The Oracle Data Block

27th October 2010

David Litchfield
(david@v3rity.com)

Introduction

Before delving into the depths of Oracle block internals, for those who are not fully comfortable with Oracle terminology we'll quickly cover some of the terms that will be used in this document. Firstly, there's the "block". Oracle data files are split into fixed size blocks and the data of each row is stored in such blocks. Blocks exist for other database objects too such as indexes and so on. An "extent" is made up of multiple blocks and these blocks are contiguous. A "segment" is made up of multiple extents but these may or may not be contiguous. Lastly, let's look at a "cluster". A cluster is a way of grouping the data of two or more tables together than share one or more columns. Oracle achieves this by storing the data from each table in the same blocks. This way, queries that require data from the two closely linked tables are speeded up. Many of the Oracle's metadata tables are stored as clusters – for example the USER$ and TSQ$ tables are stored together in the C_USER# cluster.

The Oracle Data Block

Each data block has a header, free space, used space, where rows of data can be found, "freed" space (after updates) which could be called slack or deadspace – and lastly a tail which contains a checksum of the block. The header contains many sections:

Name	Size
Block Header	20 bytes
Transaction Header	24 bytes
A list of ITLs	Each entry in list is 24 bytes
Data Header	14 bytes
A list of Table Directories	Each entry in list is 4 bytes
Row Directory	Variable length of 2 bytes offsets

The Block Header Structure

The block header structure contains general information about the block and has a fixed size of 20 bytes and takes the following form:

```
struct block_header
{
      unsigned char type;
      unsigned char format;
      unsigned char spare1;
      unsigned char spare2;
      unsigned int relative_block_address;
      unsigned int scn_base;
      unsigned short scn_wrap;
      unsigned char sequence;
      unsigned char flags;
      unsigned short checksum;
      unsigned short spare3;
}
```

There are several different kinds of data block. Whilst most of this document details blocks of type data (6), there are some more types which are of interest to this

document. Type 11 denotes a data file header block. The second block of every data file is of this type. Blocks of type 38 are System Managed Undo header blocks and contain an extent map point to extents in the segment and each block of the extents have a type of 2 – an undo data block. Undo is discussed in [1]. A full list of block types can be found in [2]. The second byte details the format the block uses and appears to be split into two halves each of 4 bits. The least significant 4 bits indicate the format is Oracle 8 or later. The most significant 4 bytes are either 0 or 0x0A. The 4 byte big relative data block address can be found at the 5th byte in. It holds both the block number and the file number. After the data block address can be found the System Commit Number of SCN. This SCN maps to a particular snapshot of the database and can be mapped to a timestamp. The SCN to timestamp mapping is stored in a table called SMON_SCN_TIME table. More details are discussed in [3]. Whenever changes are written to a data block the SCN is updated to reflect the current SCN. This can be useful for determining when a data block was last changed however it is important to note that the SCN is also updated when block cleanout occurs. This can be verified by checking the cleanout SCN in the transaction header.

The Transaction Header Structure

The transaction header can be divided into two main sections the fixed size part and the variable sized list of ITLs – or Interested Transactions. Put very simply, a Interested Transaction is a transaction that wants to change data in the block and so needs to lock the data. As noted the list of interested transactions is variable in size but this relates to how many there are; each entry in the list 24 bytes big. The fixed transaction header is 24 bytes big and takes the following form:

```
struct transaction_header
{
      unsigned char type;
      unsigned char spare1;
      unsigned char spare2;
      unsigned char spare3;
      unsigned int object_id;
      unsigned int cleanout_scn_base;
      unsigned short cleanout_scn_wrap;
      unsigned char spare4;
      unsigned char spare5;
      unsigned short number_of_itl_slots;
      unsigned char flags;
      unsigned char itl_tx_freelist_slot;
      unsigned int address_of_next_block_on_freelist;
}
```

The first byte of the header indicates the type: this is either 1 for data or 2 for an index and starting at the 5th byte, the 4 byte object ID can be found. Also of note is the number of ITL slots found 17 bytes into the header. The number in here determines the size of the variable portion of the transaction header. If the value is 2 then there will be two slots each of 24 bytes making the total size of the transaction header 72 bytes. If there are 3 the size will be 96 bytes and so on. Each slot takes the following form:

```
struct transaction_slot
{
```

```
        struct transaction_id
        {
              unsigned short undo_segment_number;
              unsigned short slot; // in Rollback Segment table
              unsigned int sequence;
        }
        struct undo_block_address
        {
              unsigned int dba;
              unsigned short sequence;
              unsigned char record_number;
              unsigned char spare1;
        }
        unsigned short flags; // KTBFTAC, KTBFUPB, KTBFIBI, KTBFCOM
        unsigned short _ktbitun;
        unsigned int base;
}
```

If any of these slots are “in use” then what can be useful here is the location of the undo block address which will hold previous values before the transaction.

Immediately after the last transaction slot is the data header.

The Data Header Structure

The data header structure is 14 bytes big and contains information about… and takes the following form

```
struct data_header
{
      unsigned char flags;
      char number_of_table_directories;
      short number_of_rows;
      short offset_to_freespace_start;
      short offset_to_freespace_end;
      short available_space;
      short available_space_after_txns_commit;
}
```

The first byte of the structure is used as a set of flags:

N	0x0001	(KDBHFNI)	pctfree hit
F	0x0100	(KDBHFNF)	do not put on free list
K	0x1000	(KDBHFFK)	flushable cluster keys

When the N flag is set it means that there’s no more space in the block and it should be removed from the free list. When F is set the block should not be placed in the free list. The second byte details the number of table directories. In non-clustered tables this is 1 but for clustered tables this may be more than 1. At the third byte is a short that holds the number of rows in the block. At byte 7 is a short that holds the offset to the start of freespace in the block. This offset is measured from the start of the data header structure. At byte 9 is another short that holds the offset to the end of the freespace; again this is measured from the start of the data header structure. The last two shorts detail available space.

After the data header is the list of table directories. If the table is non-clustered there is only one; however with clustered tables there could be more than one. If there are, each table directory contains an offset and a number of rows. This effectively divides each of the rows of data into groups, one for each table directory. A table directory takes the following form:

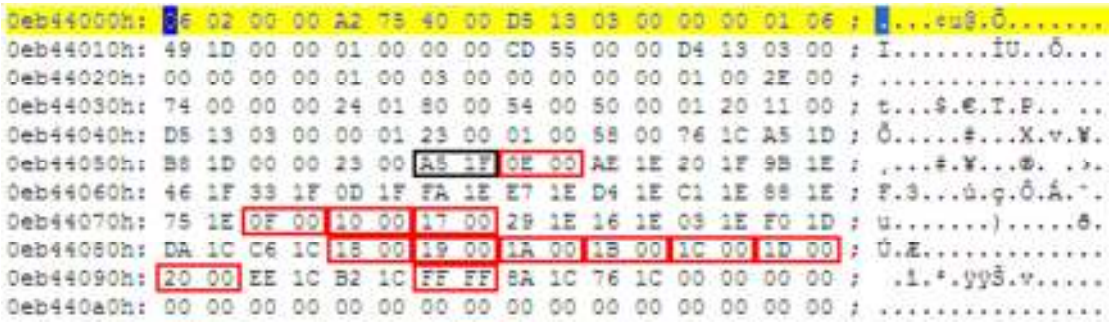
```
struct table_directory
{
      unsigned short offset;
      unsigned short number_of_rows;
}
```

The Row Directory

The row directory is a variable sized array of slots each 2 bytes in length. Each entry in the directory is an offset pointing to the start of a single row of data. This offset should be counted from the start of the data header structure. There is a caveat though. The value stored in the slot might not actually be an offset to a row but instead may be "free" and part of a "slot free linked list".

Slot Free Linked List

A row directory is divided into slots, each 2 bytes in length. When a row of data is added to the block the offset to where this data can be found in the block is placed in one of the row directory's slots. Thus this offset could be said to "point" to the row. When that row is deleted or updated and the space is "freed up" the slot may be marked as free. Free slots in a row directory are linked together in a list. Rather than containing an offset, a free slot contains the index into the row directory of the next free slot. The last free slot in linked list is denoted with a value of 0xFFFF. For example, in the hex dump below the row directory starts at 0xEB44056. The first entry in the row directory contains the value 0x1FA5, marked in the black box. This is the first slot and has an index number of 0. The next slot has a value of 0x000E, marked in red. Given that this "points" to a location inside the row directory we can tell it is a free slot. It tells us that the next free slot can be found at index 0x000E, i.e. 14. The value stored here is 0x000F, telling us that next free slot can be found at index 15. This linking continues until we get to the 0xFFFF. This tells us that this is the last free slot in the row directory.



When a dump of a data block is performed using the ALTER SYSTEM command such free slots are marked with SFLL – presumed to mean "Slot Free Linked List".

```
0xe:pti[0]      nrow=35 offs=0
0x12:pri[0]     offs=0x1fa5
```

```
0x14:pri[1]    sfll=14
0x16:pri[2]    offs=0x1eae
0x18:pri[3]    offs=0x1f20
0x1a:pri[4]    offs=0x1e9b
0x1c:pri[5]    offs=0x1f46
0x1e:pri[6]    offs=0x1f33
0x20:pri[7]    offs=0x1f0d
0x22:pri[8]    offs=0x1efa
0x24:pri[9]    offs=0x1ee7
0x26:pri[10]   offs=0x1ed4
0x28:pri[11]   offs=0x1ec1
0x2a:pri[12]   offs=0x1e88
0x2c:pri[13]   offs=0x1e75
0x2e:pri[14]   sfll=15
0x30:pri[15]   sfll=16
0x32:pri[16]   sfll=23
0x34:pri[17]   offs=0x1e29
0x36:pri[18]   offs=0x1e16
0x38:pri[19]   offs=0x1e03
0x3a:pri[20]   offs=0x1df0
0x3c:pri[21]   offs=0x1cda
0x3e:pri[22]   offs=0x1cc6
0x40:pri[23]   sfll=24
0x42:pri[24]   sfll=25
0x44:pri[25]   sfll=26
0x46:pri[26]   sfll=27
0x48:pri[27]   sfll=28
0x4a:pri[28]   sfll=29
0x4c:pri[29]   sfll=32
0x4e:pri[30]   offs=0x1cee
0x50:pri[31]   offs=0x1cb2
0x52:pri[32]   sfll=-1
0x54:pri[33]   offs=0x1c8a
0x56:pri[34]   offs=0x1c76
```

Finding the Row Directory

If you were writing a program to dump data from a block, you could calculate the start of the row directory as follows: Load the block into array and, given that the transaction header follows the fixed size block header, access block[36] and obtain the number of ITL slots. Multiply the number of slots by 24 and add this to the size of the block header (20) and the size of the fixed-size transaction header (24) – let's say this is *n*. This takes us up to the start of the data header and from here grab the number of table directories at block[*n*+1]. Multiply this by 4 to give *m*. Add *m* to *n* and also add the size of the data header structure, 14 and call this *k*. block[*k*] is the start of the row directory. To follow an entry to its data, add the entry to the start of the data header, *n*, so *n*+*offset* will take you to the row in question.

Row Header

Each row of data has its own header which, in its simplest form, looks as follows:

```
struct row_header
{
      unsigned char flags;
      unsigned char lock_status;
      unsigned char number_of_columns;
}
```

For non-clustered tables, the row header of a row has a "flag byte", a "lock byte" and a column count and is 3 bytes in length. The "flag byte", is used to denote information

about the row. The lock byte is used to denote information about the lock status of the row. The last byte, the column count, details the number of columns there are in the row. For clustered tables there is a 4th byte of the header, after the column count. For indexed clusters this byte is used for the cluster key index (cki in data file dumps) and for hash clusters it is used for the hash.

The Flag Byte

K = Cluster Key
C = Cluster table member
H = Head piece of row
D = Deleted row
F = First data piece
L = Last data piece
P = First column continues from previous piece
N = Last column continues in next piece

The Lock Byte
This byte indicates whether a row has been locked. This would happen if a transaction was attempting to change the row.

The Column Count

The Column Count indicates the number of columns that can be found in the row data. Note that this does not mean the number of columns in the table and so the column count may be less than the actual number of columns in the table. For example, assume there is a table with 3 columns. If an insert occurs into only the first column of this table then the row data will only have 1 column with the 2nd and 3rd columns, by their absence, having implicitly a NULL value. If however, an insert occurs in the 3rd column then an explicit NULL must be inserted for the 1st and 2nd columns.

Given that there is only 1 byte available for the column count this would indicate that tables can only have 255 columns as this is the maximum number a byte can hold; but tables in Oracle can have more than 255 columns so how is this dealt with at the block and row level? Well, the row is split into two or more pieces. For example, let's assume there is a table with 259 columns then the row data would be split over two rows or *pieces*. The first piece would have the Head Piece and First Data Piece flags set in the "Flag byte". Additionally, immediately after the row header is the Next Rowid (NRID). This NRID contains the block information as well as the row directory index number – a total of six bytes. Following the NRID the column data begins. The second (and last) piece would have the Last Piece flag set.

In the hex dump above, the red marked numbers are the first piece's header. Looking at the Flag byte (0x28) we can see that the Head Piece and First Data Piece flags are set. The Column Count indicates that there are 4 columns in this piece. The blue

numbers form the block number and the row directory index: this equates to block 30122, index 5. The four 0xFFs marked in grey are the 4 columns of the row – each 0xFF denoting a NULL value. The 3 bytes marked in black show the header of the next piece. The Flag Byte (0x04) indicates that only the Last Piece flag is set. Further, the Column Count (0xFF) indicates that there are 255 columns. Added to the 4 from the first piece this makes 259 – the 259 columns of the table. Note that, in this example, it just so happens that the second piece "follows" the first piece but this may not always be the case and the NRID should always be checked to locate the second piece.

The Row Data

After the row header comes the data for the row, but each column within the row is preceded by a byte specifying the length of the column. This means, the byte immediately after the row header denotes the length of the first column. There are a few caveats though. If the value of the byte is 0xFF (255) it means the column is NULL – in other words there is no data for this column of the row. Further, if the value of this byte is 0xFE (254) then it indicates that the column is longer than 255 bytes and the next two bytes will contain the length of the column. Regardless of length, let's say it's *p*, the next column of the row starts once *p* bytes have been read and the next byte is the size of the next column. For example, let's say we have 4 columns in a row – the first is 3 bytes in length, the second column is 5 bytes in length, the third is NULL and the fourth is 2 bytes in length. In this case the row would be mapped out as follows:

Bibliography

[1] http://www.databasesecurity.com/dbsec/oracle-forensics-6.pdf
[2] http://www.juliandyke.com/Internals/BlockTypes.html
[3] http://www.databasesecurity.com/dbsec/oracle-forensics-scns.pdf