specifying a different collation within the schema. CP437_BIN turns out to be the appropriate one. (This only *needs* to be done for fields that are used within keys).

Additionally the Practice Suite has a proprietary mechanism (known as TLD) for packing variable data into BTRIEVE records.

Data visibility and vulnerability

Another aspect of SQL databases is their visibility. Any suitably qualified person at a customer site could, in principle, log in to the database and manipulate it (which was clearly not true when the system used BTRIEVE). The general principle, so far as SQL databases are concerned, is that this is just a fact of life and is accepted. Apart from the inconvenience of diagnosing the problem, if a user tampers with his own data then the consequences can only rebound on him.

There are, however, some pieces of information that we very much do *not* want him to tamper with. A good example is the license expiry date of his products, or the number of users he is licensed for. Information that comes under this heading is generally stored as binary or image on the database, and the emulation layer generates/decodes this data by applying encryption to it.

BASIC EMULATION PHILOSOPHY

Mapping of files to tables

The 'order processing' example in the previous section indicates that, even with an ideal stable BTRIEVE file design there is every chance that a single file would not map to a single table in the corresponding SQL database.

However, making such a 'correct' mapping poses severe problems for an emulation strategy. Suppose, for example, that there was some sort of summary report, in the order processing example, that read all orders in a date range, counted the orders by customer type and added up the value of order items into different categories (again by customer type), depending on what was ordered.

Whilst I can write one SQL statement to deliver this information, in the BTRIEVE case I have to write a program, and that program will read the orders file sequentially by a series of 'get greater than' calls, starting from the start date. The logic of the code expects to see an order header, followed by one or more order lines, followed by the next header. It will keep cycling round this pattern until it gets to a date that is later than the end of its range. If the applications code is not to change then the SQL database must support precisely the same access transparently.

There are three potential ways of attacking this:

- The SQL database can be made 'correct' and the relevant code to do the translation can be buried inside the emulator. (This may involve the addition of special 'legacy' tables to replicate the original BTRIEVE key sequence.)
- The same philosophy as above can be adopted, but the original file can be represented as a view across the new tables (with suitable 'instead of' triggers to support insert, update and deletion access paths).
- The SQL database can be mapped one table to one file.

The first two approaches require some domain knowledge in order to do the remodelling of the data. They are also inherently more complex than the third, and may have performance implications (which can only be evaluated, or improved, with some knowledge of how the files are used by the application).

The advantage of approaches based on remodelling is that they give a path forwards to allow some degree of rewriting of parts of the application in the event of performance problems. This advantage becomes more significant if the access paths across the data are complex, and the data itself is sufficiently complex so as to make a simple mapping to a navigable SQL database difficult.

In the end, constraints of resources, timescales and lack of domain knowledge dictated that the core emulation mapping is one file to one table. A small part of the Practice

database (Time and Fees) used the first method, but all the rest (and all of Exchequer) uses the simple 1:1 mapping.

Multiple record types.

A 1:1 file to table mapping works quite well for 'simple' files with only one sort of data in them, and a well defined set of fields that go to make it up. (Such files look like relational tables already.) However, for multi record type tables things are less straightforward.

Essentially every single field that can be present in *any* record type is represented as a separate column in the table. Then, for each row in the table, those columns that are not relevant to that row are set to null.

This philosophy works reasonably well, but does tend to produce tables with large numbers of columns. In one extreme case within Exchequer it has failed because the number of columns required exceeded the maximum allowable for a table. (The solution in that case was to represent some of the groups of fields in the original BTRIEVE file as single 'binary' columns.)

Xml definition of file mappings

One of the easier things to implement within the emulation is the mapping of BTRIEVE fields to SQL columns. The code basically needs to know:

- How to work out which 'record type' a given row represents.
- Which fields are relevant to which record type.
- What format each field is, and the name and format of the column it maps to on the database.

This information is held in XML documents, one per file. The documents are maintained manually. (It proved impossible to do this automatically since there was no universal and reliable convention for the way in which the BTRIEVE file layouts were represented.)

Originally these documents were issued to the user site in a specific folder and accessed directly by the code when the emulation software initialised. This, however, introduced a significant delay in initialisation, so the XML mappings are now stored on the database itself. The pattern is thus:

- Xml mapping documents are maintained manually as part of the source of the system.
- The raw Xml files are installed into a standard place on the user's system when the system is installed.
- The installation process for the database reads these raw Xml files and copies them onto the database.
- When the live system runs it initialises its emulation control data from the database.

Xml mappings on the database are stored as binary image data. They are not user visible or amendable, and the only way that they can be updated is via the standard mechanisms.

There are additional Xml configuration files used by the emulation, specifically there is one containing the list of which files are emulated.

Caching, why and how.

The 'order processing' example given previously showed that the number of calls made on BTRIEVE to retrieve data about an entity will typically be greater than the number of calls made to a pure SQL system to get the same data. This would lead us to believe that a simple minded 1:1 emulation would have severe performance problems. (This is also borne out by practical experience.)

SQL is good at getting sets of data. The problem we have in general is that when a BTRIEVE call is made there is no indication of a context to indicate what required set of data it is part of. The only thing that is known is:

- The mode of the operation (get greater, get equal etc).
- The key that is being used.

This performance effect tends to be enhanced by virtue of the fact that the BTRIEVE file manager contains its own caching. Thus re-reading a record that was read a few instructions previously can be very efficient in the BTRIEVE case.

It is an almost self evident truth that programmers can write code that is less than optimal. It is equally true that, if the code performs accurately and the performance overhead of the inefficiency is too small to be of practical importance the code will not be changed. Thus there can equally be BTRIEVE access paths that are written

inefficiently, but are being mitigated by the internal processing of the BTRIEVE file manager itself.

The emulation strategy for improving performance is to keep a cache of records (i.e. rows) in-store, and satisfy requests for data from this, if possible. There are two types of cache:

- Records that were read as a result of a 'get equal' and may be re-read again within a short period of time.
- Records that the emulator thinks might be needed soon because the code made a sequential request (e.g. a 'get greater than').

Because of the level at which it is working the raw emulator has absolutely no way of knowing what is coming next. Hence it adopts a blind strategy for this second case. If the request is sequential it gets the next 100 rows that satisfy the key query, in the order as determined by the particular BTRIEVE key being used.

This pre-fetch mechanism will produce savings provided that the next BTRIEVE sequential access is on the same key path and starts from a key that has already been read. If this is not the case then there will be no gain and a small performance hit because of the extra volume of data transmitted back from SQL server to the emulator. The performance hit from getting 99 rows that will not be needed within SQL server itself will tend to be small, since SQL server is optimised for this sort of access.

It should be emphasised that this approach is based on hope, rather than knowledge. This is not a bad thing, if your hope is realised 50% of the time then you've just reduced the more severe problems that you have to solve by 50%.

In practice, the path of reading through a series of records in key sequence is not uncommon. In the order processing example given previously the first read to get the order header may well be a get equals, but the subsequent series of get greater than calls to read the order items will typically pre-fetch all the items in one hit. The one thing that it *can't* do is join the items (or, for that matter the header) to data in other tables that will always be required.

There will be no gain from caching if the access is 'scatter gun'. An example in the Practice Suite is when reading the main identifier file sequentially and, for each record, reading either the related business or person file randomly. The read on the identifier file will benefit from caching but, because the reads on the other two files will, on average, not be in key order it is likely that each will result in a separate select statement to SQL server.

When the code inserts a new record, or updates or deletes an existing one, the information in the cache becomes potentially out of date. A possible approach would be to scan caches and update them as necessary. However, this could get extremely complex (especially if an update modified key values). The emulator adopts a simple

minded approach and simply discards cached data if an updating operation is performed on the same file.

This approach will result in zero or low performance gains in any process that moves sequentially through a file updating records as it goes.

Ageing and timeout of cached information

The processing outlined above will recognise that cached information is potentially out of date when code *in the same process* performs some updating operation. However, all Iris products are multi user systems and the database could equally be changed by some other user on another workstation. If your code has a record in cache and keeps re-reading it (waiting for it to change as a result of some other process) then a simple minded caching mechanism will wait forever!

Cached information is potentially out of date the moment that it is read. As time passes it may be considered to be more out of date. The correct, and normal, strategy to cope with this is to discard cached information once a specified period of time has expired since it was read from the database. However, the emulator contains a timeout mechanism for cached data *based on the time since the application last requested it.* Thus, if the applications keeps asking to read data that is in the cache, then the cache will potentially never time out. This is, presumably, a result of a misunderstanding of the nature of the problem.

The original misunderstanding is a bit of a mystery. It represents a view of caching that is appropriate for a resource limited system, where there is no question of data becoming out of date, but the problem is to choose which data to discard in order to free up resources. A good example of this is virtual store paging algorithms, which is nothing like the problem we are trying to solve here.

Nonetheless, the *application* processing profile for the practice suite (where a user typically works on one practice client at once, and it is unusual for two users to work on the same client) has meant that the erroneous timeout mechanisms have not caused any problem in practice (to date). Following the sound engineering principle of 'if it ain't broke don't fix it' the erroneous mechanisms remain in place (mainly because we have no way of knowing what the performance implications of correcting them would be).

In the Exchequer case there was at least one problem arising from these mechanisms (precisely because one process was waiting for another to change a record). However, the solution adopted was to provide an API to allow the applications code to specifically request caches to be discarded, leaving the basic erroneous algorithms in place. Again this was done on the basis of maintaining code stability (and performance) at an advanced stage of testing.

Optimisation – the prefill cache mechanism.

The preceding sections have pointed out how caching can help performance in some cases, and some of the reasons that it may not.

For the cases where significant updates are performed during the processing there is no further performance gain to be had from caching, without considerable enhancement to the mechanisms to keep the caches up to date with the updates.

For the non updating processes, where caching is not helping because of the access paths involved, there are alternative strategies. These involve understanding the context of the applications query, and this can be done in one of two ways:

- By providing specific emulation code which has specific knowledge of what the application is doing and/or the structure of the data that it is dealing with.
- By generic applications code that allows the application to specify the context within which it is working.

Both approaches are used, but the first, by its very nature, tends to be product specific (and is currently only used by the Practice Suite).

The main generic mechanism, used by both the Practice Suite and Exchequer, is the prefill cache mechanism. This works as follows:

- The application creates a prefill cache by supplying the SQL select statement that is to be used to get the data. Whilst this SQL can be any valid SQL the end result must be the same structure as would arise from standard caching, i.e. all the columns in the table mapping to the specific file that subsequent BTRIEVE requests will be made against.
- On *every* subsequent BTRIEVE call that is in the context of this query the application must precede the call with a call to inform the emulator that the next call is to use the prefill cache. Requiring this positive applications action ensures that default behaviour is unaffected, and, specifically it avoids the emulator itself assuming that the prefill cache was to be used just because it existed.
- The application also has an API to allow it to discard a prefill cache when it is no longer required.

This mechanism achieves performance improvements by effectively shifting the actual SQL from the emulator to the application. The emulator, in these circumstances, is effectively not generating any SQL, but merely executing the SQL supplied by the application. Its role is reduced to mapping BTRIEVE requests onto the cache.

One side effect of this is that it bleeds information about the SQL schema into the application. We rely on the application programmers and designers to keep this

functionality localised so that a change to the schema in the future will not be too onerous.

Database locking mechanisms

BTRIEVE implements locking mechanisms. These work at the level of locking individual records. There are four variations of locks formed by the combination of two options.

- Locks can be single or multiple. When single locks are used there can be only one record locked on a file at any one time (and BTRIEVE will implicitly unlock records as necessary).
- When a lock cannot be acquired then BTRIEVE can either wait until it can, or immediately return an error code. The former case is referred to as a WAIT lock, the latter a NOWAIT lock.

Not all combinations are used by existing products. The practice suite, in particular, only uses the simplest case of single NOWAIT locks.

Mapping of these facilities to SQL is awkward. The basic SQL mechanism is pessimistic locking applied to a record set. Thus, to lock a record, you select the individual record into a record set which is set up with pessimistic locking and then read it. This more or less works although there do seem to be small windows of time in the processing during which it is not 100% watertight.

One of the failures that we have encountered is when applying a lock on a sequential read. SQL server supports quite sophisticated mechanisms for deciding what a process can see on the database when another process is, or might be changing data. These facilities, called isolation levels, are designed to present various levels of consistent view, which generally ignore changes that are outstanding, but have not yet been committed. (This is the default level which is the one that we run at.)

We have observed that, in some circumstances, rather than waiting to acquire a lock on a row during sequential access, SQL will simply skip it and get the next available row.

The net effect is that locking emulation is pretty close for a 'get equal', but unreliable on sequential reads (and we have empirical evidence that it is not 100% watertight under high contention, even for get equal).

In practice, contention is rare in the Practice Suite, and the consequences of contention failure are minor.

Exchequer has the added complexity of using NOWAIT locks. These have to be implemented by changing the lock timeout on the SQL connection to zero prior to the selection of the data.

Changing the lock timeout to zero has an unfortunate side effect that it also affects the locks that SQL server itself applies internally. This means that it becomes possible to get a lock timeout failure when executing an 'update' or 'delete' against a previously locked row (as SQL server applies page locks internally against the physical pages that it needs to update). This has necessitated putting internal 'retry' code into the emulator to repeat update/delete operations when this occurs.

Even the retry mechanism does not always work. SQL server locking is sophisticated and it is possible, with zero timeout locks, to get into an internally generated deadlock situation at the time of doing an update. We have occasionally reached the limits of practicality in trying to get SQL server to look like BTRIEVE and, in these cases, the pragmatic (and cheaper) solution is frequently to make some code change within the application itself.

The most obvious change is that a BTRIEVE 'deadlock' response can be returned by almost any BTRIEVE operation. So, whereas, in the BTRIEVE case the successful acquisition of a record lock can be assumed to guarantee the viability of a following update of the record, in the SQL case it cannot.

Use of stored procedures

Whatever emulation strategy is adopted there occurs, fairly inevitably, some circumstances for which performance continues to be inadequate. For these cases the only alternative is to resort to native SQL.

By and large this is done by writing stored procedures and having the application code invoke them. This has advantages in that it avoids bleeding knowledge about the database into the application. However, it does tend to bleed applications logic into the database.

The degree of emulation support that is provided for applications use of stored procedures varies from the simple provision of facilities to get a connection string, to total encapsulation. The latter is adopted in the Exchequer case as a matter of convenience, because Exchequer is implemented in Delphi and does not have easy access to ADO with the correct parameter types.

Stored procedures are also used under the hood within the database when implementing triggers. This is not strictly relevant to the emulator code, since the triggers are entirely below the SQL interface to the database. However, it is mentioned here simply for completeness.

Relational integrity

In principle the relationships between records on various BTRIEVE files can be defined, even though BTRIEVE itself has no way of recording them. (Such relationships are embodied in the actual code that implements database navigation within the

application.) So, in principle relational constraints could be defined in the target database. However, except in the restricted case of the remapped Time and Fees data, they are not. Here are some of the practical reasons why:

- In the 'order processing' example given previously there is a clear parent child foreign key constraint between the order and its order lines. However, given a simple 1:1 file to table mapping this would imply a foreign key relationship between a table and itself (with implied constraints on some column values in the 'parent' rows).
- There is no guarantee that the applications code will create the records in the order you think it should (because it's never had to in the past). You could therefore end up with a situation in which a 'normal' code path always violated relational integrity.
- With several thousand user database out there it must be assumed that archaeological traces of every bug in the last decade are represented somewhere. These databases will conceivably have orphans on them. It is equally conceivable that orphans are invisible to current applications paths, and hence no-one has even noticed that they are there.

It should also be noted that, from the point of view of the error generated, a constraint failure, as a result of trying to insert a duplicate entry into an index that doesn't allow it, looks pretty similar to a constraint failure resulting from trying to insert a child record with no parent. Bits of code within the emulation software see failures of this sort and *assume* that they represent duplicate index entry failures (which have to be reported back to the application in the same way that BTRIEVE would). If any extra relational constraints were to be added to the database the emulator would need enhancing in this area.

ADMINISTRATION AND CONFIGURATION

Multiple 'databases'

Both Exchequer and the Practice Suite have the concept of multiple 'databases', although they arise from different origins:

The Practice Suite has a confidential client facility that would be compromised by the free visibility of data through SQL. (In practice, the confidential client 'database' has never actually been issued for the SQL version, but the core implementation remains present in the database.)

Exchequer has historically issued a multi-company version of its software, designed specifically for bureau use.

Whilst both of these facilities can be viewed as separate logical databases, they are not necessarily implemented as such. In particular the Exchequer implementation of a bureau system is implemented on a single SQL server database.

The broad mechanism is as follows:

- The initial login (see below) gives access to a particular database which has on it a schema called Common. This login typically has access to other schemas as well.
- This schema can (and does) contain tables representing application BTRIEVE files. In the Exchequer case these are the files that are not split out into the individual company datasets. In the Practice case they constitute some of the licensing files and tables containing information to be distributed to the site as a whole.
- This schema contains the XML mapping files controlling emulation (table IrisXmlSchema).
- Finally this schema contains a table IrisDatasetConnection which contains entries for the 'logical' databases on the system. Each entry in this table maps to a schema and a connection string (suitably encrypted) to give access to that schema.

This mechanism allows multiple schemas to be created on a database with separate access permissions to each of them. However, each of these schemas must contain identical tables (because the Xml mappings are held in the common schema). Similarly, during an upgrade, all active schemas must be upgraded in parallel.

Access permissions, logins and users

All services hosting databases running Iris products are required to be set up to Windows and SQL server authentication. Connections needed to create databases,

users and logins etc are established using Windows authentication (which must be suitably available to the user performing the function). These functions typically arise during installation or, in the Exchequer case, creation of a new company.

Normal access to the database uses SQL server authentication. This means that a Windows user who has no direct access to the database can still get to the data by using the Iris product. The code within the emulator supplies the necessary credentials for these connections.

Each Iris system has a 'master' or 'admin' SQL user mapped to a specific login. This user is created at initial installation and the necessary credentials are stored in an Xml file. This is called iCoreLogin.xml for the Practice Suite, and ExchSQLLogin.xml for Exchequer. This file also contains the name of the SQL service and the database within it to connect to. Thus, the content of this file is sufficient to create an SQL server connection string.

(Note that the name ExchSQLLogin comes into effect with the 10.1 version of the Exchequer emulator – targeted for Exchequer release 6.1. Prior to this the names were the same for the two systems which, along with other name clashes was leading to problems when the Practice Suite called in to Exchequer. Exchequer release 6.0 SQL is released with the restriction that it must not be installed onto a Practice Suite SQL site.)

The login Xml file is present in the directory from which the program is loaded. This is fundamentally how the system finds it.

Information within the login Xml file is encrypted (naturally). The file is created by a utility which the installer uses (albeit indirectly via an API call). The origin of the username and password actually used is the installer itself.

In the Exchequer case there are also system stored procedures that the code needs to execute and, for this reason, the admin user is also given specific execute rights in the master database.

The admin user has rights over all schemas in the database, but specific rights to other schemas only are conferred by entries in the IrisDatasetConnection table. Each such connection can have two users/logins associated with it. (I say user/login because, under normal circumstances they will map 1:1). One user is the standard user for that connection (with full read/write permissions), and the other is a 'read only' user and is optional. The Practice Suite does not use the read only user. These entries are created by a second utility as part of installation (and, in the Exchequer case, also as part of the act of creating a new company).

Database Administration

The emulation software is responsible for providing facilities to allow databases to be created, initialised and upgraded etc. The primitive facilities are packaged up into various APIs and utilities that are available to users and the installation packages.

Code in this area tends to be complex and quite difficult to follow. This is for two reasons:

- The code is mixed managed and unmanaged. With hindsight this may not have been the best move, but the probable original reasoning was that the U/I aspects could be implemented more quickly.
- The code frequently involves lengthy processing, so runs multi threaded with a processing thread and various U/I threads. This multi threading is wrapped in common managed 'controller' classes which can make it difficult to follow.

The rest of this section describes the primitive operations and then outlines how these are built into higher level functions for use by installers etc.

Note that, where Exchequer specific file names are given for Xml configuration files, these only become effective from emulator release 10.1 (Exchequer version 6.1). Prior to this point the Practice Suite names are used for both.

Database Creation

The prerequisite conditions for the initial creation of a database are that:

- SQL server is installed (with the Express version being acceptable).
- The environment can connect to the service (with appropriate permissions) using integrated security (i.e. under Windows user name authority).

The CreateEmptyDatabase API does what it says on the tin. It creates a blank empty database of the correct name with the correct collation, size and recovery model.

The server instance and database name to be used are supplied as parameters. The basic details of the database parameters are read from an Xml file named iCoreDbInstallerConfig.xml for the Practice Suite and ExchSQLDbInstallerConfig.xml for Exchequer (emulator version 10.1 onwards).

Logins and Users

The login credentials that give access to the Iris database common schema are held (encrypted) in an Xml file (iCoreLogin.xml for the Practice Suite, ExchSQLLogin.xml for Exchequer – emulator version 10.1 onwards). At run time the emulator reads these details, and that constitutes its entry point into the database.

The API 'CreateIrisCommonLogin' has the function of setting up this login and writing the Xml file to allow the system to use it. As a precaution the code first deletes the *specified* login from the database if it already exists.

The next stage is to check if the appropriate Login.xml file already exists. If it does then the code determines what the current login name is (from the file) and drops this login from the database. Either of these steps may fail and will throw an exception if it does, thus terminating the process. (The recovery path if this happens is to delete the appropriate Login.xml file.)

The code now:

- Creates the login on the database service.
- Changes the login password policy
- Adds the specified user to the specified database, connected to the login just created.
- Sets the users role to DBO and server admin.

In the Exchequer case, for this 'admin' user it also invents a user on the master database attached to this login. This user is then given explicit execute permissions to system stored procedures sp_OACreate and sp_OAMethod. This is for a particular Exchequer requirement to create Activex objects from within database code.

The code saves the associated connection information to the appropriate Login.xml file. It then creates the tables in the database to hold the individual schema connections.

- These tables are only created if they do not already exist.
- The SQL definition of these tables is hard coded into the emulation functionality that creates them.

The 'CreateIrisLogin'API has the function of adding an entry to the 'schema' connection tables (that hold details of the login to be used when accessing a particular individual schema), and to add that user and login to SQL itself. The parameters to the API are effectively the details that need to go into the table.

This code is designed both to be used in initial installation, and to be used when adding a new schema to the database (for example, when creating a new Exchequer company). It therefore works on the basis that there could be entries already in the table, and gets these into an object before it starts its actual additions. At the end of the process the changes are saved back to the database.

The process of actually creating the login in the database follows exactly the same path as for creating the common login. (At the moment, in the Exchequer case, this appears to include mapping the login into the master database to give execute permissions to

the two system stored procedures, although this last functionality may not be strictly necessary.)

Apart from the fact that credentials must be encrypted, adding the data into the table is straightforward.

The reasoning behind these two separate sets of credentials is that it becomes possible to restrict *any* access to schema based data so that there is a degree of protection that, in the Exchequer case, keeps one Company's data safe if another Company's data is compromised. This breaks down for one or two tables that have company information in them, but are stored in the common schema (because they were only single files in the original BTRIEVE implementation).

Table Creation and Upgrade

There are two base sources of information as to what constitutes an Iris database.

- Xml configuration file iCoreConfig.xml (or ExchSQLConfig.xml for Exchequer from Emulator version 10.1).
- The individual Xml schema mapping files for each emulated BTRIEVE file.

iCoreConfig.xml contains:

- Details of the current diagnostic logging state and logging information destination.
- The relative path to use to find the individual schema files.
- A list of the emulated files and the schemas used to map them (together with choice of emulator methodology and other specific emulator parameters for each file).

During actual operation of the system the individual schemas are held on the database (for performance reasons). However, iCoreConfig.xml itself remains as a normal Windows file and is accessed as such.

In fact all access to the individual Xml schemas whilst any part of the emulator is running is normally from an instore cache. This cache is initialised, from the database, when the emulator is initialised.

The information in the individual schemas is sufficient to generate a create table statement for the 'simple' case, and this is the mechanism used to create a new empty database. Low level 'create table' functionality exists to support this.

The cases where this approach will not work include:

- Tables within the Time and Fees area (which do not have Xml schemas).
- Views that represent BTRIEVE files and require supporting triggers to support update functionality (see Practice S/W 'Specific' below)..

To cope with these more complex cases, plus the fact that the installation process must cope with both new installations and upgrades, there is a further XML configuration steering file 'iCoreDbCreateUpgradeConfig.xml' (or 'ExchSQLDbCreateUpgradeConfig.xml' for Exchequer from Emulator version 10.1). This file effectively defines the installation or upgrade process and includes:

- The ability to conditionally run SQL scripts before the process
- The ability to run SQL scripts after the process.
- Details of how individual 'files' are to be processed (specifically whether any existing data is to be overwritten and whether any data is to be initially imported into the file).
- Details of what specific upgrade scripts must be applied to existing tables to upgrade them to the next version.

In order to support specific upgrades the schema information on the database is versioned:

- The original source of the Xml schema definition for a file includes a version number.
- When the schema is loaded onto the database this version is recorded in the XML definition table (IRISXMLSchema).
- There are additional SchemaVersion tables (one per SQL schema) which just contain the Xml schema name and the version (i.e. no actual schema details).

The existence of these two separate tables may seem a little puzzling at first, but there is a logic to it.

Normally a change to an Xml schema will also involve making a change to the corresponding SQL table, and these changes must be in step. The table IRISXMLSchema is a direct copy of the raw XML file that was loaded onto the database (including the version number recorded in it). There is just one instance of this table in the common *SQL* database schema mirroring the fact that there is only one set of underlying XML files at any one time.

On the other hand there is one instance of table SchemaVersion in each *SQL* database schema and it is in step with the actual SQL table definitions in that schema. This is accomplished by including the SchemaVersion update in the actual SQL script that changes the underlying SQL table (and is run as part of the upgrade process). Both these operations occur within the context of a particular *SQL* database schema and, in the case of Exchequer, the same script is applied across all the company schemas on the database (unless it refers to a table in the common schema).

Thus a necessary (and sufficient) condition for the emulator to be able to correctly map an SQL table to the implied BTRIEVE record layout is that the IRISXMLSchema version

must agree with the SchemaVersion version. (I.e. that the SQL table has been upgraded to match the XML definition of this). Correspondingly, the emulation software *can* detect if the XML schema and SQL table definition do not agree. However, for the Practice Suite currently only the code supporting installation takes any notice of these tables. The Practice Suite run time emulator does not cross check the version tables when loading in its XML definitions of file layouts.

The Exchequer run time emulator however does do the cross check on a Company schema at the time that the company is opened.

Both versions of the code do cross checks during installation or upgrade.

The processing is necessary in principle because database table upgrades are not derived from the XML schemas (unlike initial table creation). A moment's though indicates that this is the case simply because there may be *more* to an upgrade than just adjusting the table definition.

It is also true, in general that the layout of XML schema definition for a file must also agree with the view that the applications code has of the file. This is because code accesses BTRIEVE files at the record level, so must have space in the record buffer for *all* the fields that are present at the time.

There is no database capability to check on this (other than normal QA and testing of the application). Each development department must define processes to ensure that database amendments are simultaneously applied to the code, the database and the XML schemas.

Correspondingly, since there is only one version of the code issued at any one time, the version of the tables in each SQL schema must also agree. (This is especially relevant to Exchequer multi company installations where each company is a separate SQL schema).

If you look at the definitions of the two versioning tables you will see that the database is capable of supporting multiple versions of the same schema. The code, however, is not capable of this and there should be only one version of each XML schema recorded in each table.

The general mechanism for copying the raw XML schema definitions onto the database runs from the configuration file iCoreConfig.xml (or ExchSQLConfig.xml for the Exchequer emulator from version 10.1). For each schema listed therein it accesses the file, checks whether it is already on the database or not and, if it is it compares the version in the file with the version in IRISXMLSchema. The database is updated if the schema was not previously there, or if the schema is a higher version than the database. The other way round (i.e. the database is a higher version than the schema) is treated as an error.

This general upload mechanism does *not* cross check against the SchemaVersion table at all.

The uploading of the XML schema definitions is invoked when a database is created, upgraded or converted from a prior BTRIEVE database.

BTRIEVE Database Conversion

Conversion of a BTRIEVE database to an SQL one is simple in principle. Once the database has been created each BTRIEVE file can be read sequentially and the raw records that this reads can be passed straight to the emulation software to be written to SQL. (Every BTRIEVE file that we currently support has at least one key, hence key zero is always valid.)

Of course it's rarely this simple in practice. Although the destination database has no constraints on it (in particular it has no foreign key constraints) it is still possible to get failures on insert because particular fields have invalid values in them. This can occur because there is absolutely no restriction on the bit patterns that can be written into a BTRIEVE record.

The conversion process is steered by and XML configuration file, iCoreDbConverterConfig.xml (or ExchSQLDbConverterConfig.xml for the Exchequer case from emulator version 10.1). This:

- List files that are not to be converted, including files identified by wildcard patterns. (Mature Practice Suite databases tend to have a number of ancient obsolete files in them.)
- Gives the naming patterns that identify files as part of the database, and identifies subfolders that have to be searched to find them (where relevant).
- Lists specific SQL scripts that are to be run prior to and post conversion.
- Lists specific suppressions of warnings that are not deemed to be important.

Export and Import

The emulator software provides specific functionality to allow sets of data to exported from and imported to Iris SQL databases. Although this is not strictly an emulation problem it has much in common with other aspects of emulation processing, so this is a convenient place to include it.

The sets of data to be exported are defined by configuration files. The standard configuration file is iCoreImportExportConfig.xml (or ExchSQLImportExportConfig.xml for the Exchequer case from emulator version 10.1). The configuration file maps user visible data set identifiers to sets of tables. Each table in a set can have other specific options set against it to control processing (for example, whether the table is optional, whether existing data should be overwritten on import).

This mapping is capable of including standard common file groups, so that core sets of files (like the core client data in the practice suite) can be defined just the once, and incorporated into other data sets as required.

The exported data format is XML zipped. Zipping is accomplished in line by virtue of a managed zip stream library originally acquired as free source from the web. In the best traditions of .Net the various stream objects can be just stacked on top of each other with each layer performing its own function (i.e. Xml encoding, zip compression and, finally, writing to physical media).

This works admirably for output, but, apparently had problems on input. Hence the 'import' function gives the option to 'unzip before import'. This does *not* mean that the function is capable of accepting unzipped input directly. If the option is ticked it simply means that the import file will be unzipped to a separate file which will then be imported as Xml. If the option is not ticked it means that the stream stack will be set up to include the zip component, and there will be no intermediate unzipped file created on disc. Functionally, the two options are identical – it's just that the second might not work!

The export mechanism includes the appropriate schema version (from the SchemaVersion table). Apart from this the export is blind, in the sense that it works purely from the SQL database, not any Iris specific information.

The normal selection of data for output is a 'select * from table'. There is scope within the configuration file to allow a 'where' clause to be added so that a dataset can be defined that constitutes a subset of a table. The export then takes the metadata details from the results of the selection and uses this to output the columns (eliminating 'read only' columns that cannot be subsequently imported).

The zip file is written as a zip of a series of separate files, typically one per table. These are preceded by a 'details' file that lists the other files that are included.

The import process cross checks the details file with the other entries in it. It also cross checks the schema versions recorded in the export against the contents of its SchemaVersion table. In the unlikely event of a table being present in the import, but not actually in the target database it will fall back on its XML schema definitions to create it.

The import process is destructive. Prior to adding the data from the import file it deletes rows from the target table. Normally this will be *all* rows, unless the dataset definition specifies a 'where' clause.

The import process constructs a select command to gain access to the metadata describing the target table. It uses this to cross check the import data against the table, and will throw an exception if there is a problem. Provided everything is OK the import is accomplished by constructing individual insert statements (one per row) and executing them.

It's worth emphasising that the export and import processes are not meant to be applications functionality. The selection of data is too coarse and unrelated to business functions. Also, the selection is by data type, rather than by data context. For example it is not possible to export all data for a particular Practice Suite client using this method.

The prime use of export and import functionality within the Practice Suite is twofold:

- As the means by which an initial set of data is installed onto a new Iris database, or updated standing data is issued with an upgrade.
- As a mechanism by which a customer can send in a sample data set for investigation. Currently, any data fixes that might be implied are then applied to the customer data separately by script. We do not fix databases remotely and send the results back.

You can see from the above that the use of import is constrained to database installations and upgrades, plus local use at Iris when importing customer data. In these circumstances the destructive nature of an import is not a problem.

Exchequer already uses the first mechanism, and it is anticipated that further use of import and export will not go beyond the second.

Be cautious when considering Exchequer export and import. It is tempting to imagine that this could be used to export an entire company from a multi company system. However:

- The original Exchequer implementation had each company with a separate folder with a set of BTRIEVE files within it. This set of files now maps to a set of tables in a particular SQL schema corresponding to that company.
- The original implementation also had some files that existed outside this structure. Trivially this included the file containing the list of companies. These files now map to tables in the common schema.

The unfortunate fact is that there is at least one file in this common area that contains data on a per company basis. Therefore, an export of a single company *may* turn out to be practical, provided you accept that company data held in the common schema is not needed.

Data Load for Installation/Upgrade

A 'blank' Iris database is not empty. It contains various licensing and standing data as issued by Iris itself. The mechanism for adding this data to a customer database is to run the Import function. This import function needs two things:

- A zip file with the 'blank' exported data in it.
- An import configuration file defining the core data set. For the Practice Suite this file is iCoreImportExportConfig_NewDb.xml.

The Installer APIs

The emulator software provides a number of interfaces for use by the installer. The main ones are listed in the following paragraphs.

CreateEmptyDatabase

This interface really does just create an empty database based on the information in iCoreDbInstallerConfig.xml (or ExchSQLDbInstallerConfig.xml for Exchequer emulator version 10.1 and later).

CreateIrisCommonLogin

This interface has the job of establishing the mechanism by which the application connects to the database. It writes the iCoreLogin.xml file containing the credentials to allow this to happen (or ExchSQLLogin.xml for the Exchequer emulator version 10.1 and later).

This interface also creates the IRISDatasetConnection table on the database to allow schema specific credentials to be stored.

CreateIrisLogin

This interface has the job of creating credentials to allow a specific schema to be accessed. It creates the appropriate login and user within SQL server itself and writes encrypted credentials to the IRISDatasetConnection table.

CreateIrisDatabase and CreateIrisDatabase2

These two interfaces are essentially the same functionality. CreateIrisDatabase2 is the current one and it allows the database name to be specified, plus a zip file containing data to be initially imported.

CreateIrisDatabase simply calls CreateIrisDatabase2 passing the zip file as null. (However, chasing the functionality down through the code I **don't** believe that this means that the functionality is able to skip the import.)

The functionality of this interface is to create all the tables required for an Iris database and populate them with the necessary initial values. This includes control tables holding XML schemas and the schema version table.

The database creation process is steered by an XML configuration file iCoreDbCreateUpgradeConfig.xml (or ExchSQLDbCreateUpgradeConfig for the Exchequer emulator version 10.1 and later). This configuration file defines the specific processing to be undertaken **and** specifies a default import zip file that will be used if none is explicitly specified. 'Specific' processing includes SQL scripts to be run at various stages of the process and an indication of whether specific tables on the target database are to be overwritten if already present. (This latter specification is available

because, as the name suggests, this file is used for both new installations and upgrades.)

The process also requires the login and base configuration XML files to be present so that it can access the database and set up the emulation environment.

The functionality provided by this interface is then:

- Create the IRISXMLSchema table and copy the XML schema details into it from the current XML schema files.
- Create the SchemaVersion tables and populate them.
- Run pre-creation SQL scripts.
- Create empty tables based on current schemas and configuration information.
- Import initial data into tables that require it.
- Run any post creation SQL scripts.

UpgradeAllDatabases

The function of this interface is to upgrade an Iris database. Upgrades will normally consist of running SQL scripts to change the database itself and loading newer versions of XML schemas.

The version numbers for changed objects need to be updated. For the IRISXMLSchema table this occurs because the version number is recorded in the raw XML schema source. For the SchemaVersion tables it must be done by the SQL script that changes the table structure itself.

The procedures outlined in the previous section for the CreateDatabase case are all written to create new objects only if required. Hence, all the primitive functionality is applicable to the Upgrade case as well as the Create case (with occasional minor variations of behaviour controlled by configuration settings or parameter).

In both cases the actual upgrades are controlled by comparing version numbers in the supplied configuration and information with version numbers recorded on the database.

Thus the processing of the Upgrade is identical to the processing of the initial create (including the import of data for any tables that are configured to be overwritten on upgrade), with one important exception.

As has been pointed out previously, both the Practice Suite and Exchequer are designed to work in a 'multi company' mode, where the implementation of separate 'companies' is achieved by use of separate SQL schemas with identical table structures within them. (In the Practice Suite case this capability is not currently active, but in the Exchequer case it very much is, addition of an extra company being achieved by means of the CreateCompany API.)

This functionality is then specifically designed to apply the create/upgrade functionality across the common schema and *all* 'company' schemas in turn. This was not necessary on initial database creation, because initial installation only installs one 'company'.

The list of schemas to be upgraded is simply acquired by selecting the information from the IRISDatasetConnection table.

ConvertDatabase

This interface supplies the functionality to take a set of BTRIEVE files and convert them to an SQL database. The process is controlled by the XML configuration file iCoreDbConverterConfig.xml. This file:

- Specifies, by wild card and subdirectory specifications, how to recognise which files in the specified root directory (and below) are BTRIEVE data files that require conversion.
- Specifies specific files (both explicitly and by wild card) that should be explicitly excluded from the conversion.
- Specifies specific warnings that should be suppressed for specific files.
- List SQL scripts that are to be run pre and post conversion.

The architecture of the conversion is based on abstract base classes that derive specialisations for the particular case. The two particulars are the Practice Suite and Exchequer.

This split in the implementation of variability between configuration file and explicitly coded specialisations reflects the radically different locations of BTRIEVE files between the two products, in particular the manner in which Exchequer files are located in order to implement multi company operation.

Now the ConvertDatabase interface only does multiple companies (i.e. schemas) on the basis of the specialisations provided in code. For the Practice Suite this is not a problem, there is one fixed schema corresponding to the single Iris database and the process can simply use this schema and the file selection in the configuration file to find the single set of files to be converted.

The Exchequer case is more complex, and the initial Exchequer implementation (emulator version 10.0) does not include database conversion. (All initial Exchequer SQL installations are new databases). It is for this reason that I didn't add the usual proviso about the name of the Exchequer configuration file post at emulator version 10.1 and above. There *is* an Exchequer conversion configuration file, and there *is* specialisation code to generate lists of BTRIEVE files for specific Exchequer companies. However, it's not immediately clear how this mechanism copes with

common files and, more importantly, it's unlikely that the Exchequer conversion code has ever been run in anger.

The latest information that I have indicates that database conversion is *not* a requirement for Exchequer. If this changes in the future then be aware that, although the core mechanisms are present, it is likely that a careful reconsideration (and possible reworking) will be required.

The overall pattern of processing for a conversion has much in common with that in an initial database installation. In particular it:

- Loads the XML schemas into the database.
- Creates and populates the SchemaVersion tables.

After this, for each data set (of which there is just one for the Practice Suite, and there would be at least one per company for Exchequer) it:

- Runs any specified pre-conversion scripts.
- For each file, creates the table and populates it by reading from the BTRIEVE files and writing to the SQL database.
- Runs any specified post-conversion scripts.

ImportData, ExportData and ExportData_Exchequer

The export and import functionality has been described earlier in this document (in terms of it being accessed directly by users). These interfaces package this functionality up so that it can be accessed by the installer, or tools supporting it.

Note that the installer does *not* need to use the import functionality directly during an installation, since import of the core 'blank' data is already encapsulated in the functionality provided.

BackupDatabase, RestoreDatabase, BackupDatabase_Shim and RestoreDatabase_Shim

A typical part of an installation procedure will involve backing up the user's existing data before doing something fairly substantial to it. Similarly, in the event of the 'something substantial' failing, it may wish to subsequently restore that backup. These interfaces are designed to provide that functionality.

The '_Shim' variants of the interfaces do not differ functionally from the others. They arise because of particular circumstances during an upgrade installation. The problem is essentially that managed code cannot be unloaded but the backup of the database must be done without disturbing the currently installed software, whereas later functions must be done with the newly installed software – all within the scope of the installer being loaded.

To cope with this involves a certain amount of fiddling with assembly load resolver events to avoid leaving an incorrect version of code loaded. This is a problem that we should all be grateful is already solved, and the best course of action, if any further interfaces are required, will be to simply copy the pattern that is already established.

The following paragraphs are copied from the notes within the Practice Suite code that summarise the rules for calling shim and 'normal' interfaces.

- When iCoreDbInstaller is called by the Practice Suite installer, it can be called from either a temporary directory or the application installation directory. To avoid problems we need to ensure that there is only a single instance of the dll loaded into the installer process at any one time. The following rules are designed to ensure that this is the case.
- Until the installation directory has been updated with the latest files, the installer dlls must only be loaded from the temporary directory
- When calling functions in the temporary directory, any functions that aren't purely unmanaged code must be called through the shim dll
- Once the installation directory has been updated with the latest files, the installer dll must only be loaded into the installation directory
- When calling functions in the installation directory, all functions should be called directly from iCoreDbInstaller.dll
- These rules ensure that when loaded into the temporary directory, both iCoreDbInstaller and iCoreDbInstaller_managed can be unloaded. The shim dll can't be, but it won't be used anywhere else.
- The version of iCoreDbInstaller.dll loaded from the installation directory can't be unloaded once it has called a .Net function, but we won't try to load the dll again from anywhere else, so this is acceptable.
- The reason that iCoreDbInstaller.dll wasn't unloading originally is that it was loading managed code. This meant that the assembly was being loaded into the default AppDomain, which doesn't get unloaded until the process exits. An initial thought was to create a shim dll that creates a new AppDomain and then loads DbInstaller into that to call functions that require .Net, but it turned out that some of the CRT initialisation code in a mixed mode C++ dll explicitly initialises itself in the default AppDomain (thus loading the assembly there). So, the plan of attack was to split out any .Net code into a separate assembly and use the shim dll to load this when being called in the temporary directory. This allows the temporary dlls to all be unloaded with the exception of the shim dll.

- iCoreDbInstaller.dll contains the function exports for the .Net functions as the installer still calls these in the installation directory (and some are used by Exchequer!).
- Once iCoreDbInstaller was unloading, it turned out that iCoreDAL wasn't being unloaded. This was because the settings for iCoreDbInstaller marked iCoreDAL for delay loading. Because of this the compiler generated LoadLibrary calls for iCoreDAL, which didn't have a corresponding FreeLibrary. Rather than worry about which functions may call into the DAL, the installer was changed to copy in the DAL and its dependencies when loading from the temporary directory. One side effect of this change was that the installer had to change the working directory before the UseDll call rather than immediately before the function call, otherwise the DAL wouldn't be found]

Once the code actually digs its way through to doing the actual job it is accomplished essentially by simply submitting the SQL command.

LockDatabase, UnlockDatabase and IsDatabaseLocked

It is fairly important during installation operations that no-one else is attempting to change the database. Whilst there are currently no watertight mechanisms within the applications themselves there are core facilities to support a lockout of the database by the installer. These interfaces represent the exposed capabilities.

The actual mechanism used is a database extended property called IRISDatabaseUpgradeInProgress. If this is set to TRUE then the database is considered to be locked.

Notes

The normal sequence of calls to create an Iris database from scratch is:

- CreateEmptyDatabase
- CreateIrisCommonLogin
- CreateIrisLogin

This sequence precedes the loading of initial data onto the database *or* the conversion of an existing BTRIEVE database into its SQL form.

The interfaces available to the installer for creating a new Iris database generally require an import zip file for the initial data to be loaded. This is conventionally called data.zip.

The process of creating data.zip is to export a specified subset of the database using the conventional export facilities (provided by the emulator code) from a database with the relevant data on it.

This leads to a 'chicken and egg' situation, in that something has to create this initial database. In the case of the Practice Suite the sequence is:

- The development process delivers relevant BTRIEVE files with the initial data on them.
- These files are copied into a standard blank BTRIEVE database.
- This initial BTRIEVE database is run through the standard conversion process to produce an initial SQL database.
- The data is exported from this SQL database.

For the Exchequer case this entire process is the responsibility of the Exchequer application team, rather than the emulator.

PRACTICE S/W SPECIFICS

Use of Views

There are a number of cases of data within the Practice Suite that is owned by a specific tax year. The original BTRIEVE implementation had a *separate* file for each year.

In the SQL mapping there is only a single table with the tax year as a column in it. The individual original BTRIEVE files are implemented as views onto this table. These views are then mapped to the BTRIEVE files for emulation purposes.

Views do not support update modes per se. However, they can be made to do so by the use of triggers, and this is what is done for these cases.

The upshot of this is that the creation of a new tax year file resolves to the creation of a new view.

TLD format

The Iris Practice Suite has a proprietary format for holding variable data within a BTRIEVE record. This is referred to as TLD, standing for type, length and data.

Use of TLD means that any particular field in a BTRIEVE record need not be at a fixed position in the record.

Code exists within the practice suite for packing a fixed format into a TLD format, and for unpacking it again afterwards. This code is steered by field control tables, for which, unfortunately, there is no central definition.

Within the emulation layer the pack and unpack routines are run in reverse. Thus, on reading:

- The emulator reads the individual columns from the database.
- Using its Xml mapping tables it packs the TLD part of the data into a single stream in the buffer, and returns this to the application.
- When the application receives the buffer it unpacks the data into individual fields using its own mapping tables.

And on writing:

- The application packs the TLD data into the buffer using its own mapping tables.
- When the emulator receives the buffer it unpacks the data into individual fields using its Xml mapping tables.
- The emulator uses these individual fields to update the database.

The pack and unpack routines within the emulator are not the same ones as used within the practice suite itself. Since the algorithm is fixed (and has been for over fifteen years) this is not a major problem. However, there has been at least one discrepancy in behaviour between the two.

Generic Access

There exists within the practice suite a facility known as generic access. This attempts to mimic an SQL select statement by specifying a key path, a key range and a filter function for a given set of BTRIEVE accesses. Because the correspondence with a select statement is so close there is integration between the emulator and the containing generic access object.

This is very similar to the prefill cache mechanism, in that, if there are any filter criteria the application supplies them as SQL conditions. It is different, however, in that:

- The emulator can itself determine whether a particular access is in the context of a generic access object or not, so the application does not need to specify before each API call that it is using the current contextual query.
- The generic access object actually contains the key number and key range, so the emulator can construct the key condition itself, rather than having the application supply it.

This facility was originally developed as part of the Time and Fees emulator, so the facilities for determining if an access is in the context of a containing generic access object, and for subsequently getting the details of that object, are actually implemented by that software.

Other specific emulation strategies

Within the Practice Suite the general emulation strategies have not always proved adequate. Two specific extra emulation methods are provided:

- Record cache emulation caches an entire file. It is used for small files with low volatility where there is little point in doing anything any more complex.
- ClientCache emulation caches a subset of a file based on a partial key. This is used for tax files where the subset of data that is being worked on is typically well defined (i.e. all the tax information for a client for a given year).

Time and Fees emulation

Within the practice suite an alternative emulation strategy is used for files in the Time and Fees product. These are basically the files in the 5000 series. This emulation is not described here, but it is mentioned here so that the absence of Xml mappings for these files is explained. The salient facts about the Time and Fees emulator are:

- It holds its table mappings in store compiled into the emulator itself.
- The Time and Fees SQL implementation is not 1:1 file:table. The database has been partially restructured.
- There is some relational integrity on the Time and Fees database.
- Time and Fees cache timeout is on the basis of time since data read, not time since data last accessed.
- The applications calls the Time and Fees emulator via the standard emulator, there is no separate interface.
- The Time and Fees emulator does not handle database connections, it gets its connection pointer from the standard emulator.

EXCHEQUER SPECIFICS

Computed columns

BTRIEVE allows you to index on keys made up of sets of fields, where a field is defined as any contiguous row of bytes, of any length, within the record. BTRIEVE allows you to interpret each such field as one of a fixed set of pre-defined data types.

Because BTRIEVE knows nothing about the content of the record, other than the keys, there is actually no requirement that the key 'fields' bear any relationship to the data 'fields' that go to make up a record. For example:

- It is possible to define a key to index on *part* of a character string that occurs in a record.
- It is possible to define a key to index on a set of characters as though they were an integer. (This may be unlikely for lengths greater than 1, given Intel byte reversal of integers, but older systems often play quite clever low level tricks).

Techniques of this form are used by Exchequer. The strategy adopted to cope with this is:

- The columns in the table are defined to match the fields in the record layout.
- Additional computed columns are added to represent the fields required by the key(s). Thus SQL server will generate these column values on the fly without user action.

The emulator still needs to know about the computed columns. Although they are never returned to the user in the *record* buffer, they are returned in *key* buffers, and equally may be supplied as a key by the calling code.

Get by Position

Within a BTRIEVE file every record has a unique 'position id'. BTRIEVE provides APIs to get the position id of a record, and to re-read the record by position id once known.

Position ids are not guaranteed to remain fixed forever. In particular, certain file re-organisation and recovery procedures may result in them being changed. It is therefore not good practice to persist position ids to records as foreign keys in other files.

However, access by position id (once known) is more efficient than access using a normal key. This is because a position id is effectively a direct address of the record in the file. No accesses to index blocks are required.

Exchequer uses position id access extensively.

The fact that position ids are not guaranteed to remain fixed allows an emulation strategy that does not seek to replicate the position ids in the current database, but merely has to provide a similar unique key to individual records.

The ideal candidate for this is the SQL 'identity' column, and this is what is used. Exchequer tables have identity columns that are invisible to the BTRIEVE interface, except to those APIs that work with position ids.

Multi company operation and mapping to schemas

An outline of the mechanisms for multiple 'databases' was given in the Administration and Configuration section previously in this document. This section expands this in the context of multi company operation within Exchequer.

There is one entry in the logical database table (IrisDatasetConnection) for each company.

- By this means each company maps to a separate SQL schema with, by convention, a schema name matching the Exchequer company code.
- Each such schema contains an identical set of tables, so that the Xml data mapping details (also held in the common schema) are appropriate for all company schemas.
- Because connection credentials are held within the IrisDatasetConnection table each company schema can be separately restricted.

'Company' schemas are set up by code when the user creates a new company.

Multi threading and database connections

The Exchequer application is multi-threaded. In practice it is not a complex multi threaded system, there is typically a U/I thread, monitoring and processing actions at the interface, and a worker thread for executing significant tasks in background.

This has implied a certain amount of care when mapping accesses across to ADO. ADO does not like getting SQL connections from one thread and then operating on them in a different one.

In BTRIEVE, separate streams of actions of this form are differentiated by use of a 'client id' at the API. The client id is allocated by the caller and used to separate each logical set of actions. Thus, for example, there is one 'current' record per file per client id.

Within the emulation environment sufficient objects are created to keep this level of distinction. The emulator also cross checks that objects that are created on one thread are not used on a different one.

The original Exchequer code was in the habit of opening a file in the U/I thread and passing the pos block (effectively the control block for BTRIEVE access on that file) down to the worker thread. Whilst this worked fine in the raw BTRIEVE environment it produced failures within the emulation environment because it generated actions on the same connection in different threads. The Exchequer application has now been changed so that it no longer does this.

The only other implication of multi-threading is that emulation code at the outer levels (i.e. when initially translating the API request to locate the correct underlying objects to process it) must be thread safe. This is achieved conventionally by the use of suitable critical region semaphores covering operations that would otherwise be at risk.

Zero time out locks

This has been mentioned previously in the document, but is worth emphasising.

Exchequer makes extensive use of the BTRIEVE NOWAIT lock modes. In this type of operation the code expects an attempt to read a locked record to return immediately with a failure response, rather than waiting until the lock can be acquired.

This is not default SQL behaviour. However, it is possible to get this behaviour by resetting the lock timeout interval on an SQL connection to zero. The *slight* snag is that, once you have done this you cannot set the timeout back to default on that connection. This means that, once a connection has been set to work in this way, all subsequent operations will continue to work thus.

This is of relevance because the subsequent operations (typically an update of the locked record) may themselves need to acquire more locks. This can be for a number of reasons:

- The initial pessimistic lock acquired when the row was selected only covers the row. It does not cover the page that the row exists in, nor does it cover the pages that the index entries for the row exist in. The former will be required by an update. The latter will be required if the update changes any indexed column value, and will always be required by a delete.
- SQL has a lock escalation mechanism. If the code appears to be taking a large number of locks on a table (not necessarily by virtue of explicit locking) SQL can escalate the collection of locks to a single table lock.

Acquisition of any of these implicit locks can fail when lock timeout is set to zero. This means that, in this mode of working, an update or delete can fail with a lock timeout.

It's also true that, in this case, there's nothing actually wrong with the update or delete. Given default behaviour it would have delayed slightly, until it could get access to the pages it needed, and then succeeded.

This gives us the strategy for coping with this case. We detect the failure, delay slightly and then retry. This, clearly, does not go on for ever, there is a limit on the number of times we are prepared to do this before deciding that the failure should be treated as such and returned to the invoking code.

There is a further slight complication in the delete case. Although resubmitting the SQL is valid in principle, it does not appear to work when submitted as a delete through an ADO record set. For this reason the retry in the delete case is accomplished by submitting SQL text via the ADO command object, rather than retrying the operation on the record set.

In practice this 'retry' strategy seems to work a lot of the time. However, at least one case has been met in which the conversion of internal 'intent' locks, within SQL server, to exclusive locks (when processing an update) has resulted in an internal deadlock situation. This arises because the entire purpose of 'intent' locks is to introduce a delay into update processing so that multiple updates of the same page (but for different rows) are processed serially. Having SQL server return immediately, rather than waiting, seems to completely upset this processing, event with an external retry mechanism.

Exchequer Stored Procedures

There were certain processes within Exchequer that performed poorly under emulation (in terms of performance). This is true of the practice suite as well, but a difference in the Exchequer case is that significant core update processing has been affected.

The solution has been to re-write the processing in question as stored procedures, and these procedures now include significant critical business logic for the application.

Responsibility for maintaining the stored procedures does not fall under emulation support. It actually resides with the Exchequer application team (which is correct, given the significant level of business logic involved).

The emulator does, however provide an encapsulation to allow the stored procedures to be called. This is primarily to make life easier for the Delphi code that has to call them.

SQL Emulator

How I got it debuggable..	

1) Install OS
2) Install IIS (might as well)
3) Install OS SPs
4) Install VS.NET 2008 + SP1
5) Install SQL Management Studio
6) Uninstall SQL Server 2005 Express
7) Install SQL Server 2005 Express SP3
8) Install Exchequer SQL
9) Copy soultion and all files into C:\Working\Exchequer SQL Emulator\trunk
10) Select all the project files from solution explorer in VS.NET and perform clean projects
11) Select all the project files from solution explorer in VS.NET and perform full build
12) Find all built dlls and exes from vs.net solution and copy into root exchequer directory
13) Run exchequer and log in.
14) Attach vs.net to enter1 process
15) Pat yourself on the back, you've done it.