HueDataStore File Format

This document describes the file format used by HueSpace to store all bulk binary data (VDS, property, and shape data). The format is chunk-based, and allows for sparse files. It is intended to be only a container format with minimal knowledge of the semantics of the stored data.

# High-level requirements

* The container file can only be accessed through a single HueDataStore object at a time.
* Atomic updates: a crash, full disk, etc. must never leave behind a corrupt file.
* It must be possible to group multiple updates into a transaction. Nested transactions are not supported.
* Fast direct reads/writes from/to any position within the file.
* Metadata stored for chunk must be extensible [e.g. VDSs need min/max values in the chunk].
* Individual chunks within the same file may be of different sizes.
* Support for file versioning and delta compression.

# File structure

The overall organization is a container that has a number of named files consisting of a number of chunks. For each file there is an index table which contains the file offset and size of each chunk. Additional metadata can be stored per chunk (e.g. hash values for de-duplication of data, min/max values of the data in a chunk). The index table is divided into pages so it is not necessary to read or write the entire index when updating the file.

## Header

A valid HueDataStore file starts with a 12-byte string which reads “HueDataStore”

struct DataStoreHeader

{

char magic[12]; /\* "HueDataStore" \*/

int32\_t version; /\* (major << 16) + minor \*/

int64\_t file\_table\_offset;

int32\_t file\_table\_num\_entries;

int32\_t file\_table\_name\_length;

};

## File table entry

Each file table entry describes the number of chunks in the file, the layout of the chunk index table and where to find the page table for the file. A UTF-8 encoded file name with the length described in the file header follows the information about the file. The file table name length must be a multiple of 8 to ensure alignment of the file table entries. The file names are assumed to be padded with NUL characters up to the file table name length, but need not be NUL terminated.

struct FileTableEntry

{

int64\_t head\_page\_directory\_offset;

int32\_t head\_num\_chunks;

int32\_t head\_revision\_number;

int32\_t index\_page\_num\_entries;

int32\_t file\_type; /\* file type four-CC (four ASCII-characters) \*/

int32\_t chunk\_metadata\_length;

int16\_t file\_metadata\_length;

char file\_name[/\* file\_table\_name\_length \*/];

};

All file table entries have the same number of bytes reserved for the file name.

## Index entry

Each chunk in the file has an index entry associated with it. This ensures that basic information about the chunk can be kept in memory. The chunk index makes it possible to do partial updates of files where each chunk may change size as a result of compression.

struct IndexEntry

{

int64\_t offset;

int32\_t size;

int32\_t reserved;

char metadata[/\* chunk\_metadata\_length \*/];

};

## Index page

In order to support atomic updates of files, the index entries are grouped into pages. This makes atomic updates of the index entries possible without rewriting the entire index. The number of index entries per page is found in the file table entry field index\_page\_num\_entries. Because the index pages do not have a header, it is possible to write a contiguous index and only decide how to break it up into pages when writing the page directory. If the number of chunks in the file is not a multiple of the number of entries per page, the last index page must be zero-padded.

## Page directory

At the page\_directory\_offset we find a small header and an array of int64\_t file offsets where each index page starts. The number of index pages is the number of chunks divided by the number of indexes per page (rounded up to the nearest integer). The last index page only has as many entries as there are remaining chunks in the file. If the offset for a page is zero, that means the page has not been written to the file yet, this ensures that is reasonably efficient to have a sparse file.

In addition to this, the page directory provides a basic versioning scheme by being able to link to previous versions of the page directory. The revision number must be decreasing for previous versions, and cannot be negative.

struct PageDirectory

{

int64\_t previous\_page\_directory\_offset;

int32\_t previous\_num\_chunks;

int32\_t previous\_revision\_number;

char metadata[/\* file\_metadata\_length \*/];

int64\_t index\_page\_offsets[/\* num\_index\_pages \*/];

};