Rekordbox Export Structure Analysis

James Elliott
Deep Symmetry, LLC

November 14, 2018

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/pdb.ksy

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	00	00	00	00		*len_page*				*num_tables*				*next$_u$*		
10		*unknown*				*sequence*			00	00	00	00				

Table Pointers

Figure 1: File Header

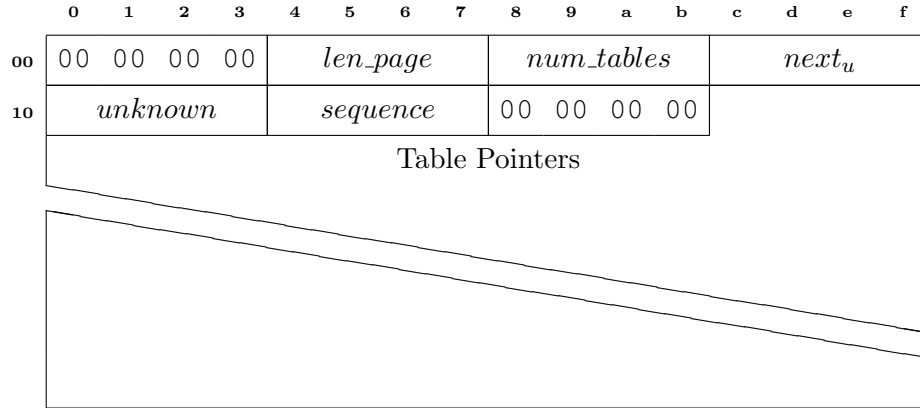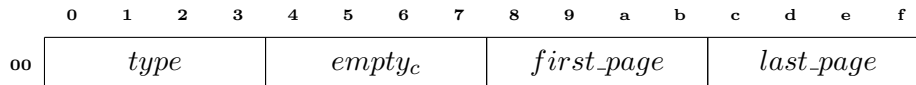|	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00		*type*				*empty$_c$*				*first_page*				*last_page*		

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.
3	albums	Albums, for reference by tracks and searching.
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.
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* × *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

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

To be continued...

2 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