Firebird Internals

Inside a Firebird Database

Norman Dunbar, Mark Rotteveel

Version 1.5, 9 February 2026

Table of Contents

1. Introduction . . . 3
2. Database Structure . . . 4
2.1. Single File Databases . . . 4
2.2. Multiple File Databases . . . 4
2.3. Shadow Files . . . 4
3. Standard Database Page Header . . . 6
4. Database Header Page — Type 0x01 . . . 8
5. Page Inventory Page — Type 0x02 . . . 15
6. Transaction Inventory Page — Type 0x03 . . . 17
7. Pointer Page — Type 0x04 . . . 19
8. Data Page — Type 0x05 . . . 22
8.1. Record Header . . . 23
8.2. Record Data . . . 25
8.3. A Worked Example . . . 25
8.4. Examining The Data . . . 29
8.4.1. Compressed Data . . . 29
8.4.2. Uncompressed Data . . . 31
8.4.3. Null . . . 32
8.4.4. NULL status bitmap . . . 32
9. Index Root Page — Type 0x06 . . . 37
10. Index B-Tree Page — Type 0x07 — YOU ARE HERE. . . . 42
10.1. B-Tree Header . . . 42
10.2. Index Jump Info . . . 43
10.3. Index Jump Nodes . . . 44
10.4. Index Nodes . . . 44
10.5. Index Data . . . 44
11. Blob Data Page — Type 0x08 — TODO . . . 46
12. Generator Page — Type 0x09 . . . 47
12.1. Creating Lots Of Sequences . . . 53
13. Write Ahead Log Page — Type 0x0a . . . 59
14. External Table Format . . . 61
14.1. General Binary Format . . . 61
14.1.1. Endianness . . . 61
14.1.2. Row Format and Virtual Position . . . 62
14.2. Datatype Formats . . . 63
14.2.1. String and Binary Datatypes . . . 63
14.2.2. Integral Datatypes . . . 65
14.2.3. Fixed-Point Datatypes . . . 66

14.2.4. Approximate Floating-Point Datatypes . . . 67
14.2.5. Decimal Floating-Point Types . . . 67
14.2.6. Date and Time Datatypes . . . 68
14.2.7. Other Datatypes . . . 70
Appendix A: Fbdump . . . 71
Appendix B: Document history . . . 72
Appendix C: License notice . . . 73

Chapter 1. Introduction

The purpose of this document is to try to explain what goes on inside a Firebird database. Much of the information in this manual has been extracted from the Firebird source code — mainly on the ODS related code and headers — and from some (partially out of date) documents on the Research part of the IBPhoenix website (https://www.ibphoenix.com/). Other questions and queries that I had were very patiently answered on the Firebird Support forums where the developers hang out.

Much hex dumping of database files was undertaken in the creation of this document, but no Firebird databases were harmed during this process.

All databases mentioned or described within this document are those with an ODS of 11.1 — in other words, Firebird 2.1 — and a page size of 4,096 bytes. There may be differences between this ODS version and previous ones and wherever possible, this has been documented.

Newer ODS versions may use different page layouts not documented in this manual.

Unless otherwise noted, the test database used for this document is empty of all user tables, indices, generators etc. It was created on a 32-bit Linux system running on Intel hardware. It is therefore little endian.

Thanks to everyone who has contributed to the manual.

Chapter 2. Database Structure

When you create a new database, be it single or multiple file, a number of things happen:

- The database file(s) are created;
- The header page is formatted and written;
- The various system tables — `RDB$` and `MON$` — and associated indices are created, and appropriate pages formatted and written to disc;
- Every page in the database is formatted with a defined page type.

The various page types are described elsewhere in this document.

A database is created and the DBA can specify the page size, or leave it to default. This action creates a database file, or files, with enough space allocated to create all the system tables and indices. New pages will be added to the end of the database file(s) as and when the user creates new tables and/or indices. For example, a brand-new database, created on a 32-bit Linux system, with a 4Kb page size allocates a total of 0xa1 pages (161 pages) for the system tables, indices and the various database overhead pages.

2.1. Single File Databases

A single file Firebird database consists of a number of pages, each the same size, and all held within one file on the underlying file system, be it NTFS, FAT32, EXT3 etc.

The first page in the database is always a header page (page type 0x01 — see below) which holds details about the database itself, the page size and so on.

The second page in the database is a Page Inventory Page or PIP (page type 0x02) which details which pages in the database are in use and which are free.

Up until Firebird 3.0, the next page is a Write Ahead Log page (page type 0x0a) but this page is wasted space, if present, and will most likely be dropped from Firebird 3.0 onwards.

The remaining pages consist of Index Root Pages (page type 0x06), Transaction Inventory Pages or TIP (page type 0x03), Pointer Pages (page type 0x04), Index BTree Pages (page type 0x07), Data Pages (page type 0x05) and so on. There is a discussion of each page type below.

2.2. Multiple File Databases

A multiple file Firebird database is almost identical to the single file database except it has more than one file on the underlying file system. Each file has the same page size as the initial file, and each file has a header page (page type 0x01) at the start of the file.

2.3. Shadow Files

Shadow files are additional files that can be used by single and multiple file databases to assist in recovery after a failure of some kind. They are not helpful in the case of a `DROP DATABASE` as the

shadow file(s) — being part of the database — are also dropped!

Shadow files are updated as the database main file(s) are updated and in this manner, the shadows are an identical copy of the database. In the event of a problem, the SYSDBA can manually activate a shadow, or have the Firebird engine activate one automatically.

Unfortunately, if a database write stores corrupt data in the database, the shadow file(s) will be identically corrupted.

Because shadow files are effectively identical copies of the database files, they will not be discussed further.

Chapter 3. Standard Database Page Header

Every page in a database has a 16-byte standard page header. Various page types have an additional header that follows on from the standard one. The C code representation of the standard header is:

```
struct pag
{
    SCHAR pag_type;
    UCHAR pag_flags;
    USHORT pag_checksum;
    ULONG pag_generation;
    ULONG pag_scn;
    ULONG reserved;
};
```

`pag_type`

One byte, signed. Byte 0x00 on the page. This byte defines the page type for the page. Valid page types are:

0x00 Undefined page. You should never see this in a database.

0x01 The database header page. Only ever seen on the very first page of the database, or, on the first page of each database file in a multi-file database.

0x02 The Page Inventory Page (PIP). This page keeps track of allocated and free pages using a bitmap where a 1 means the page is free, and a 0 (zero) shows a used page. There may be more than one PIP in a database, but the first PIP is always page 1.

0x03 Transaction Inventory Page (TIP). A page that keeps track of the stat of transactions. Each transaction is represented by a pair of bits in a bitmap. Valid values in these two bits are:

- **00** this transaction is active.
- **01** this transaction is in limbo.
- **10** this transaction is dead.
- **11** this transaction has committed.

0x04 Pointer Page. Each table has one or more of these and this page type keeps track of all the pages that make up the table. Pointer pages are owned by one and only one table, there is no sharing allowed. Each pointer in the array on these pages holds the page number for a type 5 page holding data for the table.

0x05 Data Page. These pages store the actual data for a table.

0x06 Index Root Page. Similar to a type 4 Pointer Page, but applies to indexes only.

0x07 Index B-Tree Page. Similar to the type 5 Data Page, but applies to indexes only.

0x08 Blob Page. Blobs have their own storage within the database. Very large blobs will require a sequence of pages and the type 8 page holds blob data.

0x09 Generator Page. Holds an array of 64 bit generators.

0x0a Page 2 of any database is a Write Ahead Log page. These pages are no longer used. The page will remain blank (filled with binary zero) as it is never used. This page has a standard header like all others.

`pag_flags`

One byte, unsigned. Byte 0x01 on the page. This byte holds various flags for the page.

`pag_checksum`

Two bytes, unsigned. Bytes 0x02 - 0x03. Checksum for the whole page. No longer used, always 12345, 0x3039. Databases using ODS8 on Windows NT do have a valid checksum here.

Discussions are underway on the development mailing list on reusing this field as a page number rather than a checksum. From Firebird 3.0, it is possible that this field in the page header will probably have a new name and function.

`pag_generation`

Four bytes, unsigned. Bytes 0x04 - 0x07. The page generation number. Increments each time the page is written back to disc.

`pag_scn`

Four bytes, unsigned. Bytes 0x08 - 0x0b. Originally used as the sequence number in the Write Ahead Log, but WAL is no longer used. The field was converted to be the SCN number to avoid an ODS change and is now used by `nbackup`.

`pag_reserved`

Four bytes, unsigned. Bytes 0x0c - 0x0f. Reserved for future use. It was originally used for the offset of a page's entry in the Write Ahead Log (WAL), but this is no longer in use.

Chapter 4. Database Header Page — Type 0x01

The first page of the first file of a Firebird database is a very important page. It holds data that describes the database, where its other files are to be found, shadow file names, database page size, ODS version and so on. On startup, the Firebird engine reads the first part (1,024 bytes) of the first page in the first file of the database and runs a number of checks to ensure that the file is actually a database and so on. If the database is multi-file, then each file will have a header page of its own.

The C code representation of the database header page is:

```
struct header_page
{
    pag hdr_header;
    USHORT hdr_page_size;
    USHORT hdr_ods_version;
    SLONG hdr_PAGES;
    ULONG hdr_next_page;
    SLONG hdr_oldest_transaction;
    SLONG hdr_oldest_active;
    SLONG hdr_next_transaction;
    USHORT hdr_sequence;
    USHORT hdr_flags;
    SLONG hdr_creation_date[2];
    SLONG hdr_attachment_id;
    SLONG hdr_shadow_count;
    SSHORT hdr_implementation;
    USHORT hdr_ods_minor;
    USHORT hdr_ods_minor_original;
    USHORT hdr_end;
    ULONG hdr_page_buffers;
    SLONG hdr_bumped_transaction;
    SLONG hdr_oldest_snapshot;
    SLONG hdr_backup_pages;
    SLONG hdr_misc[3];
    UCHAR hdr_data[1];
};
```

`hdr_header`

The database header page has a standard page header, as do all pages.

`hdr_page_size`

Two bytes, unsigned. Bytes 0x10 - 0x11 on the page. This is the page size, in bytes, for each and every page in the database.

`hds_ods_version`

Two bytes, unsigned. Bytes 0x12 and 0x13 on the page. The ODS major version for the database.

The format of this word is the ODS major version ANDed with the Firebird flag of 0x8000. In the example below, the value is 0x800b for ODS version 11. The minor ODS version is held elsewhere in the header page — see `hdr_ods_minor` below.

`hdr_pages`

Four bytes, signed. Bytes 0x14 - 0x17 on the page. This is the page number of the first pointer page for the table named `RDB$PAGES`. When this location is known, the database engine uses it to determine the locations of all other metadata pages in the database. This field is only valid in the header page of the *first* file in a multi-file database. The remaining files in the database have this field set to zero.

`hdr_next_page`

Four bytes, unsigned. Bytes 0x18 - 0x1b on the page. The page number of the header page in the next file of the database — if this is a multi-file database. Zero otherwise.

`hdr_oldest_transaction`

Four bytes, signed. Bytes 0x1c - 0x1f on the page. The transaction id of the oldest active (ie, uncommitted — but may be in limbo or rolled back) transaction against this database. This field is only valid in the header page of the *first* file in a multi-file database. The remaining files in the database have this field set to zero.

`hdr_oldest_active`

Four bytes, signed. Bytes 0x20 - 0x23 on the page. The transaction id of the oldest active transaction against this database, when any active transaction started. This field is only valid in the header page of the *first* file in a multi-file database. The remaining files in the database have this field set to zero.

`hdr_next_transaction`

Four bytes, signed. Bytes 0x24 - 0x27 on the page. The transaction id that will be assigned to the next transaction against this database. This field is only valid in the header page of the *first* file in a multi-file database. The remaining files in the database have this field set to zero.

`hdr_sequence`

Two bytes, unsigned. Bytes 0x28 and 0x29 on the page. The sequence number of this file within the database.

`hdr_flags`

Two bytes, unsigned. Bytes 0x2a and 0x2b on the page. The database flags. The bits in the flag bytes are used as follows:

Flag Name	Flag value	Description
`hdr_active_shadow`	0x01 (bit 0)	This file is an active shadow file.
`hdr_force_write`	0x02 (bit 1)	The database is in *forced writes* mode.
Unused	0x04 (bit 2)	Was previously for short term journaling, no longer used.
Unused	0x08 (bit 3)	Was previously for long term journaling, no longer used.

Flag Name	Flag value	Description
hdr_no_checksums	0x10 (bit 4)	Don't calculate checksums.
hdr_no_reserve	0x20 (bit 5)	Don't reserve space for record versions in pages.
Unused	0x40 (bit 6)	Was used to indicate that the shared cache file was disabled.
hdr_shutdown_mask (bit one of two)	0x1080 (bits 7 and 12)	Used with bit 12 (see below) to indicate the database shutdown mode.
hdr_sql_dialect_3	0x100 (bit 8)	If set, the database is using SQL dialect 3.
hdr_read_only	0x200 (bit 9)	Database is in read only mode.
hdr_backup_mask	0xC00 (bits 10 and 11)	Indicates the current backup mode.
hdr_shutdown_mask (bit two of two)	0x1080 (bits 7 and 12)	Used with bit 7 (see above) to indicate the database shutdown mode.

The final two database flags use a pair of bits to indicate various states of backup and shutdown.

`hdr_backup_mask`

These two bits determine the current database backup mode, as follows:

Flag Value	Description
0x00 (Both bits zero)	Database is not in backup mode. User changes are written directly to the database files.
0x400	The database is running in backup mode so all changed made by the users are written to the diff file.
0x800	The database is still in backup mode, but changes are being merged from the diff file into the main pages.
0xC00	The current database state is unknown and changes need to be read from disk.

`hdr_shutdown_mask`

The shutdown mask uses two bits to indicate the current database shutdown status, as follows:

Flag Value	Description
0x00 (Both bits 7 and 12 are zero)	Database is not shutdown. Any valid user can connect.
0x80	The database has been shutdown to, or started up in multi-user maintenance mode. The database can only be conncted to by SYSDBA or the database owner.
0x1000	The database has been fully shutdown. No connections are permitted.

Flag Value	Description
0x1080	The database has been shutdown to, or started up in single-user maintenance mode. Only one SYSDBA or database owner connection is permitted.

`hdr_creation_date`

Eight bytes, signed. Bytes 0x2c - 0x33 on the page. The date and time (in Firebird's own internal format) that the database was either originally created/rewritten or created from a backup.

`hdr_attachment_id`

Four bytes, signed. Bytes 0x34 - 0x37 on the page. The id number that will be assigned to the next connection to this database. As this is signed, the maximum value here is 2^{32} -1 and any database which reaches this maximum value must be backed up and restored in order to allow new connections. This field is only valid in the header page of the *first* file in a multi-file database. The remaining files in the database have this field set to zero.

`hdr_shadow_count`

Four bytes, signed. Bytes 0x38 - 0x3b on the page. Holds the event count for shadow file synchronisation for this database. The remaining files in the database have this field set to zero.

`hdr_implementation`

Two bytes, signed. Bytes 0x3c and 0x3d on the page. This is a number which indicates the environment on which the database was originally created. It is used to determine if the database file can be used successfully on the current hardware. This avoids problems caused by little-endian numerical values as compared with big-endian, for example.

`hdr_ods_minor`

Two bytes, unsigned. Bytes 0x3e and 0x3f on the page. The current ODS minor version.

`hdr_ods_minor_original`

Two bytes, unsigned. Bytes 0x40 and 0x41 on the page. The ODS minor version when the database was originally created.

`hdr_end`

Two bytes, unsigned. Bytes 0x42 and 0x43 on the page. The offset on the page where the hdr_data finishes. In other words, where a new clumplet will be stored if required. This is effectively a pointer to the current location of `HDR_end` (see clumplet details below) on this page.

`hdr_page_buffers`

Four bytes, unsigned. Bytes 0x44 - 0x47 on the page. Holds the number of buffers to be used for the database cache, or zero to indicate that the default value should be used. This field is only valid in the header page of the *first* file in a multi-file database. The remaining files in the database have this field set to zero.

`hdr_bumped_transaction`

Four bytes, signed. Bytes 0x48 - 0x4b on the page. Used to be used for the bumped transaction id for log optimisation, but is currently always set to 0x01. This field is only valid in the header

page of the *first* file in a multi-file database. The remaining files in the database have this field set to zero.

`hdr_oldest_snapshot`

Four bytes, signed. Bytes 0x4c - 0x4f on the page. Holds the transaction number for the oldest snapshot of active transactions. This is also documented as the *confusing and redundant variant of Oldest Active Transaction*.

`hdr_backup_pages`

Four bytes, signed. Bytes 0x50 - 0x53 on the page. Holds the number of pages in the database currently locked for a backup using nbackup. This field is only valid in the header page of the *first* file in a multi-file database. The remaining files in the database have this field set to zero.

`hdr_misc`

Twelve bytes. Bytes 0x54 - 0x5f on the page. Set to zero. These 12 bytes are currently unused.

The following is an example of a header page from a multi-file database on a little-endian system:

```
00000000  01 00 39 30 08 00 00 00  00 00 00 00 00 00 00 00  Standard header
00000010  00 10                                           hdr_page_size
00000012  0b 80                                           hdr_ods_version
00000014  03 00 00 00                                     hdr_PAGES
00000018  00 00 00 00                                     hdr_next_page
0000001c  01 00 00 00                                     hdr_oldest_transaction
00000020  02 00 00 00                                     hdr_oldest_active
00000024  05 00 00 00                                     hdr_next_transaction
00000028  00 00                                           hdr_sequence
0000002a  00 01                                           hdr_flags
0000002c  5e d7 00 00 f4 79 00 23                         hdr_creation_date
00000034  01 00 00 00                                     hdr_attachment_id
00000038  00 00 00 00                                     hdr_shadow_count
0000003c  13 00                                           hdr_implementation
0000003e  01 00                                           hdr_ods_minor
00000040  01 00                                           hdr_ods_minor_original
00000042  93 00                                           hdr_end
00000044  00 00 00 00                                     hdr_page_buffers
00000048  01 00 00 00                                     hdr_bumped_transaction
0000004c  02 00 00 00                                     hdr_oldest_snapshot
00000050  00 00 00 00                                     hdr_backup_pages
00000054  00 00 00 00 00 00 00 00 00 00 00 00             hdr_misc
00000060                                                  hdr_data[]
```

From Firebird 2.x onwards, there is a system table — `MON$DATABASE` which has a copy of all of the above data in an easy to obtain format:

```
tux> isql employee
Database:  employee
```

```
SQL> show table mon$database;
MON$DATABASE_NAME               (RDB$FILE_NAME) VARCHAR(253) Nullable
MON$PAGE_SIZE                   (RDB$PAGE_SIZE) SMALLINT Nullable
MON$ODS_MAJOR                   (RDB$ODS_NUMBER) SMALLINT Nullable
MON$ODS_MINOR                   (RDB$ODS_NUMBER) SMALLINT Nullable
MON$OLDEST_TRANSACTION          (RDB$TRANSACTION_ID) INTEGER Nullable
MON$OLDEST_ACTIVE               (RDB$TRANSACTION_ID) INTEGER Nullable
MON$OLDEST_SNAPSHOT             (RDB$TRANSACTION_ID) INTEGER Nullable
MON$NEXT_TRANSACTION            (RDB$TRANSACTION_ID) INTEGER Nullable
MON$PAGE_BUFFERS                (RDB$PAGE_BUFFERS) INTEGER Nullable
MON$SQL_DIALECT                 (RDB$SQL_DIALECT) SMALLINT Nullable
MON$SHUTDOWN_MODE               (RDB$SHUTDOWN_MODE) SMALLINT Nullable
MON$SWEEP_INTERVAL              (RDB$SWEEP_INTERVAL) INTEGER Nullable
MON$READ_ONLY                   (RDB$SYSTEM_FLAG) SMALLINT Nullable
MON$FORCED_WRITES               (RDB$SYSTEM_FLAG) SMALLINT Nullable
MON$RESERVE_SPACE               (RDB$SYSTEM_FLAG) SMALLINT Nullable
MON$CREATION_DATE               (RDB$TIMESTAMP) TIMESTAMP Nullable
MON$PAGES                       (RDB$COUNTER) BIGINT Nullable
MON$STAT_ID                     (RDB$STAT_ID) INTEGER Nullable
MON$BACKUP_STATE                (RDB$BACKUP_STATE) SMALLINT Nullable

SQL> commit;
SQL> quit;
```

`hdr_data`

The variable data area on the header page begins at offset 0x60. Data stored here is held in clumplets and there are a number of different clumplet types, see below. This area is used to store filenames for the next file and other miscellaneous pieces of data relating to the database.

The format of each clumplet is as follows:

`type_byte`

The first byte — unsigned — in each clumplet determines the type of data stored within the clumplet. There are a number of different clumplet types:

Type Name	Value	Description
HDR_end	0x00	End of clumplets.
HDR_root_file_name	0x01	Original name of the root file for this database.
HDR_journal_server	0x02	Name of the journal server.
HDR_file	0x03	Secondary file name.
HDR_last_page	0x04	Last logical page of the current file.
HDR_unlicensed	0x05	Count of unlicensed activity. No longer used.
HDR_sweep_interval	0x06	Number of transactions between sweep.
HDR_log_name	0x07	Replay log name.
HDR_journal_file	0x08	Intermediate journal filename.

Type Name	Value	Description
HDR_password_file_key	0x09	Key to compare with the password database.
HDR_backup_info	0x0a	Write Ahead Log (WAL) backup information. No longer used.
HDR_cache_file	0x0b	Shared cache file. No longer used.
HDR_difference_file	0x0c	Diff file used during the times when the database is in backup mode.
HDR_backup_guid	0x0d	UID generated when database is in backup mode. Overwritten on subsequent backups.

`length_byte`

The second byte — again unsigned — in each clumplet specifies the size of the data that follows.

`data`

The next 'n' bytes are the actual clumplet data.

The miscellaneous data stored in the header from the above database, at `hdr_data`, is shown below.

```
00000060  03                                                type = HDR_file
00000061  2b                                                length = 43 bytes
00000062  2f 75 30 30 2f 66 69 72  65 62 69 72 64 2f       data '/u00/firebird/'
00000070  64 61 74 61 62 61 73 65  73 2f 6d 75 6c 74 69 5f      'databases/multi_'
00000080  65 6d 70 6c 6f 79 65 65  2e 66 64 62 31              'employee.fdb1'

0000008d  04                                                type = HDR_last_page
0000008e  04                                                length = 4 bytes
0000008f  a2 00 00 00                                       data 0xa2 = 162

00000093  00                                                type = HDR_end.
```

From the above we can see that in our multi-file database:

- The *next* file (after this one) is named '`/u00/firebird/databases/multi_employee.fdb1`'
- The *current* file has 162 pages only — and with a 4Kb page size this means that the current file should be 663,552 bytes in size, which a quick run of `ls -l` will confirm.
- `HDR_end` is located at offset 0x93 in the page, exactly as the header field `hdr_end` told us (see above).

Chapter 5. Page Inventory Page — Type 0x02

Every database has at least one Page Inventory Page (PIP) with the first one *always* being page 1, just after the database header page. If more are required, the current PIP points to the next PIP by way of the very last bit on the page itself. The C code representation of the PIP page is:

```
struct page_inv_page
{
    pag pip_header;
    SLONG pip_min;
    UCHAR pip_bits[1];
};
```

`pip_header`

The PIP starts off with a standard page header.

`pip_min`

Four bytes, signed. Bytes 0x10 - 0x13 on the page. This is the bit number of the first page, on this PIP, which is currently free for use.

`pip_bits`

Bytes 0x14 onwards. The remainder of the page, is an array of single bits where each bit represents a page in the database. If the bit is set (1) then that page is free for use. If the bit is unset (0) then the page has been used.

If the database is large, and requires another PIP elsewhere in the database, then the last bit on this PIP represents the page number for the next PIP. For example, on a 4,096 byte page we have a total of 4,076 bytes to represent different pages in the database. As each byte has 8 bits, we have a total of 32,608 pages before we need a new PIP.

In a brand new database, a hex dump of the first few bytes of page 1, the first PIP, looks like the following:

```
Offset    Data                                            Description
--------------------------------------------------------------------------------
00001000  02 00 39 30 31 00 00 00  00 00 00 00 a1 00 00 00  Standard Header
00001010  a1 00 00 00                                      pip_min (low endian)
00001014  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  pip_bits[]
00001024  00 00 00 00 fe ff ff ff  ff ff ff ff ff ff ff ff
```

In the above, we see that `pip_min` has the value 0x000000a1 and the following 20 bytes, the first part of the `pip_bits` array, are all zero. From this, it would appear that page 0xa1 is the first available page in the database for user tables etc and that all the pages up to that one have already been used for the system tables and indices etc.

Looking at the bitmap again, page 0xa1 will be represented by byte 0x14, bit 0x01 of the bitmap. This is byte 0x00001028 bit 1. We can see that this byte currently has the value 0xfe and bit 0x00 is

already in use. So, our array is correct and so is our `pip_min` value — the next available page is indeed 0xa1.

If we look at the hexdump of that particular page, at address 0x000a1000, we see that it is actually the first byte past the current end of file, so our brand new blank database has been created with just enough space to hold all the system tables and indexes and nothing else.

Chapter 6. Transaction Inventory Page — Type 0x03

Every database has at least one Transaction Inventory Page (TIP).

The highest possible transaction number is 2,147,483,647 or 0x7fffffff in a 32-bit system. Once you hit this transaction, no more can be created, and the database needs to be shutdown, backed up and then restored to reset the transaction numbers back to zero. The reason it has this maximum value is simply because the code for allocating transaction numbers uses a signed value.

The C code representation of the TIP page is:

```
struct tx_inv_page
{
    pag tip_header;
    SLONG tip_next;
    UCHAR tip_transactions[1];
};
```

`tip_header`

The TIP starts off with a standard page header.

`tip_next`

Four bytes, signed. Bytes 0x10 - 0x13 on the page. This is the page number of the next TIP page, if one exists, within the database. Zero here indicates that the current TIP page is the last TIP page.

`tip_transactions`

Bytes 0x14 onwards. The remainder of the page, is an array of two bit values where each pair of bits represents a transaction and its status. Each transaction can have one of 4 status values:

0x00 this transaction is active, or has not yet started.

0x01 this transaction is in limbo. A two phase transaction has committed the first phase but the second phase has not committed.

0x02 this transaction is dead (was rolled back).

0x03 this transaction was committed.

Looking at a hex dump of the first few bytes of a new database, which has had a few transactions run against it, we see the following:

```
Offset    Data                                            Description
--------------------------------------------------------------------------------
000a0014  fc ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  tip_transactions[]
000a0024  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
```

```
000a0034  ff ff ff ff ff ff ff ff  ff ff ff ff ff 00 00 00
```

Now, if a new transaction starts we won't see any changes because a live transaction and one that has not started yet, shows up as two zero bits in the tip_transactions array. However, if it commits, limbo's or rolls back, we should see a change. The following is the above database after a session connected using isql and immediately exited without doing anything:

```
Offset    Data                                            Description
--------------------------------------------------------------------------------
000a0014  fc ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  tip_transactions[]
000a0024  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
000a0034  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff 00 00
```

You can see that it looks remarkably like loading up a connection to `isql` and then exiting actually executes 4 separate transactions. We can see at the end of the last line that one byte has changed from 0x00 to 0xff and with 2 bits per transaction, that equates to 4 separate transactions, all of which committed.

Other tools may run fewer or indeed, more, transactions just to connect to a database and do whatever it is that they have to do to initialise themselves.

Chapter 7. Pointer Page — Type 0x04

A pointer page is used internally to hold a list of all — or as may will fit on one pointer page — data pages (see below) that make up a single table. Large tables may have more than one pointer page but every table, system or user, will have a minimum of one pointer page. The RDB$PAGES table is where the Firebird engine looks to find out where a table is located within the physical database, however, RDB$PAGES is itself a table, and when the database is running, how exactly can it find the start page for RDB$PAGES in order to look it up?

The database header page contains the page number for RDB$PAGES at bytes 0x14 - 0x17 on the page. From experimentation, it appears as if this is always page 0x03, however, this cannot be relied upon and if you need to do this, you should always check the database header page to determine where RDB$PAGES is to be found.

The C code representation of a pointer page is:

```
struct pointer_page
{
    pag ppg_header;
    SLONG ppg_sequence;
    SLONG ppg_next;
    USHORT ppg_count;
    USHORT ppg_relation;
    USHORT ppg_min_space;
    USHORT ppg_max_space;
    SLONG ppg_page[1];
};
```

`ppg_header`

A pointer page starts with a standard page header. In the header, the pag_flags field is used and is set to the value 1 if this is the final pointer page for the relation.

`ppg_sequence`

Four bytes, signed. Offset 0x10 to 0x13 on the page. The sequence number of this pointer page in the list of pointer pages for the table. Starts at zero.

`ppg_next`

Four bytes, signed. Offset 0x14 to 0x17 on the page. The page number of the next pointer page for this table. Zero indicates that this is the final pointer page.

`ppg_count`

Two bytes, unsigned. Offset 0x18 and 0x19 on the page. This field holds the count of active slots (in the ppg_page array) on this pointer page, that are in use. As the array starts at zero, this is also the index of the first free slot on this pointer page.

`ppg_relation`

Two bytes, unsigned. Offset 0x1a and 0x1b on the page. This field holds the

RDB$RELATIONS.RDB$REALTION_ID for the table that this pointer page represents.

`ppg_min_space`

Two bytes, unsigned. Offset 0x1c and 0x1d on the page. This indicates the first entry in the ppg_page array holding a page number which has free space in the page.

`ppg_max_space`

Two bytes, unsigned. Offset 0x1e and 0x1f on the page. This was intended to indicate the last entry in the ppg_page array holding a page number which has free space in the page, but it has never been used. These two bytes are invariably set to zero.

`ppg_page`

An array of 4-byte signed values, starting at offset 0x20. Each value in this array represents a page number where a part of the current table is to be found. A value of zero in a slot indicates that the slot is not in use. Deleting all the data from a table will result in all slots being set to zero.

Page fill bitmaps

At the end of each pointer page is a bitmap array of two bit entries which is indexed by the same index as the ppg_page array. These bitmaps indicate that the page is available for use in storing records (or record versions) or not. The two bits in the bitmap indicate whether a large object (BLOB?) is on this page, and the other bit indicates that the page is full. If either bit is set (page has a large object or page is full, then the page is not used for new records or record versions.

The location of the bitmaps on each page is dependent on the page size. The bigger the page, the more slots in the ppg_page array can hold and so the bitmap is bigger. A bigger bitmap starts at a lower address in the page and so on. From looking inside a few databases with a 4Kb page size, the bitmaps begin at offset 0x0f10 on the page.

You can find the pointer page for any table by running something like the following query in isql:

```
SQL> SELECT P.RDB$PAGE_NUMBER, P.RDB$PAGE_SEQUENCE, P.RDB$RELATION_ID
CON> FROM RDB$PAGES P
CON> JOIN RDB$RELATIONS R ON (R.RDB$RELATION_ID = P.RDB$RELATION_ID)
CON> WHERE R.RDB$RELATION_NAME = 'EMPLOYEE'
CON> AND P.RDB$PAGE_TYPE = 4;

RDB$PAGE_NUMBER RDB$PAGE_SEQUENCE RDB$RELATION_ID
=============== ================= ===============
            180                 0             131
```

The page number which has RDB$PAGE_SEQUENCE holding the value zero is the top level pointer page for this table. In the above example, there is only one pointer page for the EMPLOYEE table. If we now hexdump the pointer page for the employee table, we see the following:

```
000b4000  04 01 39 30 02 00 00 00  00 00 00 00 00 00 00 00  Standard header
000b4010  00 00 00 00                                      ppg_sequence
000b4014  00 00 00 00                                      ppg_next
```

```
000b4018  02 00                                           ppg_count
000b401a  83 00                                           ppg_relation
000b401c  01 00                                           ppg_min_space
000b401e  00 00                                           ppg_max_space
000b4020  ca 00 00 00                                     ppg_page[0]
000b4024  cb 00 00 00                                     ppg_page[1]
000b4028  00 00 00 00                                     ppg_page[2]
000b402c  00 00 00 00                                     ppg_page[3]
...
000b4f10  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000b4f20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Looking at the above, we can see at address 0x0b4f10 on the page, that the byte there has the value of 0x01. This is an indicator that the page in `ppg_page[0]` — page 0xca — is full to capacity (bit 0 set) and does not have any large objects on the page (bit 1 unset). The page at `ppg_page[1]` — page 0xcb — is, on the other hand, not full up yet (bit 2 is unset) and doesn't have a large object on the page either. This means that this page is available for us.

This is confirmed by checking the value in `ppg_min_space` which has the value 0x0001 and does indeed correspond to the first page with free space. The value in `ppg_min_space` is the index into the `ppg_array` and not the page number itself.

Chapter 8. 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 0x14 on the page. The relation number for this table. This corresponds to `RDB$RELATIONS.RDB$RELATION_ID`.

`dpg_count`

Two bytes, unsigned. Offset 0x16 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 structs with two fields, each of them are 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.

8.1. Record Header

Each record's data is preceded 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.

8.2. 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.

8.3. 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 out 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 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
        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.

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 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 'abcabcabcabcabcabcabcd' do not get compressed.

8.4. Examining The Data

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

8.4.1. 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 62 43
 .  .  .  .     .  A  a  a  a  a  B  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
```

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.

8.4.2. 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 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.

8.4.3. 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.

8.4.4. 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 `NULL`s and the other with all not-`NULL` columns. The first record, all `NULL`s, 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.

Chapter 9. Index Root Page — Type 0x06

Every table in the database has an Index Root Page which holds data that describes the indexes for that table. Even tables that have no indices defined have an index root page.

The C code representation of an index root page is:

```
struct index_root_page
{
    pag irt_header;
    USHORT irt_relation;
    USHORT irt_count;
    struct irt_repeat {
        SLONG irt_root;
        union {
            float irt_selectivity;
            SLONG irt_transaction;
        } irt_stuff;
        USHORT irt_desc;
        UCHAR irt_keys;
        UCHAR irt_flags;
    } irt_rpt[1];
};
```

`irt_header`

The page starts with a standard page header. The flags byte — pag_flags — is not used on this page type.

`irt_relation`

Two bytes, unsigned. Offset 0x10 on the page. The relation id. This is the value of `RDB$RELATIONS.RDB$RELATION_ID`.

`irt_count`

Two bytes, unsigned. Offset 0x12 on the page. The number of indices defined for this table. If there are no indices defined this counter will show the value zero. (Every table in the database has an Index Root Page regardless of whether or not it has any indices defined.)

`irt_rpt`

This is an array of index descriptors. The array begins at offset 0x14 on the page with the descriptor for the first index defined for the table. Descriptors are added to the 'top' of the array, so the next index defined will have its descriptor at a higher page address than the previous descriptor. The descriptor entries consist of the following 6 fields (`irt_root` through `irt_flags`). Each descriptor is 0x0b bytes long.

`irt_root`

Four bytes, signed. Offset 0x00 in each descriptor array entry. This field is the page number where the root page for the individual index (page type 0x07) is located.

`irt_selectivity`

Four bytes, signed floating-point. Offset 0x04 in each descriptor array entry. This is the same offset as for `irt_transaction` below. In ODS versions previous to 11.0 this field holds the index selectivity in floating-point format.

From ODS version 11.0, this field is no longer used as selectivity has been moved to the index field descriptors (see below).

`irt_transaction`

Four bytes, signed. Offset 0x04 in each descriptor array entry — the same offset as `irt_selectivity` above. Normally this field will be zero but if an index is in the process of being created, the transaction id will be found here.

`irt_desc`

Two bytes, unsigned. Offset 0x08 in each descriptor array entry. This field holds the offset, from the start of the page, to the index field descriptors which are located at the bottom end (ie, highest addresses) of the page. To calculate the starting address, add the value in this field to the address of the start of the page.

`irt_keys`

One byte, unsigned. Offset 0x0a in each descriptor array entry. This defines the number of keys (columns) in this index.

`irt_flags`

One byte, unsigned. Offset 0x0b in each descriptor array entry. The flags define various attributes for this index, these are encoded into various bits in the field, as follows:

Bit 0 Index is unique (set) or not (unset).

Bit 1 Index is descending (set) or ascending (unset).

Bit 2 Index [creation?] is in progress (set) or not (unset).

Bit 3 Index is a foreign key index (set) or not (unset).

Bit 4 Index is a primary key index (set) or not (unset).

Bit 5 Index is expression based (set) or not (unset).

Each descriptor entry in the array holds an offset to a list of key descriptors. These start at the highest address on the page and extend towards the lowest address. (The array of index descriptors (`irt_rpt`) starts at a low address on the page and increases upwards. At some point, they will meet, and the page will be full.

The index field descriptors are defined as follows:

`irtd_field`

Two bytes, unsigned. Offset 0x00 in each field descriptor. This field defines the field number of the table that makes up 'this' field in the index. This number is equivalent to

RDB$RELATION_FIELDS.RDB$FIELD_ID.

`irtd_itype`

Two bytes, unsigned. Offset 0x02 in each field descriptor. This determines the data type of the appropriate field in the index. The allowed values in this field are:

0 field is numeric, but is not a 64 bit integer.

1 field is string data.

3 Field is a byte array.

4 Field is metadata.

5 Field is a date.

6 Field is a time.

7 Field is a timestamp.

8 field is numeric — and is a 64 bit integer.

You may note from the above that an `irtd_itype` with value 2 is not permitted.

`irtd_selectivity`

Four bytes, floating point format. Offset 0x04 in each field descriptor. This field holds the selectivity of this particular column in the index. This applies to ODS 11.0 onwards. In pre ODS 11.0 databases, this field is not part of the index field descriptors and selectivity is applied to the index as a whole. See `irt_selectivity` above.

The following commands have been executed to create a parent child set of two tables and a selection of indices:

```
SQL> CREATE TABLE PARENT (
CON>    ID INTEGER NOT NULL,
CON>    EMAIL VARCHAR(150)
CON> );

SQL> ALTER TABLE PARENT
CON>     ADD CONSTRAINT PK_PARENT
CON>     PRIMARY KEY (ID);

SQL> ALTER TABLE PARENT
CON>     ADD CONSTRAINT UQ_EMAIL
CON>     UNIQUE (EMAIL);

SQL> COMMIT;

SQL> CREATE TABLE CHILD (
CON>    ID INTEGER NOT NULL,
CON>    PARENT_ID INTEGER,
```

```
CON>    STUFF VARCHAR(200)
CON> );

SQL> ALTER TABLE CHILD
CON>     ADD CONSTRAINT FK_CHILD
CON>     FOREIGN KEY (PARENT_ID)
CON>     REFERENCES PARENT (ID);

SQL> COMMIT;
```

The Following command was then executed to extract the index root pages for both of these tables:

```
SQL> SELECT R.RDB$RELATION_NAME,
CON>    R.RDB$RELATION_ID,
CON>    P.RDB$PAGE_TYPE,
CON>    P.RDB$PAGE_NUMBER
CON> FROM RDB$RELATIONS R
CON> JOIN RDB$PAGES P ON (P.RDB$RELATION_ID = R.RDB$RELATION_ID)
CON> WHERE R.RDB$RELATION_NAME IN ('PARENT','CHILD')
CON>    AND P.RDB$PAGE_TYPE = 6;

RDB$RELATION_NAME RDB$RELATION_ID RDB$PAGE_TYPE RDB$PAGE_NUMBER
================= =============== ============= ===============
PARENT                        139             6             173
CHILD                         140             6             178
```

Now that the root pages are known, we can take a look at the layout of these two pages and see how the details of the various indices are stored internally:

```
tux> ./fbdump ../blank.fdb -p 173,178

FBDUMP 1.00 - Firebird Page Dump Utility

Parameters : -p 173,178 -v
Database: ../blank.fdb

DATABASE PAGE DETAILS - Page 173
        Page Type: 6
        Flags: 0
        Checksum: 12345
        Generation: 5
        SCN: 0
        Reserved: 0
PAGE DATA
        Relation: 139
        Index Count: 2

        Root Page[0000]: 174
        Transaction[0000]: 0
```

```
        Descriptor[0000]: 4088 (0x0ff8)
        Keys[0000]: 1
        Flags[0000]: 17 :Unique:Ascending:Primary Key:
        Descriptor[0000].Field: 0
        Descriptor[0000].Itype: 0 :Numeric (Not BigInt)
        Descriptor[0000].Selectivity: 0.000000

        Root Page[0001]: 176
        Transaction[0001]: 0
        Descriptor[0001]: 4080 (0x0ff0)
        Keys[0001]: 1
        Flags[0001]: 1 :Unique:Ascending:
        Descriptor[0001].Field: 1
        Descriptor[0001].Itype: 1 :String
        Descriptor[0001].Selectivity: 0.000000

DATABASE PAGE DETAILS - Page 178
PAGE HEADER
        Page Type: 6
        Flags: 0
        Checksum: 12345
        Generation: 3
        SCN: 0
        Reserved: 0
PAGE DATA
        Relation: 140
        Index Count: 1

        Root Page[0000]: 180
        Transaction[0000]: 0
        Descriptor[0000]: 4088 (0x0ff8)
        Keys[0000]: 1
        Flags[0000]: 8 :NonUnique:Ascending:Foreign Key:
        Descriptor[0000].Field: 1
        Descriptor[0000].Itype: 0 :Numeric (Not BigInt)
        Descriptor[0000].Selectivity: 0.000000
```

We can see that the `PARENT` table (relation 139) has two defined indices while the `CHILD` table (relation 140) has one.

If we examine the above output we can see that the indices do match up to those that were created above. We can also see that in the event of an index being created without a sort order (ascending or descending) that the default is ascending.

Chapter 10. Index B-Tree Page — Type 0x07 — YOU ARE HERE.

As described above for the Index Root Page (type 0x06) each index defined for a table has a root page from which the index data can be read etc. The Index Root Page field `irt_root` points to the first page (the root page — just to confuse matters slightly) in the index. That page will be a type 0x07 Index B-Tree Page, as will all the other pages that make up this index.

Indices do not share pages. Each index has its own range of dedicated pages in the database. Pages are linked to the previous and next pages making up this index.

10.1. B-Tree Header

The C code representation of an ODS 11 index b-tree page is:

```
struct btree_page
{
    pag btr_header;
    SLONG btr_sibling;
    SLONG btr_left_sibling;
    SLONG btr_prefix_total;
    USHORT btr_relation;
    USHORT btr_length;
    UCHAR btr_id;
    UCHAR btr_level;
};
```

`btr_header`

The page starts off with a standard page header. The pag_flags byte is used on these pages. The bits used and why are:

Bit 0 set means do not garbage collect this page.

Bit 1 set means this page is not propogated upwards.

Bit 3 set means that this page/bucket is part of a descending index.

Bit 4 set means that non-leaf nodes will contain record number information.

Bit 5 set means that large keys are permitted/used.

Bit 6 set means that the page contains index jump nodes.

`btr_sibling`

Four bytes, signed. Bytes 0x10 - 0x13 on the page. This is the page number of the next page of this index. The values on the next page are *higher* than all of those on this page. A value of zero here indicates that this is the final page in the index.

`btr_left_sibling`

Four bytes, signed. Bytes 0x14 - 0x17 on the page. This is the page number of the previous page of this index. The values on the previous page are *lower* than all of those on this page. A value of zero here indicates that this is the first page in the index.

`btr_prefix_total`

Four bytes, signed. Bytes 0x18 - 0x1b on the page. The sum of all the bytes saved on this page by using prefix compression.

`btr_relation`

Two bytes, unsigned. Bytes 0x1c and 0x1d on the page. The relation id (`RDB$RELATION_ID` in `RDB$RELATIONS`) for the table that this index applies to.

`btr_length`

Two bytes, unsigned. Bytes 0x1e and 0x1f on the page. The number of bytes used, for data, on this page. Acts as an offset to the first unused byte on the page.

`btr_id`

One byte, unsigned. Byte 0x20 on the page. The index id (`RDB$INDEX_ID` in `RDB$INDICES`) for this index.

`btr_level`

One byte, unsigned. Byte 0x21 on the page. The index level. Level zero indicates a leaf node.

10.2. Index Jump Info

Following on from the above, at byte 0x22 on the page, is an Index Jump Info structure. This is defined as follows:

```
struct IndexJumpInfo
{
    USHORT firstNodeOffset;
    USHORT jumpAreaSize;
    UCHAR  jumpers;
};
```

`firstNodeOffset`

Two bytes, unsigned. Offset 0x00 in the structure. This is the offset, in bytes, to the first of the Index Nodes (see below) on this page.

`jumpAreaSize`

Two bytes, unsigned. Offset 0x02 in the structure. The value here is the number of bytes left to be used before we have to create a new jump node.

`jumpers`

One byte, unsigned. Offset 0x05 in the structure. The running total of the current number of Jump Nodes on this page. There can be a maximum of 255 Index Jump Nodes on a page.

10.3. Index Jump Nodes

The Index Jump Info structure described above is followed by zero or more Index Jump Nodes. The number to be found is determined by the jumpers value in the Index Jump Info structure. Index Jump Nodes are defined as follows:

```
struct IndexJumpNode
{
    UCHAR* nodePointer; // pointer to where this node can be read from the page
    USHORT prefix;      // length of prefix against previous jump node
    USHORT length;      // length of data in jump node (together with prefix this is
prefix for pointing node)
    USHORT offset;      // offset to node in page
    UCHAR* data;        // Data can be read from here
};
```

10.4. Index Nodes

`btr_nodes`

Index nodes are described below and are used to hold the data for one entry in this index. The C code representation of an entry in the array is:

```
struct btree_nod
{
    UCHAR btn_prefix;
    UCHAR btn_length;
    UCHAR btn_number[4];
    UCHAR btn_data[1];
};
```

`btn_prefix`

One byte, unsigned. Byte 0x00 in the node. This is the size of the compressed prefix.

`btn_length`

One byte, unsigned. Byte 0x01 in the node. This is the size of the data in the index entry.

`btn_number`

Four bytes, unsigned. Bytes 0x02 - 0x05 in the node. The page number (or record number) where the data that this index entry represents, is to be found.

10.5. Index Data

`btn_data`

The data that makes up the index entry is found at bytes 0x06 onwards in the node.

Following the Index Root Page example, we can now hexdump and inspect the Primary Key index for our example table. We see from the Index Root page that the actual root of the index is on page 0x0513eb in the database. A dump of that page results in the following:

```
513eb000  07 70 39 30 02 00 00 00  00 00 00 00 00 00 00 00  Standard header
513eb010  00 00 00 00                                      btr_sibling
513eb014  00 00 00 00                                      btr_left_sibling
513eb018  1f 00 00 00                                      btr_prefix_total
513eb01c  d5 00                                            btr_relation
513eb01e  a6 00                                            btr_length
513eb020  00                                               btr_id
513eb021  02                                               btr_level
```

This looks like it is the final page in this particular index as it has no siblings, left or right. There also doesn't appear to be much space used on the page as btr_length is showing that only 0xa6 bytes have been used on this page, however, btr_level is 2 so we are not looking at a leaf node. (And we know that this is actually the root node for the entire index since the page we dumped is the root page for the index.)

Following on from the above, we have the various index nodes, starting at offset 0x22, as follows:

```
to be completed soon!
```

Chapter 11. Blob Data Page — Type 0x08 — TODO

The C code representation of a blob data page is:

```
struct blob_page
{
    pag blp_header;
    SLONG blp_lead_page;
    SLONG blp_sequence;
    USHORT blp_length;
    USHORT blp_pad;
    SLONG blp_page[1];
};
```

`blp_header`

The blob page starts off with a standard page header.

`blp_lead_page`

Four bytes, signed. Bytes 0x10 - 0x13. This field holds the page number for the first page for this blob.

`blp_sequence`

Four bytes, signed. Bytes 0x14 - 0x17. The sequence number of this page within the page range for this blob.

`blp_length`

Two bytes, unsigned. Bytes 0x18 and 0x19. The length of the blob data on this page, in bytes.

`blp_pad`

Two bytes, unsigned. Bytes 0x1a and 0x1b. Not used for any data, used as padding.

`blp_page`

This location in the page is at byte 0x1c. It has two purposes:

- An array of four byte, signed page numbers representing all the pages in this blob; or
- An array of bytes making up the blob data on this page.

If the flag byte in the standard page header (`pag_flags`) is set to 1, this blob page contains no data but acts as a pointer page to all the other blob pages for this particular blob.

Chapter 12. Generator Page — Type 0x09

Every database has at least one Generator Page, even if no generators (also known as sequences in Firebird 2.x) have been defined by the user. A blank database consisting only of system tables and indices already has a number of generators created for use in naming constraints, indices, etc.

`GENERATOR` is a non standard term that originated in Interbase. The ISO SQL standard requires the term `SEQUENCE` instead.

The C code representation of the generator page is:

```
struct generator_page
{
    pag gpg_header;
    SLONG gpg_sequence;
    SLONG gpg_waste1;
    USHORT gpg_waste2;
    USHORT gpg_waste3;
    USHORT gpg_waste4;
    USHORT gpg_waste5;
    SINT64 gpg_values[1];
};
```

`gpg_header`

The generator page starts off with a standard page header.

`gpg_sequence`

Four bytes, signed. Bytes 0x10 - 0x13. The sequence number of this generator page, starting from zero. If so many generators have been created that new generator pages are required, the sequence number will be incremented for each one.

`gpg_waste`

Twelve bytes. Bytes 0x14 to 0x1f. To quote the source code, these values are *overhead carried forward for backward compatibility*. In other words, most likely unused.

`gpg_values`

An array of 64 bit values, one for each generator in the database.

If we use `isql` to create a new blank database, we can dump out the generator page as follows:

```
tux> isql
Use CONNECT or CREATE DATABASE to specify a database

SQL> CREATE DATABASE "../blank2.fdb";
SQL> COMMIT;
SQL> EXIT;
```

We need to find the generator page next:

```
SQL> SELECT RDB$PAGE_NUMBER
CON> FROM RDB$PAGES
CON> WHERE RDB$PAGE_TYPE = 9;

RDB$PAGE_NUMBER
===============
            148

SQL> COMMIT;
```

Now we can dump out the generator page:

```
tux> ./fbdump ../blank2.fdb -p 148

FBDUMP 1.00 - Firebird Page Dump Utility

DATABASE PAGE DETAILS - Page 148
        Page Type: 9
PAGE DATA
        Sequence: 0
        Waste1: 0
        Waste2: 0
        Waste3: 0
        Waste4: 0
        Waste5: 0

        There are 9 sequences defined:

        Sequence[00000]: 9
        Sequence[00001]: 0
        Sequence[00002]: 3
        Sequence[00003]: 0
        Sequence[00004]: 0
        Sequence[00005]: 0
        Sequence[00006]: 0
        Sequence[00007]: 0
        Sequence[00008]: 0
        Sequence[00009]: 0
```

The system table RDB$GENERATORS holds the defined sequence details but no values for each one. It does have an RDB$GENERATOR_ID column and this starts from 1, not zero. And increments by 1 for each new sequence. Where does this number come from?

Looking in the blank database we created, we can see that there are 9 sequences created for system use:

```
SQL> SELECT RDB$GENERATOR_ID, RDB$GENERATOR_NAME
CON> FROM RDB$GENERATORS
CON> ORDER BY RDB$GENERATOR_ID;

RDB$GENERATOR_ID RDB$GENERATOR_NAME
================ ==================
               1 RDB$SECURITY_CLASS
               2 SQL$DEFAULT
               3 RDB$PROCEDURES
               4 RDB$EXCEPTIONS
               5 RDB$CONSTRAINT_NAME
               6 RDB$FIELD_NAME
               7 RDB$INDEX_NAME
               8 RDB$TRIGGER_NAME
               9 RDB$BACKUP_HISTORY
```

This is a clue, take a look at `Sequence[00000]`, above, and see that it contains the value 9. I suspect therefore, that the very first sequence is used to generate the `RDB$GENERATOR_ID` value when a new sequence is created. One way to find out is to create a new sequence.

```
SQL> CREATE SEQUENCE NEW_GENERATOR;
SQL> SET GENERATOR NEW_GENERATOR TO 666;
SQL> COMMIT;

SQL> SELECT RDB$GENERATOR_ID, RDB$GENERATOR_NAME
CON> FROM RDB$GENERATORS
CON> WHERE RDB$GENERATOR_ID > 9;

RDB$GENERATOR_ID RDB$GENERATOR_NAME
================ ==================
              10 NEW_GENERATOR
```

So far, so good, we see a new sequence. Time to hexdump the database file's generator page again:

```
tux> ./fbdump ../blank2.fdb -p 148

FBDUMP 1.00 - Firebird Page Dump Utility

DATABASE PAGE DETAILS - Page 148
        Page Type: 9
PAGE DATA
        ...

        There are 10 sequences defined:

        Sequence[00000]: 10
        Sequence[00001]: 0
        Sequence[00002]: 3
```

```
        Sequence[00003]: 0
        Sequence[00004]: 0
        Sequence[00005]: 0
        Sequence[00006]: 0
        Sequence[00007]: 0
        Sequence[00008]: 0
        Sequence[00009]: 0
        Sequence[00010]: 666
```

We can see that `Sequence[00010]`, that a new sequence has been created. The value in this sequence is 666 in decimal. In addition, we can see that `Sequence[00000]` has increased to 10. So it looks remarkably like the `RDB$GENERATOR_ID` is itself obtained from a sequence that *never* appears in `RDB$GENERATORS`.

The value, stored in Sequence[n], appears to be the *last value* that was used and not the *next value* to be issued. It is also the total number of sequences that have been created thus far in the database, provided, that the value in `gpg_sequence` is zero.

I wonder what happens when we drop a sequence?

```
SQL> DROP SEQUENCE NEW_GENERATOR;
SQL> COMMIT;

SQL> SELECT RDB$GENERATOR_ID, RDB$GENERATOR_NAME
CON> FROM RDB$GENERATORS
CON> WHERE RDB$GENERATOR_ID > 9;

SQL>
```

We can see that the sequence is dropped from the `RDB$GENERATORS` table, what about in the generator page in the database?

```
tux> ./fbdump ../blank2.fdb -p 148

FBDUMP 1.00 - Firebird Page Dump Utility

DATABASE PAGE DETAILS - Page 148
        Page Type: 9
PAGE DATA
        ...

        There are 10 sequences defined:

        Sequence[00000]: 10
        Sequence[00001]: 0
        Sequence[00002]: 3
        Sequence[00003]: 0
        Sequence[00004]: 0
```

```
        Sequence[00005]: 0
        Sequence[00006]: 0
        Sequence[00007]: 0
        Sequence[00008]: 0
        Sequence[00009]: 0
        Sequence[00010]: 666
```

The generator page has *not* changed. Sequence[00010] still remains at its previous value — 666 — but this 64 bits of database page representing our recently dropped sequence can never be used again. It has ceased to be a sequence and has become wasted space.

Given that `RDB$GENERATOR_ID` is itself generated from `Sequence[00000]` and cannot therefore reuse any allocated `RDB$GENERATOR_ID`, it is not surprising that the simplest way of handling a dropped sequence is simply to ignore it.

If you are creating and dropping sequences frequently, you may end up with a lot of unused sequences. You can restore these to a usable state by dumping and restoring the database:

```
tux> # Shutdown & backup the database...
tux> gfix -shut -tran 60 ../blank2.fdb
tux> gbak -backup ../blank2.fdb ./blank2.fbk

tux> # Replace (!) and restart the database...
tux> gbak -replace  ./blank2.fbk ../blank2.fdb
```

The above will cause the loss of the database if anything goes wrong. The commands used overwrite the `blank2.fdb` database from the dumpfile. If the dumpfile is corrupt, then we will lose the database as the recovery starts by wiping the database.

If we now dump the generator page as before, we see the following:

```
> ./fbdump ../blank2.fdb -p 148

FBDUMP 1.00 - Firebird Page Dump Utility

DATABASE PAGE DETAILS - Page 148
        Page Type: 9
PAGE DATA
        ...

        There are 9 sequences defined:

        Sequence[00000]: 9
        Sequence[00001]: 0
        Sequence[00002]: 3
        Sequence[00003]: 0
        Sequence[00004]: 0
```

```
        Sequence[00005]: 0
        Sequence[00006]: 0
        Sequence[00007]: 0
        Sequence[00008]: 0
        Sequence[00009]: 0
```

We now see that the deleted sequence has gone, and the value in `Sequence[00000]` has reduced by one (the number of deleted sequences) to suit. If we now create a brand new sequence, it will reuse the slot previously occupied by our deleted sequence.

```
SQL> CREATE SEQUENCE ANOTHER_SEQUENCE;
SQL> COMMIT;
```

Dumping the generator page again, we see:

```
tux> ./fbdump ../blank2.fdb -p 148

FBDUMP 1.00 - Firebird Page Dump Utility

DATABASE PAGE DETAILS - Page 148
        Page Type: 9
PAGE DATA
        ...

        There are 10 sequences defined:

        Sequence[00000]: 10
        Sequence[00001]: 0
        Sequence[00002]: 3
        Sequence[00003]: 0
        Sequence[00004]: 0
        Sequence[00005]: 0
        Sequence[00006]: 0
        Sequence[00007]: 0
        Sequence[00008]: 0
        Sequence[00009]: 0
        Sequence[00010]: 0
```

Bearing in mind that in ODS 11 onwards, a sequence is a 64 bit value, how many sequences can we store on a page? The answer will be (page size - 32 bytes)/8 and we are allowed a maximum of 32,767 sequences in any one database. With a 4K page size this would mean sequence 508 would be the first on the next page.

Because there is no apparent next and previous page numbers on a generator page, how does the database know where to find the actual page that the generator values are stored on? `RDB$PAGES` is a system table that the main database header page holds the page number for. This allows the system, on startup, to determine where its internal data can be found. For because sequences live,

as it were, in RDB$GENERATORS we can look in RDB$PAGES as follows, to find the actual page number(s):

```
SQL> SELECT *
CON> FROM RDB$PAGES
CON> WHERE RDB$PAGE_TYPE = 9;

RDB$PAGE_NUMBER RDB$RELATION_ID RDB$PAGE_SEQUENCE RDB$PAGE_TYPE
=============== =============== ================= =============
            148               0                 0             9
```

The RDB$RELATION_ID is zero because this is not actually the location of a relation (table) in the database itself, but the location of a specific page that we are after. Given that RDB$PAGE_SEQUENCE = 0 and RDB$PAGE_TYPE = 9 we see that the first generator page is located on page 148 of the database.

If there are more than one page, then the page that has gpg_sequence set to zero is the first one and the first sequence on that page is the count of all sequences created (and possibly deleted) within the database. If the gpg_sequence is non-zero, then there is no way to tell how many sequences on that page are actually valid and even when the gpg_sequence is zero, unless the database has been restored since any sequences were last deleted, it is not possible to determine which sequences on the page are still valid. (Unless you have access to the RDB$GENERATOR_ID in RDB$GENERATORS of course.)

12.1. Creating Lots Of Sequences

When you create a new blank database, the first generator page is created as part of the new database. It has to be this way because there are nine system sequences created, as described above. (Well, there are 10 actually, but no-one has access to the first one!)

When the user starts creating new sequences, they will be added to the existing generator page. However, once a new page is required things change!

Given that there can be 508 sequences, in total, on a single 4 Kb database page, then when sequence 509 is created a new page — of type 0x09 — will be required. If the new sequence is not given an initial value, then the new page is not created yet. An entry *will* be created in RDB$PAGES with RDB$PAGE_SEQUENCE set correctly (to match what will be in the gpg_sequence field in the page structure when it is finally created) and a new sequence will be stored in RDB$GENERATORS, but nothing will happen to extend the database with the required new page until such time as either:

- The sequence value is read within a transaction; or
- The sequence number is explicitly set to a new value.

It is only now that the required generator page is actually created and written to the (end of) the database file. The following explains the sequence of events that take place when a brand new blank database is extended by the creation of an additional 5,000 sequences.

1. A blank database has 10 pre-created sequences used internally — nine are visible in RDB$GENERATORS, one is hidden. A generator page exists and the details can be found in RDB$PAGES. Page 148 is the first generator page in a 4 Kb page size database. The database file is 161 pages long (659,456 bytes).

```
tux> isql
Use CONNECT or CREATE DATABASE to specify a database

SQL> CREATE DATABASE 'seq.fdb';

SQL> SHELL;

tux> ls -l seq.fdb
-rw------- 1 firebird firebird 659456 2010-05-12 11:26 seq.fdb

tux> exit

SQL> SELECT RDB$GENERATOR_ID,
CON>     RDB$GENERATOR_NAME
CON> FROM RDB$GENERATORS
CON> ORDER BY RDB$GENERATOR_ID;

RDB$GENERATOR_ID RDB$GENERATOR_NAME
================ ==================
               1 RDB$SECURITY_CLASS
               2 SQL$DEFAULT
               3 RDB$PROCEDURES
               4 RDB$EXCEPTIONS
               5 RDB$CONSTRAINT_NAME
               6 RDB$FIELD_NAME
               7 RDB$INDEX_NAME
               8 RDB$TRIGGER_NAME


SQL> SELECT *
CON> FROM RDB$PAGES
CON> WHERE RDB$PAGE_TYPE = 9;

RDB$PAGE_NUMBER RDB$RELATION_ID RDB$PAGE_SEQUENCE RDB$PAGE_TYPE
=============== =============== ================= =============
            148               0                 0             9

SQL> COMMIT;
```

2. The user creates a set of 5,000 new sequences. The database extends to accommodate the data being written into the system table RDB$GENERATORS, but there are no new generator pages written. The database is now 256 pages long (1,048,576 bytes).

   RDB$PAGES still shows that page 148 is the only generator page in the database.

```
SQL> INPUT gens.sql;

SQL> SELECT *
CON> FROM RDB$PAGES
```

```
CON> WHERE RDB$PAGE_TYPE = 9;

RDB$PAGE_NUMBER RDB$RELATION_ID RDB$PAGE_SEQUENCE RDB$PAGE_TYPE
=============== =============== ================= =============
            148               0                 0             9

SQL> SHELL;

tux> ls -l seq.fdb
-rw------- 1 firebird firebird 1048576 2010-05-12 11:28 seq.fdb

tux> exit
```

3. A transaction touches the final sequence — which has RDB$GENERATOR_ID = 5,009 — by reading its value (without changing it). At this point a new generator page is created and written to the database. The page has gpg_sequence set to 9, which is the correct page for sequence number 5,009. The database is now 257 pages in size (1052672 bytes).

```
SQL> SELECT RDB$GENERATOR_ID,RDB$GENERATOR_NAME
CON> FROM RDB$GENERATORS
CON> WHERE RDB$GENERATOR_ID = (
CON>   SELECT MAX(RDB$GENERATOR_ID)
CON>   FROM RDB$GENERATORS
CON> );

RDB$GENERATOR_ID RDB$GENERATOR_NAME
================ ==================
            5009 RANDOM_SEQ_4994


SQL> SELECT GEN_ID(RANDOM_SEQ_4994, 0)
CON> FROM RDB$DATABASE;

               GEN_ID
=====================
                    0

SQL> SHELL;

tux> ls -l seq.fdb
-rw------- 1 firebird firebird 1052672 2010-05-12 11:33 seq.fdb

tux> exit
```

RDB$PAGES shows that there are now two pages in the database with type 9. The original page 148 and a new page 256. Looking at the database file itself, however, shows that it is actually 257 pages long. Page 257, the last page, has page type zero — which is not a defined page type and, doesn't appear in RDB$PAGES.

```
SQL> SELECT *
CON> FROM RDB$PAGES
CON> WHERE RDB$PAGE_TYPE = 9
CON> OR RDB$PAGE_NUMBER = 257;

RDB$PAGE_NUMBER RDB$RELATION_ID RDB$PAGE_SEQUENCE RDB$PAGE_TYPE
=============== =============== ================= =============
            148               0                 0             9
            256               0                 9             9

SQL> SHELL;

tux> ./fbdump seq.fdb -p 257

FBDUMP 1.00 - Firebird Page Dump Utility

DATABASE PAGE DETAILS - Page 257
        Page Type: 0
```

The RDB$PAGE_SEQUENCE in RDB$PAGES for the new page, page 256, is set to 9 which corresponds to the gpg_sequence number in the actual page.

```
tux> ./fbdump seq.fdb -p 256

FBDUMP 1.00 - Firebird Page Dump Utility

DATABASE PAGE DETAILS - Page 256
        Page Type: 9
PAGE DATA
        Sequence: 9
...
```

4. A separate transaction changes the value of the sequence with RDB$GENERATOR_ID = 520, which is to be found on the second page of sequences. This page doesn't yet exist and is created with page number 257. Looking at RDB$PAGES shows that this new page exists in the database. The database file has extended now to 258 pages or 1,056,768 bytes.

   The sequence in question, however, still has the value zero. (The transaction has yet to commit.)

```
SQL> SELECT RDB$GENERATOR_NAME
CON> FROM RDB$GENERATORS
CON> WHERE RDB$GENERATOR_ID = 520;

RDB$GENERATOR_NAME
==================
RANDOM_SEQ_534

SQL> SET GENERATOR RANDOM_SEQ_534 TO 666;
```

```
SQL> SELECT *
CON> FROM RDB$PAGES
CON> WHERE RDB$PAGE_TYPE = 9;

RDB$PAGE_NUMBER RDB$RELATION_ID RDB$PAGE_SEQUENCE RDB$PAGE_TYPE
=============== ============== ================= =============
            148              0                 0             9
            256              0                 9             9
            257              0                 1             9

SQL> SHELL;

tux> ls -l seq.fdb
-rw------- 1 firebird firebird 1056768 2010-05-12 13:07 seq.fdb

tux> ./fbdump seq.fdb -p 257

FBDUMP 1.00 - Firebird Page Dump Utility

DATABASE PAGE DETAILS - Page 257
        Page Type: 9
PAGE DATA
        Sequence: 1
        Waste1: 0
        Waste2: 0
        Waste3: 0
        Waste4: 0
        Waste5: 0

        This is not the first generator page.
        Total generator count unknown.
        There are [a maximum of] 508 sequences located on this page.

        Sequence[00508]: 0
        ...
        Sequence[00520]: 0
        ...
```

Only after a commit does the sequence takes the new value of 666.

```
tux> exit

SQL> COMMIT;

SQL> SHELL;

tux> ./fbdump seq.fdb -p 257

FBDUMP 1.00 - Firebird Page Dump Utility
```

```
...
Sequence[00520]: 666
...
```

Chapter 13. Write Ahead Log Page — Type 0x0a

Every database has one Write Ahead Log page (WAL) which is currently always located at page 2.

Discussions have taken place on the Firebird development mailing list on removing this page altogether as it is not used and simply wastes space that could be better used elsewhere. From Firebird 3.0 it is likely there will not be a WAL page in any new databases.

The C code representation of the WAL page is:

```
struct log_info_page
{
    pag log_header;
    SLONG log_flags;
    ctrl_pt log_cp_1;
    ctrl_pt log_cp_2;
    ctrl_pt log_file;
    SLONG log_next_page;
    SLONG log_mod_tip;
    SLONG log_mod_tid;
    SLONG log_creation_date[2];
    SLONG log_free[4];
    USHORT log_end;
    UCHAR log_data[1];
};
```

As this structure is no longer in use within the database, it is effectively, a wasted page. Looking at a hexdump of the WAL page in a new database, we see the following:

```
tux> ./fbdump ../blank.fdb -p 2

FBDUMP 1.00 - Firebird Page Dump Utility

DATABASE PAGE DETAILS - Page 2
        Page Type: 10
PAGE DATA
        Flags: 0x00000000
        Log Control Point 1:
                Sequence: 0
                Offset: 0
                P_offset: 0
                Fn_Length: 0
        Log Control Point 2:
                Sequence: 0
                Offset: 0
```

```
                P_offset: 0
                Fn_Length: 0
        Current File:
                Sequence: 0
                Offset: 0
                P_offset: 0
                Fn_Length: 0
        Next Page: 0
        Mod Tip: 0
        Mod Transaction Id: 0
        Creation Date: COMING SOON
        Log Free Space: 0 0 0 0
        Log End: 0
```

The remainder of the page is filled with binary zeros.

Because the WAL is no longer in use, and may even be dropped completely from Firebird 3.0 onwards, it will not be discussed further.

Chapter 14. External Table Format

Firebird external tables allow you to read row data from and write them to files separate from the database. External tables use a *binary* format.

The *binary* format of external table files is similar to the in-memory image of row data, and row data on a Data Page — Type 0x05 *after* RLE decompression, but without a null bitmap.

By using **only** `CHAR(n)` with **one** *single-byte* character set for all columns, and an extra `CHAR(1)` or `CHAR(2)` column for a linebreak, the external table file format can also be treated as a *fixed-width text* format. However, using any other datatype, or mixing multiple character sets, or using a variable-length character set like `UTF8` will break any assumptions for *fixed-width text*, and make the format truly binary.

This chapter describes the format of the external table file and individual datatypes.

See section `CREATE TABLE` and specifically its subsection External Tables in the *Firebird 5.0 Language Reference* for additional information.

The external table file format has been stable for the past two decades or so, except for additions of new datatypes. However, it is effectively an internal implementation detail: the in-memory image of a row. As such, it could change in future Firebird versions.

In addition, it is possible that our description in this chapter makes assumptions about things like alignment and datatype sizes that do not hold on all platforms (CPU architecture, OS, and/or compiler).

14.1. General Binary Format

The binary format is row-oriented, without explicit separators for columns or rows (except alignment of column values). The format and length of the row is determined by its definition in DDL.

The following section describe some important aspects of the format.

14.1.1. Endianness

The column data for all datatypes except `CHAR` and `BOOLEAN` is endian-sensitive. For example, for a Firebird server running on x86-64 (AMD64), the format is little-endian.

When writing to or reading from an external table file, it is important to use the right endianness of the Firebird server. The commonly used platforms for Firebird are all little-endian (though we recommend to double-check in case of aarch64 builds distributed by third-parties).

Some datatypes are compound datatypes (e.g. compare C/C++ structs). The endianness is per element of such a compound datatype, not over the whole datatype. That is, endianness does not change the order of components, only the byte order within a component.

14.1.2. Row Format and Virtual Position

The row format of an external table file has some complications that need extra attention when reading or writing a row *if* you use datatypes other than `CHAR`:

- The null bitmap is not written in the file, but the length of the null bitmap needs to be accounted for in the virtual position in the row
- Columns must be aligned, based on the virtual position in the row, depending on the datatype
- The alignment of the first column must not be written to the file, but must be accounted for in the virtual position in the row.

We're using the term *virtual position in the row* to reflect it is not the actual position relative to the start of the row in the file on disk. In essence, it is the position in the in-memory image of the row, including a prefix that is not written to the external table file.

Null Bitmap

The in-memory image of a row has a null bitmap, which is used to determine which rows are `NULL`. Columns in an external table cannot be `NULL`, and the null bitmap of a row is not written in the external table.

Unfortunately, the length of the null bitmap **must** be accounted for in the virtual position in the row for purposes of determining the alignment of columns.

The length of the null bitmap is 4 bytes per 32 columns. It can be calculated as `4 * (1 + (column-count - 1) / 32)`. This means that 1-32 columns are 4 bytes, 33-64 columns are 8 bytes, and so on.

Alignment

Except for `CHAR` and `BOOLEAN`, all datatypes need to be aligned so they start at a multiple of 2, 4, or 8 bytes from the virtual start of the row. In an alternative interpretation, `CHAR` and `BOOLEAN` have an alignment of 1, which means they are always aligned.

A column should be aligned by writing `0x00` bytes until alignment is achieved. The number of bytes to write can be determined using `alignment - (virtual-position % alignment)`, if the result is `(0, alignment)` (or, `0 < result < alignment`).

The alignment of the first column **must not** be written to disk, but it **must** be accounted for in the virtual position in the row.

In our descriptions of the datatypes in the following section, Datatype Formats, we will note the alignment for datatypes.

It is possible that the alignment depends on the platform (CPU architecture, OS, and/or compiler) of the Firebird build. As far as we're aware, all platforms built by the Firebird project use the same alignments.

It is possible different alignments are used in Firebird builds of third-parties.

14.2. Datatype Formats

The following sections describe the encoding of the various datatypes available for external tables.

Datatypes like BLOB and arrays are not supported in external tables.

14.2.1. String and Binary Datatypes

CHAR

Length

n * *bpc* bytes

Where

n number of characters (the *n* in CHAR(n))

bpc maximum number of bytes per character of the character set

Alignment

1 byte (no alignment)

If the actual value is shorter than the column length in bytes, it must be padded with a padding byte. The padding byte is 0x20 (SPACE) for most character sets. For character set OCTETS (or type BINARY), the padding byte is 0x00 (NUL).

See appendix Character Sets and Collations of the *Firebird 5.0 Language Reference* for maximum number of bytes per character of a character set.

Variable Length Character Sets

When writing data in a variable-length character sets like UTF8 (bpc = 4) and UNICODE_FSS (bpc = 3), Unicode codepoints (characters) take 1-4 or — UNICODE_FSS — 1-3 bytes. The number of Unicode codepoints (characters) must not exceed *n*. The remaining bytes written **must** be the padding byte 0x20 (SPACE). Failure to do so will raise a string truncation error when Firebird reads the row.

In Firebird 3.0 and older, this restriction does not apply to UNICODE_FSS. In those versions, UNICODE_FSS is only limited by the byte length (3*n), not the character length *n*.

As an example, a CHAR(4) CHARACTER SET UTF8 has a length of 4*4 = 16 bytes, while a CHAR(5) CHARACTER SET UTF8 has a length of 5*4 = 20 bytes, and with value ABCD they are written as (in hex):

Example of ABCD *in* CHAR(4) CHARACTER SET UTF8

```
41 42 43 44 20 20 20 20 20 20 20 20 20 20 20 20
```

Example of ABCD *in* CHAR(5) CHARACTER SET UTF8

```
41 42 43 44 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

These padding bytes are one of the reasons why RLE compression is so important when rows are stored on a data page.

BINARY

Length

n bytes

Where

n number of characters (the *n* in `BINARY(n)`)

Alignment

1 byte (no alignment)

Given `BINARY(n)` is an alias for `CHAR(n) CHARACTER SET OCTETS`, the column format is the same as `CHAR`, with a *bpc* of 1 and a padding byte of 0x00 (NUL).

VARCHAR

Length

2 + *n* * *bpc* bytes

Where

n number of characters (the *n* in `VARCHAR(n)`)

bpc maximum number of bytes per character of the character set

Alignment

2 bytes

A `VARCHAR(n)` is a compound datatype, consisting of a 16-bit integer (a `SMALLINT`) with the actual data length in bytes, followed by `n * bpc` of string data. Even if the actual data length is shorter, you must always write the maximum byte length.

The string data part is essentially the same as for `CHAR`, except the padding up to the maximum byte length should be done with 0x00 (NUL), not 0x20 (SPACE). The section on Variable Length Character Sets also applies for `VARCHAR`.

Storage-wise, `VARCHAR(n)` requires two bytes (plus up to 1 byte for alignment) more than an equivalent `CHAR(n)`. The benefit compared to `CHAR` is that the value retains its actual length when queried from the external table; the value read is not padded with spaces — or 0x00 (NUL) in case of character set `OCTETS`/`VARBINARY(n)` — up to *n* characters.

Example of `ABCD` *in* `CHAR(5) CHARACTER SET ASCII`

```
04 00 41 42 43 44 00
```

Example of `ABCD` *in* `CHAR(5) CHARACTER SET UTF8`

```
04 00 41 42 43 44 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

See appendix Character Sets and Collations of the *Firebird 5.0 Language Reference* for maximum number of bytes per character of a character set.

`VARBINARY`

Length

2 + *n* bytes

Where

n number of characters (the *n* in `VARBINARY(n)`)

Alignment

2 bytes

Given `VARBINARY(n)` is an alias for `VARCHAR(n) CHARACTER SET OCTETS`, the column format is the same as `VARCHAR`, with a *bpc* of 1.

14.2.2. Integral Datatypes

`SMALLINT`

Length

2 bytes (16 bits)

Alignment

2 bytes

`INTEGER`

Length

4 bytes (32 bits)

Alignment

4 bytes

`BIGINT`

Length

8 bytes (64 bits)

Alignment

8 bytes

`INT128`

Length

16 bytes (128 bits)

Alignment

8 bytes

14.2.3. Fixed-Point Datatypes

The format of the fixed-point types `NUMERIC` and `DECIMAL` is the same as that of the integral type used as the backing type. The value written is the unscaled value.

`NUMERIC`

For `NUMERIC[(p[,s])]`, the defaults if not specified are `p = 9` and `s = 0` (equivalent to `INTEGER`).

Length

Varies, determined by backing integral type

Alignment

Varies, determined by backing integral type

The following integral types are used to write the unscaled value (`numeric-value * 10^s` truncated to whole numbers):

`p <= 4`	`SMALLINT`
`4 < p <= 9`	`INTEGER`
`9 < p <= 18`	`BIGINT`[1].
`9 < p <= 38`	`INT128`

`DECIMAL`

For `DECIMAL[(p[,s])]`, the defaults if not specified are `p = 9` and `s = 0` (equivalent to `INTEGER`).

Length

Varies, determined by backing integral type

Alignment

Varies, determined by backing integral type

The following integral types are used to write the unscaled value (`decimal-value * 10^s` truncated to whole numbers):

`p <= 9`	`INTEGER`
`9 < p <= 18`	`BIGINT`[1].
`9 < p <= 38`	`INT128`

14.2.4. Approximate Floating-Point Datatypes

FLOAT

Length

4 bytes (32 bits)

Alignment

4 bytes

The format is IEEE 754 binary32, a.k.a. *single precision*. This applies to FLOAT, REAL, and FLOAT(bin_prec) with 1 <= bin_prec <= 24 (Firebird 4.0 or higher).

REAL and FLOAT(bin_prec) with 1 <= bin_prec <= 24 are aliases for FLOAT without precision.

DOUBLE PRECISION

Length

8 bytes (64 bits)

Alignment

8 bytes

The format is IEEE 754 binary64, a.k.a. *double precision*. This applies to DOUBLE PRECISION, and FLOAT(bin_prec) with 25 <= bin_prec <= 53 (Firebird 4.0 or higher).

FLOAT(bin_prec) with 25 <= bin_prec <= 53 is an alias for DOUBLE PRECISION.

14.2.5. Decimal Floating-Point Types

DECFLOAT

For DECFLOAT[(dec_prec)], the default if not specified is dec_prec = 34; possible values for *dec_prec* are 16 or 34

Length

dec_prec = 16 8 bytes

dec_prec = 34 16 bytes

Alignment

8

The encoding of these datatypes is too complex to describe here, so we defer to the off-site resources linked below.

The format is:

dec_prec = 16 IEEE 754 decimal64

dec_prec = 34 IEEE 754 decimal128

More information on these datatypes can be found on General Decimal Arithmetic. For a specification of the encoding of these types, see Decimal Arithmetic Encodings, or — we assume — the IEEE 754 standard.

Firebird uses The decNumber Library for encoding and decoding `decimal64` and `decimal128` (see also https://github.com/dnotq/decNumber).

A Java implementation of the `decimal64` and `decimal128` encoding (written by Mark Rotteveel, the author of this chapter), can be found on https://github.com/FirebirdSQL/decimal-java. This library is used by Jaybird (the Firebird JDBC driver).

14.2.6. Date and Time Datatypes

`DATE`

Length

4 bytes

Alignment

4 bytes

The date is a Modified Julian Date encoded as a 32-bit signed integer. The Modified Julian Date is calculated as the number of days since November 17, 1858 (1858-11-17).

Effectively, a `DATE` is handled the same as `INTEGER`.

`TIME [WITHOUT TIME ZONE]`

Length

4 bytes

Alignment

4 bytes

The time is the number of 100 microseconds since midnight encoded as a 32-bit integer.

Effectively, a `TIME` is handled the same as `INTEGER`.

`TIME WITH TIME ZONE`

Length

8 bytes

Alignment

8 bytes

The time is the number of 100 microseconds since midnight UTC (offset 00:00) encoded as a 32-bit integer, and a 16-bit unsigned integer value for the time zone offset *or* id of the named zone (see Time zone encoding). The remaining two bytes are unused and should be 0x00 (NUL)[2].

As correct derivation of the time in a named zones requires a date, Firebird applies the time zone

rules for the date 2020-01-01.

Effectively, a `TIME WITH TIME ZONE` is a `TIME [WITHOUT TIME ZONE]` at offset +00:00 (UTC), followed by time zone information that can be used to determine the time in the specified offset or named zone.

Time zone encoding

The 16-bit unsigned integer value with the time zone information is either:

`0` – `2878`
: Time zone offset

  This is the number of minutes starting at offset -23:59 up to offset +23:59[3][4]. This means that offset +00:00 (UTC) is 1439.

`2979` – `65535`
: Time zone id of the named time zone.

  The range starts at 65535 (GMT) counting down — in Firebird 5.0.3 with its default 2025b time zone database, the lowest id is 64898 (America/Coyhaique).

  Time zone ids are stable; an id will not change to a different zone, but new ICU time zone database versions may add new time zones.

You can map a time zone id to the named zone using the table `RDB$TIME_ZONES`. The procedure `RDB$TIME_ZONE_UTIL.TRANSITIONS` can be used to find the applicable offset.

`TIMESTAMP [WITHOUT TIME ZONE]`

Length

8 bytes

Alignment

8 bytes

`TIMESTAMP [WITHOUT TIME ZONE]` is a compound datatype consisting of — in order — `DATE` and `TIME [WITHOUT TIME ZONE]`.

`TIMESTAMP WITH TIME ZONE`

Length

12 bytes

Alignment

8 bytes

`TIMESTAMP WITH TIME ZONE` is a compound datatype consisting of — in order — `TIMESTAMP [WITHOUT TIME ZONE]` with the datetime at offset +00:00 (UTC), a 16-bit unsigned integer for the time zone offset *or* id of the named zone (see Time zone encoding). The remaining two bytes are unused and should be 0x00 (NUL)[2].

Effectively, a TIMESTAMP WITH TIME ZONE is a TIMESTAMP [WITHOUT TIME ZONE] at offset +00:00 (UTC), followed by time zone information that can be used to determine the date and time in the specified offset or named zone.

14.2.7. Other Datatypes

BOOLEAN

Length

1 byte

Alignment

1 byte (no alignment)

Either 0x00 for FALSE, or 0x01 for TRUE[5].

[1] For dialect 1, DOUBLE PRECISION

[2] Firebird may write non-zero values into those two bytes

[3] When parsing datetime literals, Firebird only accepts -14:00 — +14:00

[4] In practice, real time zones have offsets in the range -12:00 to +14:00 (source)

[5] Firebird reads any non-zero value as TRUE, but only writes 0x01

Appendix A: Fbdump

Throughout some of this document you may have noticed that I've been using a tool named `fbdump` to display internal representations of Firebird Database pages. Maybe some of you are wondering where to find it in the Firebird installation directory.

`Fbdump` is a utility that I had to write myself while writing this document. I'm (almost) happy to let it loose into the wild, but it's probably the worst code you will ever have the misfortune to see. It wasn't written to a plan, I simply added bits here and there as I needed them. It's not nice.

Firebird itself comes with a page dumping mechanism, but you need to be running a debug version of Firebird in order to use it. The good news about doing it this way, rather than using `fbdump` is that the official way will keep up with ODS changes. There is no guarantee that `fbdump` will.

Sorry.

Appendix B: Document history

The exact file history is recorded in our *git* repository; see https://github.com/FirebirdSQL/firebird-documentation

Revision History

1.5	09 Feb 2026	MR	Added chapter External Table Format (#94)
1.4	02 Aug 2025	*	Improved description of `dpg_rpt` and corrected its offset — contributed by Arthur Aurajo (#225)
1.3	31 Jul 2025	*	• Fixed off-by-one range in Database Header Page — Type 0x01 for `hdr_shadow_count` — contributed by Arthur Aurajo (#224) • MR — Fix rendering error • MR — Added caution in introduction that newer ODS versions may be different
1.2	13 Aug 2021	*	Fix offset off by 2 bytes in Data Page — Type 0x05 — contributed by Rafael Estevam Reis (#161)
1.1	04 Aug 2020	MR	Conversion to AsciiDoc, minor copy-editing
1.0	03 Nov 2009	ND	Created a new manual.

Appendix C: License notice

The Original Documentation is titled *Firebird Internals*.

The Initial Writer of the Original Documentation is: Norman Dunbar.