ADO Batch Updates

by Guy Smith-Ferrier

ADO includes an implementation of batch updates which is conceptually and somewhat syntactically similar to the BDE's implementation of cached updates and to `TClientDataSet`. ADO batch updates and BDE cached updates share some similarities in terms of their goals but also some differences. In particular, ADO batch updates are a pre-requisite for disconnected recordsets, single tier applications and Remote Data Services (ADO's rough equivalent to MIDAS). ADOExpress (the ADO component suite in Delphi 5) attempts to make the logical and syntactical transition from cached updates to batch updates as seamless and painless as possible but, as you can guess from previous articles on ADO, there is some re-learning to do. This article looks at what you need to know to use ADO batch updates.

Getting Started

At first sight the syntactical conversion isn't too dramatic. Table 1 shows the `TADODataSet` batch update properties and methods and their `TBDEDataSet` counterparts.

➤ *Table 1*

TADODataSet	Description	TBDEDataSet
Supports([coUpdateBatch])	Are batch updates supported ?	N/A
LockType := ltBatchOptimistic	Enable cached updates	CachedUpdates := True
fgPendingRecords, RecordCount > 0	Determine if updates are pending	UpdatesPending
FilterGroup	Filter specific kinds of updates	UpdateRecordTypes
UpdateBatch	Update a batch	ApplyUpdates
CancelUpdates, CancelBatch	Cancel a batch	CancelUpdates
N/A	Delete Updates Delta	CommitUpdates
CancelBatch(arCurrent)	Undo changes to current record	RevertRecord
UpdateStatus, RecordStatus	Update status of the current record	UpdateStatus
N/A	Specifies how records are found for updating	UpdateMode

The essential approach is quite straightforward: enable a recordset for batch updates, make changes to records (updates, inserts and deletes) and apply the changes (using `TADODataSet.UpdateBatch`).

There are a few differences worth mentioning. ADO allows you to determine if the dataset supports batch updates using:

```
TADODataSet.Supports(
  coUpdateBatch)
```

Batch updates are enabled by setting the `LockType` (see Issue 52) to `ltBatchOptimistic` before opening the record set. Hereafter changes are made to a local memory-based cache (just as they are with the BDE's cached updates). This step also affects the locking mechanism. As records are changed on the client side, changes are made in memory, thus no record locking takes place. When the batch is applied an optimistic locking scheme is used, ie records are only successfully updated if they have not been changed by another user (more about this later).

There are two ways to cancel batch updates: `CancelUpdates` and `CancelBatch`. The former is a direct equivalent to `TBDEDataSet.CancelUpdates` and the latter is the ADO equivalent, which accepts an optional parameter indicating which kind of updates should be cancelled.

`TADODataSet` has no equivalent to `TBDEDataSet.CommitUpdates` because the updating of the batch always deletes the update delta.

The `TADODataSet.UpdateStatus` property can be used in exactly the same way as its BDE counterpart. Listing 1 shows a `TDBGrid`'s `OnDrawColumnCell` method being used to colour modified rows

➤ *Listing 1*

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const Rect: TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
begin
  case ADOTable1.UpdateStatus of
    usModified : DBGrid1.Canvas.Brush.Color:=clBlue;
    usDeleted  : DBGrid1.Canvas.Brush.Color:=clRed;
    usInserted : DBGrid1.Canvas.Brush.Color:=clGreen;
  end;
  DBGrid1.DefaultDrawDataCell(Rect, Column.Field, State);
end;
```

either blue, red or green depending on whether they have been modified, deleted or inserted. TBDEDataSet.UpdateRecordTypes is used to filter a result set to show unapplied updates of certain kinds. For example, to show unapplied deleted and modified records you could use the following line:

```
Table1.UpdateRecordTypes:=
  [rtDeleted, rtModified];
```

TADODataSet's nearest equivalent is FilterGroup. However, TADODataSet.Filter Group is not a set of enumerated values but a single value. Table 2 shows the possible TFilterGroup values.

There is certainly a high degree of similarity between UpdateRecordTypes and FilterGroup, but each is capable of feats which the other is not.

Update Conflicts

So far so good, but I have always found a healthy dose of paranoia to be an excellent programming quality and I always want to know what happens when things don't work the way they should. For example, what happens when two updates conflict? Well, the answer depends on the nature of the conflict.

Assume that two users want to change the same record. One user wants to change the CONTACTNAME field and another user wants to change the CONTACTPHONE field. In this example, both updates will succeed. ADO locates the original record by searching for a record which matches the original primary key and the original value of the changed field(s). This is the same technique as the BDE uses when TBDEDataSet.UpdateMode is set to upWhereChanged. The difference is that ADO has no facility to allow the programmer to change this setting, so you are stuck with upWhereChanged.

Now assume that two users want to change the same field of the same record at the same time. When the second user updates their batch they will get the error *'The specified row could not be located for updating. Some values may have been changed since it was last read'*. This is understandable and the user is left with an unapplied update.

However, this is not the only error which the user could receive. The same problem can result in an *'Errors occurred'* error. We need to take a closer look into how ADO applies the batch update. Assume that a user makes changes to several records. Also assume that another user successfully updates one of these records before the first user applies their update. Now when the first user applies their update all of the changes which do not conflict will be successfully applied. Only the conflicting records will not be applied.

This behaviour is different to that of the BDE. Typically, BDE cached updates are applied from a TDatabase object. As a result, the updates are encased within a transaction. If any single update fails then all updates are rolled back. This is not the case in ADO batch updates, as TADOConnection has no UpdateBatch method. However, the code to enclose the updates in a transaction is very simple (Listing 2).

fgUnassigned	Specifies that no filtering is in effect
fgNone	Removes any current filtering and all rows are visible
fgPendingRecords	Shows just the rows that have been changed and not applied
fgAffectedRecords	Shows just the rows affected by the last update
fgFetchedRecords	Shows just the rows in the current update cache
fgPredicate	Shows just deleted rows
fgConflictingRecords	Shows just the rows that could not be applied due to errors on the apply attempt

➤ *Table 2*

Now to get back to the distinction between the two error messages which I talked about a moment ago. The difference is caused by the UpdateBatch method continuing to apply the remaining updates even after one or more updates have failed. If all updates succeed except for the last one, the reason for failure is reported accurately (ie you receive the error *'The specified row could not be located for updating. Some values may have been changed since it was last read.'*). If an error occurs and it is not the last record updated or more than one error occurs then ADO simply reports *'Errors occurred'*.

You can view the sequence of events using the TCustomADODataSet.OnRecordChangeComplete event. This receives a number of parameters, including the reason for the event (Reason: TEventReason) and the status of the event (EventStatus: TEventStatus).

When a record is first changed on the client side an OnRecordChangeComplete event is generated with a reason of erFirstChange, indicating that this is the first time the record has been changed since it was read from the database. In addition, the event status is esOk, which indicates that the record was changed successfully (on the client side). When the batch is finally updated (using

➤ *Listing 2*

```
ADOConnection1.BeginTrans;
try
  ADOTable1.UpdateBatch;
  ADOConnection1.CommitTrans;
except
  ADOConnection1.RollbackTrans;
  ShowMessage('Errors occurred.'+#13+'Complete batch rolled back');
end;
```

method UpdateBatch) an OnRecordChangeComplete event is generated again for each of the updates. For each update the reason parameter is erUpdate, indicating that an existing row was being modified. When the update fails (because another user changed the data first) the event status is esErrorsOccurred, indicating that the update was not successfully applied. What is very useful about this event is that it enables you to see that even after one update has failed ADO continues to keep applying subsequent updates.

Resolving Conflicts

The standard approach to resolving batch update conflicts in the world of ADO is to update the batch as normal and then, after the complete batch has been updated and one or more records in the batch have failed to update, review the failed records. To filter the recordset to show only those records which failed to be applied use the following code:

```
ADOTable1.FilterGroup :=
  fgConflictingRecords;
ADOTable1.Filtered := True;
```

then use the NewValue, CurValue and OldValue properties of Delphi's TField class to allow the user to review the differences and decide upon a solution. Table 3 shows your available options.

You can see from this table the ADO equivalents to the TField class properties. One of the problems which Delphi programmers using ADO must overcome is that Borland, quite deliberately, only documents ADOExpress and not ADO (or MDAC, OLE DB, ODBC etc). Although this can be frustrating for the Delphi programmer trying to learn ADO, it is the only realistic approach that Borland can take. It really doesn't make sense to document something all over again which is already documented (although it would have been helpful to include the MDAC SDK on the Delphi CD). Thus the reason for including the ADO Field class equivalents to the Delphi TField class is that all of the documentation on ADO talks in ADO terms and not Delphi terms (quite reasonable, really).

ADO Field Class	Description	Delphi TField Class
Value	The client's current value	NewValue
UnderlyingValue	The new value from the database	CurValue
OriginalValue	The value when first read from the database	OldValue

➤ *Table 3*

The Delphi TField properties do, of course, retrieve their values from the ADO Field class properties, so there really isn't much difference. However, you should be aware that all ADOExpress features are represented using the familiar Delphi TDataSet architecture. What this means is that TDataSet descendants can buffer data. If you bypass the ADOExpress class and go directly to the ADO class (eg to get the UnderlyingValue or OriginalValue of ADO's Field class) you will get the values of the last record read. This might not be the same as the ADOExpress class's current record. You can either avoid this problem altogether by not bypassing the ADOExpress class and going directly to the underlying ADO class (more preferable) or by resynchronizing the ADO class with the Delphi class using TDataSet.UpdateCursorPos (which is less preferable).

However, as with all things in ADO, the specification simply states how any particular feature is supposed to work, but the OLE DB Provider in question may or may not actually implement the said feature. This is the case for the UnderlyingValue/CurValue property. The Jet 4.0 OLE DB Provider (Access 97 and Access 2000) returns the OriginalValue/OldValue for UnderlyingValue/CurValue instead of the value from the database as set by another user (the SQL Server OLE DB Provider returns the correct value for UnderlyingValue/OldValue).

If you are familiar with TClientDataSet you may be wondering about whether you can make use of TReconcileErrorForm or if there is any ADO equivalent. To recap: TReconcileErrorForm is a very convenient dialog supplied with Delphi which is intended for resolving update conflicts when using TClientDataSet.ApplyUpdates. TReconcileErrorForm uses DBClient (and therefore MIDAS) and DBTables, but with a little amendment of the uses clause and a small amount of copying (TReconcileAction, EDBClient and EReconcileError) from DBClient the error reconciliation form can be made to compile without TClientDataSet. However, this is a non-starter, as TCustomADODataSet does not have the required OnReconcileError event where TReconcileErrorForm is used. Unfortunately, this is a show stopper, because the only feedback the programmer receives during the process of updating the batch is an OnRecordChangeComplete event and the OnRecordChangeComplete event has no way of reconciling the error. In short, ADO was not designed to resolve this problem this way and, at least for the moment, it appears the problem cannot be resolved this way.

Master/Detail Relationships

Master/detail or parent/child relationships caused their fair share of problems in the BDE world and, as batch updates are conceptually the same as cached updates, this problem, like many others, remains the same. In the world of cached updates the question was always about in which order the updates should be applied. The order could be specified by the order in which datasets were passed to the TDatabase.ApplyUpdates method and usually this meant that updates were applied

to the parent first and then to the child. When TClientDataSet replaced cached updates as the recommended solution to the problem, the approach to solving the problem changed. Instead the parent table on the client side contained the complete set of parent records *plus* the complete set of child records. The TClientDataSet became a conduit through which all updates on the client side passed.

There are two ADO solutions to the master/detail problem (though both solutions are very similar) which have similarities with both cached updates and TClientDataSet. The first solution is the most straightforward and is the same as for the BDE's cached updates: start a transaction, apply updates to each table individually and commit the transaction. Listing 3 shows the code to achieve this.

The second solution is to use the Microsoft Data Shape Provider (supplied with ADO 2.0 upwards). This approach has much more in common with TClientDataSet. The Data Shape Provider uses a set of extensions to SQL to 'shape' data. Listing 4 shows an example of the data shape language to return a Customer/Orders master/detail relationship.

This SQL can be used with a TADODataSet or a TADOQuery. To get at the child data you'll need to add persistent fields to the dataset. Then add another TADODataSet and set its DataSetField property to the TDataSetField added to the first data set (eg ADODataSet1ORDERS).

All appears to be well and the problem happily solved except for the small detail that the process of updating the detail dataset isn't automatic, as it is when using TClientDataSet. Instead you have to update the detail's batch as well as the parent's batch. As a result, the code to update both the master and the detail is the same using this second solution as it was for the first solution (see Listing 3 again). Although the Data Shape Provider is certainly a very useful tool, it doesn't solve many problems in the world of batch updates.

➤ *Listing 3*

```
ADOConnection1.BeginTrans;
try
  ADODataSet1.UpdateBatch;
  ADODataSet2.UpdateBatch;
  ADOConnection1.CommitTrans;
except
  ADOConnection1.RollbackTrans;
end;
```

Auto-Incremented Keys

As I have mentioned, many of the problems which BDE cached updates face reappear with ADO batch updates. Auto-incremented keys, such as Access AutoNumber and SQL Server Identity fields, are another example. The problem here is one of timing. When using batch updates the underlying database is not updated until the BatchUpdate method is called. It is at this point that the database gets a chance to generate the next number in the sequence for a newly inserted record. The problem is that this may be too late for the client application if it needs this information when the record is added on the client (instead of updated on the server). ADO batch updates suffer the same fate as the BDE's cached updates, in that your application simply has to make allowances for this behaviour.

However, ADO does make some effort to help in retrieving the newly auto-incremented values from the database after the batch has been updated. You may recall that ADO has the concept of 'dynamic properties' which allow you to interrogate an object's features. One such dynamic property is Update resync which determines how the recordset will be updated by the database after a batch update. The all important value which you are searching for here is adResyncAutoIncrement (the default) which specifies that auto-increment fields of the recordset will be updated from the database for newly added records. You can check this setting using:

```
if ADOTable1.Properties[
  'Update Resync'] =
  adResyncAutoIncrement then
  Caption:='Resync '+
  'Auto-Incremented fields';
```

Of course, if you're ahead of the game, by now you should be guessing that this is simply how it is supposed to work and not necessarily that it *will* always work this way with all providers. In fact, it is really quite awkward trying to work out whether this feature will work or not, because it is dependent on the combination of OLE DB Provider, database engine, database format version number, whether the data source is indexed, whether the cursor location is client-side or server-side, the cursor type and the lock type. For example, the Jet 4.0 OLE DB Provider will not automatically resync auto-incremented fields when used with Access 97 and a client-side cursor, but it will (in nearly all cases) when used with Access 2000. If you use Access 97 and auto-incremented fields then after you have added a new record with an AutoNumber field, updated the batch and refreshed the data, you will receive the error *'The value for this row has been changed or deleted at the data source. The local row is now deleted'*. This is because the server has changed the primary key but the client is unaware of the change. The client can no longer find the local record which it has on the server.

Updating Read-Only Result Sets

One of the benefits touted for BDE cached updates is the ability to update result sets which are read only by nature. Typically this means a JOIN between two tables but cached updates are not limited just to updating simple JOINs. The process involves making use of a TUpdateSQL component and the judicious clicking of a couple of buttons to generate the necessary

➤ *Listing 4*

```
SHAPE {SELECT * FROM CUSTOMERS}
APPEND ({SELECT * FROM ORDERS}
RELATE CustomerID TO CustomerID) AS ORDERS
```

SQL to resolve the ambiguity of updating a result set which refers to more than one table. In terms of solving the problem, cached updates score 10 out of 10. However, the mechanism it uses is not wholly intuitive and it is rather fragile in the face of changes to the database structure, as the SQL needs to be regenerated each time the database structure changes.

ADO's solution to this problem is simple: there is no problem. ADO joins are naturally updateable (both sides of the join are updateable) without the need to resort to any other components or setting any properties at all. In terms of solving the problem it gets a 7 out of 10, and in terms of the solution's elegance it gets a 10 out of 10. I only gave it a 7 for solving the problem for two reasons. Firstly, although updating the result set is simple, inserting and deleting rows in the result set is more problematical. Secondly, it only gets a 7 because having manual control over the updating process gives you the potential to handle other, less common, situations which ADO batch updates cannot handle. For example, take an SQL statement which contains a `GROUP BY` clause. This is a read-only result set, because each logical row refers to many physical rows in a table or view somewhere. With cached updates you could say that an update to a logical row should be applied to all of the corresponding physical rows.

Disconnected Recordsets

One of the main goals behind ADO's batch updates is to allow programmers to disconnect recordsets. This leads to other features, like recordset persistence (ie the briefcase model) and N-tier applications using RDS. A disconnected recordset is one which has been disconnected from the database and is operating independently from the database. The most important point here is that the connection to the database has been dropped. This allows the client to get on with the job of communicating with the user and letting the user modify, insert and delete data without tying up a connection to the database. ADO excels in this subject, as the ADO `RecordSet` class was designed with this goal in mind. Disconnecting from the database is simple and can be seen in Listing 5, reconnecting is shown in Listing 6.

ADO's solution is exceptionally elegant because the recordset continues to function in the same manner when disconnected as when connected. There is no comparison with BDE cached updates because the connection cannot be dropped but even if you contrast this with `TClientDataSet`, which does allow the connection to be dropped, ADO is still a more elegant solution because you are not forced to change dataset components (ie from `TTable/TQuery` to `TClientDataSet`).

Conclusion

ADO's implementation of batch updates is well thought out and offers similar functionality to that of the BDE's cached updates and also of `TClientDataSet`. In addition, it can be seen as a pre-requisite for other ADO features such as disconnected recordsets, recordset persistence and RDS. However, as always with ADO you should take great care to determine that the OLE DB Provider you are using supports the features you want to use. I'll be back soon with more help for your ADO projects; meanwhile, have fun!

```
// Disconnect recordset
// from the Connection
ADOTable1.Connection := nil;
// Drop connection to database
ADOConnection1.Connected := False;
```

➤ *Listing 5*

```
// Connect to the database
ADOConnection1.Connected:=True;
// Connect recordset to Connection
ADOTable1.Connection :=
  ADOConnection1;
```

➤ *Listing 6*

Guy Smith-Ferrier is Technical Director of Enterprise Logistics Ltd (www.EnterpriseL.com), a training company specialising in Delphi which is now running ADO courses. He can be contacted at gsmithferrier@EnterpriseL.com