RFC: Onion VFD

Songyu "Ray" Lu
John Mainzer
Jacob "Jake" Smith
Dana Robinson

There is a desire to introduce and track modifications to an HDF5 file while preserving or having access to the file as it existed prior to a particular set of modifications. To this end, this RFC proposes an Onion Virtual File Driver (VFD) as an effectively in-file revision management facility. Users will be able to open a particular revision of the file, read from and make modifications to the file, and write to file as a new revision. The name "Onion" derives from a mnemonic: the original file exists with data layered atop one another from an original file to the most recent revision.

1 Introduction

At present, the HDF5 library offers no support for version control or provenance management[1] – there is no mechanism to store unique batches of data in the same logical file space, nor to track when or by whom a particular modification was made. While this is not a significant deficit for most HDF5 applications, it is a major concern for experimental and observational data, where the original data must be preserved, and any changes tracked and attributed.

The primary reason for implementing this ourselves (as opposed to, for example, relying on an external version control program) is that doing so will allow for a significantly smaller footprint on the storage (e.g., disk). Major contenders of external programs such as SVN, Git, or Mercurial, would store each revision as a full-sized binary file of the file at the revision state – this is clearly unacceptable with large files (gigabyte-plus each) and many revisions.

An obvious and powerful way to address this use case is to implement an "infinite undo" facility along the lines of that offered by some text editors. In the context of HDF5, such a facility would allow reconstruction of earlier versions of an HDF5 file on an API call-by-API call basis, with each API call constituting a unique revision of the file. While this is doable, implementation would be a major exercise, and it would add significant complexity to the HDF5 library with the obvious implications for long term maintainability. Thus, this paper presents a sketch design for a simpler and cheaper option.

[1] Throughout the remainder of this document, these terms – version control and/or provenance management – will be collectively referred to as revision control.

If we make the simplifying assumption that it will be sufficient to track changes on a per file open-close cycle, the problem becomes much more tractable. More precisely, it will allow us to address the revision control at a very low level of the HDF5 library, making it transparent to the vast majority of the HDF5 code base. This in turn will greatly reduce the implementation effort required, simplify long-term maintenance, and allow us to implement revision control as an optional module in the form of a Virtual File Driver.

2 Basic Concepts

We will move forward with the above assumptions that a virtual file driver (VFD) is suitable for the use case. Here we discuss the notion of revisions and how they interact with the idea of a file modified over time.

VFDs present the underlying HDF5 file, regardless of its backing store (how the file is persistently stored), as an extensible range of bytes. The driver is responsible for mapping a byte or byte-range in the logical file to the appropriate byte or bytes in the backing store. With the Onion VFD, rather than having strictly one target for a logical file location "key", the target depends on which revision is being accessed – an extension of the VFD concept.

When writing new data to the backing store, the extant data is preserved and the new data is inserted in a known way. This operation at large is "copy-on-write", and may be implemented in one of two ways: The extant data is copied to another location and the replacement data takes its place, or the extant data remains in place and the replacement data is inserted elsewhere; in both cases, references are maintained to the location of both extant and replacement data. Because of the preference for modifying extant data as little as possible, this RFC assumes the latter behavior: leaving extant data untouched, and inserting the replacement data elsewhere.

There will be an original file revision – either an extant HDF5 file with legacy data or a newly-created empty HDF5 file. Subsequent to the original revision, revisions will be added. The file's logical state at a given revision will be the "appearance" of the file from that revision inward, with the content of more recent revisions superseding any data from earlier revisions "below". An alternative mental image to the onion might be a glass paperweight: the appearance from the outside (i.e., the opened revision) is a result of all its visible constituent layers.

Figure 1. File logical view with amendments color-coded by revision.

Each revision, once committed, cannot itself be modified – rather, a subsequent revision must supply the amended content. Because revisions will be hierarchical and immutable once committed, it

becomes possible to reconstruct the history of a file, much how an archaeologist or geologist "goes back in time" by descending through layers of rock or sediment. It also becomes possible to create divergent histories – branching, or forking – within the file by opening an arbitrary revision of the file and making modifications to it in a new revision. This branching complicates the issue of opening the "most recent" revision, as there are now potentially multiple end states, but this will likely be a minor technical issue, discussed in §Browsing Revisions and §History Metadata. The user is responsible for using this feature correctly[2].

There is presently no design provision for merging of divergent histories.

Figure 2. Color-coded logical view of file revisions with divergent histories (branching).

2.1 Revision Index

Integral to the concept of the Onion VFD is the ability to locate the "most recent" data, as seen from the chosen revision. To achieve this, we will create an index that maps locations in the logical file to the correct bytes in the backing store. This index must handle the case where a range of bytes read from the logical file is comprised of data from multiple revisions, possibly including the original data.

When the file is opened, an archival index – or "dead" index – will be constructed, which correctly associates any given byte in the logical file with the appropriate data in the backing store. If a given byte is present in the index, it will be read from the amended data in the appropriate revision; if absent from the index, it will be read from the original file data. Naturally, if the read extends outside of the logical file size, the read is erroneous and must fail.

When in write mode, a second, working index – or "live" index – will also be created, representing modifications to the file since file-open: the new revision to be written. This live index is actively updated as any modifications to the file are added or adjusted. This also adds an additional step when reading from the logical file, as any byte present in the live index is by definition the "most recent" and must take precedence if it intersects with the read range.

[2] With great power comes great responsibility.

2.2 Browsing Revisions – a new API: H5FDfctl()

There is one use case that cannot be satisfied efficiently with the existing API: browsing for which revision to open.

In a slow and crude fashion, the existing API could make iterative attempts at opening all the revisions in the file and eventually finding the one desired, but for any but the most trivial case the time to perform this selection is unconscionable. What the user would want is to 'peek' at the revision history of a file before committing to a particular open, and making a decision based on that lighter-weight peek.

Implementing this feature requires intimate knowledge of the Onion implementation, suggesting that it belongs as part of the driver itself; at best, an external implementation would result in a maintenance issue, where changes in the driver must be addressed separately. As such, we will introduce a new API that can potentially be utilized by other drivers to collect information about a driver, or files related to that driver, in a generalized fashion. With increasing cloud-based storage, asking a driver to query the health of (or connection with) the server or remote host (as in the S3 VFD) is an obvious use case outside of the Onion VFD.

3 Architectural Options

We must decide how to store amended data: as runs of contiguous amended data or fixed-size verbatim pages. We must decide where to store the revised information (and revision metadata): as part of the original file, or as one single or multiple external files. We must decide how to organize the amended data and the revision metadata within the storage location.

3.1 Amended Data Blobs: Pages over Runs

As minimizing stored data is a priority, an obvious first choice would be to record revised data as contiguous runs, merging amendments in working memory and writing only exactly those amended runs to the backing store. However, after some analysis, it becomes clear that this approach incurs excessive overhead at runtime. This analysis and conclusion is presented in an appendix.

The chosen solution for the first cut of the Onion VFD will instead be to store data as discrete 'pages' of data – blocks of a fixed number of bytes of which at least one byte has been amended from a prior revision. These pages will be stored verbatim, the entire contents of the page – amended or otherwise – present. While this approach will use excess space on the backing store (unmodified bytes in a page will be duplicated between revisions), we anticipate the runtime performance improvement (and simplicity) well worth the cost.

This implementation may be revisited in a later version of the VFD if there is sufficient interest.

3.2 Storage Location

There are three possible strategies for where to store the onion data:

1) Combine the entire revision history with the original file (i.e., appended), or
2) Store the entire revision history in a single separate file, or
3) Scatter the revision history across multiple separate files as:
    a. Complete per-revision

i. Each file represents a revision containing a copy of all amended data as seen from that revision.
ii. Wastes a potentially-vast amount of space on backing-store with repeated data from past revisions, but may improve read time from the backing store (worst case, read spans two revision blocks: original and revised data).

b. Amended per-revision
i. Each file represents a revision containing only the data amended in this revision.
ii. Complicates and slows construction of the archival index; slows reads spanning multiple revisions (must open each relevant revision file to access the amended data).

c. Individual amendment
i. Each file represents a discrete blob of amended data, e.g., a page. A revision that amends 100 pages would introduce 100 new (small) files to the backing store, plus perhaps additional files for revision metadata.
ii. Much slower to build the archival index and to read amended data.
iii. Never do this.

None of the separate multiple file approaches are particularly attractive for obvious reasons. In addition to the complications given for each approach, they introduce further fragility in managing the backing store – if a user wishes to rename or move an HDF5 file, an arbitrary number of associated files must also be handled.

The same-file appended approach clearly has much to recommend it, as it keeps only one file on the backing store, making it easier to move and maintain, and legacy VFDs will still be able to open the file (the history data is appended, leaving all original data pristine).

The separate single file approach is also attractive: it leaves the original file completely alone, and is in some ways easier to implement (as it "owns" the entire history file). As a minor complication, the driver must now track both files, and the user (or manager of the backing store) is responsible for two associated files on the backing store instead of only one file – the HDF5 file and the "onion file".

For the first cut of the Onion VFD, we will prioritize the separate-single storage case. This has the dual benefit of simplifying the implementation and keeping any original file data unaltered in any way.

3.3 History Metadata and Organization

The onion history is constructed out of three primary elements in sequence: an identifying header, one or more blocks of revision data, and a whole-history record.

The Onion Header contains information identifying the onion-ness of the file, the onion settings, and further information required to access the onion data.

Each revision must have, in addition to the amended data, its own metadata in a Revision Record consisting of, at a minimum: a representation of the index, the revision ID of the direct parent, and the size of the metadata. To satisfy the provenance requirements (i.e., provide time and author information

about the revision), we will also include time of creation and some form of user ID to the revision metadata. Additional elements may be added as appropriate.

Additionally, it also is readily apparent that an abstract Whole History record/summary is desirable, collecting the locations of all the revision records in one location. By reading a whole history, locating an arbitrary revision becomes trivial, and, if divergent histories are enabled, identifying multiple "latest" revisions becomes much easier. The contents of the whole history will include at a minimum: count of revisions, and map of revision ID to revision record location for each revision.

If we assume that we will want to move bytes around in the history file as little as possible, it becomes evident that each revision record should be written following the data amended during that revision, and that the whole history should be located at the end of the file. This allows for append-only writing of the history metadata.

The amended data within the revision will by default be unordered, written to the backing store in order of creation. Modified amended data (written more than once during a single open-close cycle) is updated in-place if it already exists[3].

3.3.1 Page/Block Alignment of History Metadata

It may be desirable to align history metadata with a given page boundary (multiple of a power of two), such as 4096 bytes on Linux systems. To achieve this, all history metadata will have unused padding bytes following its contents to fill out to the page boundary. At the expense of unused space on the backing store (on the order of one-half page per revision, plus a partial page with the onion history header, plus – in the case of a single HDF5+onion file – a partial page between HDF5 and onion data), this alignment can improve I/O performance on some systems, especially when involving direct I/O[4].

3.4 Page Buffering

As an advanced design consideration, including some form of page buffering may be desirable to reduce latency between the user and the backing store.

Commented [JS1]: TODO: user story to help clarify

This will likely be implemented as a separate "module" within the VFL, to be utilized freely by any VFD which may benefit (such as an optimized S3 VFD).

This will not be addressed in the initial version of the Onion VFD.

4 Implementation Details

We will proceed with the following assumptions:

[3] Update-in-place can lead to data fragmentation if multiple adjustments intersect, resulting in slower read performance.

[4] "Direct I/O is a feature of the file system whereby file reads and writes go directly from the applications to the storage device, bypassing the operating system read and write caches."
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/global_file_system/s1-manage-direct-io
http://people.redhat.com/msnitzer/docs/io-limits.txt

1. The VFD approach is suitable, accepting its limitation of open-close cycle granularity between revisions.
2. Verbatim per-page storage is acceptable for the first pass. The simplicity of implementation has much to recommend this approach, and the "wasted space" of repeated page data between revisions can be analyzed to inform the need to optimize.
    a. That the revision index (live index, for write mode) will be implemented as a hash table.
3. Revision history stored as a separate single, external file, e.g., "onion file", is the priority for the first pass.
    a. Implementation will provide for alternative architectural options as discussed in §Storage Location, but those options may not be supported initially.

4.1 Revision Index

The revision index (working, or live index) is responsible for managing the association between the logical location of a page (offset of start-of-page) in file and the "physical" location of the page's data in the backing store. If implementing amended data as runs, the index must also manage the size of the run; the paged approach uses the page size as a fixed implicit run size.

There are two competing options for the revision index: binary search tree and hash table, the deciding factor largely falling into the speed-versus-size decision matrix. We will elect for speed and ease of implementation for the first version of the Onion VFD.

Considerations for a binary search tree implementation of the index is presented as an appendix

4.1.1 Hash Table Implementation of Revision Index

Hash tables are tricky to get right, but conceptually straightforward and potentially very powerful. They make sense as the live index only if using pages, where the page ID of the index entry is used as the hashing key.

The current proposal for the hashing function involves chaining – collisions are stored in a data structure – which will increase the implementation complexity somewhat for a potentially very fast hashing function relying on bit-shifts:

1) The size of the hash table is always a power of 2.
2) The hashing function is H_i = `i & (table_size - 1)`, where `i` is the page number.
3) Collisions are stored in a data structure – e.g., linked list or binary search tree – of elements.
    a. A naïve binary search tree will degenerate into a linked list in the worst case. True fanatics may push for a self-balancing tree implementation. Deletion cannot occur, simplifying BST implementation. However, a linked list implementation remains simpler.
4) When adding a new entry would fill the table to half or more of its capacity, the table is reallocated to the next larger power of two and its contents re-hashed, and then the entry is inserted.

4.1.2 Revision Index Entry

While more concise implementations can be envisioned, creating a standard structure for the index entries that can adapt to multiple index construction has much to recommend it; further, it contains all data about an entry in a single "object" (contrast with a hash table that maintains several independent arrays).

```
/* ----------------------------------------------------------------------------
 * Structure      H5FD_onion_revindex_entry
 *
 * Purpose:       Stores information about an entry in the revision index.
 *                Pointers to this structure are the population of the
 *                "live index" used for amended data introduced in write mode.
 *
 * version:       Version number of this structure, informs component membership.
 *                Used to future-proof modifications to this structure.
 *
 * logic_page:    The page number of the data within the logical file.
 *                "Physical offset" of the entry start in the file is the product
 *                of page number times page size.
 *                This value is used as the hashing key.
 *
 * physic_addr:   The offset of the entry start in the backing store.
 *
 * length:        Length of the page data.
 *                A sanity-checking value, MUST equal the page size set for the
 *                onion history.
 *
 * checksum:      32-bit checksum of the amended data in store.
 *
 * file_id:       File identifier for the file containing the relevant entry.
 *                Exists primarily as a future-proofing consideration in the event
 *                of some form of separate multiple file storage of amended data.
 *
 *                For the initial implementation, this is utilized as a sanity-
 *                check: must be zero (0) if onion history data is stored as
 *                part of the HDF5 file (appended to original data), or
 *                one (1) if stored as a separate single file.
 *                Any other value will be considered invalid.
 *
 * chaining structure info:
 *                Pointers to other revision index entry structures. Details TBD.
 * ----------------------------------------------------------------------------
 */
struct H5FD_onion_revindex_entry {
    uint8_t  version;
    uint64_t logic_page;
    hoff_t   physic_addr;
    uint64_t length;
    uint32_t checksum;
    uint64_t file_id;
    TODO: chaining structure info;
};
```

Commented [JS2]: Presumes using pages. Would need to be renamed if dealing with raw logical addresses

Commented [JS3]: TBD
Could be singly-linked list, e.g.
struct H5FD_onion_revindex_entry *next;

Or binary tree, e.g.
struct H5FD_onion_revindex_entry *smolr;
struct H5FD_onion_revindex_entry *biggr;

4.2 Archival "Dead" Index

Before committing the live index to the revision history, the revision "live" index is converted to a sorted array, ordered by increasing logical address, and merged with the previous archival index. This sorted, merged array is written to the file as part of the revision record and will be read as the archival index for the revision[5].

On file open, the revision's index is ingested "as-is" as a sorted array. Locating an element then becomes a straightforward binary search operation.

4.3 Onion File Access Property List

We will use a structure to put all the driver configuration information into a single element to pass into and out of the FAPL set- and get-driver routines `H5Pset_fapl_onion()` and `H5Pget_fapl_onion()`.

```
/* ------------------------------------------------------------------------
 * Structure    H5FD_onion_fapl_info_t
 *
 * Purpose:     Encapsulate info for the Onion driver FAPL entry.
 *
 * version:     Future-proofing identifier. Informs struct membership.
 *              Must equal H5FD_ONION_FAPL_VERSION_CURR to be considered valid.
 *
 * backing_fapl_id:
 *              Backing or 'child' FAPL ID to handle I/O with the
 *              underlying backing store. If the onion data is stored as a
 *              separate file, it must use the same backing driver as the
 *              original file.
 *
 * page_size:   Size of the amended data pages. If opening an existing file,
 *              must equal the existing page size or zero. If creating a new
 *              file or an initial revision of an existing file, must be a
 *              power of 2.
 *
 * store_target:
 *              Enumerated/defined value identifying where the history data is
 *              stored, either in the same file (appended to HDF5 data) or a
 *              separate file. Other options may be added in later versions.
 *
 *              + H5FD_ONION_FAPL_STORE_MODE_SEPARATE_SINGLE (1)
 *                      Onion history is stored in a single, separate "onion
 *                      file". Shares filename and path as hdf5 file (if any),
 *                      with only a different filename extension.
 *
 * revision_id: Which revision to open. Must be 0 (the original file) or the
```

Commented [JS4]: typedef to drop 'struct' and add '_t' suffix, in keeping with convention with other HDF5 FAPLs.

Commented [DR5]: We should consider renaming this to simply be "revision" as "ID" is already overloaded in the library.

[5] This has the effect of making each revision index larger than is strictly necessary, but greatly accelerates time to open the file. If there is demand, we may revisit this decision and prioritize on-store space requirement.

```
 *              revision number of an existing revision.
 *              Revision ID -1 is reserved to open the most recently-created
 *              revision in history.
 *
 * force_write_open:
 *              Flag to ignore the write-lock flag in the onion data
 *              and attempt to open the file write-only anyway.
 *              This may be relevant if, for example, the library crashed
 *              while the file was open in write mode and the write-lock
 *              flag was not cleared.
 *              Must equal H5FD_ONION_FAPL_FLAG_FORCE_OPEN to enable.
 *
 * creation_flags:
 *              Flag used only when instantiating an Onion file.
 *              If the relevant bit is set to a nonzero value, its feature
 *              will be enabled.
 *
 *              + H5FD_ONION_FAPL_CREATE_FLAG_ENABLE_DIVERGENT_HISTORY
 *                (1, bit 1)
 *                      User will be allowed to open arbitrary revisions
 *                      in write mode.
 *                      If disabled (0), only the most recent revision may be
 *                      opened for amendment.
 *
 *              + H5FD_ONION_FAPL_CREATE_FLAG_ENABLE_PAGE_ALIGNMENT (2, bit 2)
 *                      Onion history metadata will align to page_size.
 *                      Partial pages of unused space will occur in the file,
 *                      but may improve read performance from the backing store
 *                      on some systems.
 *                      If disabled (0), padding will not be inserted to align
 *                      to page boundaries.
 *
 *              + <Remaining bits reserved>
 *
 * comment:     User-supplied NULL-terminated comment for a revision to be
 *              written.
 *              Cannot be longer than H5FD_ONION_FAPL_COMMENT_MAX_LEN.
 *              Ignored if part of a FAPL used to open in read mode.
 *
 *              The comment for a revision may be modified prior to committing
 *              to the revision (closing the file and writing the record)
 *              with a call to H5FDfctl().
 *              This H5FDfctl overwrite may be used to exceed constraints of
 *              maximum string length and the NULL-terminator requirement.
 *
 * ------------------------------------------------------------------------
 */



typedef struct H5FD_onion_fapl_info_t {
    uint8_t                         version;
    hid_t                           backing_fapl_id;
```

```
    uint32_t                          page_size;
    H5FD_onion_target_file_constant_t store_target;
    uint64_t                          revision_id;
    uint8_t                           force_write_open;
    uint8_t                           creation_flags;
    char                              comment[H5FD_ONION_FAPL_COMMENT_MAX_LEN + 1];
} H5FD_onion_fapl_info_t;
```

4.4 H5FDfctl() Details

A new API function, `H5FDfctl()`, will be added to the VFL. This will allow any driver to implement features as appropriate that may not otherwise be possible with the current API. The function's name is derived from conventional operating system calls, "file control". A new function component will be added to the driver class, each driver implementing (or eliding) the feature as appropriate.

```
/* ----------------------------------------------------------------------------
 * Function:    H5FDfctl
 *
 * Purpose:     Provide a general-purpose interface for driver-defined behavior
 *              in interacting with a virtual file handle.
 * ----------------------------------------------------------------------------
 */
herr_t H5FDfctl (H5FD_t *handle, uint32_t op_code, const void *input, void *result);
```

Commented [JS6]: Documentation comment TODO

With the Onion VFD, three procedures are supported:

1. Retrieve the information about the currently-opened revision – inspect its record.
2. Browse the onion revision history without committing to opening any revision in advance.
3. Modify revision comment before writing of file (write mode only).

Procedures one and two are a two-step process, with the first step determining the size of the buffer required to store the result and the second step collecting the actual result in the user-allocated buffer.

4.4.1 Get Onion Info About Currently-Opened Revision

From the file already opened at a given revision, inspect information about the revision.

```
H5FD_t *handle = NULL;
struct H5FD_onion_fctl_info_revision  *record_ptr = NULL;
uint64_t        size_out = 0;
void   *buf = NULL;

/* Get the virtual file handle
*/
H5Fget_vfd_handle(file_id, onion_enabled_fapl_id, &handle);

/* Determine required buffer size
 * Pass in File ID with pointers to
 *   IN   N/A
 *   OUT  uint64_t
 */
H5FDfctl (handle, H5FD_ONION_FCTL_OP_REVISION_INFO_SIZE, NULL, &size_out);
HDassert(size_out > sizeof(H5FD_onion_fctl_info_revision));
    /* minimum two bytes extra for empty username and comment strings */
```

```
/* Allocate buffer and get info
 * Pass in File ID with pointers to
 *   IN   uint64_t  : size of the user-allocated buffer.
 *   OUT  void/char : user-allocated buffer for record struct and varlen data.
 */
buf = HDmalloc(size_out);
H5FDfctl (handle, H5FD_ONION_FCTL_OP_REVISION_INFO_GET, &size_out, buf);
record_ptr = (struct H5FD_onion_fctl_info_revision *)buf;
HDassert(record_ptr->magic == RECORD_MAGIC);

/* inspect record components and release resources */
. . .
record_ptr = NULL;
HDfree(buf);
buf = NULL;
handle = NULL;
```

Commented [JS7]: Symbolic name TODO

4.4.2 Browse Onion History

Without committing to opening a file, examine a file's onion history to choose which revision to open.

```
H5FD_t *handle = NULL;
struct H5FD_onion_fctl_info_history_filter   hist_filter = {. . .};
struct H5FD_onion_fctl_info_history_size     size_out;
struct H5FD_onion_fctl_info_history_size_filter      size_filter = {. . .};
struct H5FD_onion_fctl_info_revision *records = NULL;
void   *buf = NULL;

handle = H5FDopen("myfile.h5", H5F_ACC_RDONLY, onion_enabled_fapl_id, MAXADDR);

/* Determine required buffer size
 * Pass in FAPL ID with pointers to
 *   IN   H5FD_onion_fctl_info_history_size_filter
 *   OUT  H5FD_onion_fctl_info_history_size
 */
size_out.magic = TODO;
size_out.version = TODO;
H5FDfctl(handle, H5FD_ONION_FCTL_OP_HISTORY_SIZE, &size_filter, &size_out);

if (size_out.count > 0) {
    /* Allocate buffer and get info
     * Pass in FAPL ID with pointers to
     *   IN   H5FD_onion_fctl_info_history_filter
     *   OUT  char/void pointer to user-allocated buffer
     */
    buf = HDmalloc(size_out.size);
    hist_filter.expected_count = size_out.count;
    H5FDfctl(file_id, H5FD_ONION_FCTL_OP_HISTORY_GET, &hist_filter, buf);
    HDassert(hist.count == size_out.count);

    records = (struct H5FD_onion_fctl_info_revision *)buf;
    HDassert(records[0] && records[0].magic == H5FD_ONION_FCTL_REVISION_INFO_MAGIC);
```

Commented [JS8]: Symbolic names TODO

```
    . . . /* inspect record components */

    /* Done -- release resources */
    records = NULL;
    HDfree(buf);
    buf = NULL;
}
H5FDclose(handle);
handle = NULL;
```

4.4.3 Set or Update Revision Comment

With a file opened in write mode, the user may wish to update or create a comment for the revision, which may have been supplied in the FAPL on file-open or by a previous comment-set with `H5FDfctl()`. Rather than abusing the FAPL, we will make use of this new API to implement this feature.

```
H5FD_t *handle = NULL;
struct H5FD_onion_fctl_info_comment comment = {. . .};
comment.comment = "ABNORMAL BRAIN DO NOT USE";
comment.len = strlen(comment.comment);
H5Fget_vfd_handle(write_mode_opened_file_id, onion_enabled_fapl_id, &handle);
H5FDfctl(handle, H5FD_ONION_FCTL_OP_COMMENT_SET, &comment, NULL);
handle = NULL;
```

4.4.4 Onion Get-Info Structures

Structures are presented in alphabetical order.

```
/* ----------------------------------------------------------------------------
 * Structure:   H5FD_onion_fctl_info_comment
 *
 * Purpose:     Pass a new revision comment into the working file.
 *
 * magic:       4-byte, semi-unique number identifying this structure.
 *              Must equal SYMBOLC_NAME_TODO to be considered valid.
 *
 * version:     4-byte number; future-proofing guard, informs struct membership.
 *
 * len:         Length of the comment string.
 *
 * comment:     String to use as the new comment.
/* ----------------------------------------------------------------------------
struct H5FD_onion_fctl_info_comment {
    uint32_t magic;
    uint32_t version;
    uint64_t len;
    char    *comment;
};

/* ----------------------------------------------------------------------------
 * Structure    H5FD_onion_fctl_info_history_filter
 *
```

```
 * Purpose:     Supply selection information and criteria to H5FDfctl() for
 *              the Onion VFD, when accessing an unopened file.
 *
 *              Not present in the initial version, but it stands to reason
 *              that the user will expect this 'filter' to do much of the
 *              heavy lifting for them; pass in some selection criteria and
 *              receive only relevant results.
 *              This may be added in a later version.
 *              If included, a history-size filter will also be required to
 *              repeat the in-call filtering.
 *
 * magic:       4-byte semi-unique "magic" number identifying structure.
 *              Must equal SYMBOLIC NAME TODO to be considered valid.
 *
 * version:     4-byte number; future-proofing guard, informs struct membership.
 *
 * expected_count:
 *              This should be set to the count returned by the history size
 *              result. If the count found when browsing this time,
 *              (e.g., the user modified the selection or the
 *              file has been modified), the operation should fail in a
 *              controlled fashion.
 * ----------------------------------------------------------------------------
 */
struct H5FD_onion_fctl_info_history_filter {
    uint32_t magic;
    uint32_t version;
    uint64_t expected_count;
};

/* ----------------------------------------------------------------------------
 * Structure:   H5FD_onion_fctl_info_history_size
 *
 * Purpose:     Store the result of history-size-filter operation.
 *              The implementation will walk through the history
 *              once, and determine which revisions (if any) match the
 *              selection criteria; the count of those revisions and the space
 *              required to copy their contents is recorded in this struct.
 *
 *              The resulting size will be the size of a record struct times
 *              the count of records PLUS the total space required for the
 *              variable-length data in those records.
 *
 *              The user will be expected to allocate a buffer not smaller than
 *              this size and pass it as part of the info_out pointer
 *              to obtain the history view.
 *
 * magic:       4-byte semi-unique "magic" number identifying structure.
 *              Must equal SYMBOLIC NAME TODO to be considered valid.
 *
 * version:     4-byte number; future-proofing guard, informs struct membership.
 *
 * n_bytes: Number of bytes to contain all the relevant record data.
```

Commented [JS9]: Discussed in an appendix

Commented [JS10]: TODO: details

```
 *
 * n_revisions: Number of revisions in history.
 * ------------------------------------------------------------------------
 */
struct H5FD_onion_fctl_info_history_size {
    uint32_t magic;
    uint32_t version;
    uint64_t n_bytes;
    uint64_t n_revisions;
};

/* ------------------------------------------------------------------------
 * Structure:   H5FD_onion_fctl_info_revision
 *
 * Purpose:     A structure representation of the revision record as found
 *              on the backing store.
 *
 *              Stores the result from the onion info record filter input.
 *              Will usually be a region in a user buffer passed into
 *              H5FDfctl(), with the residual space storing the
 *              variable-length data from the revision record.
 *
 * magic:       4-byte, semi-unique number identifying this structure.
 *              Must equal H5FD_ONION_FCTL_REVISION_INFO_MAGIC to be considered
 *              valid.
 *
 * version:     4-byte number; future-proofing guard, informs struct membership.
 *
 * revision_id: Revision ID of this revision.
 *
 * parent_revision_id:
 *              Revision ID of the immediate parent of this revision.
 *              The 'original' revision (revision_id == 0) is a unique case
 *              in that its parent revision ID is also 0 (itself). Users
 *              must take care to handle this self-reference.
 *
 * time_of_creation:
 *              ISO-8601-formatted string (NOT NULL-terminated!) giving the
 *              time when the revision was written to the backing store.
 *
 *              Granularity is seconds, meaning that more than revision may
 *              have identical times of creation.
 *
 *              Format: four-digit year, two-digit month, two-digit date,
 *              the character capital-T, two-digit hour, two-digit minute,
 *              two-digit second, character capital-Z.
 *              e.g., '20200418T154712Z' == 18 April 2020, 3:47:12pm GMT
 *
 * user_id:     4-byte user ID of revision creator.
 *
 * username_size:
 *              Bytes required to contain the revision creator username.
 *
```

Commented [JS11]: If implemented, the number of revision matching the filtering criteria.

```
 * comment_size:
 *              Bytes required to contain the revision comment.
 *
 * username:    NULL-terminated string of user name.
 *
 * comment:     NULL-terminated string of user comment for this revision.
 *              Can be empty.
 * ---------------------------------------------------------------------------
 */
struct H5FD_onion_fctl_info_revision {
    uint32_t magic;
    uint32_t version;
    uint64_t revision_id;
    uint64_t parent_revision_id;
    char     time_of_creation[16]; /* caution: not null-terminated */
    uint32_t user_id;
    uint32_t username_size;
    uint32_t comment_size;
    char *username;
    char *comment;
};
```

4.5 Additional Considerations

4.5.1 Concurrent Writing Processes

To prevent multiple concurrent writes (or writing processes), we will include a flag in the onion header which "locks" the file once it is opened write-only. If this is set, attempts to open the file for writing should fail. Provisions will be made for error-recovery, such that a crash with the write-lock set will not corrupt or permanently lock the file.

4.5.2 File-Specific Divergent History Setting

Divergent history (branching) will be enabled or disabled "permanently" upon onion creation. A flag will be set in the header indicating whether or not forking histories are allowed. If allowed, any revision can be opened for writing, and the "latest" shortcut (revision ID -1) is not guaranteed to be the desired tail (rather, it is the most recently-committed revision); if disallowed, only the latest revision may be opened for writing, but the latest ID shortcut (-1) always accesses the only tail.

4.5.3 Append-Only Whole-History Issue

Implemented in a naïve fashion, the whole-history will be deleted when a writing process begins to amend data for a new revision. This presents the problem of possible corruption (unable to access the file at all) in the event of a program or system crash.

An obvious solution is to skip over the whole-history, rather than overwriting it. This leaves unused whole-histories scattered throughout the file, one following each revision record, which will bloat the file somewhat. However, this has the direct benefit of the file always being in a readable state – the address of the Whole History in the onion header is updated only when a new revision (pages, revision record, and new whole-history) is committed and written to store.

4.5.4 Modification of Original File

It is conceivable that a user may, deliberately or otherwise, open the "original" HDF5 contents of an oniony file in write mode with a non-onion VFD.

In the case of separate single onion storage, there is nothing that can be done – the user will overwrite the original data, effectively destroying the goals of onionizing in the first place.

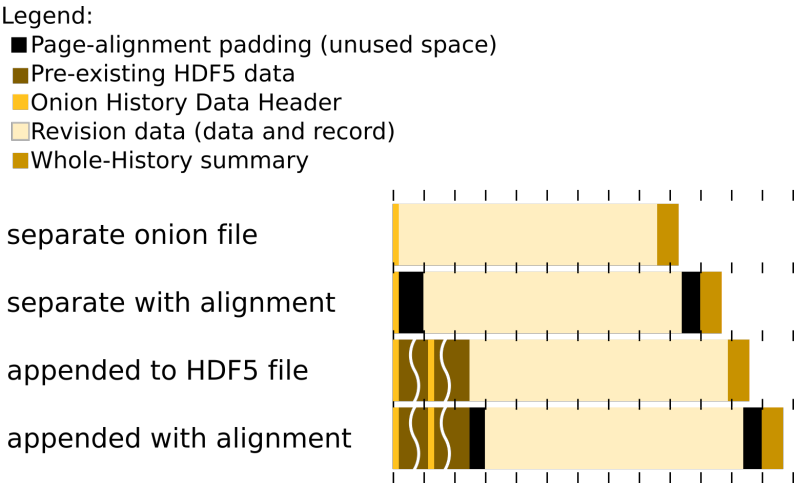
In the second case, where the onion history is appended to the HDF5 file, the Onion Header will be part of the superblock extension message – the library will be able to read this and deduce that the file must not be opened in write mode without an Onion VFD.

4.5.5 Single-File Storage and Userblock

With the design proposal below, the Onion Header will occupy the Userblock, which prefaces the HDF5 file proper[6]. This is a limitation of the onion implementation: If a file is oniony with single-file storage, the Userblock cannot be used for another purpose; if the original file already exists with a Userblock, it cannot be made oniony with single-file storage.

5 File Format Specification

This section describes the file format of the history data. The history data is consistent between storage as one file (appended to the HDF5 file) or in a separate single file. All multi-byte data is stored in little-endian word order, following the convention of HDF5 file metadata. The contents of the Onion data may be page-aligned on the backing, where unused padding bytes shifts data to align with a page boundary – relevant to accelerate I/O performance on some systems.



[6] TODO: Pointer to documentation on Userblock.

Figure 4. Visual overview of the Onion file format.

5.1 Onion History Data Header Format

The Onion Header must be located at offset zero (0) in the versioning file. If onion data is stored as a separate single file, this is trivial. If instead stored in the HDF5 file, it also exists at offset zero (occupying the user block) *and* is duplicated in the superblock extension message.

Version 0 (zero) of this header has the following format:

byte	**byte**	**byte**	**byte**
Signature			
Version number	Flag bitfield		
Page size			
Origin file size (8 bytes) …			
Address of Whole-History Record (8 bytes) …			
Size of Whole-History Record (8 bytes) …			
Checksum			

The fields of the Version 0 Onion history data header are described in the following table.

Field Name:	**Description:**
Signature	Magic number indicating that this is an Onion VFD revision data header. Must be set to 'OHDH' (Onion History Data Header). (4 bytes)
Version number	The version number of the format for the versioning file. (1 byte)
Flag bitfield	Reserved space for various binary flags. (3 bytes) • Write-Lock Flag (1, bit 1, bit 55 in file) Indicates whether this file has been opened in write mode. If not zero (0), the file has been opened in write mode by the Onion VFD and set this flag. This flag should be reset to zero when closed from write mode. If not zero, it is very likely unsafe to open the file in write mode due to possible interlaced writing of page data. • Branching Support Flag (2, bit 2, bit 54 in file) Set on onion instantiation. If zero (0), only the most recently-created revision in history may be opened in write mode, forcing a single chain if history from origin to most recent revision. If not zero, any revision may be opened in write mode.

	• Align-to-Page Flag (4, bit 3, bit 53 in file) Set on onion instantiation. If not zero, onion data in the backing file will be aligned to intervals of page size, introducing unused spaces within the file as necessary. • Remaining bits reserved.
Page size	Size of pages containing amended data. Must not be modified after onion instantiation. (4 bytes)
Origin file size	Number of bytes in the 'origin' canonical (HDF5) file. Required to correctly write pages or partial pages extending the logical file. (8 bytes)
Address of Whole-History Record	The location of the whole-history record; offset from start of file. (8 bytes)
Size of Whole-History Record	The size of the whole-history record in bytes. Whole-History address + size must point to the last byte in the file. (8 bytes)
Checksum	The checksum of this header. (4 bytes)

5.2 Whole-History Record Format

The whole-history record is located in the end of the onion history file (which may be the same as the original HDF5 file). It contains the addresses of the revision records in the history. Version 0 (zero) has the following format:

byte	**byte**	**Byte**	**byte**
Signature			
Version number	*unused space reserved (3 bytes)*		
Number of revisions (8 bytes) …			
List of Record Pointers (variable size) ……			
Checksum			

The fields of the Version 0 whole-history summary record are described in the following table:

Field Name:	**Description:**
Signature	'Magic number' identifying this structure. Must equal 'OWHR' (Onion Whole-History Record). (4 bytes)
Version number	Version number of the format for this whole-history. (1 byte)
Number of revisions	Count of all the revisions present in the history. (8 bytes)
List of record pointers	List of pointers to revision records on store. (pointer size * count bytes).

Checksum	The checksum of this header. (4 bytes)

5.2.1 Record Pointer Format

The [revision] record pointer has the following format.

byte	byte	Byte	byte
Physical address (8 bytes) …			
Record Size (8 bytes) …			
Checksum			

The fields of the [revision] record pointer are described as follows.

Field Name:	Description:
Physical address	Location of the revision record in the backing store Offset is given from start of the file – not necessarily the start of the history data. (8 bytes)
Record size	Size in bytes of the record in the backing store. (8 bytes)
Checksum	Checksum of the address and size. (4 bytes)

5.3 Revision Record Format

The revision record for a given revision is located after the data pages of the revision. It contains the following information.

byte	byte	Byte	byte
Signature			
Version number	*unused space reserved (3 bytes)*		
Revision ID (8 bytes) …			
Parent Revision ID (8 bytes) …			
Time of Creation (16 bytes) ……			
Logical file size (8 bytes) …			
Page size			
User ID			

Number of index entries (8 bytes) …
User name size
Comment size
List of index entries (variable size) ……
User name (variable size) ……
Comment (variable size) ……
Checksum

The fields of the revision record are described in the following table:

Field Name:	**Description:**
Signature	'Magic number' identifying this structure. Must equal 'ORRS' (Onion Revision Record Signature)
Version number	Version number of the revision-record format. (1 byte)
Revision ID	Unique ID of the revision. (8 bytes)
Parent Revision ID	Unique ID of the revision of which this is an immediate descendant. (8 bytes)
Time of Creation	A **not** NULL-terminated ASCII char array of format ISO-8601 for time and date (from UTC, Coordinated Universal Time) when this record was written. Format: four-digit year, two-digit month, two-digit date, the character capital-t (T), two-digit hour, two-digit minute, two-digit second, character capital-z (Z). e.g., 20200418T154712Z for 10:47:12am on April 18, 2020 CDT (15:47, or 3:47:12pm UTC; CDT = UTC-5:00) (16 bytes)
Logical file size	Size of the logical file – equivalent to EOF/EOA. (8 bytes)
Page size	Size in bytes of a page of logical file data. Must equal the value in the Onion History Data Header; duplicated here for sanity-checking and internal use. (4 bytes)
User ID	The 32-bit value of the UID that created this version. (4 bytes)
Number of index entries	The count of index entries for this revision. The index is a sorted list based on the logical address (page number / offset start) in the HDF5 file. (8 bytes)
User name size	The length of the variable-sized user name. (4 bytes)
Comment size	The length of the variable-size comment. (4 bytes)

List of index entries	List of entries, each giving the information required to identify and locate amended data. (entry_size * count bytes)
User name	The user name of variable size. (NULL-terminated string)
Comment	The variable-size comment. (NULL-terminated string)
Checksum	The checksum of this record. (4 bytes)

5.3.1 Index Entry Format

The index entry has the following format.

byte	byte	Byte	byte
Logical address (8 bytes) ...			
Physical address (8 bytes) ...			
TODO: Page Data Checksum			
Checksum			

The fields of the index entry are described as follows.

Field Name:	Description:
Logical address	Location of the amended data within the logical HDF5 file. (8 bytes)
Physical address	Location of the amended data in the backing store. Offset is given from the start of the file – not necessarily the start of the history data. (8 bytes)
Page Data Checksum	Checksum of the page data in the store (raw bytes). (4 bytes)
Checksum	Checksum of the entry's data in store (logical and physical address). (4 bytes)

6 API Functions

Name: H5Pset_fapl_onion

Signature:

herr_t `H5Pset_fapl_onion` (*hid_t* `fapl_id`, *const H5FD_onion_fapl_info_t* *`onion_fa_info`)

Commented [JS12]: TOOD? `struct H5FD_onion_fapl_info

Purpose:

Set the provided FAPL to use the Onion VFD with the given configuration.

Description:

`H5Pset_fapl_onion` sets the file access property list `fapl_id` to use the Onion virtual file driver with the given configuration. The info structure may be modified or deleted after this call, as its contents are copied into the FAPL.

The details of the input info pointer `onion_fa_info` is discussed in §Onion File Access Property List with the structure documentation.

Parameters:

hid_t `fapl_id`	IN: File access property list identifier.
const H5FD_onion_fapl_info_t *`onion_fa_info`	IN: Configuration info structure.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5Pget_fapl_onion
Signature:

herr_t `H5Pget_fapl_onion` (*hid_t* `fapl_id`, *H5FD_onion_fapl_info_t* *`onion_fa_info`)

Purpose:

Retrieve the information of the version control driver.

Description:

`H5Pget_fapl_onion` retrieves the FAPL information pertaining to the Onion virtual file driver configuration. If successful, the information as found in the FAPL is copied into `onion_fa_info`. The details of the input info pointer `onion_fa_info` is discussed in §Onion File Access Property List with the structure documentation.

This function is for use only when accessing an HDF5 file written as a set of files with the Onion VFD.

Parameters:

hid_t `fapl_id`	IN: File access property list identifier.
H5FD_onion_fapl_info_t *`onion_fa_info`	OUT: Configuration info structure.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Name: H5FDfctl
Signature:

herr_t `H5FDfctl` (*const H5FD_t* *`handle`, *uint32_t* `op_code`, *const void* *`input`, *void* *`result`)

Purpose:

Retrieve the information of a VFD file.

Description:

`H5FDfctl()` provides a general-purpose interface for driver-defined behavior in interacting with a virtual file handle.

The details of the `op_code`, `filter` and `result` parameters are necessarily driver-dependent.

`result` is passed in by the user and used to collect values returned by the call. The user is responsible for managing this pointer and any memory region(s) it references.

Parameters:

H5FD_t *`handle`	IN: Virtual File Handle on which to operate.
uint32_t `op_code`	IN: Numeric constant for which operation to perform.
const void *`filter`	IN: Information informing the driver's fctl operation.
void *`result`	OUT: User-allocated buffer for the results of the call.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

7 Using the Onion VFD

7.1 Instantiate Onion History

7.1.1 Instantiate with No Existing HDF5 Origin

If storage mode/target is a separate, single file (onion file), a blank file with .h5 extension is created, and all data is written to the onion file (sharing the file name, with .onion extension).

If storage mode/target is the HDF5 file (single file), a file with .h5 extension is created. Its contents are exactly those of an onion file.

TODO: should the "original data" be written into the .h5 file, e.g. as only a superblock and userblock extension message?

7.1.2 Instantiate from Existing HDF5 Origin

If storage mode/target is separate, single file (onion file), the original data is untouched, and a new file with the same name but .onion extension is created which will contain the onion history. An onion header and "empty" whole-history record is written to the onion file, and the empty whole-history record is copied to the temporary location for error-recovery (.onion.recovery extension?), as with all Onion VFD write-mode opens; this temporary recovery whole-history record file is deleted upon file close.

If the storage mode/target is the HDF5 file (single file), the userblock extension message is first inspected – if it is already set, the file cannot be onionized in this manner.
If the userblock extension message can be set, it is populated with the onion header; the onion header is duplicated at the start of the file (moving all existing file contents as necessary); the (empty) whole-history record is appended to the file and copied in the temporary recovery location.

Commented [JS13]: This seems really dangerous and slow. Maybe have the header only in the userblock extension message?

7.2 Open File for Reading

1. Verify that it is an onionized file.
    a. Onion file may not exist.
    b. Userblock extension message may not be set.
    c. Prior write may have failed, leaving whole-history file.
2. Open files as appropriate.
    a. Chosen revision may not exist (want 15, only 0..12 exist, e.g.)
3. Perform reads.
4. Close files.

7.3 Open File for Writing

1. Verify that it is an onionized file.
    a. Onion file may not exist.
    b. Userblock extension message may not be set.
    c. Prior write may have failed, leaving whole-history recovery file.
2. Open "origin" .h5 file and .onion file as appropriate.
    a. Chosen revision may not exist (want 15, only 0..12 exist, e.g.)
3. Set write-lock flag in onion header.
4. Copy whole-history summary to temporary location on backing store.
5. Commence modifications to file.
6. Write revision record to file.
7. Copy and update whole-history from recovery location to end of file.
8. Update whole-history address and unset write-lock flag in onion header.
9. Close files.

8 Recommendation

The Onion VFD approach to revision control has the advantages of relative simplicity and modularity, although its utility rests on the presumption that file open-close cycle resolution is adequate for the vast majority of applications.

For the prototype implementation, storing revision history in a separate single file is prioritized – it has the capacity of accommodating multiple versions as well as protecting the original data, while avoiding having to create, open, or generally manage too many files on the filesystem.

Acknowledgements

Development of this RFC and the prototype implementation for the Onion VFD is supported by the EOD project.

Revision History

February 10, 2020:	Version 1 circulated for comment within The HDF Group.
June 19, 2020:	Version 2 circulated for comment within The HDF Group.
???	Version 3 includes updated details from actual implementation.
December 5, 2021	Version 4 minor updates to match library code (requires more work) This version was added to the feature/onion_vfd branch in the repo

Appendix: Amended Data as Contiguous Runs

If the goal is strictly to minimize the footprint of amended data, using runs is clearly the best option – storing only exactly the contiguous region of amended bytes. No additional data will be stored, and each amendment serves as a diff from the previous revision. If two runs would overlap or 'touch' in working memory (write mode, e.g.), then the most recent data would overwrite any previous data in the run, and the runs merge into a single contiguous run.

In the live index, each run entry would have an offset and range in the logical file, mapped to a location in the backing store (its length implicit). The live index would be updated and consolidated in real time, merging runs in a new revision as necessary before committing the history, keeping the index and storage space at an absolute minimum. To be efficient, a binary search tree quickly recommends itself, allowing access in O(log(N)) – a self-balancing tree, such as from the red-black family, is an obvious choice.

However, consolidating the working data – already tentatively committed to the file – in real time is not trivial, and incurs significant overhead in allocating space for and copying, joining, and relocating run data. A naïve implementation might simply copy prior data into a new block, which results in unused space within the backing store (from the original, partial copy of the run data) – a possible solution would be to 'defragment' the storage space, which would involve potentially many I/O operations and be accordingly quite slow.

Due to the endemic complication of reorganizing working data (the result of this reorganization being a prime motivator for electing for runs), this will not be the approach chosen for the initial cut of the Onion VFD.

Appendix: Binary Search Tree Implementation of Revision Index

A binary search tree of the revision index could be implemented as a form of self-balancing tree, rather than the hash table discussed above. A binary search tree is uniquely applicable to both page- and run-based approaches (whereas a hash table is not).

In the case of the paged approach, the data in each node may comprise of solely either the staring offset of the page in the logical file, or the ID of the page (i.e., page offset / page size). A more generalized approach would include the length of the entry, being either the page size, multiple of page size, or exact length of the run. Note that including page length when each node is exactly one page introduces overhead of unused memory space. In any event, `offset+length` of a give node `k` must always be less than offset of node `k+1` (e.g., any byte offset in the HDF5 file maps to at most one entry in the tree).

Pros:

- Smaller index size
    - Unlike a hash table, all or near-all of its allocated memory is used (with the uncommon removed-but-not-deleted node, depending on the implementation, serving as unused placeholders/cruft).
- Accommodates runs
    - In both paged and run data, it is not difficult to consolidate elements that might otherwise require multiple hash table entries. An example using the paged approach, a single amendment that spans multiple consecutive pages could be recorded as a single "run" of those pages as a single node.

Cons:

- Slower performance
    - Slower to access an item in the index, and much slower to access an item *not* in the index.
    - Depending on the implementation, could have further delay with each node access incurring overhead from cache-misses, compounded by the O(log2(N)) accesses.

If this approach is to be implemented, it may require a modified version of the revision index entry structure.

Appendix: Simpler, Slower Hash Table for "Live" Index

This alternative hashing algorithm replaces chaining with quadratic probing, and bit-shifts with modulo.

1) The size of the hash table is always prime – first prime greater than a power of 2.
2) The hash table never has more than half its entries populated.
3) Hashing function is page ID modulo the size of the hash table.
4) If a collision occurs, use quadratic probing to combat primary clustering: '$H_i = H_0+i^2$ (mod M)'. The square operation may be simplified the sequential operation 'H_{i-1} + 2*i - 1 (mod M)',

where the 'mod M' operation also simplifies to 'if H_i > M then H_i -= M' with each iteration. (H is the hashing value, M is the size of the hash table.)

5) When necessary, new table is allocated at approximately twice the size (first prime greater than the next greater power of 2) and all extant entries are re-hashed into the new table; the old table is then deallocated.

Appendix: In-Call Filtering for History Browsing with H5FDfctl()

Below is a possible component extension to the relevant filters to the call, which would add a pointer to this structure type as part of their definition.

The implementation would restrict results of the browsing call(s) to revisions with revision record data conforming to the intersection of all user-supplied criteria, potentially greatly reducing the work required by the user.

```
/* ----------------------------------------------------------------------------
 * Structure:   H5FD_onion_info_history_selection
 *
 * Purpose:     Encapsulate the filtering criteria used when browsing the
 *              Onion whole-history. Intended to be used by the Onion
 *              implementation to actively filter results when accessing
 *              history data from the file.
 *
 *              For the first cut of the Onion VFD, this feature set is
 *              unlikely to be fully implemented.
 *
 * magic:       4-byte semi-unique "magic" number identifying structure.
 *              Must equal SYMBOLIC NAME TODO to be considered valid.
 *
 * version:     4-byte number; future-proofing guard, informs struct membership.
 *
 * time_of_creation:
 *              Null-terminated string of formatted ISO-8601 timestamps.
 *              Used to find revisions created at a specific time (x),
 *              before a specific time (-x), after a specific time (x-),
 *              between two given times (x-x), or a combination (x-x, x-).
 *              Each timestamp must be fully-formed, but whitespace between
 *              entries and the separator characters ('-', ',') is ignored.
 *              If the string is empty or malformed, it is ignored.
 *
 * username_regex:
 *              Null-terminated regular expression string to select
 *              revisions where usernames match the pattern.
 *              If the string is empty, it is ignored.
 *
 * creation_user_id:
 *              If not zero (0), selects revisions created by the given User ID.
 *
 * comment_regex:
 *              Null-terminated regular expression string to select
 *              revisions where the comment matches the given pattern.
```

```
 *              If empty, it is ignored.
 *
 * relation_id: Revision ID of a revision, used with relation_mode to
 *              select from the history relative to a given revision ID.
 *
 * relation_mode:
 *              One-byte bitmask informing how to interpret relation_id.
 *                      + ONION_SELECTION_RELATION_NONE (0x00)
 *                              relation_id is ignored.
 *                      + ONION_SELECTION_RELATION_CHILDOF (0x01)
 *                              Find children of relation_id.
 *                      + ONION_SELECTION_RELATION_PARENTOF (0x02)
 *                              Find parents of relation_id.
 *                      + ONION_SELECTION_RELATION_DISJOINT (0x04)
 *                              Find revisions that are neither parents nor
 *                              children of relation_id -- revisions that are
 *                              part of any other branch in the history.
 *                              Only meaningful if divergent histories enabled.
 * ------------------------------------------------------------------------
 */
struct H5FD_onion_info_history_selection {
    uint32_t magic;
    uint32_t version;
    char    *time_of_creation;
    char    *username_regex;
    uint32_t creation_pid;
    char    *comment_regex;
    uint64_t relation_id;
    uint8_t  relation_mode;
};
```

Commented [JS14]: Full symbolic names TODO