ADO Transaction Processing

by Guy Smith-Ferrier

In my previous article on ADO (Issue 49, September 1999) I gave an overview of the components and features of the ActiveX Data Objects support which was added to Delphi 5. In this article I want to examine one specific feature: ADO transaction processing. Most database programmers from almost any background will be familiar with the essential idea and mechanics of transaction processing but with the switch of middleware from the BDE to ADO the deck is slightly reshuffled. This article is for BDE programmers learning ADO.

Getting Started

I mentioned in the last article that Delphi 5's `TADOConnection` component is analogous to the BDE's `TDatabase` component. However, I also mentioned that the designers of the ADO components chose to maintain compatibility with ADO rather than with Delphi's BDE components, so the names of properties and methods of `TADOConnection` are not compatible with those of `TDatabase`. With that difference in mind, the translation is simple. Table 1 shows `TADOConnection`'s transaction processing properties and methods and their BDE counterparts.

The methods are used in the same way as for `TDatabase`. Listing 1 shows a `TADOConnection` component being used to ensure that both sides of a transfer of money from one account to another account either succeed or both sides fail.

Determining Support

However, this is where the differences start. ADO takes a different attitude to databases than the BDE. Both pieces of middleware offer portability from one database engine to another (hence the BDE is based on IDAPI, which stands for *Independent Database API*). As a result, portability is only achievable by programming down to the lowest common denominator. Sometimes the lowest common denominator is so feature-light that this is a 'challenge'. However, the BDE attempts to offer better portability by raising the level of the lowest common denominator to make it more palatable and to make portability easier. ADO does not take this approach. Instead it offers the features that the underlying database offers and does not add back missing features.

The immediate consequence of this is that when using ADO with the Paradox ODBC driver any transaction processing method will give an exception: '*The operation requested by the application is not supported by the provider*'. So it is up to the programmer to first determine the transaction processing facilities of the OLE DB Provider and, if relevant, the ODBC driver in use. You can do this using the `Transaction DDL` dynamic property shown in Listing 2.

Table 2 shows the ADO constants which determine the kind of transaction processing supported by the provider. The difference between the constants simply determines how DDL (Data Definition Language: `CREATE`, `ALTER`, `DROP`) used within a transaction is treated.

Nested Transactions

ADO supports nested transactions. This is a simple concept of allowing one transaction to be 'nested' inside another. The inner transaction can be committed or rolled back independently of the

➤ *Listing 1*

```
ADOConnection1.BeginTrans;
try
  ADOCompanyAccounts.Edit;
  ADOCompanyAccounts.FieldByName('Balance').AsFloat:=
  ADOCompanyAccounts.FieldByName('Balance').AsFloat + 100;
  ADOCompanyAccounts.Post;
  ADOCustomerAccounts.Edit;
  ADOCustomerAccounts.FieldByName('Balance').AsFloat:=
  ADOCustomerAccounts.FieldByName('Balance').AsFloat - 100;
  ADOCustomerAccounts.Post;
  ADOConnection1.CommitTrans;
except
  ADOConnection1.RollbackTrans;
end;
```

➤ *Table 1*

ADO	Description	BDE Equivalent
TADOConnection.BeginTrans	Begins a transaction	TDatabase.StartTransaction
TADOConnection.CommitTrans	Commits a transaction	TDatabase.Commit
TADOConnection.RollbackTrans	Rolls back a transaction	TDatabase.Rollback
TADOConnection.IsolationLevel	Transaction isolation level	TDatabase.TransIsolation
TADOConnection.Properties['Transaction DDL']	Retrieves transaction support level	N/A
TADOConnection.InTransaction	Is a transaction in progress ?	TDatabase.InTransaction
TADOConnection.Attributes	Should a transaction be started after a commit or rollback ?	N/A

outer transaction. Providers which support nested transactions typically place no limit on the number of levels of nesting, or place a limit which is beyond all practical need. Of course, this brings up the point that whereas ADO supports the concept of nested transactions, not all providers support it. For example, ODBC does not support nested transactions. SQL Server 7 does support nested transactions but SQL Server 6.5 only supports what is sometimes referred to as 'fake nesting'. Fake nesting is achieved by allowing the programmer to create nested transactions but the data is not actually committed until the outermost transaction is committed. Rather surprisingly I have not found any way of determining whether nested transactions are supported other than by trying a nested transaction in a `try..except` block and observing whether it fails or not.

One of `TADOConnection`'s properties which has a bearing on nested transactions is called `Attributes`. This is a set of `TXactAttributes` enumerated types. `TXactAttributes` has only two possible values: `xaCommitRetaining` and `xaAbortRetaining`. The first of these instructs ADO to start a new transaction as soon as the old transaction has been completed, and `xaAbortRetaining` instructs ADO to start a new transaction as soon as the old transaction has been rolled back. By default the set is empty but it can easily be set programmatically:

```
ADOConnection1.Attributes:=
  [xaCommitRetaining,
  xaAbortRetaining];
```

Incidentally, remember that I mentioned the ADO datasets where ADO-like instead of BDE-like? Well, when it comes to a choice between being ADO-like instead of Delphi-like then the datasets are Delphi-like. The `Attributes` property is a good example. Delphi uses enumerated types and sets of enumerated types and these make reading and writing this code very easy. If we had simply used the ADO type library directly as I showed in my article on using ADO in Delphi 4 (Issue 46) then the code would have involved using bitwise operators with the ADO constants:

```
Connection.Attributes :=
  adXactCommitRetaining or
  adXactAbortRetaining;
```

This solution is less appealing. But back to nested transactions. The reason why the `Attributes` property has a bearing on nested transactions is because if `Attributes` includes either `xaCommitRetaining` or `xaAbortRetaining` and a nested (or 'inner') transaction is started then the outermost transaction can never be committed or rolled back. The reason for this is obvious when you think it through in steps.

First, the outermost transaction starts. Then a nested (or 'inner') transaction starts. The inner transaction is either committed or rolled back. Then a new transaction is automatically started because of the setting in the `Attributes` property.

As a result it is not possible to get back to the outermost transaction because a new transaction is always being started.

Cursor Locations, Cursor Types And Lock Types

Cursor locations, cursor types and lock types are all interwoven properties. The setting of one directly affects the setting of the others. As such all three must be covered together.

Transaction DDL Constant	Value	Description
DBPROPVAL_TC_NONE	0	Transactions are not supported
DBPROPVAL_TC_DML	1	Transactions can contain DML. DDL causes an exception
DBPROPVAL_TC_DDL_COMMIT	2	Transactions can contain DML. DDL causes transactions to commit
DBPROPVAL_TC_DDL_IGNORE	4	Transactions can contain DML. DDL is ignored
DBPROPVAL_TC_ALL	8	Transactions can contain DML and DDL

➤ *Table 2*

```
if ADOConnection1.Properties['Transaction DDL'].Value > DBPROPVAL_TC_NONE then
  Caption:='Transaction processing is supported';
```

➤ *Listing 2*

`Connections` and `Recordsets` both have a `CursorLocation` property which determines whether the cursor is a 'client-side' cursor (`clUseClient`) or a 'server-side' cursor (`clUseServer`). Client-side cursors are managed by the Microsoft Client Cursor Library and offer a level of flexibility and features similar to those of `TClientDataSet`. For example, client-side cursors can be sorted and re-sorted without re-querying the data, and client-side cursors are always bi-directional. Server-side cursors are managed by the OLE DB provider and are typically an extension of the cursor handle of the underlying database. Server-side cursors can offer better performance and are necessary for large result sets (where the client machine has insufficient disk space to cache the complete result set) but not all providers offer bi-directional cursors.

Why am I mentioning all of this? Well, the cursor location has a direct bearing on the kinds of cursor types which a recordset can use. Table 3 shows the cursor types supported by ADO.

As you can see from the table, the cursor type affects whether the result set is bi-directional, whether records added by other users are visible, and whether

	ADO Constant Value	Forwards / Backwards	Records Added By Others Are Visible	Records Deleted By Others
ctUnspecified		N/A	N/A	N/A
ctOpenForwardOnly	0	Forwards only	?	?
ctKeyset	1	Both	No	Inaccessible
ctDynamic	2	Both	Yes	Yes
ctStatic	3	Both	No	No

➤ *Table 3*

records deleted by other users are deleted in your own record set. For 'keyset' cursors (ie, `CursorType := ctKeySet`) you can specify whether your own result set should see your own inserts and deletions using the `Remove Deleted Records` and `Own Inserts Visible` dynamic properties.

The `CursorLocation` property has a bearing on `CursorType` because the OLE DB provider in use will automatically change the cursor type from one which it doesn't support to one which it does. For all OLE DB providers if you specify a client-side cursor then the cursor type will always be changed to `ctStatic`. For server-side cursors the amending of the cursor type varies from OLE DB provider to OLE DB provider. For example, the SQL Server OLE DB provider often changes keyset and static cursors to dynamic, the Jet OLE DB provider changes most cursors to keyset, the Oracle OLE DB provider changes all cursors to forward only, the ODBC OLE DB provider changes cursors depending on the ODBC driver in use.

The locking scheme employed by the BDE is typically determined by the driver and also depends on whether cached updates are being used. For example, when using the Paradox (STANDARD) driver the BDE employs pessimistic locking (ie records are locked as editing begins). Using cached updates modifies the scheme slightly by not releasing the record lock after the edit is completed. When using the InterBase driver (or any SQL Links driver) the BDE employs optimistic locking (ie a record lock is placed only when the update is finally attempted). In ADO the programmer has some choice of which locking scheme is used. Table 4 shows the lock types supported by ADO.

ltUnspecified	The lock type has not been specified yet
ltReadOnly	Read-only
ltPessimistic	Pessimistic locking
ltOptimistic	Optimistic locking
ltBatchOptimistic	Used for batch updates

➤ *Table 4*

Essentially the choices are read-only (in which case there is no locking scheme), pessimistic, optimistic and batch optimistic. The last is used for batch updates, which are to ADO what cached updates are to the BDE (I hope to cover batch updates in a future article). Just as the `CursorLocation` property had a bearing on the `CursorType` property, so it also has a bearing on the lock type. Client-side cursors only support read-only and batch optimistic lock types. If you specify a client-side cursor and specify a lock type other than one of these then the lock type will be changed to batch optimistic.

Transaction Isolation Levels

As the name implies, a transaction isolation level specifies how isolated a transaction is from changes made by other users and *vice versa*. The three transaction isolation levels supported by the BDE are well known by BDE database programmers who use the SQL Links drivers. However, it is also well known that these transaction isolation levels are a simplification of the choices typically offered by the underlying database engines (as InterBase developers often point out).

ADO is forced to make similar generalisations but it is slightly less restrictive than the BDE. Table 5 shows the transaction isolation levels which are supported by `TADOConnection`.

ADO often uses two constants to mean the same thing and, as a result, the Delphi enumerated type also contains multiple names to specify the same transaction isolation level. The table also shows the equivalent BDE transaction isolation level. The default for ADO is the same as for the BDE: `ilCursorStability`. Just as for SQL Links drivers for the BDE, OLE DB providers can force a higher transaction isolation level if the requested level is not supported.

Assuming that an OLE DB provider offers all transaction isolation levels, your choice of which level to use is determined by the transaction anomalies which you want to prevent and the price you are prepared to pay to prevent them. Transaction anomalies can be categorised in three ways. First, *dirty reads* occur when a transaction reads data that has not yet been committed. Second, *non-repeatable reads* occur when a transaction reads the same row twice and the data is different each time. Third, *phantoms* are rows that match the criteria used to compose a result set, but are not initially visible.

Table 6 shows the transaction isolation levels and the anomalies which occur when they are used.

Delphi TIsolationLevel	Equivalent ADO Constant	ADO Constant Value	Equivalent TDatabase.TransIsolation	Description
ilUnspecified	adXactUnspecified	-1		Server is using an isolation level other than what was requested and the specific isolation level cannot be determined
ilChaos	adXactChaos	16		Changes from more highly isolated transactions cannot be overwritten by the current connection
ilReadUncommitted	adXactReadUncommitted	256	tiDirtyRead	Uncommitted changes in other transactions are visible
ilBrowse	adXactBrowse	256	tiDirtyRead	Exactly the same as ilReadUncommitted
ilCursorStability	adXactCursorStability	4096	tiReadCommitted	Changes from other transactions only visible after being committed (Default)
ilReadCommitted	adXactReadCommitted	4096	tiReadCommitted	Exactly the same as ilCursorStability
ilRepeatableRead	adXactRepeatableRead	65536	tiRepeatableRead	Changes made in other transactions not visible, but requerying can retrieve new recordsets
ilSerializable	adXactSerializable	1048576		Transactions conducted in isolation from other transactions
ilIsolated	adXactIsolated	1048576		Exactly the same as ilSerializable

➤ *Table 5*

Isolation Level	Dirty Reads	Non-repeatable Reads	Phantoms
ilReadUncommitted	Yes	Yes	Yes
ilReadCommitted	No	Yes	Yes
ilRepeatableRead	No	No	Yes
ilSerializable	No	No	No

➤ *Table 6*

The `ilSerializable` transaction isolation level is likely to be unfamiliar to many BDE programmers so I will explain it here.

First, let's recap on what `ilRepeatableRead` does. When a transaction starts it places read locks on all rows that it reads. For example, `SELECT * FROM CUSTOMER` would place a read lock on every row in the table. As `INSERT`s, `UPDATE`s and `DELETE`s are performed it holds write locks on the records affected. Other transactions are unable to modify any of the records modified by the first transaction because the modified records are still write locked. As a result the first transaction avoids non-repeatable reads because no other transactions can modify the data which it has modified. The read locks and write locks are released when the transaction commits or rolls back.

The `ilSerializable` isolation level does everything that `ilRepeatableRead` does and, in addition, it does not allow any other transactions to insert records that would be included in the result set, nor does it allow other transactions to modify any records such that they either move into or out of the original result set. Thus, if the original result set was constructed from `SELECT * FROM CUSTOMER` then no one can add any records to the table. If the original result set was constructed from `SELECT * FROM CUSTOMER WHERE AREACODE=6` then records with `AREACODE=6` cannot be inserted and no existing record can be modified such that it is either changed to `AREACODE=6` when it wasn't before or that it was `AREACODE=6` before and then the `AREACODE` was changed to be something other than `6`. As a result it avoids 'phantom' records.

Conclusion

The implementation of transaction processing in ADO (and its support in Delphi) is easily recognisable by programmers already familiar with transaction processing in the BDE. It is fair to say that there are probably as many similarities between the two pieces of middleware as there are differences. Also, the subject matter is as much like a 'knowledge onion' in ADO as it is in the BDE: the problem may appear simple at first but, as each layer is stripped away, so another more complex layer to be learnt is then revealed. I hope I have helped you avoid some of the tears involved in working with this particular onion!

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