InterBase, Firebird and Blobs - A Technical Overview

By Paul Beach - IBPhoenix

Definition

InterBase pioneered the use of Blobs in relational database technology, an InterBase or Firebird Blob can be considered as an arbitrarily long sequence of bytes that can be used and manipulated whilst under transaction control. The name blob itself does not mean anything, it wasn't an originally defined as an acronym for Binary Large Object, or Basic Large Object, the name actually comes from the "B" film "The Blob", the Blob being a creature from outer space that ate large chunks of the USA.

According to Jim the sequence of Blob creation is as follows:
1. Barry Robinson, my boss at DEC, was prone to wandering around muttering "blobs, blobs, we gotta have blobs." When I asked what a blob was, he pointed out that I was the architect and that was my job.
2. Marooned in Colorado Springs (where Barry lived) because of a snow storm in Massachusetts (where I lived), and unable to derive the grand theory of transaction consistency, I invented the blob instead. Ah ha! A concept to hang on a wonder name!
3. The Rdb/VMS guys declared war on my sublime creation, the blob. First, they argued, they never lost a sale because they didn't have blobs. Second, documents and images didn't belong in databases; that's what sequential files were for. Third, if you did want to store a document, the right way to handle it was to normalize the lines. [No, I'm not making this up. Ask Ann. She almost never lies.] Finally, the term "blob" was unprofessional.
4. The retired bouncer hired to head the DSRI process rules that blobs were in, but had to be renamed "segmented strings" to avoid offending Clevelandites (or whatever).
5. Much later, Terry McKiever, an Apollo marketing gnome, fell in love with the concept of blob, but felt the blob needed to be an acronym. She started calling them "basic large objects." Apollo never private labelled the product, so this should be irrelevant. Unfortunately, her ravings got the attention of Informix (I think) who announced that they would support "binary large objects" at some future time. The damage was done.
6. Somebody asked a DEC product manager if and when Rdb/VMS would support blobs. "Sometime in the future" was the "product guidance." (The DEC, now Oracle, development group still know them as blob.).
7. Ashton Tate buys Interbase, Borland buys Ashton Tate. Some demented Borland type styles the humble blob "BLOb". Jim gags.

SQL Support SQLIII

SQL III is catching up to in the need for support of Binary Large Objects. SQL III defines the following datatypes:

- BLOB Binary Large Object BLOB
- CLOB Character Large Object BLOB SUB_TYPE TEXT
- NCLOB National Character Large Object BLOB SUB_TYPE TEXT CHARACTER SET <national>

Blob Sub-types

All blobs are sequences of bytes, however this sequence can have different interpretations. Firebird provides a sub-type declaration as a convenient mechanism to track the potential different uses for blobs. A blob can exist in a variety of forms, or sub-types of the general blob type. Knowing which sub-type of a blob field to use is useful when creating database applications that incorporate Firebird blob fields. The documentation refers to two pre-defined subtypes, however in actuality there are more. Blob fields come in a number of pre-defined varieties:

Sub-type 0 through sub-type 8 (the pre-defined sub-types), and then there are your own user-defined sub-types.

The pre-defined blob sub-types can be found in types.h

TYPE ("TEXT", 1,nam_f_sub_type)
TYPE ("BLR", 2,nam_f_sub_type)
TYPE ("ACL", 3,nam_f_sub_type)
TYPE ("RANGES", 4,nam_f_sub_type)
TYPE ("SUMMARY", 5,nam_f_sub_type)
TYPE ("FORMAT", 6,nam_f_sub_type)
TYPE ("TRANSACTION_DESCRIPTION", 7,nam_f_sub_type)
TYPE ("EXTERNAL_FILE_DESCRIPTION", 8,nam_f_sub_type)

Sub-type 0 blob fields are created by default when a CREATE command is issued and a sub-type is not specified. It is possible to explicitly indicate that the blob field is to be of sub-type 0 when a type 0 blob is being created. This sub-type of a blob field is typically used for the storage of binary data or data of an indeterminate type.

Firebird makes no analysis of the data stored in a blob, it just stores it in the blob field on a byte-for-byte basis. The most common intended use for blob fields in various applications is the storage of binary image data, typically for display e.g. for an image. Use the blob field sub-type 0 or create your own user-defined sub-type for this purpose.

Sub-type is 1.
This blob field sub-type is designed for the storage and manipulation of text. Typically, this is free-form memo or notes data. Typically you would use this blob field sub-type is for storing large amounts of text data. This is more convenient that a large VARCHAR because, unlike a VARCHAR, there is no 32K limit.

Sub-type is 2.
Used to store BLR (Binary Language Representation), Firebird's internal compiled language. All requests to the database engine are ultimately made in BLR. You can find this column definition where SQL or GDML syntax has been converted into BLR for the engine to use. E.g. triggers and stored procedures. Look in RDB$PROCEDURES for column RDB$PROCEDURE_BLR. This is what gets executed when the stored procedure runs, not the original SQL or GDML source code.

Sub-type is 3.
An ACL, or access control list, it was used to manage security for pre 4.0 databases.

Sub-type is 4
RANGES. Refresh ranges, comes from the PC semantic engine, allowing the declaration of a data range, rows, tables, pages to allow a client system to detect when the “data” has

changed. Not used.

Sub-type is 5.
Is a compiled, binary form of a relation's metadata to avoid expensive join processing when loading that metadata (includes C runtime format descriptors, etc.)

Sub-type is 6.
FORMAT Used in the system table rdb$formats. It describes the physical layout of record.

Sub-type is 7.
Contains the description of a multi-database (two phase commit) transaction that failed.

Sub-type is 8.
Not implemented. A way of storing additional information about an external file.

Another specialized type of blob is in fact an array. The blob mechanism is actually used to store both blob and array data.

In SQL the sub-type 1 blob is created by defining after the blob keyword the appropriate SUB_TYPE. e.g.

```
CREATE TABLE EMPLOYEES
(
NAME CHAR(20),
HISTORY BLOB SUB_TYPE TEXT
);
```

Or

```
CREATE TABLE EMPLOYEES
(
NAME CHAR(20) NOT NULL PRIMARY KEY,
HISTORY BLOB SUB_TYPE 1
);
```

As well as the predefined blob field sub-types, you can define your own sub-types. User-defined sub-types are designated by a negative integer value in association with the sub-type keyword. The actual integer value, that you use, is entirely arbitrary. But it should be negative e.g. use sub-type -1 to sub-type -32,678. The only issue, when creating and using user-defined sub-types is making sure that the same type of binary data is stored for every row in the table for a blob field of a given user-defined sub-type.

Firebird will not evaluate whether this criteria is met, and no error will be generated from the database if an incorrect type of binary data is stored in a user-defined blob field sub-type, but an application can have problems if it is expecting to find one type of binary data but finds something else.

```
CREATE TABLE IMAGE_DATA
(
FILENAME CHAR(12) NOT NULL PRIMARY KEY,
BMP BLOB SUB_TYPE -1,
JPEG BLOB SUB_TYPE -2
);
```

If you hadn't thought about it before, you now have a small understanding of how you would consider implementing blob filters that would allow the database to convert data

from one sub type to another (see the separate talk on this subject). The blob filter mechanism relies on knowing what the various sub-types are to provide its functionality. For example the ability that Firebird has that allows you to read and write to a textual blob by default is functionality that is implemented in the engine by a simple text blob filter.

When defining a blob and its sub-type, you can also define its segment length. A segment is a chunk of a blob, or a unit of a blob that can be written or read in a pre-determined amount. A text blob by default will automatically have a segment length of 80 bytes, historically the size of a terminal screen. The largest segment length that you can define is 32k or 32,767 bytes.

```
CREATE TABLE EMPLOYEES
(
NAME CHAR(20) NOT NULL PRIMARY KEY,
HISTORY BLOB SUB_TYPE 1 SEGMENT SIZE 80
);
```

The documentation generally refers to the fact that you must read or write to blobs a segment at a time. This is not specifically true, as there are internal utilities in Firebird that do make the manipulation of blobs a little easier. Equally there are functions for manipulating blobs as streams.

Blob Internal Storage

Blobs are created as part of a data row, but because a blob could be of unlimited length, what is actually stored with the data row is a blobid, the data for the blob is stored separately on special blob pages elsewhere in the database.

The blobid is an 8 byte value that allows Firebird to uniquely identify a blob and locate it. The blobid's can be either temporary or permanent, a temporary blob is one which has been created, but has not yet been stored as part of a table, permanent blobs have been stored in a table. The first 4 bytes represent the relation id for the blob (like data rows, blobs are bound to a table), the second four bytes represent the id of the blob within the table. For temporary blobs the relation id part is set to 0.

A blob page stores data for a blob. For large blobs, the blob page could actually be a blob pointer page, i.e. be used to store pointers to other blob pages. For each blob that is created a blob record is defined, the blob record contains the location of the blob data, and some information about the blobs contents that will be useful to the engine when it is trying to retrieve the blob. The blob data could be stored in three slightly different ways. The storage mechanism is determined by the size of the blob, and is identified by its level number (0, 1 or 2). All blobs are initially created as level 0, but will be transformed to level 1 or 2 as their size increases.

A level 0 blob, is a blob that can fit on the same page as the blob header record, for a data page of 4096 bytes, this would be a blob of approximately 4052 bytes (Page overhead - slot - blob record header).

Although the documentation states that the segment length does not affect the performance of Firebird, the actual physical size of a blob, or its segment length can become useful in trying to improve I/O performance for the blob, especially if you can size the segment (typically) or blob to a page.

This is especially true if you plan to manipulate the blob using certain low level Firebird Blob calls.

When a blob is too large to fit on a single page (level 1), and the data will be stored on one or more blob data pages, then the initial page of the blob record will hold a vector of blob page numbers.

A level 2 blob occurs when the initial page of the blob record is not big enough to contain the vector of all the blob data page numbers. Then Firebird will create blob pointer pages, i.e. multiple vector pages that can be accessed from the initial blob header record, that now point to blob data pages. The maximum size of a level 2 blob is a product of the maximum number of pointer pages, the number of data pages per pointer page, and the space available on each data page.

Blob pages are released when the blob is released, because the record that owns it is deleted or because the blob itself was modified

There are three different types of objects that use the blob storage mechanism. Segmented blobs are broken up into segments, or pieces of data (chunks). A segment is stored on disk as a two byte segment length followed by the segment. A stream blob on the other hand, is stored exactly as it was passed to the engine. An array is implemented as a stream blob, except that a description of the array is stored first, followed by the array data.

Manipulating Blobs

Above we defined a simple table containing a text blob, outputting blob data is easy, however inputting and updating text into that blob can be a little bit more complicated.....

In SQL a normal SELECT statement will automatically retrieve the textual data (and any other blob data for that matter). Use the command at the ISQL command line:

SET BLOB [ALL | n]

ALL will display all blob sub-types whilst n can be used to define a specific subtype.

However inserting and updating a blob in SQL is another matter, and the documentation states that this can only be done programmatically. E.g. by using a tool that automatically understands the basic blob sub-types e.g. Delphi or C++Builder, or by using specialist Firebird components, or by utilising a 3rd party User Defined Functions library that allows for some blob manipulation functionality.

Rather than try and go into all the various work-arounds, add-ins etc that there are available, the purpose of this paper is to look specifically at how blobs can be manipulated simply and programmatically.

But before we do that, it might be worth letting you all know, that you can put text directly into a blob on windows by doing the following.

At the DOS prompt.

SET EDITOR=edit

Remember to set ISC_USER and ISC_PASSWORD variables too.

```
QLI
QLI> ready employee.gdb
```

QLI> store employees
Enter name: Test

Up comes your editor, to allow you to insert history automatically.

On other platforms, just define the EDITOR to be your favourite editor e.g. vi, emacs etc.

The InterBase Programmers Guide, Language Reference and API guide all refer to the manipulation of blobs using embedded SQL and cursors that allow you to manipulate a blob a segment at a time.

All fairly complex and time consuming, mainly because the SQL standard still doesn't really support blobs. Please refer to the documentation on how this is done.

However before InterBase 4.0 there were a number of simple basic blob library routines available that would allow a developer to manipulate blobs fairly easily. These functions allowed the interchange of data between blob fields and operating system files.

Pre 4.0 Blob Routines

To compile the sample program, use the following:

1. Make sure you pre-compile the program using gpre. e.g

Gpre -c -either -manual -no sample.e

Gpre -c (extended C program), -either (accept upper case or lower case DML in C) -manual (do not automatically attach to the database -no (do not generate C debug lines) sample.e

This will create a .c file (sample.c), that can compiled by your favourite C/C++ compiler.

Sample Code to:

1. Load the contents of an operating system file into a blob: BLOB_load.
2. Dump the contents of a blob into an operating system file: BLOB_dump.
3. Read a blob field, dump it a file, and call an editor with the contents of a blob: BLOB_display.
4. Write or update the contents of a blob via an editor: BLOB_edit.

Simple Database Definition

```
SET SQL DIALECT 3;
/* CREATE DATABASE*/
Create database 'D:\testTest.gdb' PAGE_SIZE 1024
/* DEFAULT CHARACTER SET */
/* Table: BLOB_TEST, Owner: SYSDBA */
CREATE TABLE "BLOB_TEST"
(
"PKEY"  INTEGER NOT NULL,
"DATA"  BLOB SUB_TYPE 0 SEGMENT SIZE 80,
PRIMARY KEY ("PKEY")
);
```

Sample Code

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char* argv[])
{
char filename[40];
EXEC SQL WHENEVER SQLERROR GOTO ExitError;
EXEC SQL SET DATABASE DB1 = 'd:/test/test.gdb';
EXEC SQL BEGIN DECLARE SECTION;
BASED ON BLOB_TEST.PKEY pkey;
BASED ON BLOB_TEST.DATA blob_data;
EXEC SQL END DECLARE SECTION;
EXEC SQL CONNECT DB1 USER 'SYSDBA' PASSWORD 'masterkey';
printf("Database is openn");
EXEC SQL SET TRANSACTION;
printf("Starting a transaction\n");

/* BLOB_load Example */
STORE A IN BLOB_TEST USING A.PKEY = 1;
BLOB_load (&A.DATA, DB1, gds__trans, "a.txt");
END_STORE;

/* BLOB_dump Example */
FOR A in BLOB_TEST WITH A.PKEY = 1
BLOB_dump (&A.DATA, DB1, gds__trans, "b.txt");
END_FOR;

/* BLOB_display Example */
FOR A in BLOB_TEST WITH A.PKEY = 1
BLOB_display(&A.DATA, DB1, gds__trans, "");
END_FOR;

/* BLOB_edit Insert Example */
STORE A IN BLOB_TEST USING A.PKEY = 1;
BLOB_edit(&A.DATA, DB1, gds__trans, "edit");
END_STORE;

/* BLOB_Edit Update Example */
FOR A IN BLOB_TEST WITH A.PKEY = 1
MODIFY A USING
BLOB_edit(&A.DATA, DB1, gds__trans, "edit");
END_MODIFY;
END_FOR;

printf("n");

EXEC SQL COMMIT;
printf("Commit the transaction\n");

EXEC SQL DISCONNECT DB1;
printf("Close the database\n");
```

```
ExitError:
if (SQLCODE)
{
isc_print_sqlerror(SQLCODE, isc_status);
EXEC SQL ROLLBACK;
EXEC SQL DISCONNECT DEFAULT;
exit(1);
}

}
```

Blobs and Varchars

Varchars are limited to 32k in length whilst a blob is effectively unlimited in size, however the page size of a database will determine maximum size as determined by the

e.g.

- 1Kb page size => 64 Mb
- 2Kb page size => 512 Mb
- 4Kb page size => 4 Gb
- 8Kb page size => 32 Gb
- 16Kb page size => Big enough:-)

A normal row in Firebird is limited to max 64k, and a varchar is part of a row, so this limits the use of multiple varchar columns within a single table, wheras a blobid is only 8 bytes.

Varchars are fetched as part of the row in full, whilst blobs are fetched separately.

Varchars are compressed via RLE, blobs are not, unless you compress them yourself via some kind of filter mechanism.

Blobs can't be indexed, and access via search is limited to LIKE, STARTING and CONTAINING, also blobs cannot be compared using =, <,>, etc whilst varchars do support =, <.>, BETWEEN, IN as well as case sensitive LIKE and STARTING and case insensitive containing.

You cannot define collation sequences for Blobs.

Blobs do not support UPPER, CAST, MIN, MAX etc.

Blobs do not support sorting, as well as GROUP BY, DISTINCT, UNION, JOIN ON.

Blob columns cannot be concatenated.

To retrieve data from a table, you need to have the SELECT privilege.

To retrieve blob, you only need to know its blobid. But there isn't a check made against the table that the blob is assigned to, so if you know the blobid anybody can effectively read the blob.

Delphi and C++Builder

When defining TField objects for Firebird blob fields in Delphi or C++Builder, the various blob field sub-types are assigned TField derivative types as follows:

- Sub-type 0: TBlobField
- Sub-type 1: TMemoField
- User-defined: TBlobField

Because both the predefined sub-type 0 and user-defined sub-types are recognised as TBlobField objects, make sure that when you design an application that you do not confuse a field of one sub-type for that of another. The only way to differentiate between a field of sub-type 0 from that of a user-defined type is by viewing the metadata information for the relevant table.

Acknowledgements

Ivan Prenosil: http://www.volny.cz/iprenosil/interbase/ip_ib_strings.htm