efficient.

Filtering in this manner can include multiple expressions connected with AND or OR operators, but cannot include functions like UPPER(). Keep in mind that while the syntax is like an SQL WHERE clause, it is not SQL and you cannot use SQL functions (or any function for that matter) within the Filter property.

OnFilterRecord Event Handler

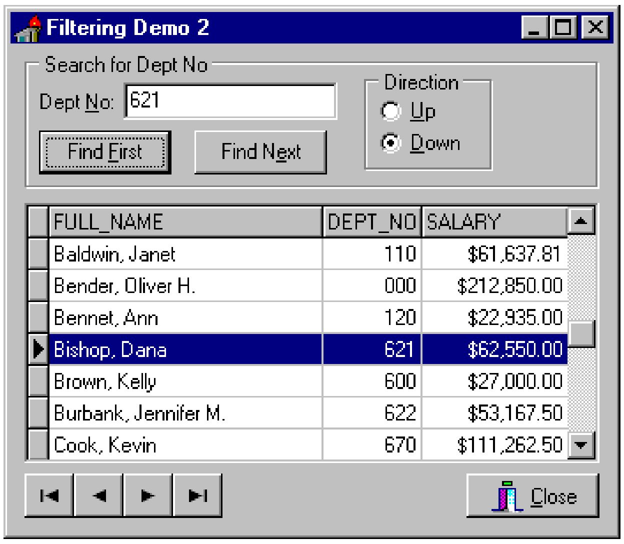
Alternatively, we could have performed the same filtering by using the dataset's OnFilterRecord event handler instead of the Filter property. When Filtered is set to True, the OnFilterRecord event handler is called once for each record in the dataset. You simply provide the code necessary to identify those records to be excluded from the filtered result.

For example, the same filtering shown in Listing 3 could have been accomplished by the code which is shown in Listing 4, where qryGetEmployeesFilterRecord is the query's OnFilterRecord event handler. If the handler's Accept parameter is set to True (the default), then the record is included in the

```
procedure TForm1.qryGetEmployeesFilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  Accept := DataSet['Job_Country'] = 'USA';
end;
procedure TForm1.btnWorldwideClick(Sender: TObject);
begin
  qryGetEmployees.Filtered := False;
end;
procedure TForm1.btnUSAOnlyClick(Sender: TObject);
begin
  qryGetEmployees.Filtered := True;
end;
```

➤ *Listing 4: OnFilterRecord event handler*

➤ *Figure 3*



Dept No: 621

Up [✓ Down]

FULL_NAME	DEPT_NO	SALARY
Baldwin, Janet	110	$61,637.81
Bender, Oliver H.	000	$212,850.00
Bennet, Ann	120	$22,935.00
Bishop, Dana	621	$62,550.00
Brown, Kelly	600	$27,000.00
Burbank, Jennifer M.	622	$53,167.50
Cook, Kevin	670	$111,262.50

```
procedure TForm1.edtSearchMaskChange(Sender: TObject);
begin
  with qryGetEmployees do
    if edtSearchMask.Text = '' then
      Filter := ''
    else
      Filter := Format('Dept_No = ''%s''', [edtSearchMask.Text]);
end;
procedure TForm1.btnFindFirstClick(Sender: TObject);
begin
  with qryGetEmployees do begin
    if btnDown.Checked then
      FindFirst
    else
      FindLast;
    if not Found then
      MessageDlg('No matches found.', mtError, [mbOk], 0);
  end;
end;
procedure TForm1.btnFindNextClick(Sender: TObject);
begin
  with qryGetEmployees do begin
    if btnDown.Checked then
      FindNext
    else
      FindPrior;
    if not Found then
      MessageDlg('No matches found.', mtError, [mbOk], 0);
  end;
end;
```

➤ *Listing 5: Using filtering for searches*

filtered result set. By using the `OnFilterRecord` event handler rather than the `Filter` property, you have all the functions and logic Delphi provides at your disposal in writing a filter. But beware, this code is applied to every record in the dataset, so it should be streamlined for performance.

Implementing Searches

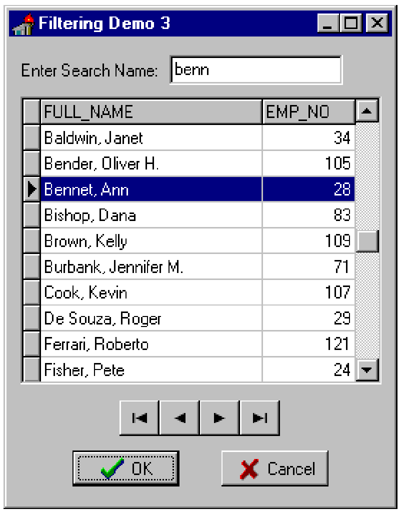
You can do much more with filtering than just reduce the visible result set. You can leave the entire result set visible and employ filtering to move the record pointer through all the rows matching the filter. In this manner, you can implement the standard *Find* and *Find Next* operations as illustrated in Figure 3. In this example, you enter a particular department number, and press the `Find` button. The record pointer instantly moves to the first employee in that department, while all the employees remain visible. Pressing the `Find Next` button moves the record pointer to the next matching employee, and so forth.

Listing 5 shows how to accomplish this. The `OnChange` event for the edit control sets the query's `Filter` property. The `Find First` button then calls the query's `FindFirst` or `FindLast` method depending on the search direction chosen by the user. These methods move the record pointer to the first (or last) record in the dataset that meets the filter criteria. Finally, the query's `Found` property indicates whether the `Find...` method actually found anything. Note that `Filtered` is not used in this operation: that would make all non-matching rows invisible.

Incremental Searches

This same search technique can be refined to allow incremental searches. What makes this possible is the last piece of the filtering puzzle: the `FilterOptions` property. This property comprises a set of two values, `foCaseInsensitive` and `foNoPartialCompare`, which are both present by default. By turning off `foNoPartialCompare`, partial matches to the filter criteria can be performed.

Figure 4 shows an example program that incrementally searches the employee name. The only code necessary to implement this is shown in Listing 6. Note that an asterisk is appended to the end of



Enter Search Name: benn

FULL_NAME	EMP_NO
Baldwin, Janet	34
Bender, Oliver H.	105
Bennet, Ann	28
Bishop, Dana	83
Brown, Kelly	109
Burbank, Jennifer M.	71
Cook, Kevin	107
De Souza, Roger	29
Ferrari, Roberto	121
Fisher, Pete	24

➤ *Figure 4*

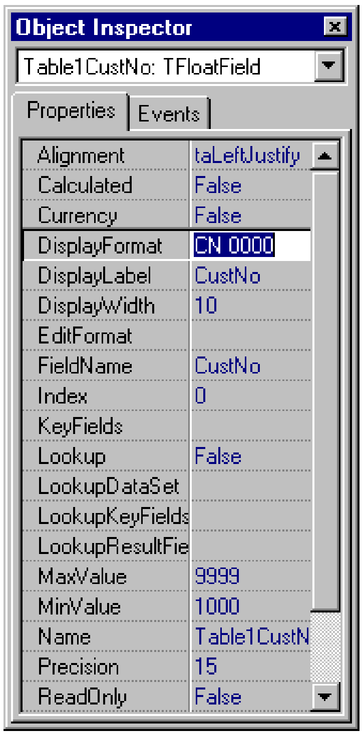


Table1CustNo: TFloatField

Alignment	taLeftJustify
Calculated	False
Currency	False
DisplayFormat	CN 0000
DisplayLabel	CustNo
DisplayWidth	10
EditFormat	
FieldName	CustNo
Index	0
KeyFields	
Lookup	False
LookupDataSet	
LookupKeyFields	
LookupResultFie	
MaxValue	9999
MinValue	1000
Name	Table1CustN
Precision	15
ReadOnly	False

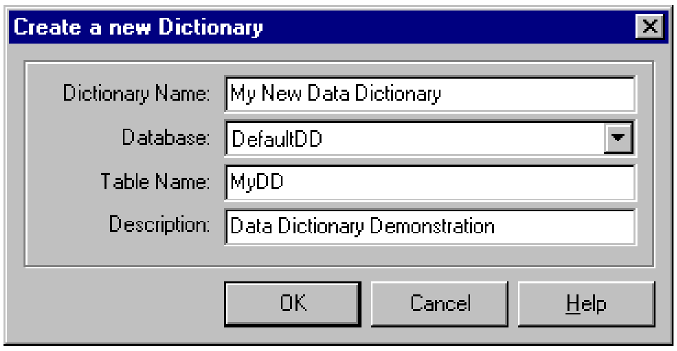
➤ *Figure 5*

the search value. This is a required wildcard to perform partial matches (something that the Delphi documentation neglects to mention).

Client/Server Considerations

While these filtering techniques are certainly very helpful, you must keep in mind that, like most of the database functionality in Delphi, they were designed with desktop databases in mind. You must exercise good judgment in how and when you implement dataset filtering with client/server databases.

➤ *Figure 6*



Dictionary Name:	My New Data Dictionary
Database:	DefaultDD
Table Name:	MyDD
Description:	Data Dictionary Demonstration

```
procedure TForm1.edtSearchMaskChange(Sender: TObject);
begin
  with qryGetEmployees do begin
    Filter := Format('Full_Name = ''%s*''', [edtSearchMask.Text]);
    FindFirst;
  end;
end;
```

➤ *Listing 6: Partial match filtering*

Keep in mind that dataset filtering is applied on the result set at the client end. It is still up to you to retrieve a manageable result set from the server before dataset filtering can be applied. An incremental search dialog with dataset filtering is still a bad idea if you are simply going to bind an unrestricted `TTable` to a million row customer table.

Using The Data Dictionary

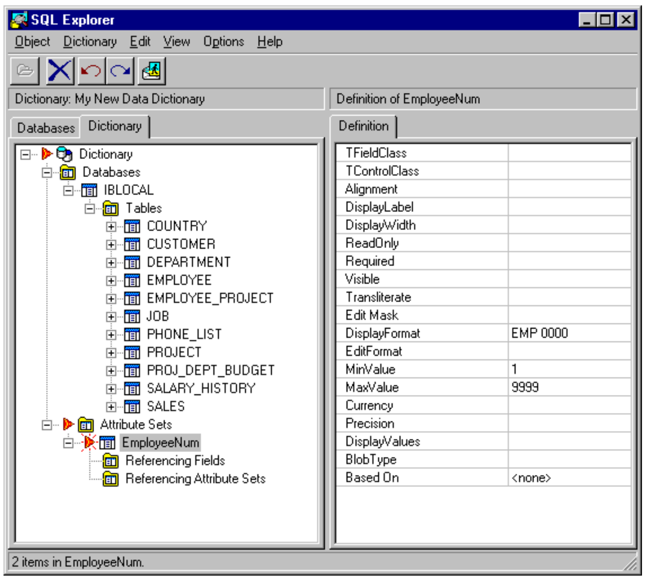
As with version 1, Delphi 2 allows you to customize the attributes for any or all of a dataset's field components by using the Fields Editor and changing field attributes through the Object Inspector (see Figure 5). However, with version 2 you can drag and drop the field from the Fields Editor onto the form and Delphi automatically provides a data-aware control and corresponding label for that field (and a `TDataSource` component, if needed, to link the control and dataset).

What's also new is the Data Dictionary which is used to predefine attributes for selected fields outside of the application so that the field is consistently presented throughout a project or across several projects. Before we discuss how to create a data dictionary for your projects, let's use the example one already provided by Borland to get a feel for what we'll use it for.

Delphi 2 ships with an example data dictionary already in place called the Borland Database Engine Sample Data Dictionary. This is set up for the `DBDEMOS` example database. To see how the Data Dictionary works, let's create a new form in Delphi 2, drop a `TTable` component on it, and hook the `TTable` to the `DBDEMOS` alias and the `CUSTOMER.DB` table. Next, double-click on the `TTable` to pull up its Fields Editor, right-click the Fields Editor, and select `Add Fields` from the speedmenu to add all the fields in the table to the Fields Editor.

Now is where the Data Dictionary part comes in. Click the `CUSTNO` field in the Fields Editor and take a look at the Object Inspector (see Figure 5). If you have a sharp eye, you'll see that some of the properties do not have their usual default values. Specifically, the `Display-Format`, `MaxValue`, `MinValue` and `Required` properties have changed. This is because these properties have been assigned specific values in the Data Dictionary for this particular field in the database.

Basically, the Data Dictionary allows you to predefine field component default property values for any field in the database. These defaults are accessible by any Delphi project using the database. You are free to change any or all of the values in the field component itself the Data Dictionary simply provides new default values.



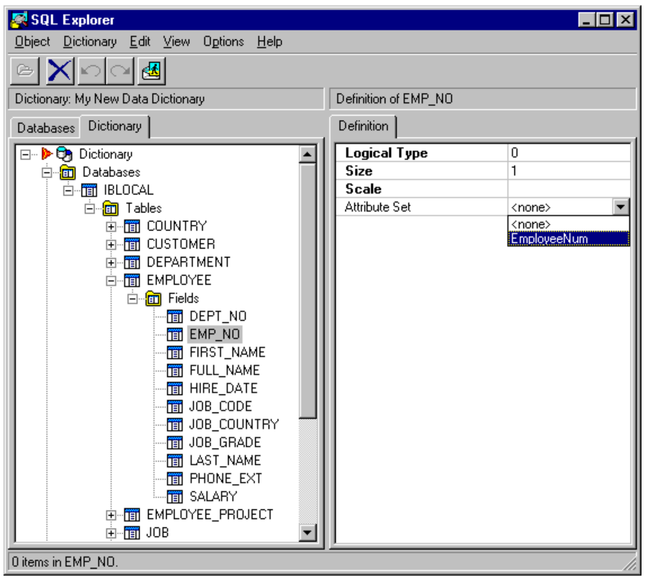
Dictionary: My New Data Dictionary

- Dictionary
  - Databases
    - IBLOCAL
      - Tables
        - COUNTRY
        - CUSTOMER
        - DEPARTMENT
        - EMPLOYEE
        - EMPLOYEE_PROJECT
        - JOB
        - PHONE_LIST
        - PROJECT
        - PROJ_DEPT_BUDGET
        - SALARY_HISTORY
        - SALES
  - Attribute Sets
    - EmployeeNum
      - Referencing Fields
      - Referencing Attribute Sets

Definition of EmployeeNum

TFieldClass	
TControlClass	
Alignment	
DisplayLabel	
DisplayWidth	
ReadOnly	
Required	
Visible	
Transliterate	
Edit Mask	
DisplayFormat	EMP 0000
EditFormat	
MinValue	1
MaxValue	9999
Currency	
Precision	
DisplayValues	
BlobType	
Based On	<none>

2 items in EmployeeNum.



Dictionary: My New Data Dictionary

- Dictionary
  - Databases
    - IBLOCAL
      - Tables
        - COUNTRY
        - CUSTOMER
        - DEPARTMENT
        - EMPLOYEE
          - Fields
            - DEPT_NO
            - EMP_NO
            - FIRST_NAME
            - FULL_NAME
            - HIRE_DATE
            - JOB_CODE
            - JOB_COUNTRY
            - JOB_GRADE
            - LAST_NAME
            - PHONE_EXT
            - SALARY
        - EMPLOYEE_PROJECT
        - JOB

Definition of EMP_NO

Logical Type	0
Size	1
Scale	
Attribute Set	<none>

<none>
EmployeeNum

0 items in EMP_NO.

➤ *Left: Figure 7*
➤ *Right: Figure 8*

What you gain by using the Data Dictionary is that you can define selected field attributes independently of any particular form or project so that the field definition can be applied consistently to all occurrences of that field throughout all your projects. For example, you might have special edit masks for order numbers, customer numbers, or credit card numbers. You can even override the default TDBEdit control that gets dropped on the form when dragging the field from the Fields Editor. Want a TDBLookupCombo instead? No problem, it can be assigned in the Data Dictionary.

Creating A Data Dictionary

So how do you setup a data dictionary for your own databases? Start with SQL Explorer (Database Explorer for you Delphi Developer types) by selecting Database | Explore from the main menu in Delphi 2. Then select Dictionary | New from the SQL Explorer main menu. This takes you to the dialog shown in Figure 6.

Dictionary Name simply identifies this dictionary when we have to select between ours and the Borland Sample Dictionary, for example. Database is the alias for the database in which we will store the dictionary information. Note that this is not necessarily the alias for the database we are creating the dictionary for. The Borland Sample Dictionary applies to the DBDEMOS database, but Borland chose to store the dictionary itself in a database called DefaultDD. For our purposes here, we will also store our dictionary in the DefaultDD database. In practice you could just as easily store the dictionary within the same database you're describing. It's important to understand which database we are referring to throughout this process.

Table Name is the name of the table that will be created in DefaultDD to contain all the information for the data dictionary we are creating. DefaultDD happens to be a Paradox database, so we have to be sure our table name is legal for Paradox and obviously should not conflict with any existing table in that database. Description is an arbitrary text field that further defines our dictionary.

Now that we've created the dictionary, we're going to populate it. The first thing we need to do is associate a particular database with the dictionary. From the main menu select Dictionary | Import From Database and select the IBLOCAL alias for the example InterBase database. This is the database for which we'll be creating the dictionary. Note that we could add any number of databases to our data dictionary.

Now that we have a database, we need to setup some field definitions. The Data Dictionary refers to these as *Attribute Sets*. In the outline, right-click on Attribute Sets and select New. Enter EmployeeNum as the name for the new attribute set. In the right-hand pane (see Figure 7) are all the possible properties for this attribute set. In our case we know that the employee number is an integer value between 1 and 9999 so we set the MaxValue and MinValue fields as shown. Note that the TControlClass property is where we would define a different data control (such as TDBLookupCombo) for a field in the Data Dictionary. When we are done setting properties, right-click on our attribute set in the left-pane (EmployeeNum) and select Apply from the speedmenu to save what we've done.

The last thing we need to do is bind the attribute set with one or more fields in the database. In the outline, drill-down through Dictionary | Databases | IBLOCAL | Tables | EMPLOYEE | Fields | EMP_NO (see Figure 8). In the right-hand pane you'll see the property Attribute Sets. Pull down the drop down list and select the EmployeeNum attribute set to bind it to this field. Right-click on the EMP_NO field in the outline and select Apply from the speedmenu to save the binding. Now do the same thing for the EMP_NO field in the EMPLOYEE_PROJECT table.

If you return to the outline in the left-hand pane, you'll now see these two fields listed under Attribute Sets | EmployeeNum | Referencing Fields. You will probably have to select `Refresh` from the outline speedmenu first. Now, whenever you reference this database in a Delphi project, the BDE links with the Data Dictionary to obtain default property values for any field component you may define for the `EMPLOYEE.EMP_NO` field.

If you change the characteristics of an attribute set defined in the Data Dictionary, any existing field components defined in Delphi projects are not affected in any way. Only the creation of new field components for the changed field will respond to the new values in the Data Dictionary. However, the field component retains knowledge of its link to the Data Dictionary so you can refresh the component's properties by selecting `Retrieve Attributes` from the field's speedmenu.

Delphi's Data Dictionary supplies a mechanism for external field definitions, but does nothing to aid you in managing changes to the dictionary by way of identifying or cascading those changes in affected projects. It is entirely up to you to manually identify each affected field in all your projects and decide whether to apply the changes for each and every field.

Conclusion

With Delphi 2 come some significant improvements in database handling that greatly simplify many common challenges faced by database application developers. However, Delphi still retains a decidedly desktop database flavor. Client/server developers can certainly benefit from features like cached updates and filtering if used carefully. If used indiscriminately, these features can undermine some principal advantages of the client/server model.

Next month we'll see some tips and tricks to increase your productivity and effectiveness with SQL.

Postscript

An issue that I hear from readers quite frequently is *"How can I write a Delphi program to access any backend server with minimal changes?"* The question is a legitimate one. It is of great value to have an application that is portable across more than one backend server, particularly when customers may have already invested heavily in a particular RDBMS. The answer is not simple and I have decided to address the topic in this column. In fact, the answer is so un-simple that I need your help in compiling it.

There are a number of backends that can be used with Delphi and my experience is limited to only InterBase and Microsoft SQL Server. We have developed a means to support both of these servers for our development needs, but I would like to hear from others who have tackled the same issues to get a broader feel for the problems involved and more generalized solutions.

If you would like to participate in this "group-study" then I would like to hear from you. I would especially like to hear from anyone who has created a significant Delphi app that allows the backend to be either Paradox or a SQL database. What obstacles did you face and how did you solve them? Please e-mail me at one of the addresses below.

Steve Troxell is a software engineer with TurboPower Software where he is developing Delphi client/server applications for the casino industry. Steve can be contacted at stevet@tpower.com or on CompuServe at 74071,2207