Designing An Article Index Database System

by Bob Swart

As your collection of copies of *The Delphi Magazine* grows, you want to be sure you can find all those nuggets of information quickly and easily, just when you need them, don't you? A database of all the articles, reviews, tips and clinic items is what's needed and of course Delphi is the ideal tool to create such a system!

This article describes the development of *The Delphi Magazine Article Index Database* (TDMAid). The full source code is on this month's disk, along with the data files with details from all the back issues. We'll be updating both the software and data regularly and you will be able to download new versions from *The Delphi Magazine* Web site. In fact, as this issue goes to press, we're continuing to add new features to the software, so check out

http://ourworld.compuserve.com/homepages/DelphiMagazine

now for the latest version!

Analysis

First of all, we need to define the information we want to store in the database. At the least, we need a table for the article information and one for the authors. The basic information that is needed in the ARTICLE table is:

- ➢ Article ID (integer key)
- ➢ Article Type (one of: Article, Review, Tip, Clinic, Misc)
- ➢ Title (string of 64 characters)
- ➢ Author1 ID (integer foreign key)
- ➢ Author2 ID (integer foreign key)
- ➢ Author3 ID (integer foreign key)
- ➢ Issue Number (integer)
- ➢ Issue Date (month, year) *for display, calculated field from Issue Number*
- ➢ Page Number of start of article (integer)
- ➢ Summary (memo field)

As noted, another table with author information would be most informative as well (including a picture of the author). The information that we would like to see in the AUTHOR table is:

- ➢ Author ID to link to ARTICLE table (integer key)
- ➢ Name (string of 64 characters)
- ➢ Email address (memo field, generally with one or two lines of bio info and e-mail address)
- ➢ Photograph (graphic blob)

We also decided to create a separate KEYWORD table:

- ➢ Article ID (foreign integer key)
- ➢ Keyword / Key Phrase (string of 28 characters)

Data Design

Now that we've defined the three tables "on paper", it's time to build them. The easiest way to build a new empty table is to use the Database Desktop from Delphi 1 (note that the code will work with both versions of Delphi without any problems, I've used the Delphi 1 tools to make sure every reader is able to re-create this example as we proceed. If you have Delphi 2 then you can use the Database Explorer of course.

Normally, when using Borland Database Desktop to create tables for a new project, it would be advisable to create an alias first. An alias is a logical name to specify the location of database tables and connection parameters for database servers. Using an alias, you are not required to specify the full path for your database tables, which is most helpful. Aliases are stored in your local IDAPI.CFG file, which is one reason why I'm not using an alias for this project: everyone would need to define an alias and I want the source code for this project to compile and run without a single change.

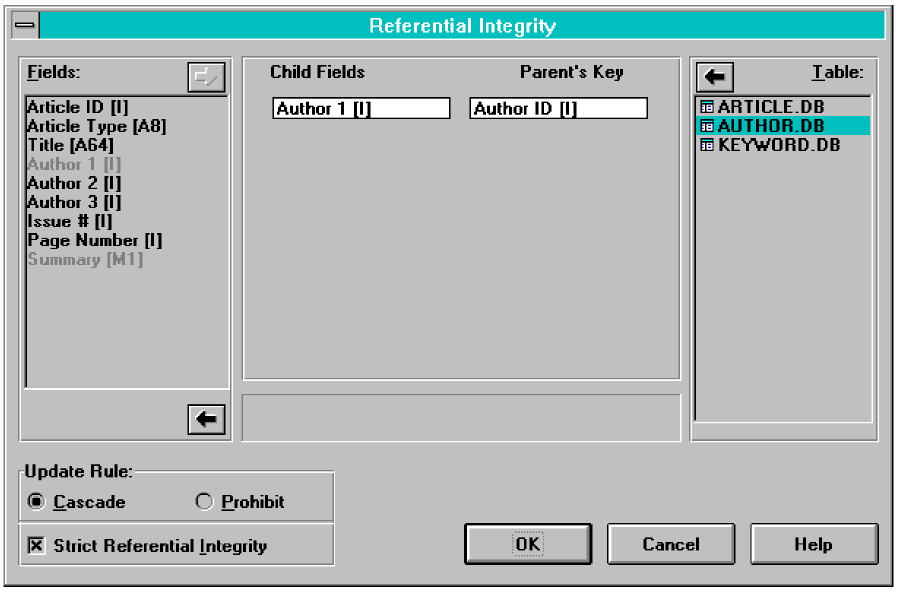
So, while we could tell the user to create an alias using the BDE Configuration Utility BDECFG.EXE, or create an alias programmatically by calling DbiAddAlias, we prefer to use no alias at all: our program just expects the database tables to be in the same directory as the executable, so we can call ExtractFilePath(ParamStr(0)) and set the result to the DatabaseName property of our tables and queries.

Of course, for design time, we should use an alias. For this project I used an alias called TDMAID, set to the directory in which I've placed all my tables and source files. If you want to follow the development of the TDMAid application here, then I suggest you do the same.

I've chosen to use the Paradox 5.0 for Windows table type, as it's somewhat richer than dBASE, has automatic referential integrity capabilities and has native BDE support. To save space, I won't go through the details of creating the tables, as it's very straightforward.

Note that I defined a *picture mask* for the Article Type field in the ARTICLE table, comprising a set of five possible values: Article, Review, Tip, Clinic and Misc. Only one of these five values (or a NULL value, since it's not a required field) is accepted by the BDE as a valid value for this field.

To define the referential integrity relationship between the ARTICLE and AUTHOR tables we close both tables in the Database Desktop and change our working directory to be the directory where the tables are saved (in case this wasn't our current working directory). Then, we open the ARTICLE table and choose Restructure Table. In the Table Properties combobox we select Referential Integrity and click the Define button. In the Referential Integrity dialog (Figure 1), select the Author 1 field as the field for which we want to define a referential integrity rule, namely that this field



Fields:
- Article ID [I]
- Article Type [A8]
- Title [A64]
- Author 1 [I]
- Author 2 [I]
- Author 3 [I]
- Issue # [I]
- Page Number [I]
- Summary [M1]

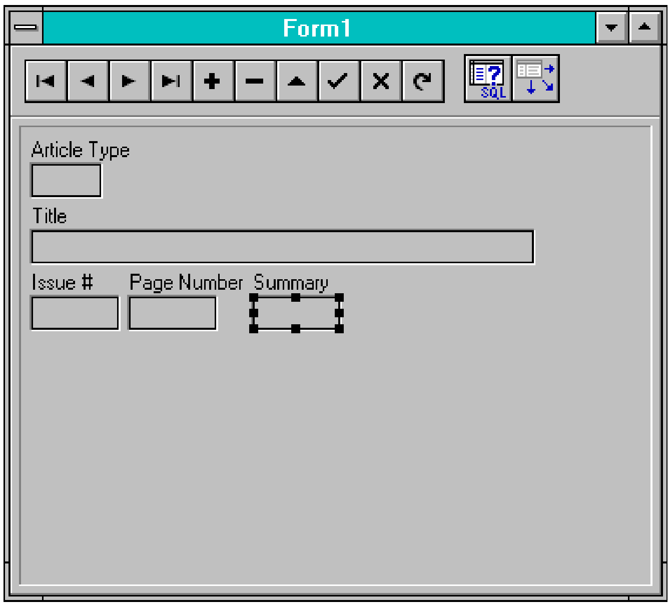
Child Fields: Author 1 [I]
Parent's Key: Author ID [I]

Table:
- ARTICLE.DB
- AUTHOR.DB
- KEYWORD.DB

Update Rule: [✓ Cascade] Prohibit

[✓ Strict Referential Integrity]

➤ *Figure 1*



➤ *Figure 2*

should be connected to the Author ID field of the AUTHOR table. When we select the AUTHOR.DB table in the right listbox, to indicate that this is the *master* table for our referential integrity rule, we automatically get the key from that table (the Author ID field) that's being connected to the Author 1 field. This will now ensure that we cannot delete an Author from the AUTHOR table unless all his/her articles have been deleted in the ARTICLE table as well. More background theoretical information on the use and implications of referential integrity can be found in works on relational database theory, by Codd or Date for example, and in the Borland Database Desktop Help (which will also explain why we want to use the Strict Referential Integrity checkbox).

If we click on the OK button on the dialog we can give this referential integrity rule a name (use Author1 in this case). We then need to do exactly the same for the Author 2 and Author 3 fields, naming the rules appropriately. Saving the ARTICLE table again will update the ARTICLE.VAL file with the referential integrity information.

You should now be able to figure out how to add the referential integrity rule named Article to the Article ID field of the KEYWORD table, connected to the ARTICLE table (to prevent a user removing an article unless all keywords connected to the article are removed). This all seems pretty obvious, and it may seem unnecessary (as we're only writing a database viewer in this article), but for the data-entry people at *The Delphi Magazine* offices it sure makes a lot of difference to know that it's the database itself that holds referential integrity rules that guard over the integrity of the data in the tables.

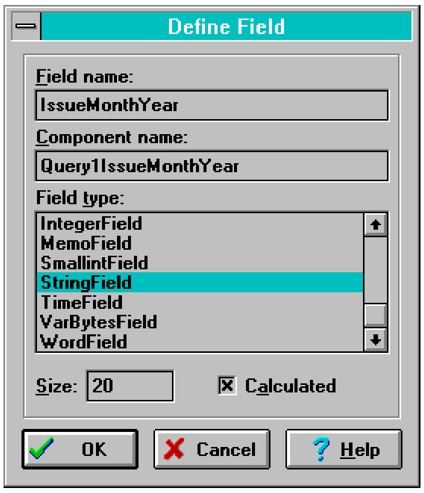
Main Form Design

Now it's time to let Delphi generate some nice forms to browse the data in our tables. By default, Delphi will start with a new, empty project, with an empty form staring at us. Because we're going to use the Database Form Expert, we don't need this empty form, so we'd like to remove it from our project. To do that, select the View | Project Manager menu, then close the empty form Form1 by clicking on the line with Unit1 and Form1, followed by a click on the Remove button. Delphi will then ask if we would like to save the changes to UNIT1.PAS. Just answer No and the empty form disappears. Close the Project Manager and we are left with a truly empty project!

Database Form Expert

We now start the Database Form Expert to create a new form, by selecting it from the Help menu. In Form Options we'll choose a simple form. Because we want to add some search facilities to our application, we'll create a form using TQuery objects instead of TTable.

In the next page, select the ARTICLE table, as this will be the first one we will put on the form, then on the third page of this dialog, we can select the fields for which we want Delphi to generate visual data-aware controls on the form. We don't want to see every field. In fact, we'd rather like to hide the key ID fields from both the ARTICLE and AUTHOR tables. On the fourth page, we'll pick the default horizontal field layout. After clicking on the Next button one more time, we agree that this should be our main form and the initial result is shown in Figure 2.


Define Field

Field name: IssueMonthYear

Component name: Query1IssueMonthYear

Field type:
- IntegerField
- MemoField
- SmallintField
- StringField
- TimeField
- VarBytesField
- WordField

Size: 20 [✓ Calculated]

➤ *Figure 3*

If we look closely, we can see some unexpected results. The fields all seem a little smaller than we expected, and the Summary field seems all wrong: isn't this supposed to be a memo data-entry field? It indeed is of type TDBEdit and not of type TDBMemo. Why? I don't know, but we'll replace it with a genuine TDBMemo, setting its DataSource property to DataSource1 and DataField to Summary. Shameless plug: *Marco and Dr. Bob's Database Expert* (Issue #7, Listing 3, page 14) would not have assigned a TDBEdit to a memo field!

Calculated Field

The next thing we need to do is add the field IssueMonthYear, which is calculated from the Issue # field. To add a calculated field, double click the Query component to start the Fields Editor, then click on the Define button to start the Define Field dialog, in which we can enter a new calculated field with the Field name IssueMonthYear (the dialog itself will make up the Component name Query1IssueMonthYear) and select the field type of StringField with a size of 20 (Figure 3).

Next, we drop a TDBLabel and a TDBEdit between the Issue # and Page Number fields on the form, assign their DataSource to DataSource1, assign their DataField to IssueMonthYear (the calculated field we've just defined) and drag the controls around a bit to give the form a somewhat more satisfying look. Since we're doing this, let's work on the naming as well.

```
procedure TForm1.ArticleCalcFields(DataSet: TDataset);
begin
  case QArticleIssue.Value of
    1: QArticleIssueMonthYear.Value := 'April 1995';
    2: QArticleIssueMonthYear.Value := 'July 1995';
    3: QArticleIssueMonthYear.Value := 'September 1995';
    4: QArticleIssueMonthYear.Value := 'November 1995';
  else
    if QArticleIssue.Value >= 5 then { monthly from now on }
      QArticleIssueMonthYear.Value := Format('%s %d',[
            LongMonthNames[((QArticleIssue.Value-5) mod 12)+1],
            1996 + (Pred(QArticleIssue.Value-4) div 12)])
    else
      QArticleIssueMonthYear.Value := 'unknown'
  end
end;
```

➤ *Listing 1*

```
procedure TForm1.TabSetAuthorClick(Sender: TObject);
begin
  QAuthor.Active := False { reset last query };
  case (Sender AS TTabSet).TabIndex of
    0: QAuthor.ParamByName('AUTHOR').AsInteger := QArticleAuthor1.Value;
    1: QAuthor.ParamByName('AUTHOR').AsInteger := QArticleAuthor2.Value;
    2: QAuthor.ParamByName('AUTHOR').AsInteger := QArticleAuthor3.Value
  end;
  QAuthor.Active := True { execute! }
end;
```

➤ *Listing 2*

Click on the Query1 component and rename it to QArticle (all the fields for the query will automatically be renamed).

To define the value of the calculated field (which is now called QArticleIssueMonthYear, we need to go to the Events page of the QArticle query component and write our code in the OnCalcFields event handler (Listing 1).

The algorithm used to calculate the field is pretty simple. For the first four issues, we use a kind of lookup table: the first issue had a cover date of April 1995, then followed July, September and November for issues 2, 3 and 4. From Issue 5, which had a cover date of January 1996, the magazine has been monthly, so for any issue number >= 5, we can use:

```
Month = 1 +
  ((IssueNumber - 5) mod 12)
Year = 1996 +
  ((IssueNumber - 4) div 12)
```

Authors

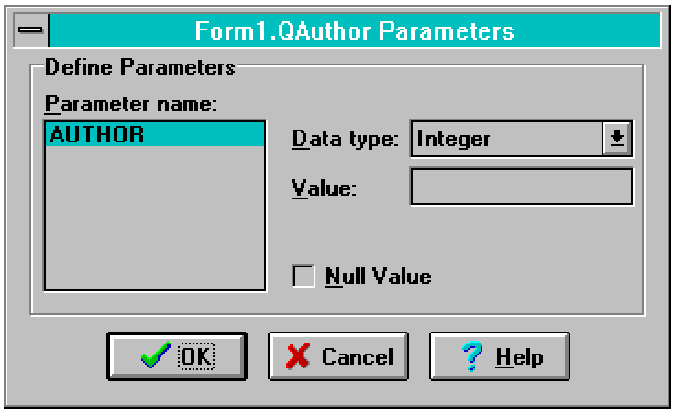
Now it's time for us to add the list of authors for each article to the form. We do this by dropping another TQuery (renamed to QAuthor), a TDataset and a TTabSet (renamed to TabSetAuthor) on the form. Note that we won't need a NoteBook to come with the TabSet, as we just use the TTabSet to define which Author we're looking for (the first, second or third author). We connect the DataSet2 component to QAuthor and prepare the SQL query as follows:

```
SELECT * FROM AUTHOR WHERE
  (AUTHOR.'Author ID' =
  :AUTHOR)
```

The :AUTHOR part is a parameter in this SQL query, which we need to define and fill with a sensible value of Author ID before executing the query (Figure 4). We will set this parameter when one of the three tabs is clicked and also set the Active property of QAuthor to True. See Listing 2.

Now that we can execute the QAuthor query, we need some data-aware controls on the form to show the contents of the data fields. But first we need to make sure that we can actually access the individual fields: double-clicking on the QAuthor component opens up the Field Editor, where we can Add all the fields.

We need a TDBEdit to hold the name of the author, a TDBMemo to hold the Author's bio and email



➤ *Figure 4*

```
procedure TForm1.DBNavigatorClick(Sender: TObject; Button: TNavigateBtn);
begin
  TabSetAuthor.Tabs.Clear;
  if BtnAuthorName.Caption = '1' then begin
    if QArticleAuthor1.Value > 0 then TabSetAuthor.Tabs.Add('first author');
    if QArticleAuthor2.Value > 0 then
      TabSetAuthor.Tabs.Add('second author');
    if QArticleAuthor3.Value > 0 then
      TabSetAuthor.Tabs.Add('third author')
  end else begin
    { names of authors }
    Screen.Cursor := crSQLWait;
    if QArticleAuthor1.Value > 0 then begin
      QAuthor.Active := False;
      QAuthor.ParamByName('AUTHOR').AsInteger := QArticleAuthor1.Value;
      try
        QAuthor.Active := True;
        TabSetAuthor.Tabs.Add(QAuthorName.Value)
      except
        TabSetAuthor.Tabs.Add('first author')
      end
    end;
    if QArticleAuthor2.Value > 0 then begin
      QAuthor.Active := False;
      QAuthor.ParamByName('AUTHOR').AsInteger := QArticleAuthor2.Value;
      try
        QAuthor.Active := True;
        TabSetAuthor.Tabs.Add(QAuthorName.Value)
      except
        TabSetAuthor.Tabs.Add('second author')
      end
    end;
    if QArticleAuthor3.Value > 0 then begin
      QAuthor.Active := False;
      QAuthor.ParamByName('AUTHOR').AsInteger := QArticleAuthor3.Value;
      try
        QAuthor.Active := True;
        TabSetAuthor.Tabs.Add(QAuthorName.Value)
      except
        TabSetAuthor.Tabs.Add('third author')
      end
    end;
    Screen.Cursor := crDefault
  end;
  if TabSetAuthor.Tabs.Count > 0 then
    TabSetAuthor.TabIndex := 0
end;
```

➤ *Listing 3*

address, and a TDBImage to hold the photo (we don't need the Author ID). We can place the three controls on the form just between the article summary memo control and the TTabSet. And then of course we link up the controls to DataSource2 and set the relevant fields.

Finally, since we only want to *browse* the data, we can remove the Insert, Delete, Edit, Post, Cancel and Refresh buttons from the Navigator.

Now we're almost done. At design time, the form should resemble Figure 5. Actually, this has some extra things we haven't addressed quite yet (apart from the fact that this is a Win95 screenshot, to show again that this project will work with Delphi 1 and 2, and can run on Windows 3.1x, Windows 95 and NT). We've added a right mouse click TPopupMenu (with an AboutBox and extra information) and three extra TSpeedButtons, one to exit the program and two for special search options.

Author Tabs

The SpeedButton with the A label is used to switch between showing First Author etc and the actual author names on the TabSet. In the latter case, a query for each author is needed to get to the author name and on a slow machine this may take too long for comfort.

For completeness, the method that executes the code to get the true names (or fixed names) for the authors and assigns them to the tabset's tabs is DBNavigatorClick, since this is the place where we move from one record to another. The first part of this routine is pretty simple, we assign a fixed label to the tabset in case an author is present. We can see if an author is present if we check to see if the QArticleAuthor field is greater than zero (valid Author IDs are from 0 up). See Listing 3.

We could alternatively have checked each QArticleAuthor field for a NULL value, but checking on the actual value helps check for the validity of the Author Key.

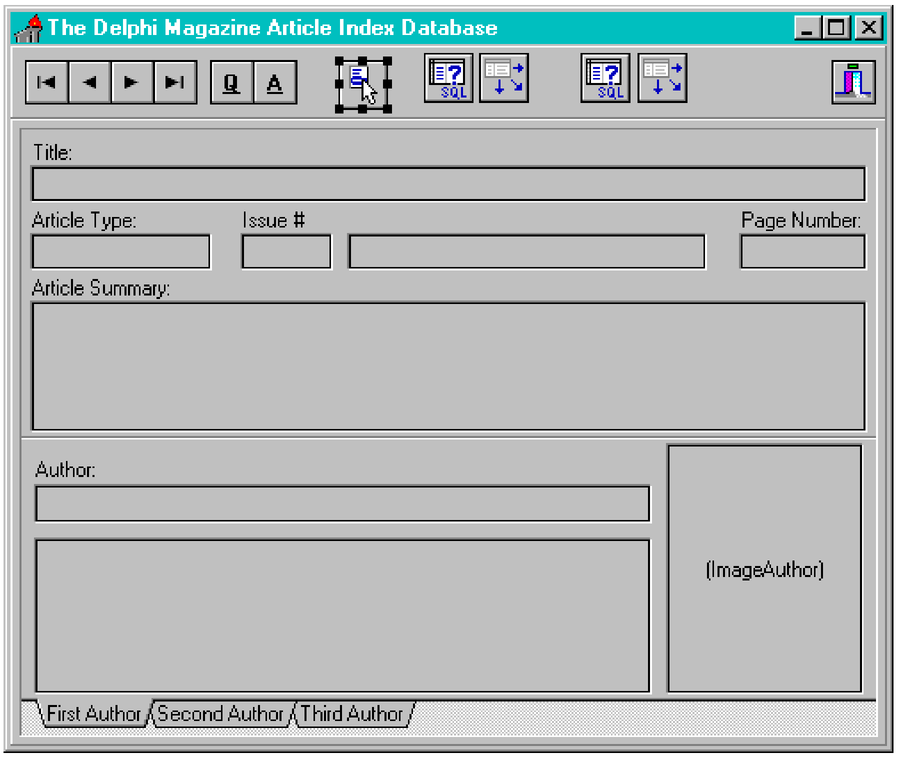
As can be seen in the listing, the caption of the BtnAuthorName button decides for us whether we should use the three fixed labels or the actual names of the authors. Getting these actual names is what the rest of the DBNavigatorClick method is all about. We set the Screen.Cursor to an SQL hourglass and then check for each of the three authors if the Author ID field is greater than zero, to indicate that an author is present. If so, we set the AUTHOR parameter of the QAuthor query to the value of the QArticleAuthor field, set the Active property of the QAuthor to True to start the query, get the result, and set it as the name of the next new tab in the TabSet. If the query fails, we go back and use the fixed name instead (in a try-except block, so the user doesn't notice anything).

Finally, if all the authors have been assigned to a tab, the first tab is selected by setting the TabIndex property of the TabSetAuthor to 0. This will cause the TabSet to actually collect the information for the selected author as implemented in the TabSetAuthorClick method (Listing 2).

A final thing that might be helpful when clicking on the DBNavigator is to somehow have an indication of which record number we are on and how many records there actually are in the resulting query. The latter can be collected as QArticle.RecordCount, of course, but how do we know which record we are at? It appears that we have to use some native BDE calls to get this information and it's different if you're using a dBASE or a Paradox TDataset. See Listing 4 for the function CurrentRecordNumber.

Note that in case of an inactive dataset we call DBError which in turn will raise an exception that we catch ourselves, so we can optimise this routine for that particular instance. We use the value of CurrentRecordNumber as the first statement in the DBNavigatorClick:

```
ToolBar.Caption :=
  Format('Article %d/%d',
  [CurrentRecordNumber(
    QArticle),
  QArticle.RecordCount]);
```



➤ *Figure 5*

```
function CurrentRecordNumber(DataSet: TDataSet): LongInt;
  { Gives current record number as result for dBase/Paradox datasets only }
var CursorProps: CurProps;
    RecordProps: RECProps;
begin
  Result := 0;
  with DataSet do
  try
    if State = dsInactive then DBError(SDataSetClosed);
    Check(DbiGetCursorProps(Handle, Cursorprops));
    UpdateCursorPos;
    Check(DbiGetRecord(Handle, dbiNOLOCK, nil, @RecordProps));
    case CursorProps.iSeqNums of
      0: Result := RecordProps.iPhyRecNum; {dBase}
      1: Result := RecordProps.iSeqNum   {Paradox}
    end
  except
    { skip errors - return 0 }
  end
end;
```

➤ *Listing 4*

Article Query

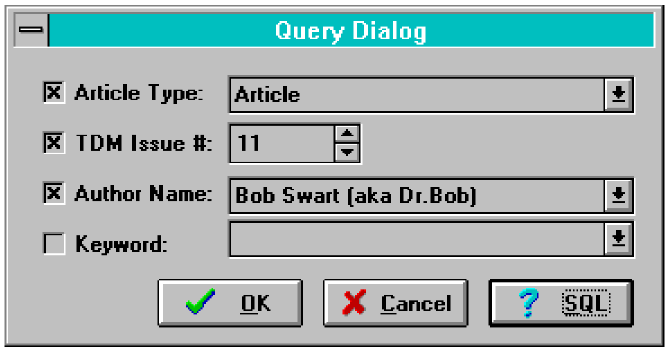
Having a nice browser to walk through all articles, with corresponding authors, is one thing. Being able to find what you're looking for is another! The last SpeedButton on the form, the one with the Q caption, is used to define and execute a search query. Figure 6 shows our query dialog.

Since we're already using a TQuery to walk through the list of Articles, we just need to re-define the SQL query every time we re-define our search criteria.

But before we can even try to generate the SQL query, we must initialise the comboboxes with the choices that the user can make. For Issue Numbers we can use a simple TSpinEdit that starts at 0 and runs up to 100. The Article Type is equally simple: we can fill the combobox with the five article types.

But what about the Author Name and Keyword? Well, for those we need to actually look inside the AUTHOR and KEYWORD tables and pick every author name and keyword. In short, we just open the relevant table, go to the first record, get the field value we need, put it in the combobox and go to the next record until we're at the end of the table. Pretty straightforward, eh? See Listing 5.

Unfortunately, if we store only the author name in the combobox we will need to look for the author ID later when performing the query. Alternatively, we would need to do a JOIN between the ARTICLE and AUTHOR tables. However, JOINs take far more time than normal queries, which means we can win some efficiency if we find a way not to use one! So, we would in fact like to use something like a TDBLookupCombo, one that shows us the names of the authors, but where the actual values are the IDs of the authors. Unfortunately, the TDBLookupCombo component itself won't work for us in this case, since


Query Dialog

[✓] Article Type: Article
[✓] TDM Issue #: 11
[✓] Author Name: Bob Swart (aka Dr.Bob)
[ ] Keyword:

➤ *Figure 6*

```
procedure TQueryDlg.FormCreate(Sender: TObject);
begin
  with TTable.Create(nil) do begin
    DatabaseName := DataPath;
    TableName := 'AUTHOR.DB';
    ReadOnly := True;
    try
      Active := True;
      First;
      while not Eof do begin
        AuthorName.Items.Add(Format('%-64s%d',
          [FieldByName('NAME').AsString,
          FieldByName('AUTHOR ID').AsInteger]));
        Next
      end
    finally
      Active := False
    end { authors };
    TableName := 'KEYWORD.DB';
    try
      Active := True;
      First;
      while not Eof do begin
        Keyword.Items.Add(FieldByName('KEYWORD').AsString);
        Next
      end
    finally
      Active := False;
      Free
    end { keywords }
  end
end;
```

➤ *Listing 5*

```
procedure TQueryDlg.CheckArticleTypeClick(Sender: TObject);
begin
  if not CheckArticleType.Checked then
    ArticleType.ItemIndex := -1
  else
    if (ArticleType.ItemIndex = -1) then
      if (ArticleType.Items.Count > 0) then
        ArticleType.ItemIndex := 0
      else
        CheckArticleType.Checked := False
end;
```

➤ *Listing 6*

```
procedure TQueryDlg.ArticleTypeChange(Sender: TObject);
begin
  CheckArticleType.Checked := ArticleType.ItemIndex >= 0
end;
```

➤ *Listing 7*

it insists on sorting on the lookup value instead of the display value (we can't set a secondary index on the QAuthor either).

With a little trick we can get the best of both worlds: both the names of the authors (in alphabetical order) and the author ID values, hidden from the end-user but present in the combobox. The only thing we need to do is to pad the author names with enough spaces to push the author ID out of the visible range of the combobox. I stretched all the names to 64 characters, padding them with spaces where needed, and appended the author ID after the 64th character – present but not visible. The Sorted property of the combobox will ensure that it's sorted on name, not ID. After the user has selected a name we just remove the first 64 characters and convert it to an integer Author ID. Hey presto: an artificial secondary index!

Now that we have filled each combobox let's focus on the interaction between the checkboxes and the comboboxes. If an item is selected in one of the comboboxes then the checkbox is checked. If we uncheck the checkbox, the item in the combobox is de-selected. So either both of them are selected (checked) or not. As an example, the code for the article type checkbox is shown in Listing 6. We need a corresponding routine for when the selection in the combobox changes. See Listing 7.

SQL

Now, finally, we can rest back and focus on generating the SQL query. It consists of three parts: the SELECT part, the WHERE part and the ORDER BY part. We've already seen the SELECT part, which was generated by the Database Form Expert.

Now, what about the WHERE part? If we check an Article Type, Issue Number and so on, we need to add a WHERE clause to the query. For an Article Type, this is actually a pretty simple addition; in pseudo code it is:

```
WHERE ARTICLE."Article Type" =
  ArticleType.Items[
  ArticleType.ItemIndex];
```

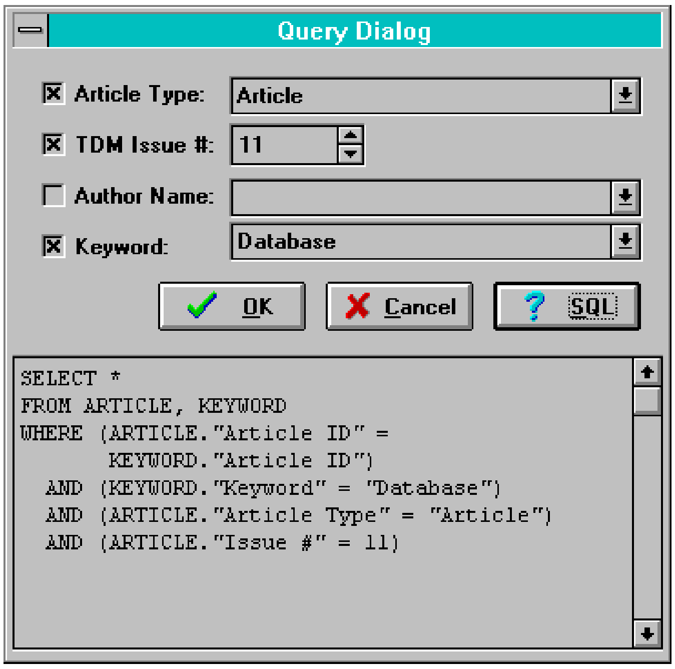
We have to generate the SQL query and fill in the value of the selected item of the combobox. Which is the main reason why the checkboxes and comboboxes must be in sync at all times: if we decide to add a WHERE clause to the query, the combobox must have a valid selection to build the query with. For Issue Number, we have an equally simple addition to the query:

```
WHERE ARTICLE."Issue #" =
  IssueNumber.Value
```

We'll skip the Author Name for now and look at the Keywords first. Since the `ARTICLE` table doesn't know anything about Keywords, it seems we have no choice but to use a `JOIN` here; in pseudo-code:

```
WHERE (ARTICLE."Article ID" =
  KEYWORD."Article ID")
  AND (KEYWORD."Keyword" =
  Keyword.Items[
    Keyword.ItemIndex])
```

If we click on the SQL button in the Query Dialog, the dialog is resized to reveal a hidden memo field that holds the text of the SQL query that will be generated and executed. For an Article Type, Issue Number and Keyword, the generated SQL query would be as in Figure 7. Note the `FROM ARTICLE, KEYWORD` part, which denotes the `JOIN` in the query.



```
SELECT *
FROM ARTICLE, KEYWORD
WHERE (ARTICLE."Article ID" =
       KEYWORD."Article ID")
  AND (KEYWORD."Keyword" = "Database")
  AND (ARTICLE."Article Type" = "Article")
  AND (ARTICLE."Issue #" = 11)
```

➤ *Figure 7*

OR-Bug?

Now, time to move on to the last part: the Author Name. It doesn't sound that difficult: we can get the Author Name from the combobox and we've hidden the Author ID in the text starting at position 65. So, what's the difficulty this time? Well, the problem is that the Author Name we've selected can be the first, second or third author of the article. So we actually need an `OR` statement this time:

```
WHERE (ARTICLE."Author 1" =
    AuthorID)
  OR (ARTICLE."Author 2" =
    AuthorID)
  OR (ARTICLE."Author 3" =
    AuthorID)
```

At first sight, this seems to work fine. However, as soon as we combine this statement with the other parts, like the Article Type, we get into trouble. It seems that when we combine an `AND`-part and an `OR`-part, even if we use parentheses, the query will not work correctly:

```
SELECT * FROM ARTICLE
WHERE (ARTICLE."Article type" =
  Article)
  AND ((ARTICLE."Author 1" = 2)
   OR (ARTICLE."Author 2" = 2)
   OR (ARTICLE."Author 3" = 2))
```

The combined query above will, in practice, also deliver any reviews written by two or more authors where the ID of the second or third author is 2. It seems the query *internally* becomes:

```
SELECT * FROM ARTICLE
WHERE (ARTICLE."Article type" =
  Article) AND
  (ARTICLE."Author 1" = 2)
  OR (ARTICLE."Author 2" = 2)
  OR (ARTICLE."Author 3" = 2)
```

where each part separated by an `OR` in the `WHERE` clause is considered a valid alternative. Is this a bug? I couldn't find enough documentation, but I think it's a bug (using ODBC the query is executed correctly). So, we amend the SQL:

```
SELECT * FROM ARTICLE
WHERE (ARTICLE."Article type" =
 Article) AND
  (ARTICLE."Author 1" = 2)
  OR (ARTICLE."Article type" =
   Article) AND
   (ARTICLE."Author 2" = 2)
  OR (ARTICLE."Article type" =
   Article) AND
   (ARTICLE."Author 3" = 2)
```

We'll move on now to the final part of the SQL query: the `ORDER BY` clause. As stated before, it would be a good idea to have the resulting articles of the query sorted by Issue number and Page number. To do that, we just need:

```
SELECT * FROM ARTICLE
ORDER BY ARTICLE."Issue #",
  ARTICLE."Page Number"
```

You can check the code which generates the final SQL query in the routine `TQueryDlg.BtnSqlClick` in the file QUERY.PAS on the disk.

Conclusion

Well, by now you must be tired and only want to see the end result! On this month's disk, in directory TDMAID, you will find two ZIP files: one with the source code and a second with the data files. An enhanced executable version is on our Web site. Happy hunting!

Bob Swart (aka Dr.Bob, find him at http://www.pi.net/~drbob/) is the Delphi Specialist for Bolesian in The Netherlands and a freelance technical author. Bob is co-author of *The Revolutionary Guide to Delphi 2*, published by WROX Press. In his spare time Bob likes to watch videos of Star Trek Voyager and Deep Space Nine with his 2-year old son Erik Mark Pascal.