Rekordbox Export Structure Analysis

James Elliott
Deep Symmetry, LLC

February 26, 2019

Abstract

The files written to external media by rekordbox for use in player hardware contain a wealth of information that can be used in place of queries to the remotedb server on the players, which is important because they can be obtained from the players' NFS servers, even if there are four players in use sharing the same media. Under those circumstances, remotedb queries are impossible. This article documents what has been learned so far about the files, and how to interpret them.

Contents

1 Database Exports

The starting point for finding track metadata from a player is the database export file, which can be found within rekordbox media at the path `PIONEER/rekordbox/export.pdb` (if you are using the Crate Digger `FileFetcher` to request this file, use that path as the `filePath` argument, and use a `mountPath` value of `/B/` if you want to read it from the SD slot, or `/C/` to obtain it from the USB slot).

The file is a relational database format designed to be efficiently used by very low power devices (there were deployments on 16 bit devices with 32K of RAM). Today you are most likely to encounter it within the Pioneer Professional DJ ecosystem, because it is the format that their rekordbox software uses to write USB and SD media which can be mounted in DJ controllers and used to play and mix music.

The file consists of a series of fixed size pages. The first page contains a file header which defines the page size and the locations of database tables of different types, by the index of their first page. The rest of the pages consist of the data pages for all of the tables identified in the header.

Each table is made up of a series of rows which may be spread across any number of pages. The pages start with a header describing the page and linking to the next page. The rest of the page is used as a heap: rows are scattered around it, and located using an index structure that builds backwards from the end of the page. Each row of a given type has a fixed size structure which links to any variable-sized strings by their offsets within the page.

As changes are made to the table, some records may become unused, and there may be gaps within the heap that are too small to be used by other data. There is a bit map in the row index that identifies which rows are actually present. Rows that are not present must be ignored: they do not contain valid (or even necessarily well-formed) data.

The majority of the work in reverse-engineering this format was performed by Henry Betts[1] and Fabian Lesniak[2], to whom I am hugely grateful.

1.1 File Header

Unless otherwise stated, all multi-byte numbers in the file are stored in little-endian byte order. Field names used in the Figures match the IDs assigned to them in the Kaitai Struct specification[3], unless that is

[1] `https://github.com/henrybetts/Rekordbox-Decoding`

[2] `https://github.com/flesniak/python-prodj-link`

[3] `https://github.com/Deep-Symmetry/crate-digger/blob/master/src/main/kaitai/rekordbox_pdb.ksy`

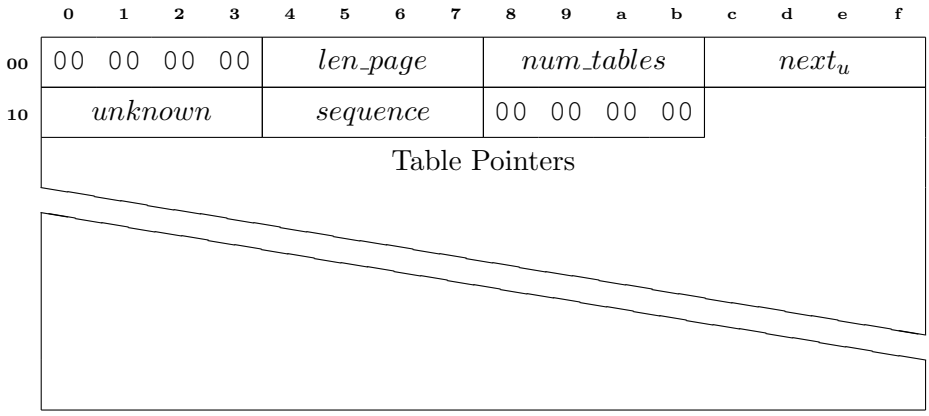

Figure 1: File Header

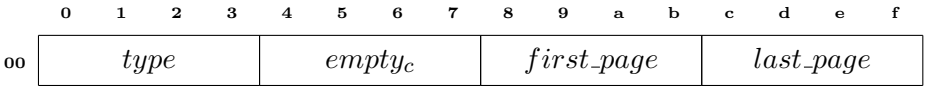

Figure 2: Table Pointer

too long to fit, in which case a subscripted abbreviation is used, and the text will mention the actual ID.

The first page begins with the file header, shown in Figure 1. The header starts with four zero bytes, followed by a four-byte integer, *len_page* at byte `04`, that establishes the size of each page (including this first one), in bytes. This is followed by another four-byte integer, *num_tables* at byte `08`, which reports the number of different tables that are present in the file. Each table will have a table pointer entry in the "Table pointers" section of the file header, described below, that identifies and locates the table.

The four-byte integer $next_u$ at byte `0c` has an unknown purpose, but Mr. Lesniak named it `next_unused_page` and said "Not used as any `empty_candidate`, points past the end of the file." The four-byte integer *sequence*, at byte `14`, was described "Always incremented by at least one, sometimes by two or three." and I assume this means it reflects a version number that rekordbox updates when synchronizing to the exported media.

Finally, there is another series of four zero bytes, and then the header ends with the list of table pointers which begins at byte `1c`. There are as many of these as specified by *num_tables*, and each has the structure shown in Figure 2.

Each Table Pointer is a series of four four-byte integers. The first, *type*, identifies the type of table being defined. The known table types are shown in Table 1. The second value, at byte `04` of the Table Pointer, was called `empty_candidate` by Mr. Lesniak. It may link to a chain of empty pages if the database is ever garbage collected, but this is speculation on my part.

Type	Name	Row Content
0	tracks	Track metadata: title, artist, genre, artwork ID, playing time, etc.
1	genres	Musical genres, for reference by tracks and searching.
2	artists	Artists, for reference by tracks and searching, Section 1.3.2.
3	albums	Albums, for reference by tracks and searching, Section 1.3.1.
4	labels	Music labels, for reference by tracks and searching.
5	keys	Musical keys, for reference by tracks, searching, and key matching.
6	colors	Color labels, for reference by tracks and searching.
7	playlist_tree	Describes the hierarchical tree structure of available playlists and folders grouping them.
8	playlist_entries	Links tracks to playlists, in the rightp order.
13	artwork	File paths of album artwork images, Section 1.3.3.
16	columns	Details not yet confirmed.
19	history	Records the tracks played during performances.

Table 1: Table Types

Other than the type, the two important values are $first\_page$ at byte `08` and *last_page* at byte `0c`. These tell us how to find the table. They are page indices, where the page containing the file header has index 0, the page with index 1 begins at byte *len_page*, and so on. In other words, the first page of the table identified by the current Table Pointer can be found within the file starting at the byte $len\_page \times first\_page$.

The table is a linked list of pages: each page contains the index of the next page after it. However, you need to keep track of the *last_page* value for the table, because it tells you not to try to follow the next page link once you reach the page with that index. (If you

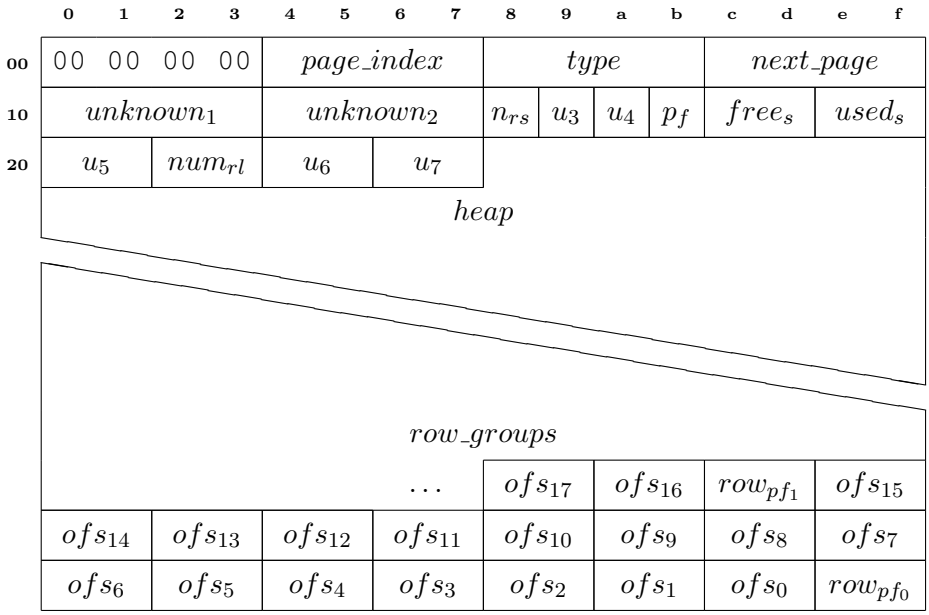

Figure 3: Table Page

do keep going, you will start reading pages of some different table.) The structure of the table pages themselves are described in the next section.

As far as we know, the remainder of the first page after the table pointers is unused.

1.2 Table Pages

The table header is followed by the table pages themselves. These each have the size specified by *len_page* in Figure 1, and the structure shown in Figure 3.

Data pages all seem to have the header structure described here, but not all of them actually store data. Some of them are "strange" and we have not yet figured out why. The discussion below describes how to recognize a strange page, and avoid trying to read it as a data page.

The first four bytes of a table page always seem to be zero. This is followed by a four-byte value *page_index* which identifies the index of this page within the list of table pages (the header has index 0, the first actual data page the index 1, and so on). This value seems to be redundant, because it can be calculated by dividing the offset of the start of the page by *len_page*, but perhaps it serves as a sanity check.

This is followed by another four-byte value, *type*, which identifies the type of the page, using the values shown in Table 1. This again seems redundant because the table header which was followed to reach this page also identified the table type, but perhaps it is another sanity check, or an alternate way to tell, when following page links, that you have reached the end of the table you are interested in. Speaking of which, the next four-byte value, *next_page*, is that link: it identifies the index at which the next page of this table can be found, as long as we have not already reached the final page of the table, as described in Section 1.1.

The exact meaning of $unknown_1$ is unclear. Mr. Flesinak said "sequence number (0→1: 8→13, 1→2: 22, 2→3: 27)" but I don't know how to interpret that. Even less is known about $unknown_2$. But *num_rows_small* at byte `18` within the page (abbrviated n_{rs} in Figure 3) holds the number of rows that are present in the page, unless *num_rows_large* (below) holds a value that is larger (but not equal to `1fff`). This seems like a strange mechanism for dealing with the fact that some tables (like playlist entries) have a lot of very small rows, too many to count with a single byte. But then why not just always use *num_rows_large*?

The purpose of the next two bytes are is also unclear. Of u_3 Mr. Flesniak said "a bitmask (first track: 32)", and he described u_4 as often 0, sometimes larger, especially for pages with a high number of rows (e.g. 12 for 101 rows).

Byte `1b` is called *page_flags* (abbrviated p_f in Figure 3). According to Mr. Flesniak, "strange" (non-data) pages will have the value `44` or `64`, and other pages have had the values `24` or `34`. Crate Digger considers a page to be a data page if $page\_flags \& \texttt{40} = 0$.

Bytes `1c-1d` are called *free_size* (abbrviated $free_s$ in Figure 3), and store the amount of unused space in the page heap (excluding the row index which is built backwards from the end of the page); *used_size* at bytes `1c-1d` (abbrviated $used_s$) stores the number of bytes that are in use in the page heap.

Bytes `20-21`, u_5, are of unclear purpose. Mr. Flesniak labeled them "(0→1: 2)."

Bytes `20-21`, *num_rows_large* (abbrviated num_{rl} in Figure 3) hold the number of entries in the row index at the end of the page when that value is too large to fit into *num_rows_small* (as mentioned above), and that situation seems to be indicated when this value is larger than *num_rows_small*, but not equal to `1fff`.

u_6 at bytes `24-25` seems to have the value `1004` for strange pages, and `0000` for data pages. And Mr. Flesniak describes u_7 at bytes `26-27` as "always 0 except 1 for history pages, num entries for strange pages?"

After these header fields comes the page heap. Rows are allocated within this heap starting at byte `28`. Since rows can be different sizes,

there needs to be a way to locate them. This takes the form of a row index, which is built from the end of the page backwards, in groups of up to sixteen row pointers along with a bitmask saying which of those rows are still part of the table (they might have been deleted).

The number of row index entries is determined by the value of either *num_rows_small* or *num_rows_large* as described above.

The bit mask for the first group of up to sixteen rows, labeled row_{pf_0} in Figure 3 (meaning "row presence flags group 0"), is found in the last two bytes of the page. The low order bit of this value will be set if row 0 is really present, the next bit if row 1 is really present, and so on. The two bytes before these flags, labeled ofs_0, store the offset of the first row in the page. This offset is the number of bytes past the end of the page header at which the row itself can be found. So if row 0 begins at the very beginning of the heap, at byte `28` in the page, ofs_0 would have the value `0000`.

As more rows are added to the page, space is allocated for them in the heap, and additional index entries are added at the end of the heap, growing backwards. Once there have been sixteen rows added, all of the bits in row_{pf_0} are accounted for, and when another row is added, before its offset entry ofs_{16} can be added, another row bit-mask entry row_{pf_1} needs to be allocated. And so the row index grows backwards towards the rows that are being added forwards, and once they are too close for a new row to fit, the page is full, and another page gets allocated to the table.

1.3 Table Rows

The structure of the rows themselves is determined by the *type* of the table, using the values shown in Table 1.

1.3.1 Album Rows

Album rows hold an album name and ID along with an artist association, with the structure shown in Figure 4. The unknown value at bytes `00-01` seems to usually have the values `80 00`. It is followed by a two-byte value Mr. Flesniak called $index\_shift$, although I don't know what that means, and another four bytes of unknown purpose. But at bytes `08-0b` we finally find a value we have a use for: *artist_id* holds the ID of an artist row associated with this track row. This is followed by *id*, the ID of this track row itself, at bytes `0c-0f`. We assume that there are index tables somewhere that would let us locate the page and row index of a record given its table type and ID, but we have not yet found and figured them out.

This is followed by five more bytes with unknown meaning, and the final byte in the row, ofs_name is a pointer to the track name (labeled

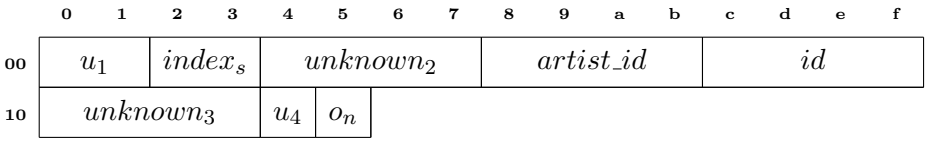

Figure 4: Album Row

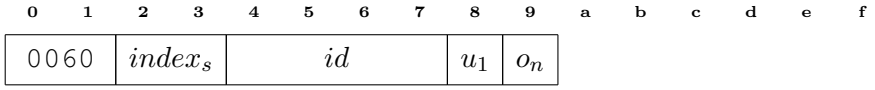

Figure 5: Artist Row with Nearby Name

o_n in Figure 4. To find the location of the name, add ofs_name bytes to the address of the start of the track row itself. The name itself is encoded in a surprisingly baroque way, explained in Section 1.4

1.3.2 Artist Rows

Artist rows hold an Artist name and ID, with the structure shown in Figure 5 or Figure 6. The *subtype* value at bytes `00-01` determines which variant is used. If the artist name was allocated close enough to the row to be reached by a single byte offset, offset, *subtype* has the value `0060`, and the row has the structure in Figure 5. If the name is too far away for that, *subtype* has the value `0064` and the row has the structure in Figure 6.

In either case, *subtype* is followed by the unexplained two-byte value found in all row types that Mr. Flesniak called $index\_shift$, and then by *id*, the ID of this artist row itself, at bytes `04-07`, an unknown value, and ofs_name_near (labeled o_n in Figure 5), the one-byte name offset used only in the first variant.

If *subtype* is `0064`, the value of ofs_name_near is ignored, and instead the two-byte value ofs_name_far (labeled o_{far} in Figure 6) is used.

Whichever name offset is used, it is a pointer to the artist name. To find the location of the name, add the value of the offset to the address of the start of the artist row itself. This gives the address of a DeviceSQL string holding the name, with the structure explained in Section 1.4

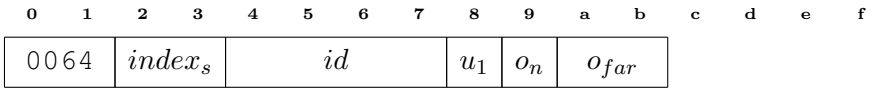

Figure 6: Artist Row with Far Name

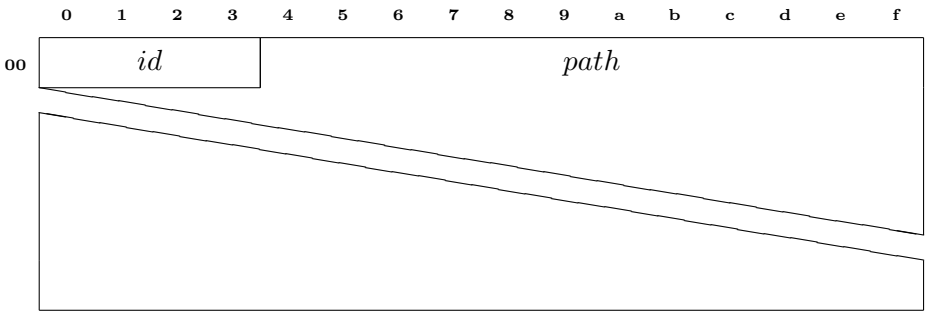

Figure 7: Artwork Row

1.3.3 Artwork Rows

Artwork rows hold an *id* (which tracks refer to) and the path at which the corresponding album art image file can be found, with the structure shown in Figure 7. Note that in this case, the DeviceSQL string *path* is embedded directly into the row itself, rather than being located elsewhere in the heap through an offset. The structure of the string itself is still as described in Section 1.4.

1.4 DeviceSQL Strings

Many row types store string values, sometimes by directly embedding them, but more often by storing an offset to a location elsewhere in the heap. In either case the string itself uses the strange structure described in this section. Strings can be stored in a variety of formats. The first byte of the structure, labeled *length_and_kind* in the parsed Kaitai Struct, identifies the encoding type and, when the value is odd, also the length (for short ASCII strings), as detailed in Section 1.4.3.

1.4.1 Long ASCII Strings

If *length_and_kind* has the value `40`, it is followed by a two-byte *length* field, and then followed by that many bytes of ASCII-encoded string data, as shown in Figure 8.

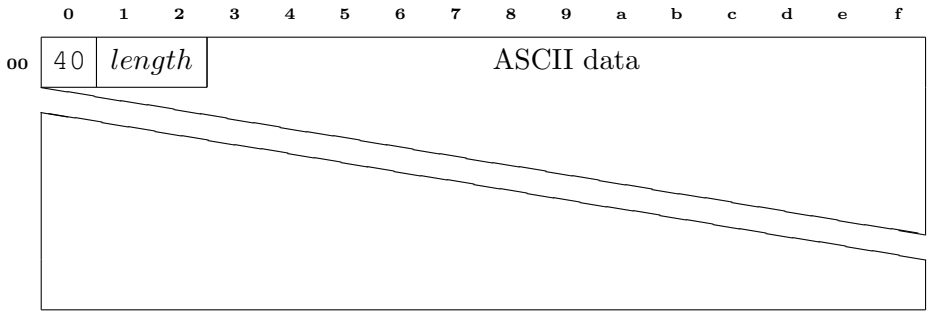

Figure 8: Long ASCII DeviceSQL String

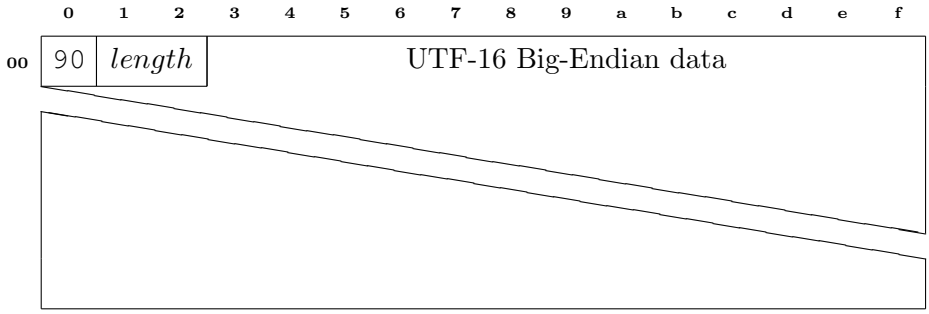

Figure 9: Long UTF-16-BE DeviceSQL String

1.4.2 Long UTF-16 Big-Endian Strings

If *length_and_kind* has the value `90`, it is followed by a two-byte *length* field, and then followed by that many bytes of UTF-16 big-endian encoded string data, as shown in Figure 9.

1.4.3 Short ASCII Strings

If *length_and_kind* has an odd value it is a *mangled_length*, labeled m_l in Figure 10. This means we are dealing with a short ASCII DeviceSQL string. To find the length of the string data (which immediately follows this byte), subtract 1 from *mangled_length*, divide it by 2, and subtract 1 again.

2 Analysis Files

To be written...

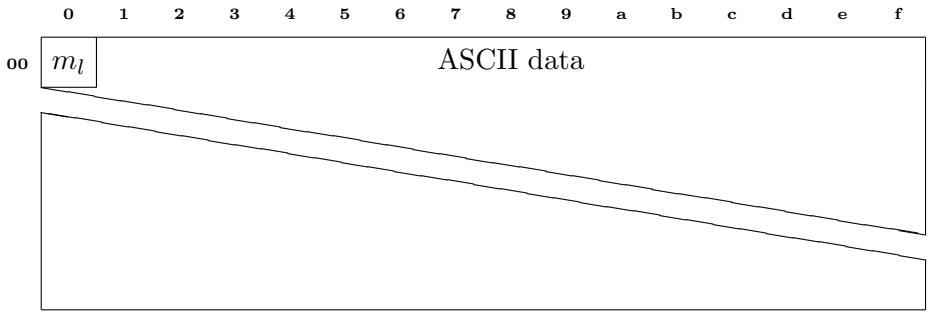

Figure 10: Short ASCII DeviceSQL String

3 Crate Digger

You can find a Java library that can parse the structures described in this research, and that can retrieve them from players' NFS servers, at: `https://github.com/deep-symmetry/crate-digger`

The project also contains Kaitai Struct specifications for the file structures, which were used to automatically generate Java classes to parse them, and which can be used to generate equivalent code for a variety of other programming languages.

There are also ONC RPC specification files which were similarly used to generate Java classes to communicate with the NFSv2 servers in the players, and which can likely be used to generate structures for other languages as well.

List of Figures

List of Tables

http://deepsymmetry.org