Structure of a data page

By Paul Beach

(With thanks to Dave Schnepper and Deej Bredenberg) A database is considered to be a collection of pages, each page has a pre-defined size, this size is determined when the database is created by a database parameter that is passed in the isc_database_create call (gds_dpb_page_size). Pages are identifed by a page number (4 byte unsigned integer), stating at 0 and increasing sequentially from the beginning of the first database file to the end of the last database file.

Page 0 of a database is always the database header page, which contains the information that is needed when you attach to a database. Page 1 is the first PIP page (Page Inventory Page) and the first WAL page is always page 2. By convention, page 3 is the first pointer page for the RDB$PAGES relation, but that location is described on the header page so it could (in theory) change.

Except for the header page there is no specific relationship beetween a page number and the type of data that could be stored on it.

The types of pages are defined in ods.h and are as follows:

```
#define pag_header 1            /* Database header page */
#define pag_pages 2             /* Page inventory page */
#define pag_transactions 3      /* Transaction inventory page */
#define pag_pointer 4           /* Pointer page */
#define pag_data 5              /* Data page */
#define pag_root 6              /* Index root page */
#define pag_index 7             /* Index (B-tree) page */
#define pag_blob 8              /* Blob data page */
#define pag_ids  9              /* Gen-ids */
#define pag_log  10             /* Write ahead log information */
```

Pages are located in the database by seeking within the database file to position page_number*bytes_per_page. The structure of a data page, as defined in ods.h is as follows:

All pages have a page header, the page header consists of,

```
typedef struct pag {
   SCHAR pag_type;
   SCHAR pag_flags;
   USHORT pag_checksum;
   ULONG pag_generation;
   ULONG pag_seqno;     /* WAL seqno of last update */
   ULONG pag_offset;    /* WAL offset of last update */
} *PAG
```

1	**2**	Length, bytes	Description
pag_type	Page Type	1	=pag_data

pag_flags	Page Flags	1	e.g. Data page is orphaned (it doesn't appear on any pointer page), page is full, or a blob or an array exist on the page.
pag_checksum	Page Checksum	2	Always 12345 for known versions.
pag_generation	Page Generation	4	how many times has the page been updated.
pag_seqno	Page Sequence Number	4	WAL sequence number of last update, unused.
pag_offset	Page Offset	4	WAL offset of last update, unused.

The remainder of the page (less the 16 bytes above) is used to store page-specific data.

A data page holds the actual data for a table, and a data page can only be used by a single table, i.e. it is not possible for data from two different tables to appear on the same data page. Each data page holds what is basically an array of records (complete or fragmented). Below the header is 8 bytes of:

- Page Sequence (dpg_sequence 4 bytes) sequence number of the data page in a table, used for integrity checking.
- Page's Table/Relation id (dpg_relation 2 bytes) this id is also used for integrity checking.
- Number of Records or record fragments that exist on the data page (dpg_count 2 bytes).

This is then followed by an array of descriptors each of the format: offset of record or fragment, length of record or fragment. This descriptor describes the size and location of records or fragments stored on a page. For each record or fragment that is stored on the page there is an equivalent record descriptor at the top of the page. As records get stored the array grows down the page, whilst the records or fragments are inserted backwards from the end of the page. The page is full when they meet in the middle.

```
typedef struct dpg {
     struct pag dpg_header;
     SLONG dpg_sequence;    /* Sequence number in relation */
     USHORT dpg_relation;   /* Relation id */
     USHORT dpg_count;      /* Number of record segments on page */
     struct dpg_repeat
     {
     USHORT dpg_offset;     /* Offset of record fragment */
     USHORT dpg_length;     /* Length of record fragment */
     }  dpg_rpt [1];
 } *DPG;
```

Obviously data records can vary in size, so the number of records that may fit on a page can vary. Equally records may get deleted, leaving gaps on a page.

The page free space calculation works by looking at the size of all of the records that exist on a page. If space can be created on the page for a new record, then the records will get compressed i.e. shifted downwards to fill the gaps that would get created during normal insert, update and deletion of data. When the free space is less than the size of the smallest possible fragment - then the page is full.

A record may be uniquely identified by its record number (rdb$db_key).

The record header structure is,

1	Length, bytes	Description
rhd_transaction	4	Record header transaction. The transaction id that wrote the record.
rhd_b_page	4	Record header back pointer. Page number of the back version of the record.
rhd_b_line	2	Record header back line. Line number of the back version of the record.
rhd_flags	2	Record header flags. Possible flags are:
		• rhd_deleted - the record has been logically deleted, but hasn't yet been garbage collected.
		• rhd_chain - this record is an old version, a later version points backwards to this one.
		• rhd_fragment - the record is a fragment of a record.
		• rhd_incomplete - the initial part of the record is stored here, but the rest of it may be stored in one or multiple fragments.
		• rhd_blob - the record stores data from a blob.
		• rhd_stream_blob - the record stores data from a stream blob.
		• rhd_delta - the prior version of this record must be obtained by applying the differences to the data stored in this array.
		• rhd_large - this is a large record object such as a blob or an array.
		• rhd_damaged - the record is known to be corrupt.
		• rhd_gc_active - the record is being garbage collected as an unrequired record version.
rhd_format	1	Record header format. The metadata version of the stored record. When a record is stored or updated, it is marked with the current format number for that table. A format is a description of the number and physical order of fields in a table and the data type of each field.

When a field is added or dropped, or the datatype of a field is changed, a new format is generated for that table. A history of all of the formats for a table is stored in RDB$FORMATS. This allows the database to reconstruct records that were stored at any time based on the format that existed for the table at that time. Metadata changes, such as the above do not directly affect the records when the metadata change itself takes place, only when the records are actually next visited.

Record header data (hd_data size n as needed) is the actual record data and is compressed by RLE (Run Length Encoding). When a run takes place the compression algorithm will use 1 extra byte per 128 bytes, to represent the run length followed by one or more bytes of data. A positive run length indicates that the next sequence of bytes should be read literally, whilst a negative run length indicates that the following byte is to be repeated ABS(n) times.

```
typedef struct rhd {
    SLONG rhd_transaction;    /* transaction id */
    SLONG rhd_b_page;         /* back pointer */
    USHORT rhd_b_line;        /* back line */
    USHORT rhd_flags;         /* flags, etc */
    UCHAR rhd_format;         /* format version */
    UCHAR rhd_data [1];
} *RHD;
```