S/W Support Document

Overview of the Standard Iris SQL Emulator

Thursday, June 24, 2010
Version 1 Draft

Prepared by
Dave Elliott

Senior Architect - Practice Architecture

Revision and Signoff Sheet

Change Record

Date	Author	Version	Change reference
2/10/08	Dave Elliott	.1	Initial draft for review/discussion

Reviewers

Name	Version approved	Position	Date
Manjit Sagoo		Head of –Architecture Team – Practice S/W	

Table of Contents

Introduction and History

The 'current' generation of the Iris Practice Suite was developed mainly during the first half of the 1990s, on hardware that was (relative to today's machines) quite low powered, and for the MS-DOS operating system.

By the end of the decade the Practice Suite (still using the same architecture) had expanded by the addition of further products and reached about 2 million lines of C and C++ code.

The architecture of the product reflected the architectural thinking of its origin, and the constraints of the size of machines it was to run on. In particular, both U/I and data access tend to be embedded within the main code paths.

At the time that the technology stack for the products was established, although there were SQL environments available on microcomputers (Ingres, Informix etc) they were quite limiting, both in terms of the restrictions that they placed on the size of the system and the performance that they offered. Accordingly Iris used the dominant ISAM file handler of the time, which was BTRIEVE.

When, later, the strategic decision was taken to shift the data storage technology to an SQL base (in this case SQL server) it posed the problem of how to make the transition. It was deemed impractical to re-write the entire Practice Suite (because of its size) so an emulation approach was chosen. The ideal objective of this approach is that the application can continue to work as if it were still using BTRIEVE and an intermediate emulation layer translates BTRIEVE calls into actions on the SQL database.

This approach has advantages in terms of the effort required and, probably more importantly, stability of the core application on which the business is based. It was (and continues to be) the transition strategy proposed by Microsoft. (This, however, should always be seen in the light of Microsoft wishing to encourage such a transition).

The approach also has problems (which the original, *but not current*, Microsoft guidelines highlighted). These will emerge during the rest of this document, but generally revolve around performance issues and quite detailed differences in behaviour between the two products.

(Note that, for applications written against BTRIEVE files that are designed consistently with a relational implementation there is an alternative strategy of moving to Pervasive SQL, which is an SQL based database using BTRIEVE files as its physical layer. However, the 'relational' requirement is quite severe, and is not satisfied by Practice Suite files.)

The approach used for the Practice Suite emulation is generic, although only a subset of BTRIEVE functionality is implemented (defined by what is required by the Practice Suite itself). The transition problem that it aims to solve is not unique, and was also faced by another product in the Iris family (i.e. Exchequer). The same basic software was therefore used for emulation of this product.

Initially only a single set of emulation software was used for the two cases. However, there are sufficient differences between the subsets of functionality required for the two products that the continued development of one had the real possibility of upsetting the stability of the other. The single emulator was therefore split into two separate ones which, thereafter, followed separate development paths.

This document gives the background to this emulation layer. It also gives a brief description of BTRIEVE itself (which you can skim if you are already familiar with it – although it does make comments on the differences between BTRIEVE and SQL that are of relevance when considering the difficulties that emulation must face).

This document does not describe SQL, ADO or C/C++. You should definitely understand SQL in order to read this document and, if you become involved in maintenance or enhancement of the emulation software you will need to understand ADO and be happy reading and writing C++.

Whilst the main emulator is written in standard C++ there are some ancillary tools and functions that have been written in managed code. This is a mixture of C# and managed C++ (the latter to mitigate between the managed and unmanaged code).

BTRIEVE

What is BTRIEVE? What can/can't it do?

BTRIEVE is a file handler that supports ISAM (Indexed Sequential Access). It is **not** a database. In particular:

- All BTRIEVE APIs are in terms of connection to a single file, and every file is completely independent of all other files.
- There is no schema, and no containing software to handle management of the 'database' as a whole. In particular there are no backup and restore facilities provided by BTRIEVE itself.
- There is no separate query language for accessing BTRIEVE files, the only way to access them is via the applications API.

A BTRIEVE file equates 1:1 to a normal file, and appears in folders as such. Each BTRIEVE file is self contained (i.e. the data and any indexes supporting it reside in the same physical file).

The BTRIEVE API contains mechanisms for creating files, but none are required (or present) to delete them. The operating system facilities for deleting files are adequate.

From the point of view of the information visible at the API, a BTRIEVE file consists of a set of records. Beyond the information needed to derive key values from the information in the record, BTRIEVE has **no** knowledge of what is actually in any record. Correspondingly, apart from the 'key' fields there is no requirement that all records in a file are of the same format. (Indeed, an application could put completely different information in each record in a file if it wished).

Similarly, records on a file do not have to be all the same size.

Keys and position ids.

Whilst BTRIEVE does give functionality to allow an application to read though a file serially, it is more usual to access data on the basis of a key.

A key is an ordered set of fields, from within the record data, that define an index value for that record. The collection of all the index values for all records for a specific key gives an index that supports ISAM access.

BTRIEVE recognises different types of field when working out what a key means. For example it correctly deals with x86 byte reversal of binary numeric fields, and can recognise that characters after the null in a null terminated string are not significant.

The key structure is constant across all records in a single file. However, in practical terms, there can be as many separate keys associated with a file as you wish (subject to some overall large limit and the constraints of the performance hit associated with inserting new records with large numbers of keys).

Keys confer full ISAM access modes. These are best understood by thinking of a key as defining the *order* that records appear in the file. ISAM lets you move through the file (forwards or backwards) on the basis of this order. It also lets you search through the file for a particular position in the key sequence. Such a search can be for equality (i.e. all parts of the key match), but does not have to be. Thus, for example, there is an ISAM mode that will return the record with the lowest key that is greater than the key supplied for comparison.

It is not possible to read a BTRIEVE file in any *defined* order other than those defined by the keys.

There is no *simple* way of pre-filtering records returned by the file handler. The best that the application can normally do is to read a sequence of records (defined by a range of key values) and discard those that it isn't interested in.

Key values (for one particular key) *can* be forced to be unique within one file, but they do not have to be. Special access modes are provided to allow retrieval of the set of records that all have the same key value.

In addition to access by key, every record is considered to have a unique position id within its file. Once this is known it is possible to retrieve that record directly by this identifier.

Normal design patterns for ISAM file layout and access.

ISAM files have a long history, and, in their early incarnations, were quite primitive. The design patterns that have arisen round them exploit their strengths and seek to minimise use of system resources (like file connections). Whilst some of these approaches may appear without foundation in the current technological environment, this was not the case when they first evolved.

A common design pattern is to put all information relating to one thing into one file. Thus, to take the oft quoted example of an order processing system, the orders would be entirely contained in an orders file. This would hold the order header information *and* details of all the individual lines in the order. The key would be arranged so that the header would come first, followed by the individual lines, followed by the header for the next order.

If different order lines (in this example) required different information (because they were for radically different things) this would not be a problem, since the file manager doesn't actually know anything about the content of the record. Similarly, if a new type of order line came into being, whilst it would require code changes to cope with it, it could be implemented and issued with no change to the data file and with full backwards compatibility.

It would equally be possible to define another key that indexed the order headers by customer identifier (and probably date).

Another aspect that can be illustrated by this example is the occurrence of denormalised fields. Suppose that we wish to list orders by customer showing the order total for each. We can easily get all the order headers by customer, but to get the order totals we have to flip to use a different key and read the order item records. This might be inefficient or, more importantly, we'd have to write some code to do it. Hence, the simple convenience of getting at the information may lead us to put denormalised order totals on the order header record.

(Contrast this with the relational design which would see an Order table containing the order header, and an OrderItem table containing the individual order lines, with a foreign key relationship between them to tie the two together. Simple built in SQL aggregation functions will return the order totals without the need to hold them in denormalised form.)

Multiple record types

As the preceding example illustrates, a sequence of records on an ISAM file will frequently consist of a set of records *of varying formats*, but all relating to the same real world object. For example our order processing key *could* look like this:

Field	**Use**
Order number	**Unique number for the order as allocated by the system**
Item number	**Defined to be zero on order headers and non zero on order items. In the latter case it serves to define the sequence in which items appear on order.**
Item type	**Determines what type of item this order item is for (with the field being defined to be zero on an order header).**

The remainder of the data in the record may have differing formats depending on the values of the key items. Thus:

- If Item number = 0 it's a header format
- If item number > 0 it's an order item format, but you'll have to look at Item Type to decide precisely which.

This pattern is frequently referred to as a 'multi record type' file. Each of the different formats is referred to as a different record type. If you're lucky then there may be a single 'record type' field that will tell you what format to use. If you're not so lucky then you may have to apply an algorithm to one or more fields (as above).

Note two things:

- As you're reading through the order items you don't know what format each one will be until you've actually read it.
- You won't know that you've read all the items until you read the next record and discover that it's the header for the next order.

Other awkward things you can easily do on a BTRIEVE file include:

- Packing the record data to minimise its length, involving subsequent unpacking code after the record has been read.
- Putting arrays of values into the record.

The BTRIEVE API

There are two important facts to grasp about the BTRIEVE API

- Any 'get' operation (that reads data from the file) will only return a single record.
- Any update or delete operation will operate on the record that was last read on that file (specifically it does not pay any attention to any key values that may be around).

Contrast this with SQL which

- Typically returns a *set* of data that satisfies the criteria presented to it.
- Normally processes each SQL command independently of previous ones (so that update and delete determine what to operate on on the basis of the criteria given to them, not any previously successful operation). (Although updating an SQL database via an ADO RecordSet does bear some similarity to the BTRIEVE case.)

Thus BTRIEVE coding tends to be chatty when compared to the equivalent SQL code. For example, suppose that we wished to display an order (from the example above) on the screen.

The SQL data access would probably involve one singleton select to retrieve the order header plus any ancillary information (like the customer name and address details), followed by a select to create a recordset of all the order items (potentially joined with other secondary information such as a product description derived from a product code). That's two round trips to the database with the database engine providing all the brute force data acquisition and selection.

By contrast the BTRIEVE data access code would look something like this:

```
Get order header record from order file
Get related customer record from customer file
Get related address record from address file
While (
       (Get next record on order file not EoF)
       &&
       (Acquired record is for the correct order)
      )
{
  Get product information from product file
  Add item to results
}
```

Which is 4 + 2n separate calls, where n = the number of items in the order.

This gives the frustrating problem that is common to all emulation approaches. As the emulator processes a single API call, it has no knowledge of the context (beyond knowing what the current record on the file is, in case it is asked to do an update).

So, whereas the underlying database could efficiently assemble the information required, the poor old emulator is in no position to ask it to, because the only information it gets is at the low level API interface. If this proves inadequate in particular cases then the underlying objective of not changing applications code generally has to be compromised.

Use of one field to represent different types of data

Since all the weight of interpreting the data held in records falls to the application, the only restrictions on how that data is interpreted are those imposed by the language of implementation. In the case of the Practice Suite this is C/C++, in the case of Exchequer it is Delphi. Both are powerful languages, so the restrictions are essentially none.

Equally, both products are interacting with the data at a fairly physical level. Given their lengthy history they can also make assumptions about the encoding of character strings. Both products, in fact, assume ASCII and feel free to assume that a character will always be a byte. Both products will occasionally put non printable characters into strings (for specific applications purposes).

Within the body of a record (because BTRIEVE has no knowledge of any format therein) there is complete freedom to interpret any particular row of bytes however you wish. Even within the key there is considerable scope to still do this, so long as the field length remains the same and the key values (of whatever format) can continue to make sense to BTRIEVE in terms of the format that *it* believes they have.

This is not to say that there is anarchy in the application! The norm is that the layout of the record (in terms of fields) is well defined, but that the applications code will occasionally put values into fields that are apparently inconsistent with the type that the field is defined to have.

Other formatting issues and emulation strategies

There are several other formatting issues related to this freedom in interpreting values in records:

- BTRIEVE (or, at least, the original versions of it) does not recognise the concept of a 'null' value. Hence the application tends to adopt conventions in individual cases. For example, it is frequently (but not always) the case that zero is used to indicate that an integer 'foreign key' is 'not set'.
- The date formats used by the products are proprietary. In the case of Exchequer the format is a Delphi date, in the case of the Practice Suite it is an integer that is encoded/decoded by the Practice Suite software itself. Clearly, the desired target format on the database is an SQL date. Occasionally, the Practice Suite will use 'reserved' values (like -1) in these date fields to confer special meaning.
- For historical reasons the Practice Suite BTRIEVE files contain ASCII strings in IBM format, rather than ANSI. The differences all occur in top half of the encoding (i.e. with bit 7 set), but they *are* significant because they typically include the £ sign. The target SQL database is ANSI, so that 3rd party access will be correct, and the emulation layer is required to perform the relevant translations.
- BTRIEVE integer keys can be defined as signed or unsigned, and the key ordering will take due note of this. SQL server has no concept of unsigned integers, so could interpret the values wrongly if the most significant bit was set. (This could happen because one of the frequent use of 'unsigned' is when storing bitmasked flag fields). The solution is normally to define the SQL server field larger than is needed for the BTRIEVE field (e.g. int mapping to bigint).
- Both products sometimes use quite complex structures within records, and sometimes these are simply stored as image data, rather than translating them. In at least one case this has been forced on us by the limit of the number of columns that an SQL table can support. (It goes without saying that 'image' data is not amenable to being used in queries.)
- It is common practice to declare arrays of fields within BTRIEVE records. Since SQL does not support arrays the only solution is to map each array element as a separate field (unless the entire array is simply stored as an image).
- When BTRIEVE orders records by a key containing ASCII strings it does so on the basis of the binary values of the ASCII characters. This does not match default SQL server collation. Fortunately this one can simply be solved by