Data Page - type 0x05

A data page belongs exclusively to a single table. The page starts off, as usual, with the standard page header and is followed by an array of pairs of unsigned two byte values representing the 'table of contents' for this page. This array fills from the top of the page (lowest address, increasing) while the actual data it points to is stored on the page and fills from the bottom of the page (highest address, descending).

The C code representation of a data page is:

```
struct data_page
{
    pag dpg_header;
    SLONG dpg_sequence;
    USHORT dpg_relation;
    USHORT dpg_count;
    struct dpg_repeat {
        USHORT dpg_offset;
        USHORT dpg_length;
    } dpg_rpt[1];
};
```

Dpg_header: The page starts with a standard page header. In this page type, the pag_flags byte is used as follows:

- *Bit 0* - dpg_orphan. Setting this bit indicates that this page is an orphan - it has no entry in the pointer page for this relation. This may indicate a possible database corruption.
- *Bit 1* - dpg_full. Setting this bit indicates that the page is full up. This will be also seen in the bitmap array on the corresponding pointer page for this table.
- *Bit 2* - dpg_large. Setting this bit indicates that a large object is stored on this page. This will be also seen in the bitmap array on the corresponding pointer page for this table.

Dpg_sequence: Four bytes, signed. Offset 0x10 on the page. This field holds the sequence number for this page in the list of pages assigned to this table within the database. The first page of any table has sequence zero.

Dpg_relation: Two bytes, unsigned. Offset 0x12 on the page. The relation number for this table. This corresponds to RDB$RELATIONS.RDB$RELATION_ID.

Dpg_count: Two bytes, unsigned. Offset 0x14 on the page. The number of records (or record fragments) on this page. In other words, the number of entries in the dpg_rpt array.

Dpg_rpt: This is an array of two byte unsigned values. The array begins at offset 0x18 on the page and counts upwards from the low address to the higher address as each new record fragment is added.

The two fields in this array are:

Dpg_offset: Two bytes, unsigned. The offset on the page where the record fragment starts. If the value here is zero and the length is zero, then this is an unused array entry. The offset is from the

start address of the page. For example, if the offset is 0x0fc8 and this is a database with a 4Kb page size, and the page in question is page 0xcd (205 decimal) then we have the offset of 0xcdfc8 because 0xcd000 is the actual address (in the database file) of the start of the page.

Dpg_length: Two bytes, unsigned. The length of this record fragment in bytes.

The raw record data is structured into a header and the data.

back to top of page

Record header

Each record's data is preceeded by a record header. The format of the header is shown below. Note that there are two different record headers, one for fragmented records and the other for unfragmented records.

```
// Record header for unfragmented records.
struct rhd {
    SLONG rhd_transaction;
    SLONG rhd_b_page;
    USHORT rhd_b_line;
    USHORT rhd_flags;
    UCHAR rhd_format;
    UCHAR rhd_data[1];
};

/* Record header for fragmented record */
struct rhdf {
    SLONG rhdf_transaction;
    SLONG rhdf_b_page;
    USHORT rhdf_b_line;
    USHORT rhdf_flags;
    UCHAR rhdf_format;
    SLONG rhdf_f_page;
    USHORT rhdf_f_line;
    UCHAR rhdf_data[1];
};
```

Both headers are identical up to the rhd_format field. In the case of an unfragmented record there are no more fields in the header while the header for a fragmented record has a few more fields. How to tell the difference?

See the details of the rhd_flags field below.

Rhd_transaction: Four bytes, signed. Offset 0x00 in the header. This is the ID of the transaction that created this record.

Rhd_b_page: Four bytes, signed. Offset 0x04 in the header. This is the record's back pointer page.

Rhd_b_line: Two bytes, unsigned. Offset 0x08 in the header. This is the record's back line pointer.

Rhd_flags: Two bytes, unsigned. Offset 0x0a in the header. The flags for this record or record fragment.

The flags are discussed below.

Flag name	Flag value	Description
rhd_deleted	0x01 (bit 0)	Record is logically deleted.
rhd_chain	0x02 (bit 1)	Record is an old version.
rhd_fragment	0x04 (bit 2)	Record is a fragment.
rhd_incomplete	0x08 (bit 3)	Record is incomplete.
rhd_blob	0x10 (bit 4)	This is not a record, it is a blob. This bit also affects the usage of bit 5.
rhd_stream_blob/rhd_delta	0x20 (bit 5)	This blob (bit 4 set) is a stream blob, or, prior version is differences only (bit 4 clear).
rhd_large	0x40 (bit 6)	Object is large.
rhd_damaged	0x80 (bit 7)	Object is know to be damaged.
rhd_gc_active	0x100 (bit 8)	Garbage collecting? a dead record version.

Rhd_format: One byte, unsigned. Offset 0x0c in the header. The record format version.

Rhd_data: Unsigned byte data. Offset 0x0d in the header. This is the start of the compressed data. For a fragmented record header, this field is not applicable.

The following only apply to the fragmented record header. For an unfragmented record, the data begins at offset 0x0d. Fragmented records store their data at offset 0x16.

Rhdf_f_page: Four bytes, signed. Offset 0x10 (Padding bytes inserted). The page number on which the next fragment of this record can be found.

Rhdf_f_line: Two bytes, unsigned. Offset 0x14. The line number on which the next fragment for this record can be found.

Rhdf_data: Unsigned byte data. Offset 0x16 in the header. This is the start of the compressed data for this record fragment.

back to top of page

Record data

Record data is always stored in a compressed format, even if the data itself cannot be compressed.

The compression is a type known as Run Length Encoding (RLE) where a sequence of repeating characters is reduced to a control byte that determines the repeat count followed by the actual byte to be repeated. Where data cannot be compressed, the control byte indicates that “the next n characters are to be output unchanged”.

The usage of a control byte is as follows:

- Positive n - the next n bytes are stored 'verbatim'.
- Negative n - the next byte is repeated n times, but stored only once.
- Zero - if detected, end of data. Normally a padding byte.

The data in a record is not compressed based on data found in a previously inserted record - it cannot be. If you have the word Firebird in two records, it will be stored in full in both. The same applies to fields in the same record - all storage compression is done within each individual field and previously compressed fields have no effect on the current one. (In other words, Firebird doesn't use specialised 'dictionary' based compression routines such as LHZ, ZIP, GZ etc.).

Repeating short strings such as abcabcabc are also not compressed.

Once the compression of the data in a column has been expanded, the data consists of three parts - a field header, the actual data and, if necessary, some padding bytes.

Obviously, when decompressing the data, the decompression code needs to be able to know which bytes in the data are control bytes. This is done by making the first byte a control byte. Knowing this, the decompression code is easily able to convert the stored data back to the uncompressed state.

The following section shows a worked example of an examination of a table and some test data.

back to top of page

A worked example

The shows an internal examination of a Firebird data page. For this very simple example, the following code was executed to create a single column test table and load it with some character data:

```
SQL> CREATE TABLE NORMAN(A VARCHAR(100));
SQL> COMMIT;

SQL> INSERT INTO NORMAN VALUES ('Firebird');
SQL> INSERT INTO NORMAN VALUES ('Firebird Book');
SQL> INSERT INTO NORMAN VALUES ('666');
SQL> INSERT INTO NORMAN VALUES ('abcabcabcabcabcabcabcabcd');
SQL> INSERT INTO NORMAN VALUES ('AaaaaBbbbbbbbbbCccccccccccccccDD');
SQL> COMMIT;

SQL> INSERT INTO NORMAN VALUES (NULL);
SQL> COMMIT;
```

We now have a table and some data inserted by a pair of different transactions, where is the table (and data) stored in the database? First of all we need the relation ID for the new table. We get this from RDB$RELATIONS as follows:

```
SQL> SELECT RDB$RELATION_ID FROM RDB$RELATIONS
CON> WHERE RDB$RELATION_NAME = 'NORMAN';
```

```
RDB$RELATION_ID
==============
           129
```

Given the relation ID, we can interrogate RDB$PAGES to find out where the pointer page (page type 0x04) lives in the database:

```
SQL> SELECT * FROM RDB$PAGES
CON> WHERE RDB$RELATION_ID = 129
CON> AND RDB$PAGE_TYPE = 4;

RDB$PAGE_NUMBER RDB$RELATION_ID RDB$PAGE_SEQUENCE RDB$PAGE_TYPE
=============== =============== ================ =============
            162             129                0             4
```

From the above query, we see that page number 162 in the database is where the pointer page for this table is to be found. As described above, the pointer page holds the list of all the page numbers that belong to this table.

If we look at the pointer page for our table, we see the following:

tux> ./fbdump ../blank.fdb -p 162

```
Page Buffer allocated. 4096 bytes at address 0x804b008
Page Offset = 663552l

DATABASE PAGE DETAILS
=====================
       Page Type:              4
       Sequence:               0
       Next:                   0
       Count:                  1
       Relation:               129
       Min Space:              0
       Max Space:              0
       Page[0000]:             166

Page Buffer freed from address 0x804b008
```

We can see from the above that this is indeed the pointer page (type 0x04) for our table (relation is 129). The count value shows that there is a single data page for this table and that page is page 166. If we now dump page 166 we can see the following:

```
tux> ./fbdump ../blank.fdb -p 166

Page Buffer allocated. 4096 bytes at address 0x804b008
Page Offset = 679936l

DATABASE PAGE DETAILS
=====================
       Page Type:              5
```

```
       Sequence:               0
       Relation:               130
       Count:                  6
       Page Flags:             0: Not an Orphan Page:Page has space:No Large
Objects

       Data[0000].offset:      4064
       Data[0000].length:      30

       Data[0000].header
       Data[0000].header.transaction:   343
       Data[0000].header.back_page:     0
       Data[0000].header.back_line:     0
       Data[0000].header.flags:   0000:No Flags Set
       Data[0000].header.format:
       Data[0000].hex:    01 fe fd 00 0a 08 00 46 69 72 65 62 69 72 64 a4 00
       Data[0000].ASCII:   .  .  .  .  .  .  .  F  i  r  e  b  i  r  d  .  .

       Data[0001].offset:     4028
       Data[0001].length:     35

       Data[0001].header
       Data[0001].header.transaction:   343
       Data[0001].header.back_page:     0
       Data[0001].header.back_line:     0
       Data[0001].header.flags:   0000:No Flags Set
       Data[0001].header.format:
       Data[0001].hex:    01 fe fd 00 0f 0d 00 46 69 72 65 62 69 72 64 20
                          42 6f 6f 6b a9 00
       Data[0001].ASCII:   .  .  .  .  .  .  .  F  i  r  e  b  i  r  d
                           B  o  o  k  .  .

       Data[0002].offset:     4004
       Data[0002].length:     24

       Data[0002].header
       Data[0002].header.transaction:   343
       Data[0002].header.back_page:     0
       Data[0002].header.back_line:     0
       Data[0002].header.flags:    0000:No Flags Set
       Data[0002].header.format:
       Data[0002].hex:    01 fe fd 00 02 03 00 fd 36 9f 00
       Data[0002].ASCII:   .  .  .  .  .  .  .  .  6  .  .

       Data[0003].offset:     3956
       Data[0003].length:     47

       Data[0003].header
       Data[0003].header.transaction:   343
       Data[0003].header.back_page:     0
```

```
       Data[0003].header.back_line:     0
       Data[0003].header.flags:    0000:No Flags Set
       Data[0003].header.format:
       Data[0003].hex:    01 fe fd 00 1b 19 00 61 62 63 61 62 63 61 62 63
                          61 62 63 61 62 63 61 62 63 61 62 63 61 62 63 64
                          b5 00
       Data[0003].ASCII:   .  .  .  .  .  .  .  a  b  c  a  b  c  a  b  c
                           a  b  c  a  b  c  a  b  c  a  b  c  a  b  c  d
                           .  .

       Data[0004].offset:     3920
       Data[0004].length:     36

       Data[0004].header
       Data[0004].header.transaction:   343
       Data[0004].header.back_page:     0
       Data[0004].header.back_line:     0
       Data[0004].header.flags:    0000:No Flags Set
       Data[0004].header.format:
       Data[0004].hex:    01 fe fd 00 03 20 00 41 fc 61 01 42 f7 62 01 43
                          f2 63 02 44 44 bc 00
       Data[0004].ASCII:   .  .  .  .  .     .  A  .  a  .  B  .  b  .  C
                           .  c  .  D  D  .  .

       Data[0005].offset:     3896
       Data[0005].length:     22

       Data[0005].header
       Data[0005].header.transaction:   345
       Data[0005].header.back_page:     0
       Data[0005].header.back_line:     0
       Data[0005].header.flags:    0000:No Flags Set
       Data[0005].header.format:
       Data[0005].hex:    01 ff 97 00 00 00 00 00 00
       Data[0005].ASCII:   .  .  .  .  .  .  .  .  .
```

```
Page Buffer freed from address 0x804b008
```

We can see from the above, the records appear in the order we inserted them. Do not be misled - if I was to delete one or more records and then insert new ones, Firebird could reuse some or all of the newly deleted space, so record 1, for example, might appear in the “wrong” place in a dump as above.

Note: This is a rule of relational databases, you can never know the order that data will be returned by a SELECT statement unless you specifically use an ORDER BY.

We can also see from the above that Firebird doesn't attempt to compress data based on the contents of previous records. The word Firebird appears in full each and every time it is used.

We can see, however, that data that has repeating characters - for example 666 and AaaaaBbbbbbbbbbCccccccccccccccDD - do get compressed - but records with repeating consecutive

strings of characters - for example abcabcabcabcabcabcabcabcd do not get compressed.

back to top of page

Examining the data

Looking into how the compression works for the above example is the next step.

Compressed data

Record number 4 has quite a lot of compression applied to it. The stored format of the record's data is as follows:

```
        Data[0004].offset:      3920
        Data[0004].length:      36
        Data[0004].header
        Data[0004].header.transaction:    343
        Data[0004].header.back_page:      0
        Data[0004].header.back_line:      0
        Data[0004].header.flags:    0000:No Flags Set
        Data[0004].header.format:
        Data[0004].hex:    01 fe fd 00 03 20 00 41 fc 61 01 42 f7 62 01 43
                           f2 63 02 44 44 bc 00
        Data[0004].ASCII:   .  .  .  .  .     .  A  .  a  .  B  .  b  .  C
                            .  c  .  D  D  .  .
```

If we ignore the translated header details and concentrate on the data only, we see that it starts with a control byte. The first byte in the data is always a control byte.

In this case, the byte is positive and has the value 0x01, so the following one byte is to be copied to the output. The output appears as follows at this point with ASCII characters below hex values, unprintable characters are shown as a dot:

```
        fe
         .
```

After the unchanged byte, we have another control byte with value 0xfd which is negative and represents minus 3. This means that we must repeat the byte following the control byte abs(-3) times. The data now looks like this:

```
        fe 00 00 00
         .  .  .  .
```

Again, we have a control byte of 0x03. As this is positive the next 0x03 bytes are copied to the output unchanged giving us the following:

```
fe 00 00 00 20 00 41
 .  .  .  .     .  A
```

The next byte is another control byte and as it is negative (0xfc or -4) we repeat the next character 4 times. The data is now:

```
fe 00 00 00 20 00 41 61 61 61 61
 .  .  .  .     .  A  a  a  a  a
```

Repeat the above process of reading a control byte and outputting the appropriate characters accordingly until we get the following:

```
fe 00 00 00 20 00 41 61 61 61 61 42 62 62 62 62 62 62 62 62 62 43
 .  .  .  .     .  A  a  a  a  a  B  b  b  b  b  b  b  b  b  b  C

63 63 63 63 63 63 63 63 63 63 63 63 63 63 44 44
 c  c  c  c  c  c  c  c  c  c  c  c  c  c  D  D
```

Note: I've had to split the above over a couple of lines to prevent it wandering off the page when rendered as a PDF file.

We then have another control byte of 0xbc which is -68 and indicates that we need 68 copies of the following byte (0x00). This is the 'padding' at the end of our actual data (32 bytes in total) to make up the full 100 bytes of the VARCHAR(100) data type.

You may have noticed that the two consecutive characters' DD did not get compressed. Compression only takes place when there are three or more identical characters.

back to top of page

Uncompressed data

The first record we inserted is 'uncompressed' in that it has no repeating characters. It is represented internally as follows:

```
Data[0000].offset:      4064
Data[0000].length:      30
Data[0000].header
Data[0000].header.transaction:    343
Data[0000].header.back_page:      0
Data[0000].header.back_line:      0
Data[0000].header.flags:    0000:No Flags Set
Data[0000].header.format:
Data[0000].hex:    01 fe fd 00 0a 08 00 46 69 72 65 62 69 72 64 a4
                   00
Data[0000].ASCII:   .  .  .  .  .  .  .  F  i  r  e  b  i  r  d  .
                    .
```

The offset indicates where on the page this piece of data is to be found. This value is relative to the

the start of the page and is the location of the first byte of the record header.

The length is the size of the compressed data piece and includes the size of the header as well as the data itself.

In the above, the record header details have been translated into meaningful comments. The data itself starts at the location labelled Data[0000].hex:.

When restoring this data to its original value, the code reads the first byte (0x01) and as this is a control byte (the first byte is always a control byte) and positive, the following one byte is written to the output unchanged.

The third bye is a control byte (0xfd) and as this is negative (-3) , it means that the next byte is repeated three times.

Byte 5 (0x0a) is another control byte and indicates that the next 10 bytes are copied unchanged.

Finally, the second to last byte is another control byte (0xa4) and is negative (-92) it indicates that the final byte (0x00) is to be repeated 92 times.

We can see that even though the actual data could not be compressed, Firebird has managed to reduce the VARCHAR(100) column to only a few bytes of data.

back to top of page

NULL

The final record inserted into the table is the one with no data, it is NULL.

The internal storage is as follows:

```
        Data[0005].offset:      3896
        Data[0005].length:      22

        Data[0005].header
        Data[0005].header.transaction:    345
        Data[0005].header.back_page:      0
        Data[0005].header.back_line:      0
        Data[0005].header.flags:    0000:No Flags Set
        Data[0005].header.format:
        Data[0005].hex:    01 ff 97 00 00 00 00 00 00
        Data[0005].ASCII:   .  .  .  .  .  .  .  .  .
```

We can see that in the record header, the transaction ID is different to the other records we inserted. This is because we added a COMMIT before we inserted this row.

The NULL data expands from the above to:

```
        ff 00 00 00 <followed by 102 zero bytes>
```

The first four bytes are the field header, the next 100 zeros are the data in the VARCHAR(100) field (actually, they are not data as a NULL has no data) and then two padding bytes.

back to top of page

NULL status bitmap

From the above description of how the fields appear when compressed and again, when uncompressed, we can see that each record is prefixed by a 4 byte (minimum size) NULL status bitmap. This is an array of bits that define the NULL status of the data in the first 32 fields in the record. If a table has more than 32 fields, additional bits will be added in groups of 32 at a time. A record with 33 columns, therefore, will require 64 bits in the array, although 31 of these will be unused.

As this example table has a single field, only one bit is used in the array to determine the NULL status of the value in the field, the bit used is bit 0 of the lowest byte (this is a little endian system remember) of the 4.

The bit is set to indicate NULL (or “there is no field here”) and unset to indicate that the data is NOT NULL.

The following example creates a 10 field table and inserts one record with NULL into each field and one with NOT NULL data in each field.

```
SQL> CREATE TABLE NULLTEST_1(
CON>    A0 VARCHAR(1),
CON>    A1 VARCHAR(1),
CON>    A2 VARCHAR(1),
CON>    A3 VARCHAR(1),
CON>    A4 VARCHAR(1),
CON>    A5 VARCHAR(1),
CON>    A6 VARCHAR(1),
CON>    A7 VARCHAR(1),
CON>    A8 VARCHAR(1),
CON>    A9 VARCHAR(1)
CON> );
SQL> COMMIT;

SQL> INSERT INTO NULLTEST_1 (A0,A1,A2,A3,A4,A5,A6,A7,A8,A9)
CON> VALUES (NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
SQL> COMMIT;

SQL> INSERT INTO NULLTEST_1 VALUES
('0','1','2','3','4','5','6','7','8','9');
SQL> COMMIT;
```

I have not shown the process for determining the actual data page for this new table here, but, in my test database, it works out as being page 172. Dumping page 172 results in the following output:

```
tux> ./fbdump ../blank.fdb -p 172
```

```
Page Buffer allocated. 4096 bytes at address 0x804c008
Page Offset = 704512l

DATABASE PAGE DETAILS
=====================
        Page Type:              5
        Sequence:               0
        Relation:               133
        Count:                  2
        Page Flags:             0: Not an Orphan Page:Page has space:No
Large Objects

        Data[0000].offset:      4072
        Data[0000].length:      22

        Data[0000].header
        Data[0000].header.transaction:   460
        Data[0000].header.back_page:     0
        Data[0000].header.back_line:     0
        Data[0000].header.flags:   0000:No Flags Set
        Data[0000].header.format: '' (01)
        Data[0000].hex:    02 ff ff d7 00 00 00 00 00
        Data[0000].ASCII:   .  .  .  .  .  .  .  .  .

        Data[0001].offset:      4012
        Data[0001].length:      57

        Data[0001].header
        Data[0001].header.transaction:   462
        Data[0001].header.back_page:     0
        Data[0001].header.back_line:     0
        Data[0001].header.flags:   0000:No Flags Set
        Data[0001].header.format: '' (01)
        Data[0001].hex:    2b 00 fc 00 00 01 00 30 00 01 00 31 00 01 00 32
                           00 01 00 33 00 01 00 34 00 01 00 35 00 01 00 36
                           00 01 00 37 00 01 00 38 00 01 00 39
        Data[0001].ASCII:   +  .  .  .  .  .  .  0  .  .  .  1  .  .  .  2
                            .  .  .  3  .  .  .  4  .  .  .  5  .  .  .  6
                            .  .  .  7  .  .  .  8  .  .  .  9

Page Buffer freed from address 0x804c008
```

Taking the first record where all fields are NULL, we can expand the raw data as follows, we are only interested in the first 4 bytes:

```
        Data[0000].hex:    ff ff 00 00 ..................
```

The first two bytes are showing all bits set. So this indicates that there is NULL data in the first 16 fields, or, that some of the first 16 fields have NULL data and the remainder are not actually present.

Looking at the NOT NULL record next, the first 4 bytes expand as follows:

```
        Data[0001].hex:    00 fc 00 00 ..................
```

Again, only the first 4 bytes are of any interest. This time we can see that all 8 bits in the first byte and bits 0 and 1 of the second byte are unset. Bits 3 to 7 of the second byte show that these fields are not present (or are NULL!) by being set.

Next, we will attempt to see what happens when a table with more than 32 fields is created.

In this case, I'm using a record with 40 columns.

```
SQL> CREATE TABLE NULLTEST_2(
CON>  A0 VARCHAR(1),  A1 VARCHAR(1),  A2 VARCHAR(1),  A3 VARCHAR(1),
CON>  A4 VARCHAR(1),  A5 VARCHAR(1),  A6 VARCHAR(1),  A7 VARCHAR(1),
CON>  A8 VARCHAR(1),  A9 VARCHAR(1), A10 VARCHAR(1), A11 VARCHAR(1),
CON> A12 VARCHAR(1), A13 VARCHAR(1), A14 VARCHAR(1), A15 VARCHAR(1),
CON> A16 VARCHAR(1), A17 VARCHAR(1), A18 VARCHAR(1), A19 VARCHAR(1),
CON> A20 VARCHAR(1), A21 VARCHAR(1), A22 VARCHAR(1), A23 VARCHAR(1),
CON> A24 VARCHAR(1), A25 VARCHAR(1), A26 VARCHAR(1), A27 VARCHAR(1),
CON> A28 VARCHAR(1), A29 VARCHAR(1), A30 VARCHAR(1), A31 VARCHAR(1),
CON> A32 VARCHAR(1), A33 VARCHAR(1), A34 VARCHAR(1), A35 VARCHAR(1),
CON> A36 VARCHAR(1), A37 VARCHAR(1), A38 VARCHAR(1), A39 VARCHAR(1)
CON> );
SQL> COMMIT;

SQL> INSERT INTO NULLTEST_2 (
CON>  A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,
CON> A10,A11,A12,A13,A14,A15,A16,A17,A18,A19,
CON> A20,A21,A22,A23,A24,A25,A26,A27,A28,A29,
CON> A30,A31,A32,A33,A34,A35,A36,A37,A38,A39
CON> )
CON> VALUES (
CON> NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
CON> NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
CON> NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
CON> NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
CON> );

SQL> INSERT INTO NULLTEST_2 VALUES (
CON> '0','1','2','3','4','5','6','7','8','9',
CON> '0','1','2','3','4','5','6','7','8','9',
CON> '0','1','2','3','4','5','6','7','8','9',
CON> '0','1','2','3','4','5','6','7','8','9'
CON> );
SQL> COMMIT;
```

Once again, the test data is a simple pair of records, one with all NULLs and the other with all NOT NULL columns. The first record, all NULLs, dumps out as follows:

```
        Data[0000].hex:    fb ff 80 00 de 00 00 00 00
```

Decompressing the above, gives the following

```
        Data[0000].hex:    ff ff ff ff ff 00 00 00 00 .... 00
```

It is difficult to tell from the all NULL record where the NULL bitmap array ends and the real data begins, it's easier in the NOT NULL record as shown below, however, the first 8 bytes are the interesting ones. We have defined the record with more than 32 fields, so we need an additional 4 bytes in the bitmap, not just 'enough to hold all the bits we need'.

The NOT NULL record's data is held internally as:

```
        Data[0001].hex:    f8 00 7f 01 00 30 00 01 00 31 00 01 00 32 00 01
                           00 33 00 01 00 34 00 01 00 35 00 01 00 36 00 01
                           00 37 00 01 00 38 00 01 00 39 00 01 00 30 00 01
                           00 31 00 01 00 32 00 01 00 33 00 01 00 34 00 01
                           00 35 00 01 00 36 00 01 00 37 00 01 00 38 00 01
                           00 39 00 01 00 30 00 01 00 31 00 01 00 32 00 01
                           00 33 00 01 00 34 00 01 00 35 00 01 00 36 00 01
                           00 37 00 01 00 38 00 01 00 39 00 01 00 30 00 01
                           00 31 20 00 01 00 32 00 01 00 33 00 01 00 34 00
                           01 00 35 00 01 00 36 00 01 00 37 00 01 00 38 00
                           01 00 39
```

And this expands out to the following, where again,. we only need to look at the first 8 bytes:

```
        Data[0001].hex:    00 00 00 00 00 00 00 00 01 00 30 00 01 00 31 00
.....
```

Again, this makes it difficult to determine where the data starts and where the bitmap ends because of all the zero bytes present at the start of the record, so a sneaky trick would be to insert a NULL in the first and last columns and dump that out. This results in the following, when expanded:

```
        Data[0002].hex:    01 00 00 00 80 00 00 00 00 00 00 00 01 00 31 00
.....
```

The first field in the record is NULL and so is the 40th. The bit map now shows that bit 0 of the first byte is set indicating NULL and so is bit 7 of the fifth byte. Five bytes equals 40 bits and each field has a single bit so our number of bits matches up to the number of fields in each record.