Surviving Client/Server: Cached Updates

by Steve Troxell

Last month I promised to take you on a tour of all the spiffy new database stuff in Delphi 2.0 and how it relates to client/server development. Well, space and time limits mean we'll be able to take a look at just one new feature this month: cached updating with `TDataSet` components. This refers to buffering all changes made to one or more datasets and committing them to the database all at once. Ironically, that short, simple sentence opens the door to a whole slew of possibilities for your database front ends, whether they are based on a client/server database or a desktop database.

To illustrate how this differs from Delphi 1.0 (and Delphi 2.0 without caching enabled), let's take the example of a data-aware grid used to edit a dataset. As you modify fields or add new records in this grid, Delphi posts the changes to the database whenever you explicitly or implicitly post the record (by scrolling to a new record, for example). The same thing happens when you delete from the dataset: Delphi posts the change to the database immediately.

When caching is enabled (the dataset's `CachedUpdates` property is `True`), then these changes are held locally within the application until they are explicitly posted by the `ApplyUpdates` method of `TDatabase`. This sends the necessary update, insert, or delete statements for each record changed within the dataset, all encased within a single transaction. `ApplyUpdates` accepts one or more `TDataSet` components as parameters and will apply the cached changes for all of these datasets as a single transaction. This allows you to post related changes as a single unit, as in a master/detail screen.

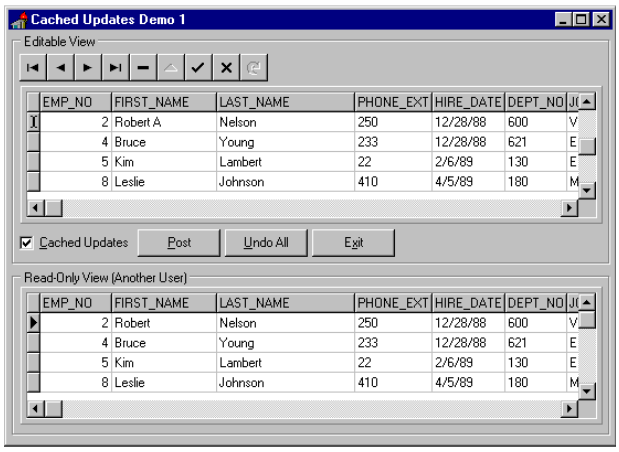
The program shown in Figure 1 illustrates this concept. This is the CACHE1 example on this issue's disk, which I encourage you to experiment with. The top grid is an editable dataset, the bottom grid represents another user looking at the same dataset. When the `Cached Updates` checkbox is cleared, if you make a change to a record and scroll off of it (or post it with the navigator), you will immediately see the change in the lower grid. When the `Cached Updates` checkbox is checked, the changes are not seen in the lower grid until you click the `Post` button. Alternatively, you can abandon all changes by clicking the `Undo All` button. The event handlers for these buttons are shown in Listing 1, where `dbEmployee` is the `TDatabase` component and `qryGetEmployees` is the `TQuery` component populating the upper grid.

Note that the three example programs in this article can be found in the SURVIVE directory on this issue's disk. Since they allow modification of the example InterBase database EMPLOYEE.GDB, they are coded to use an alias called `MYEMPL`, which is intended to be a copy of EMPLOYEE.GDB. You should either create this alias or change the `AliasName` property of the `TDatabase` components in the two programs.

How do cached updates fit into your client/server application plans? Rather than a spurious barrage of queries being sent off to the server at uneven intervals, a single grouping of queries is sent all at once. The same total number of queries is generated, but they are packaged more efficiently for two reasons.

First, a single large transmission of data may result in less overall network congestion than the same data broken out into a series of disjointed smaller transmissions. The reason being that at least one network packet must be created and transmitted for each query sent singly, but all the queries could possibly fit in one network

➤ *Figure 1: CACHE1 example program*



➤ *Listing 1: Posting/abandoning cached updates*

```
procedure TfrmMain.btnPostClick(Sender: TObject);
begin
  dbEmployee.ApplyUpdates([qryGetEmployees]);
end;
procedure TfrmMain.btnUndoAllClick(Sender: TObject);
begin
  qryGetEmployees.CancelUpdates;
end;
```

packet if sent at once. In any case, the total number of packets needed to transmit all the queries at once will nearly always be less than the total number of packets needed to transmit them one at a time. This can become a concern in a high volume environment with a large number of users and/or changes per screen.

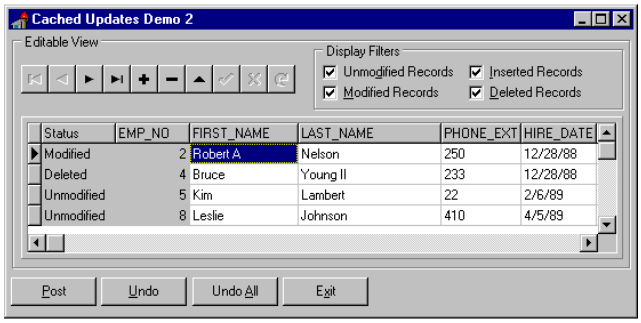
Second, when data-aware controls are used in a case like the CACHE1 program, the tendency is to start a transaction at the beginning of the edit process and then either commit or rollback all the changes depending on whether or not the user posts them. Each update holds a lock on the part of the database being updated until the entire transaction is completed. This is not so much a concern with InterBase as it is with most other SQL servers. With uncached updates, the duration of the transaction is dependent on how long the user takes to complete their changes, and this can lead to contention problems with other users. With cached updates, the same locks are occurring but the transaction length is very short (the duration of the `ApplyUpdates` call) and contention is low because the locks are dropped quickly.

Caching updates allows for more practical data entry screen development. As just described above, the user can make any number of changes to a screenful of data and decide whether to apply them or not at the click of a button. However, if the user decides to abandon their changes, the local cache merely needs to be cleared with a call to `CancelUpdates`. The data-aware controls will automatically be restored to their original values. Without caching, you must either wrap transaction control around all the changes (with the attendant pitfalls described above), or load non-data-aware controls and explicitly post the changes on a `save` action and reload original values from the database on an `abandon` action. The cached updates solution is much easier to program, reduces database contention, and provides much crisper response to the user.

Identifying Classes Of Changes

In looking at Figure 2, the CACHE2 program, you'll see that the first column of the grid shows the type of change being made to that record. When pointing to a given row, that row's changes can be cancelled with the `Undo` button. Notice also the checkboxes at the top alluding to the fact that you can selectively display any subset of changed records you want, even to include showing deleted records. It may be valuable to a user to be able to selectively cancel a deletion of a row (or any row change for that matter) before posting the whole set of changes. It may also be helpful to get a filtered list of just what was deleted, changed, or inserted prior to posting the whole batch. With clever coding of the grid's `OnDrawColumnCell` event handler, you could even paint different background colors or use different

➤ *Figure 2: CACHE2 example program*



➤ *Listing 2*

```
procedure TfrmMain.chkFilterClick(Sender: TObject);
var UpdateFilters: TUpdateRecordTypes;
begin
  UpdateFilters := [];
  if chkUnmodified.Checked then
    Include(UpdateFilters, rtUnmodified);
  if chkModified.Checked then
    Include(UpdateFilters, rtModified);
  if chkInserted.Checked then
    Include(UpdateFilters, rtInserted);
  if chkDeleted.Checked then
    Include(UpdateFilters, rtDeleted);
  qryGetEmployees.UpdateRecordTypes := UpdateFilters;
  if UpdateFilters = [] then begin
    ShowMessage(
      'At least one display filter must be checked');
    UpdateFilters := qryGetEmployees.UpdateRecordTypes;
    chkUnmodified.Checked := (rtUnmodified in UpdateFilters);
    chkModified.Checked := (rtModified in UpdateFilters);
    chkInserted.Checked := (rtInserted in UpdateFilters);
    chkDeleted.Checked := (rtDeleted in UpdateFilters);
  end;
end;
procedure TfrmMain.qryGetEmployeesCalcFields(
  DataSet: TDataSet);
begin
  { UpdateStatus is for the current record,
    contrary to what the online help says }
  with qryGetEmployeesStatus do begin
    case DataSet.UpdateStatus of
      usUnmodified: AsString := 'Unmodified';
      usModified: AsString := 'Modified';
      usInserted: AsString := 'Inserted';
      usDeleted: AsString := 'Deleted';
    end;
  end;
end;
```

```
procedure TfrmMain.dsGetEmployeesDataChange(
  Sender: TObject; Field: TField);
begin
  { Enable/disable the buttons depending on
    whether there are changes }
  if ChangingData then btnPost.Enabled := True
  else btnPost.Enabled := qryGetEmployees.UpdatesPending;
  btnUndoAll.Enabled := btnPost.Enabled;
  btnUndo.Enabled := (qryGetEmployees.UpdateStatus <>
    usUnmodified) or ChangingData;
end;
procedure TfrmMain.qryGetEmployeesStartChanges(
  DataSet: TDataSet);
begin
  ChangingData := True;
end;
procedure TfrmMain.qryGetEmployeesEndChanges(
  DataSet: TDataSet);
begin
  ChangingData := False;
end;
procedure TfrmMain.btnPostClick(Sender: TObject);
begin
  dbEmployee.ApplyUpdates([qryGetEmployees]);
end;
procedure TfrmMain.btnUndoClick(Sender: TObject);
begin
  qryGetEmployees.RevertRecord;
end;
procedure TfrmMain.btnUndoAllClick(Sender: TObject);
begin
  qryGetEmployees.CancelUpdates;
end;
```

fonts to show different classes of changes to the records.

Ok, so how to you accomplish all this? Simplicity itself with cached updates. All the key code for the CACHE2 program is in Listing 2.

To control which classes of records are displayed in a given dataset, use the property `TDataSet.UpdateRecordTypes`, which accepts a set of `TUpdateRecordTypes`: `rtUnmodified`, `rtModified`, `rtInserted` or `rtDeleted`. Whatever set members are included in this property, those row types are made visible in the dataset. In Listing 2, this is handled through the `chkFilterClick` method which is tied into the `OnClick` event handler for each checkbox on the form.

The rows are always stored internally with the dataset, this property just controls their visibility (including whether you can move to them in code). The default filter for `UpdateRecordTypes` is all members except `rtDeleted`. Also, if you attempt to assign an empty set to this property, it will default to `rtUnmodified`.

Similarly, the `TDataSet.UpdateStatus` property shows the class of modification to which the current row belongs: `usUnmodified`, `usModified`, `usInserted` or `usDeleted`. The CACHE2 program merely examines this property in the dataset's `OnCalcFields` event handler and sets the `Status` column to the appropriate text.

Another handy property is `UpdatesPending`, which is simply a `Boolean` property indicating whether or not there are updates in the cache waiting to be applied to the database. CACHE2 uses this property to enable or disable the buttons on the form depending upon whether or not there are any changes. This can be seen in the `dsGetEmployeesDataChange` method which is the `OnDataChange` event handler for the datasource.

There's also a little trickery going on here to make sure that field changes trigger the buttons' enabled state, not just record changes. The `qryGetEmployeesStartChanges` method is tied into the `TQuery`'s `AfterEdit` and `AfterInsert` event handlers, which are triggered when the user begins modifying the record. This sets a form-level `Boolean` variable to indicate that the current record is being changed. This variable gets reset by the `qryGetEmployeesEndChanges` method which is the `AfterCancel` and `AfterPost` event handler. It is only when the changes to the record are posted to the cache that `UpdateStatus` becomes available for that record.

Finally, in the `btnPostClick`, `btnUndoAllClick` and `btnUndoClick` methods you can see the code needed to post all changes to the database, abandon all changes or discard the current row's changes, respectively.

TUpdateSQL

Cached updates are made even more flexible by the inclusion of the `TUpdateSQL` component. With this component, you can specify separate SQL statements for each function: update, insert, or delete. The SQL statements appear respectively in the `ModifySQL`, `InsertSQL` and `DeleteSQL` properties of `TUpdateSQL`.

Why go to the trouble of coding modification statements when Delphi would do it itself anyway (and transparently for different backends)? First, you regain control over the exact SQL being sent. Given the particular circumstances of your dataset, you may be able to produce a more efficient or refined SQL statement for your needs.

Second, you can funnel implicit dataset changes through stored procedures where you might be performing additional server-based tasks not feasible with Delphi's direct manipulation. Or, you may have sensitive data requiring read-only permissions for the user but which the application can access and modify through a stored procedure. This would not be possible for implicit dataset changes through Delphi.

Finally, and the principal reason why `TUpdateSQL` was provided, when the nature of the SQL query prohibits a live result set, you can update it anyway with `TUpdateSQL`.

You bind the `TUpdateSQL` component to a dataset by assigning its component name to the dataset's `UpdateObject` property. Now, whenever the dataset changes are finally applied to the database, Delphi uses the corresponding SQL statement from the `TUpdateSQL` component rather than its internal logic.

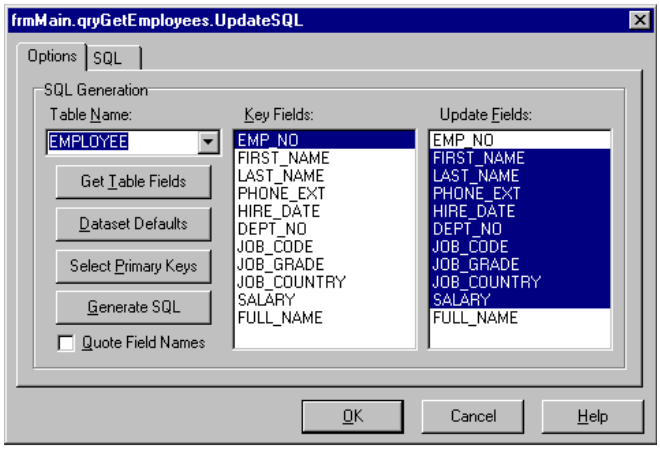
The SQL statements you supply to `TUpdateSQL` function just like any other SQL statement processed through a `TQuery` with two additional considerations. The parameters used in the SQL statements must match the field names of the dataset being updated. When applying the updates, Delphi feeds the dataset's fields into the appropriate SQL statement by matching the field names to the parameter names.

Also, each parameter can have a companion parameter of the same name with the prefix `Old_`. This parameter contains the field value before it was changed by the user. This is sometimes necessary if the field used to locate the record was itself changed to a new value. In this case, you must use the field's old value to locate the original record before applying the updates. Typically, you should be in the practice of using the `Old_` parameters for all `WHERE` clauses, whether or not those fields can be modified.

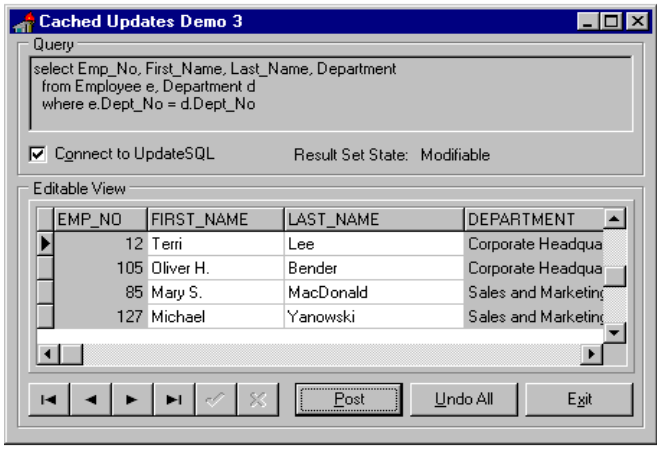
Once a `TUpdateSQL` component is bound to a dataset component's `UpdateObject` property, you can right-click on the `TUpdateSQL` component at design time and select *UpdateSQL Editor* to get the nifty little tool you see in Figure 3. This editor constructs your SQL statements for you after you've selected the appropriate fields for modification and row selection. Also, in the SQL tab, you can easily switch between the text of the three SQL statements through a set of radio buttons. This really makes short work of the tedious aspect of this component.

Modifying Read-Only Result Sets

By using cached updates and a `TUpdateSQL` component, you can effectively make changes to an otherwise read-only dataset. For example, not all SQL queries produce live result sets which can be edited



➤ *Figure 3*



➤ *Figure 4*

through code or data aware controls. This obstacle now can be overcome because the updates applied through the `TUpdateSQL` component are independent of the underlying dataset.

Take a look at the CACHE3 program shown in Figure 4. The query shown at the top of the screen is used to populate the grid at the bottom. Because this query involves a join between two tables, it violates Delphi's requirements for a live result set and would normally be read-only.

The `Connect to UpdateSQL` checkbox programmatically assigns or un-assigns a `TUpdateSQL` component to the query's `UpdateObject` property (the SQL statement used is shown in Listing 3). When the `UpdateObject` property is set, the result set becomes modifiable and the grid allows changes to the first name and last name fields. When the `UpdateObject` property is cleared, the result set becomes dead and the grid no longer allows edits. This transformation is handled entirely by the code shown in Listing 4. Delphi transparently manages the change in state completely based on the presence or absence of a valid `TUpdateSQL` component bound to the dataset.

The reason why this works is straightforward. The various constraints on the SQL statements used to produce live result sets ensure that Delphi has enough information to decide on its own what fields to update in what tables and how to locate them. In the case of the join query shown in Figure 4, Delphi only sees a result set of four fields, some of which belong to the `Employee` table and some of which belong to the `Department` table. When it comes time to update a 'row' in this result set, no single query can update both tables and Delphi would have to decide which fields belonged to which table.

With the `TUpdateSQL` component, we are in complete control over the SQL used to update the result set. In this example, we know that the `Department` field is merely a reference field and should not be part of the update. As shown in Listing 3, only the fields relevant to the `Employee` table are modified. We'll see next month how we can use `TUpdateSQL` to apply updates to both tables at once.

Conclusion

Cached updates greatly simplify the development of data entry screens and allows us to give even more control to the user without overly encumbering the database. But you'll have to take care. Since the updates are cached, other users will not see the changes until they have been posted. And other users may very well have made their own changes to the same records behind your back.

In the next issue we'll look at how we can address these concerns and wrap up our look at cached updates. In addition, we'll continue our venture into Delphi 2.0 by examining client-side filtering of datasets and the new data dictionary.

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

➤ *Listing 3: TUpdateSQL.ModifySQL*

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

➤ *Listing 4*

```
procedure TfrmMain.chkConnectUpdateSQLClick(Sender: TObject);
begin
  with qryGetEmployees do begin
    DisableControls;
    try
      Close;
      { Set the UpdateObject property of qryGetEmployees }
      if chkConnectUpdateSQL.Checked then
        UpdateObject := UpdateSQL1
      else
        UpdateObject := nil;
      Open;
    finally
      EnableControls;
    end;
  end;
end;
```