HDF5 Single-Writer/Multiple-Reader (SWMR) Feature Design and Semantics

The single-writer/multiple-reader (SWMR) feature

This feature will add support to the HDF5 C library that will allow multiple reader processes to inspect a file that is concurrently being written to by a single writer process, without requiring any inter-process communication ("SWMR semantics"). Due to the complexity of this feature and the technical challenges involved in the implementation, development is proceeding in several phases. This document supports phase I of the project, which has the following user-level characteristics:

- The only file modification allowed by the writer is appending data to pre-existing datasets.
- Variable-length and region reference datatypes are not supported.
- HDF5 file and reader process consistency are a major consideration.
- Write throughput should not be significantly by SWMR.
- Read performance and reader/writer latency are less of a concern and will be optimized in a later phase.

Phase I is intended to be a demonstration of the SWMR concept and a test bed for future development. A platform-independent, concurrent testing and debugging harness will be created as a part of this phase's work.

The SWMR feature is intended to be a part of the future HDF5 1.10 release, due in 2014 or later. It is unclear at this time if will be possible to create files under SWMR that are 1.8 compatible.

An overview of the document

The purpose of this document is to describe the operation of HDF5 C library internals and the layout of HDF5 file data structures as pertains to appending data to pre-existing datasets under SWMR semantics (defined below).

Purpose of Document

This document has been prepared as a part of the first stage of the single-writer/multiple-reader (SWMR) feature project.

The document begins with a description of the problem, describes the layout, behavior, and potential flush dependency issues of the data structures that are used as chunk indexes in the current 1.8 and future 1.10 versions of the HDF5 library. Since the internals of the HDF5 library are not well documented, this whitepaper will serve to communicate this information to project managers, testing engineers, and customers, who may not have in-depth knowledge of the pertinent library internals. In particular, this information will be important for ensuring that the SWMR feature is thoroughly and efficiently tested.

Background

The main problem that must be solved for the SWMR feature to work properly is ensuring that the central HDF5 file is kept consistent at all times. An HDF5 file contains a significant amount of file metadata that can include internal file pointers. If any of these pointers resolve to a metadata object that has not been propagated to disk, the file will be in an inconsistent state and a reader that attempts to resolve the object located at that offset will likely crash. In addition to the file offset issue, there are other cases in the library where stored metadata values can cause erroneous behavior (e.g., the version 2 B-tree storing the number of children in the node).

The general solution to this problem is to modify the writer's metadata cache to ensure that metadata that is referred to by another metadata object is flushed from the cache first. This ensures that a reader will never attempt to resolve an invalid offset. The parent-child relationship between the metadata objects is called a *flush dependency* and is described in more detail below.

When appending to datasets, the only file objects that are created or modified are the indexes that the library uses to quickly locate a particular chunk in the HDF5 file. There are four of these:

Data Structure	Version	Usage
B-Tree (v1)	1.8, 1.10	(1.8) All chunk indexes. (1.10) All chunk indexes when 1.8 compatibility is in effect (default!).
B-Tree (v2)	1.10	Datasets with **more than one** unlimited dimension.
Extensible Array	1.10	Datasets will **one** unlimited dimension.
Fixed Array	1.10	Datasets with **no** unlimited dimensions.

Scope and Limitations

This document only concerns the layout, behavior, and potential flush dependency issues of the internal HDF5 data structures that are used as chunk indexes in the 1.8 and 1.10 versions of the HDF5 library. All other internal HDF5 data structures are ignored. Some of the described data structures are also used in a context outside of chunk indexing (e.g., B-tree (version 1) symbol tables), but that usage is largely ignored in this document. Testing of these data structures in a SWMR context is described in a separate document.

In the discussion of each data structure, delete operations are ignored since neither file object deletion not dataset shrinkage will be supported in the first stage of the SWMR project. In the B-tree discussion, standard B^+-tree and B*-tree operations are not discussed in detail as this information is readily available in textbooks and on the web.

SWMR Semantics

Copied almost verbatim out of the progress report. Need to flesh this out and describe flock-like behavior/implementation.

- Multiple readers can open the same file for reading when no writer has the file open for writing.
- No reader can open the file for reading when a writer is accessing the same file for writing.
- No writer can open the file for writing when reader(s) are accessing the same file for reading.
- No writer can open the file for writing when a writer already holds the file open for writing.
- Multiple readers can open the same file for "reading and SWMR-read" when a writer opens the file for "writing and SWMR-write".

Metadata Cache Flush Dependencies

Background

The metadata cache in the HDF5 library is used to hold pieces of file format metadata that are recently accessed by the library. Each piece of metadata is stored as an entry in the metadata cache and the cache attaches information about the entry's type (B-tree node, heap block, etc.), encoded offset and length in the file, recency of last access, whether the metadata has been modified (i.e. its "dirty" status), etc. to each entry in the cache. Data structures in the file, such as object headers, B-trees, heaps, etc., are composed of multiple pieces of file metadata, all of which are accessed through the metadata cache interface within the library.

Purpose

For SWMR-safe file modifications to work correctly, metadata for each file data structure must be written to the file in a particular order. The library code that manages each file data structure determines which pieces of its metadata are affected, and the order that those pieces of metadata should be written. To support the data structure management code, the metadata cache exposes library-internal interfaces that enable the definition of a write-ordering between two entries in the metadata cache. These write-orderings are called "flush dependencies" within the library.

Operation

When a flush dependency is created between two metadata cache entries, one entry is designated as the "parent" entry and the other as the "child" entry. Multiple flush dependencies may be created for each entry in the metadata cache (i.e. each parent entry may have multiple child entries), and each cache entry can be a parent, a child, or both.

The principal function of a flush dependency between two cache entries is to define an ordering between write operations of those entries. Parent entries that have been modified (i.e. are dirty) may not be written to the file until all of their child entries are clean (i.e. if a child entry is dirty, it must be written to the file). Circular parent-child flush dependencies are not allowed. There is no write ordering between cache entries that don't have a flush dependency defined between them: either entry could be written first. Additionally, when a cache entry is a parent in a flush dependency, it is pinned in the cache (i.e. it can't be evicted) until it has no child entries.

Example

Consider the following example of flush dependencies between metadata cache entries, shown in Figure 1 below:

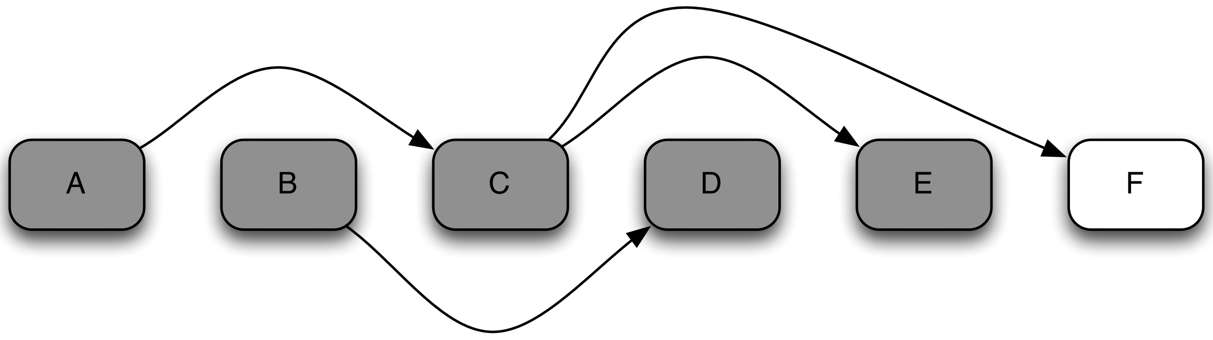


Figure 1

This diagram shows 6 metadata cache entries, with several flush dependencies defined: A is the parent of C, B is the parent of D, and C is the parent of both E and F. In the diagram, dirty entries are shaded (entries A-E) and clean entries are unshaded (entry F). With this configuration, entry A could not be written to the file until entry C is clean, and entry C could not be written to the file until both entries E and F are clean. Likewise, entry B could not be written to the file until entry D was clean. So, with this configuration, when the metadata cache is attempting to flush dirty entries to the file (perhaps when flushing all the metadata for the file), either entries D and E could be written to the file first (making then clean), then entries B and C, and finally entry A. (Although if C was written before B, A would be able to be flushed before B, also).

Implementation

The internal API routines to operate on flush dependencies are:

herr_t H5AC_create_flush_dependency(void *parent, void *child);

herr_t H5AC_destroy_flush_dependency(void *parent, void *child);

Each pointer parameter for the flush dependency calls is a pointer to a metadata cache entry, but the type of each entry varies, so they are passed as void *'s. Once the dependency is set up, the relationship is managed by the cache and doesn't need further interaction from file data structure manager code and persists until the child entry in the relationship is evicted from the cache, or the relationship is explicitly destroyed.

Testing

The flush dependency code in the metadata cache is extensively tested by the existing metadata cache testing code and does not need additional tests.

Chunk Indexing

Datasets (Object Headers)

Datasets in the HDF5 file are, at the lowest level, represented by an object header that contains and/or refers to internal metadata about the dataset and also points to the data, either directly in unchunked or compact datasets[1], or via the chunk index in chunked datasets. Since the SWMR prototype is only concerned with the latter (extendable datasets must be chunked), we only consider them and ignore the simpler forms.

The object header is

The chunk index is created lazily by default. No chunk index storage is allocated until the first dataset write. On first write, the initial index storage is written out to the disk and its offset in the file is written into the object header. For SWMR, this requires a flush dependency between the "root" of the chunk index and the object header.

Chunk Proxies

When used as a chunk index, *chunk proxies* must also be considered. These chunk proxies are stubs that allow flush dependencies to be set up on chunk data. They ensure that flushed chunk index objects will not refer to data that has not yet been flushed from the chunk cache. Proxies are not needed for symbol table entries since these data exist in the metadata cache.

Datatype Limitations

Some HDF5 datatypes will not be supported in the initial SWMR prototype due to missing flush dependencies that could cause errors in reader processes. They are mentioned here since their non-trivial storage would require interaction between the dataset's file objects and other HDF5 file objects.

The non-supported datatypes are:

Variable-Length Types.

A dataset of a variable-length type stores offsets into a global heap, where the actual data is contained. Flush dependencies between the chunk proxies and the global heap have not been implemented at this time. When this is implemented, the global heap will have to be flushed before the chunk proxy.

Region References

A dataset of region reference type stores offsets into a heap, where the actual data is contained. Flush dependencies between the chunk proxies and the global heap have not been implemented at this time.

[1] Compact datasets actually store the data in the object header so "refer" is not technically correct.

When this is implemented, the global heap will have to be flushed before the chunk proxy. The problem of ensuring that the referred-to region in chunked datasets is flushed to the disk before the region will also need to be addressed.

Object references are supported as long as the objects exist before SWMR operations begin. No new objects can be created under SWMR operations.

Other HDF5 datatypes, including compound, array, enum, and fixed-length strings, are simply stored as contiguous bytes in the dataset and there are no interactions with other file objects. These are all supported in the SWMR prototype.

B-Trees (version 1)

Overview

The data structure is a relatively straightforward implementation of a standard B^+-tree data structure with sibling pointers to facilitate leaf traversals. In addition to its use as a chunk index, it is also used as a symbol table (group) index. It has been available since the initial implementation of the library and remains in both the current 1.8 and future 1.10 versions of the library.

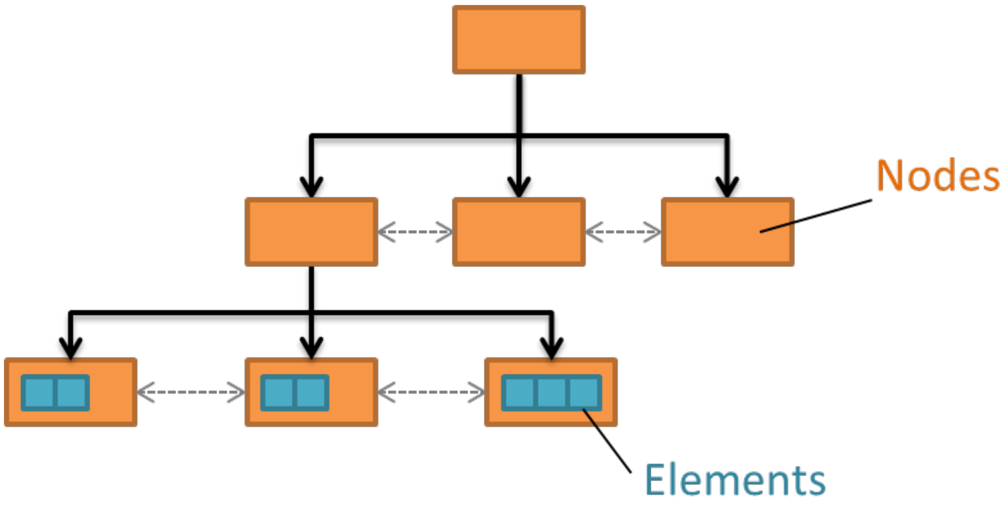
Layout and Operation

A version 1 B-tree is implemented as a collection of identical nodes. Each node conceptually consists of n values surrounded by n+1 keys[2]. The keys are (essentially) the coordinates of the element closest to the origin in the chunk and the values are file offsets, pointing either to other B-tree nodes in internal nodes or to chunks in leaf nodes. These values are associated with the key to the right. The leftmost key in the node is interpreted as the 0th chunk[3]. Each node also includes pointers to its left and right siblings in the level[4]. These pointers are only followed during leaf-level iteration, though this has been changed in current versions of the library and the sibling pointers are no longer used (though they must be maintained for backward compatibility). Each node also stores a value that indicates its level (leaves = 0, nodes on the level immediately above leaves = 1, etc.) as well as a few other values that have no impact on SWMR operations. The detailed layout of a B-tree node is described in the HDF5 file format specification and is identical in both the 1.8 and 1.10 versions of the library.

B-tree nodes are created as needed, when chunks are added to the dataset. When a node is not large enough to contain a newly added value, it is split into two nodes at the same level and the links (and possibly other splits) are propagated up the tree as far as needed. Nodes are allocated at their "full" size so they do not have to be resized and possibly moved in the file as the number of key-value pairs they

[2] n can be set by the user using the H5Pset_istore_k and H5Pset_sym_k API functions, for chunks and groups, respectively. The default is currently 64 for chunk indexes (8 for symbol table indexes).
[3] This is historical. For the record, in symbol table indexes, the keys are heap IDs referring to link information in the group's local heap (which uses the link names for comparisons) and leaf node values are symbol table entries. The values are associated with the key on the left and the "extra" key on the right is interpreted as an ASCII NUL (\0) character.
[4] The "undefined address" is stored in sibling pointers of nodes that are on the edge of the tree.

contain grows. The depth of the tree only changes when the root node splits. When this happens, the root node's values are copied into two new nodes, the root node's values are zeroed out, and the offsets of the two new nodes are set as the first two values of the root node. This is done so that the root node's offset never changes, and its offset in the object header never needs to be updated.



Cache Objects, References, and Flush Dependencies

The version 1 B-tree is represented, both in the file and in the metadata cache, by a single type of cache object and each node is a separate cache object.

The flush dependencies in a version 1 B-tree are straightforward as a flush dependency only exists between each child node and its parent (i.e., the children must be flushed to the disk before the updated parent node can be flushed). When a node splits, two new nodes are created and both are flushed before their common parent is flushed. Sibling pointers are not considered for flush dependencies since the current implementation of the library does not use them.

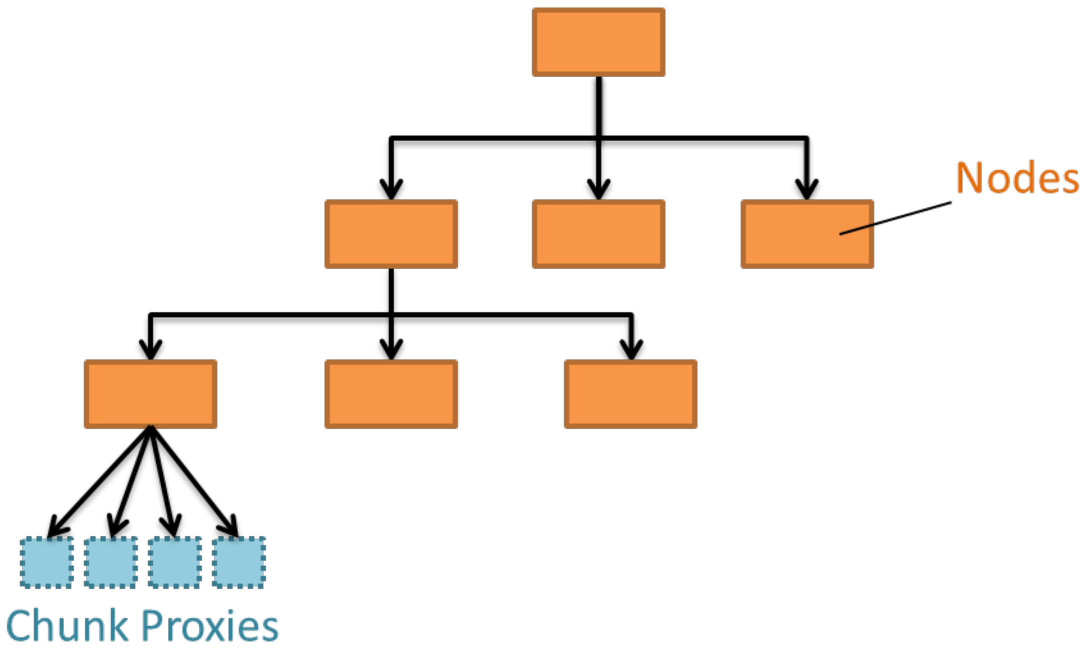


Table 1: Parent-->Child flush dependencies between version 1 B-tree objects in the metadata cache.

Parent	Child	Reason
internal node	child node (internal or leaf)	Tree values store node offsets
leaf node	chunk proxy/symbol table node	Tree values store file offset of a chunk or symbol table node.

B-Trees (version 2)

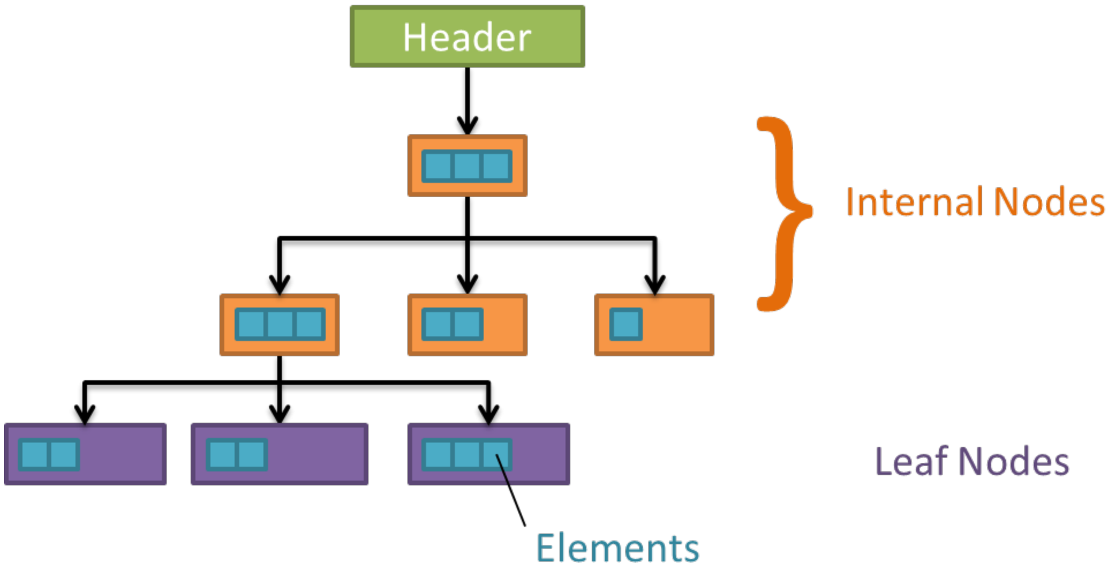
Purpose

The version 2 B-tree is used in more contexts in the library than the version 1 B-tree. There are currently around ten indexing uses for the version 2 B-tree, each with its own particular record type, but we only consider its use as a chunk indexing structure here. Other contexts will be explored in later stages of the SWMR project.

In the 1.10 version of the library, the version 2 B-trees are used for indexing chunks in datasets with two or more unlimited dimensions. The data structure is a relatively straightforward implementation of a standard B*-tree data structure. Note that this differs from the version 1 B-tree subtype, as the version 2 implementation attempts to rebalance nodes. Also unlike the version 1 B-trees, there are no sibling pointers to facilitate leaf traversals. The version 2 B-trees are not used as a chunk index data structure in the 1.8 version of the library.

Layout and Operation

A version 2 B-tree has a slightly more complicated structure than its version 1 counterpart. A tree consists of a header, a collection of internal nodes, and a collection of leaf nodes. Unlike the version 1 tree, records can be stored directly in an internal node. The records contained in the tree are also more variable and complicated than the records stored in the version 1 B-trees, which reflects their more varied use. Unlike the v1 B-tree, the v2 B-tree does not store separate keys and values. Instead, nodes simply store records from which key information is extracted depending on usage. For chunk indexing, the key part of the record is the chunk coordinates.



The differences between the two types of B-trees are illustrated in the table below.

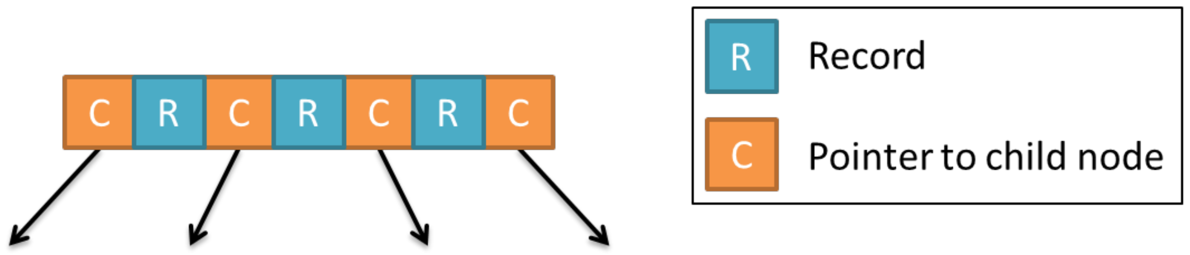
Table 2 - Differences between the version 1 and 2 B-trees

Characteristic	version 1	version 2
B-tree sub-type	B^+-tree	B*-tree
Usage	Chunk Indexes Symbol Table Indexes	Chunk Indexes Dense Symbol Table Indexes Dense Attribute Indexes Fractal Heap Large Object Indexes
Components	Nodes	Header Internal Nodes Leaf Nodes
Keys	Size of chunk Filter mask Offsets of chunk in all dims	Offsets of chunk in all dims
Values	Offsets of chunk in all dims	Offsets of chunk in all dims Chunk size (if filtered) Filter mask (if filtered)

Sibling node pointers?	Yes	No
Data stored in internal nodes?	No	Yes
# of children rebalanced on inserts?	No	Yes

The header contains the offset of the root node, the total number of records in the root node and in the tree, and some miscellaneous structural information.

Internal nodes contain information some bookkeeping information and (conceptually) a collection of records contained between child node pointers (see figure). The default number of records is 512.



Each record consists of the chunk coordinates in all dimensions, the chunk's offset in the file, and, if filtered, the chunk size and filter exclusion mask. Each child node pointer also maintains the number of immediate and total children below it. These counts allow the library to quickly go to the nth child in the structure.

Leaf nodes only store records, which are identical to those stored in the internal nodes. The number of records in a leaf node is identical to that stored in the internal nodes.

As in the version 1 case, B-tree nodes are created as needed, when records are added to the container. When a node is not large enough to contain a newly added value, the library first attempts to rebalance the node by redistributing records to its siblings. If this is unsuccessful, the node is split into two nodes at the same level and the links (and possibly other splits) are propagated up the tree as far as needed. Nodes are allocated at their "full" size so they do not have to be resized and possibly moved as the number of key-value pairs they contain grows. The depth of the tree only changes when the root node splits. Unlike the version 1 B-trees no special handling of the root node takes place since the version 2 B-tree includes a header that never changes location in the file.

Cache Objects, References, and Flush Dependencies

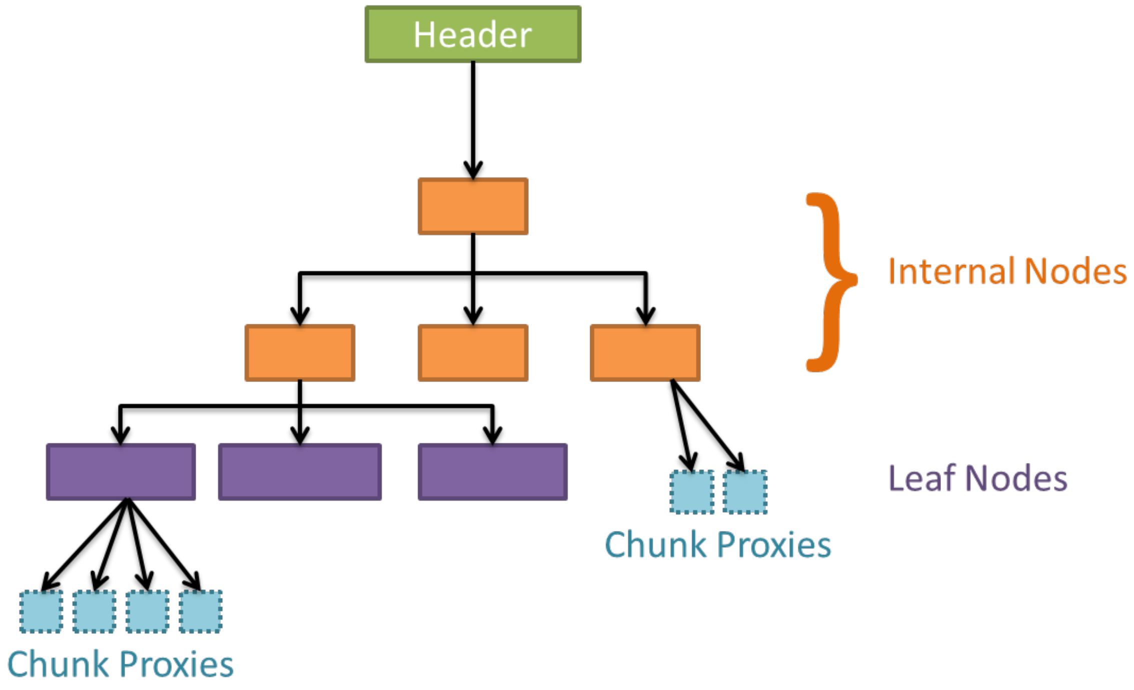
The version 2 B-tree has more complicated flush dependency issues than the version 1 B-tree due to the requirement to maintain the child counts in the nodes. Since both the child offsets and these counts must be correct for the set/get algorithms to succeed, there is no set of atomic write operations that maintains a consistent B-tree on an insert that results in a node split or rebalance. To handle these cases, the parents (all the way up to the header, which does not move in the file) are shadowed in the cache and written separately to the file. This allows any readers that are currently inspecting the B-tree to always have a consistent view of the data structure. A shadow operation also requires re-evaluating the flush dependencies due to the new node structure.

This shadowing has negative implications for SWMR as B-tree nodes will often be "duplicated" in the file instead of overwritten, increasing its size. Note that the number of writes does NOT change, however.

Aside from this issue, the flush dependencies in a version 2 B-tree are straightforward as a flush dependency only exists between each child node and its parent (i.e., the children must be flushed to the disk before the updated parent node can be flushed). The header, internal, and leaf nodes are represented as different types of cache objects.

Table 3: Parent-->Child flush dependencies between version 1 B-tree objects in the metadata cache.

Parent	Child	Reason
header	root internal node	Header stores offset of root
internal node	internal or leaf node	Node stores offsets of children
internal and leaf nodes	chunk proxy	Records store chunk offsets



Extensible Array

Purpose

The extensible array is used in the 1.10 version of the library as a chunk index for datasets with a single unlimited dimension and any number of fixed-size dimensions. It is a variant of a deterministic skip list and is optimized for appending along the unlimited dimension. The extensible array does not exist in the 1.8 version of the library.

Layout and Operation

The idea behind the extensible array is that a particular data object can be located via a lightweight indexing structure of fixed size for a given address space. This indexing structure requires only a few (2-3) file operations per element lookup and gives good cache performance. Unlike the B-tree structure, however, the extensible array is optimized for appends. Where a B-tree would always add at the rightmost node under these circumstances, either creating a deep tree (v1) or requiring expensive rebalances to correct (v2), the extensible array has already mapped out a pre-balanced internal structure. This idealized internal structure is instantiated as needed, when chunk records are inserted into the structure.

On the disk, the extensible array consists of several components:

Header

Each extensible array has a header, which contains bookkeeping information about the array and stores the offset of the index block. The element count is stored here as well.

Index Block

The index block primarily serves as a dictionary into the internal index nodes, called super blocks.

Data Block Pages (Data Blocks)

Data block pages store the bulk of the data in an extensible array. They are very simple and store the elements along with a small amount of internal bookkeeping data. Data block pages are called data blocks when accessed directly through the index block (see below).

Super Blocks

The super blocks are the internal nodes of the extensible array. Each superblock stores offsets into a set of data block pages.

Elements

Each element in the array consists of the offset of the chunk in the file and, if filtered, the chunk size and filter exclusion mask.

General Concept

The figure below shows the layout of the extensible array. In a 32-bit extensible array, the index block contains 32 superblock offsets. The highest bit set in the index number determines which superblock offset is used to find the data. This superblock will contain 2^n-1 / 1024 data block page offsets, where n is the pointer level. Any item in the array can be located in two deterministic steps.

Optimizations

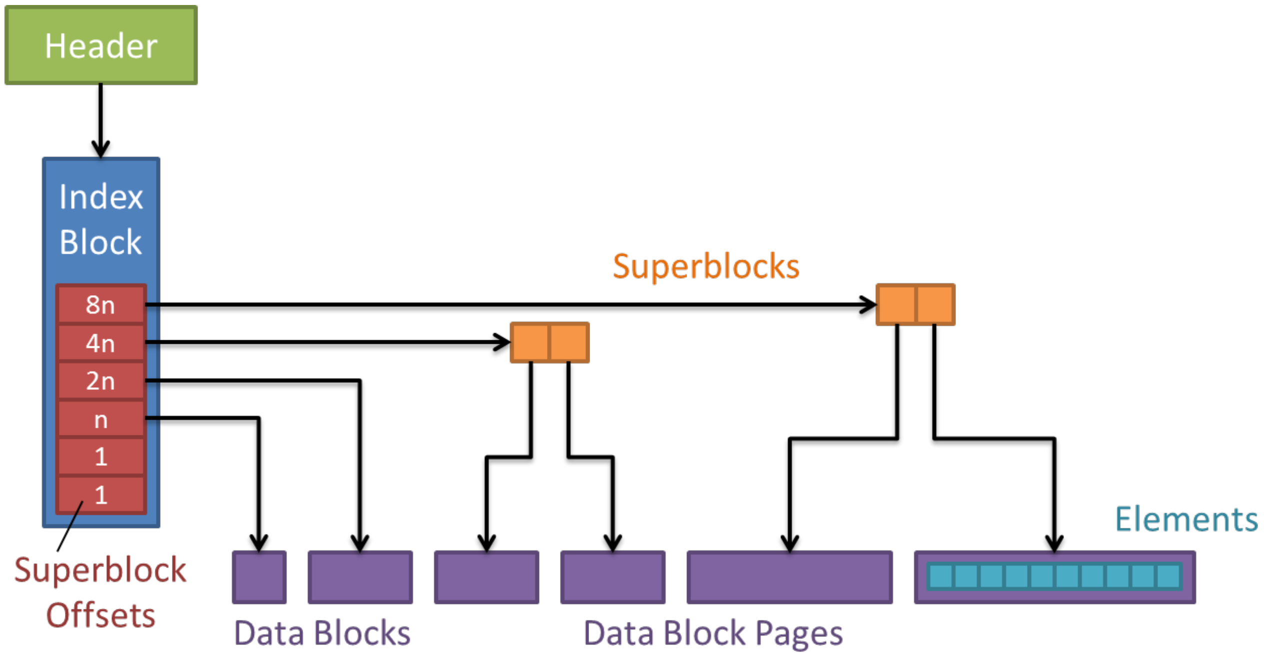
- The first 4 super blocks store their elements directly in the index block. This saves two lookups and significant space for very small datasets.
- The next 8 super blocks store their data block page pointers directly in the super block. This saves one lookup at the cost of a slightly larger index block. Instead of being called data block pages, they are called data blocks.

Example:

To find element 12345 (not considering the optimizations):

1) The super block is determined by looking at the highest bit set in 12345 (0b11000000111001), which is 14, so the 14th super block is loaded from the disk.
2) That superblock stores 8192 elements, which are spread across eight 1024-element data block pages. The element is the 4153rd element in this super block, which is stored on the 5th page.

The element was located deterministically in two steps; requiring 3 I/O operations (load index block, super block, data block page) if none of the data structures were in the metadata cache. This is true for all elements in the array, even the 4294967295th.

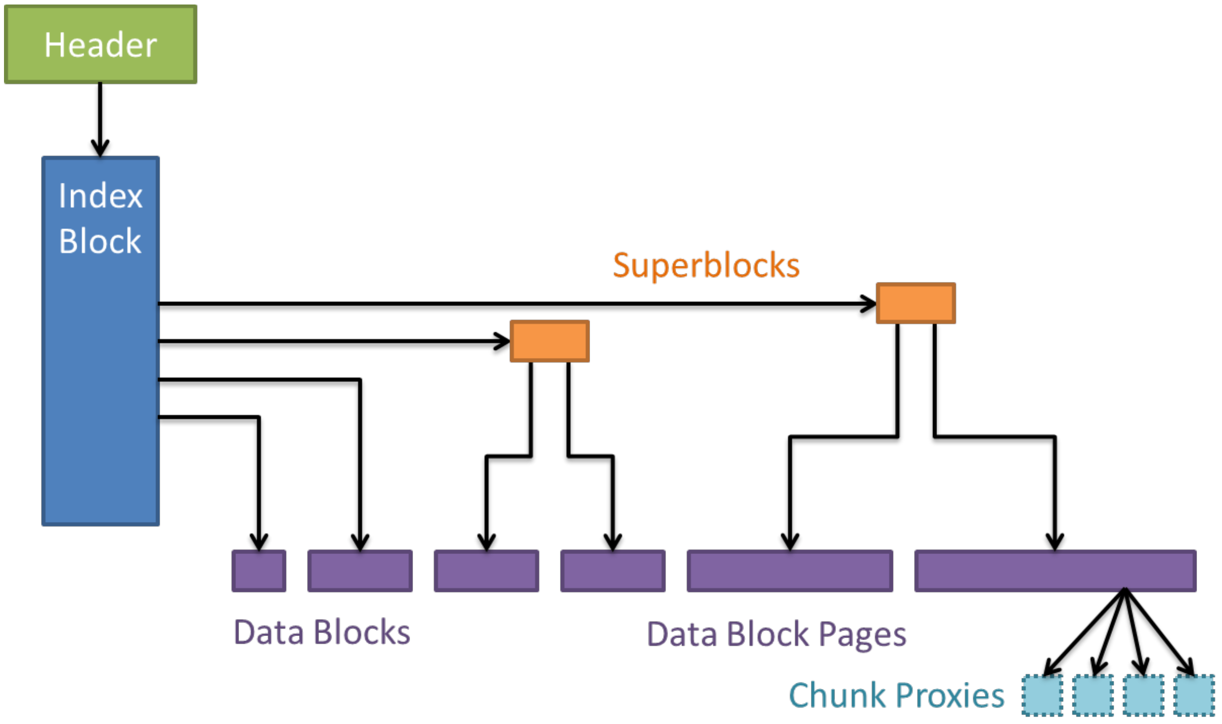


Cache Objects, References, and Flush Dependencies

The extensible array has separate cache object types for each component part:

- header
- index block
- super blocks
- data blocks
- data block pages

The flush dependencies that exist between the metadata cache objects are shown in table X and expressed graphically as arrows in figure X. In each listed flush dependency, the children must be flushed from the metadata cache before the parent to ensure readers do not resolve an invalid file offset.



NOTE: Many of these file/cache objects include the offset of the header, but this is only intended for use by file integrity and repair tools. These do not require an <object>-->header flush dependency since low-level file analysis and reconstruction will not be compatible with SWMR.

Table 4: Parent-->Child flush dependencies between extensible array objects in the metadata cache.

Parent	Child	Reason
header	index block	Header stores file offset of index block
index block	chunk proxies	Index block stores file offset of a few chunks
index block	data blocks	Index block stores file offset of a few data blocks
index block	super blocks	Index block stores file offset of super blocks
super block	data block page(s)	Super block stores file offset of data block page(s)
data block /page	chunk proxies	Data block/page elements store file offset of a chunk

Fixed Array

Purpose

The fixed array is an on-disk data structure that represents a one-dimensional array containing a fixed number of identically-sized elements. It is a simple data structure that offers relatively fast lookups, a smaller total size on disk, and does not require expensive maintenance operations such as relocating or splitting nodes. It can only be used when the number of elements is fixed, will never change, and is known *a priori*. Consequently, it is only used to index chunks in datasets where there are no unlimited dimensions. The fixed array index that is created is large enough to index the dataset at the maximum possible size. The dataset may be resized, although not outside of the maximum sizes for each dimension that were stated at dataset creation, and the index's size and element-->chunk mapping never changes.

The fixed array is only used in the 1.10 version of the library and does not exist in the 1.8 version.

Layout and Operation

The fixed array consists of a header and a data block, which can be paged or unpaged. The header contains some basic information about the data structure such as the number and size of the stored elements and the on-disk sizes of the fixed array data objects. The contents of the data block vary, depending on the number of elements stored in the array. Smaller arrays[5] use a single, unpaged data block that directly stores the data elements. Larger arrays use a paged data block composed of a "master" data block followed by multiple data block pages for more efficient cache performance (each page is a separate metadata cache item). The master data block object does not contain any elements. Instead, it contains a bitmap representing which data block pages are "in use" and have been written to at least once[6]. The elements in large arrays are stored in data block pages that are located contiguously after the master data block object. Elements are offsets to the stored dataset chunks, with a filter exclusion mask and the chunk size added when filters are used.

[5] Currently defined as those that store <1024 elements, but this number can be changed by customizing the fixed array's creation parameters. Note that "small" is defined as "number of elements", not "size in bytes", which we may want to reconsider. Also note that this parameter is not exposed outside of the library and is unavailable to users.

[6] This allows the library do lazy initialization of the pages in sparse arrays. It does not save disk space since the total space for the array (data block + pages) is allocated at creation time.

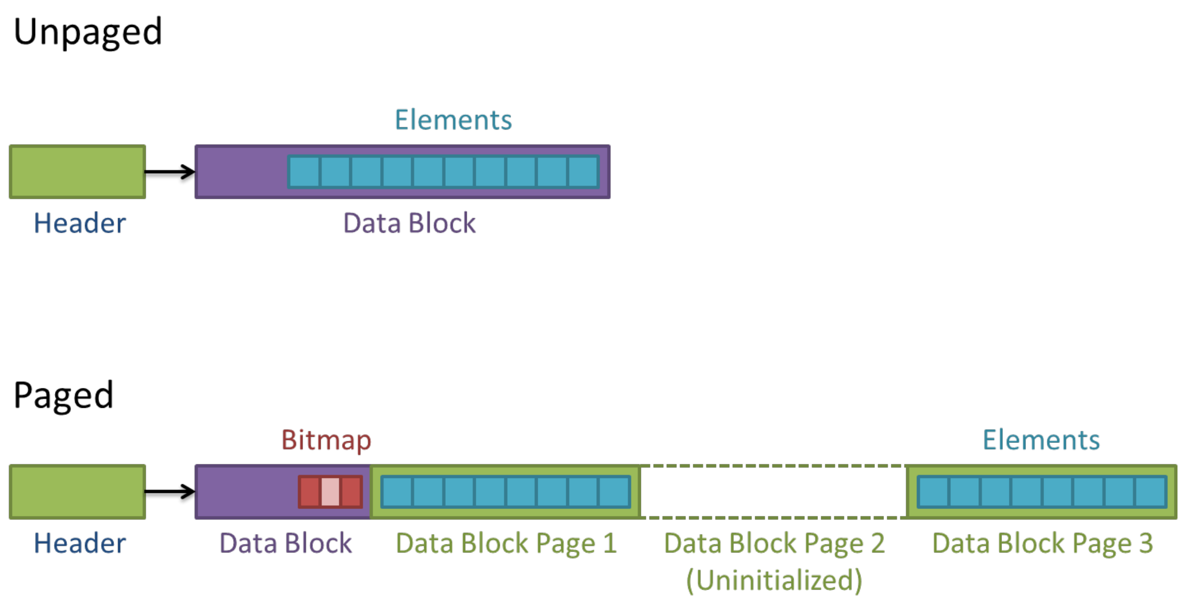


Figure 2: On-disk arrangements of fixed array file objects.

Cache Objects, References, and Flush Dependencies

The fixed array is represented by three types of object in the metadata cache. The *header*, *data block*, and *data block pages* make up the fixed array objects. When used as a chunk index, *chunk proxies* must also be considered. These chunk proxies are stubs that allow flush dependencies to be set up on chunk data. They ensure that flushed chunk index objects will not refer to data that has not yet been flushed from the chunk cache.

The flush dependencies that exist between the metadata cache objects are shown in table 1 and expressed graphically as arrows in figure 2. In each listed flush dependency, the children must be flushed from the metadata cache before the parent to ensure readers do not resolve an invalid file offset.

NOTE: The fixed array data block also includes the offset of the header, but this is only intended for use by file integrity and repair tools. This does not require a data block-->header flush dependency since low-level file analysis and reconstruction will not be compatible with SWMR.

Table 5: Parent-->Child flush dependencies between fixed array objects in the metadata cache.

Parent	Child	Reason
header	data block	Header stores file offset of data block
data block (paged)	data block page (used for first time)	Bitmap in data block indicates in-use/valid data block pages
data block (unpaged) or data block page (paged)	chunk proxy	Data block/page elements store file offset of a chunk

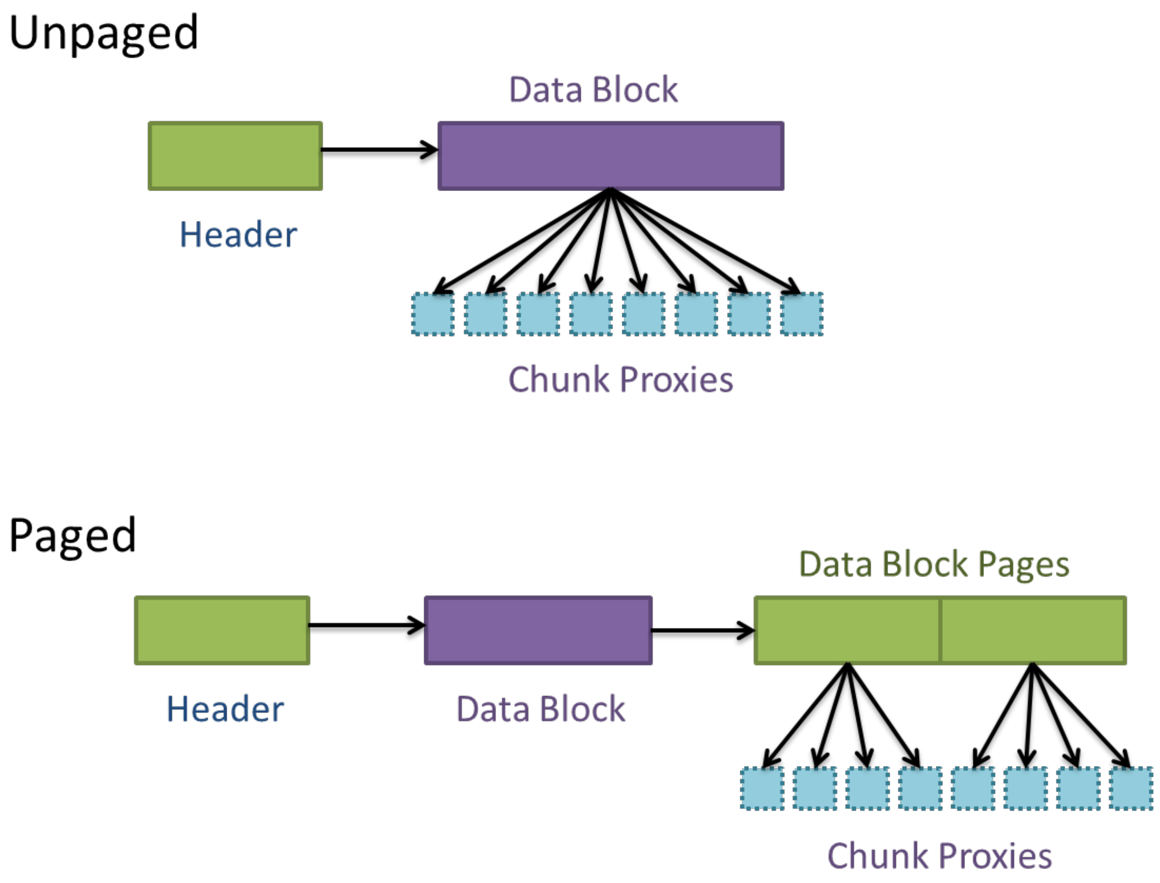


Figure 3: Flush dependencies between fixed array metadata cache objects for both paged and unpaged fixed arrays. The base of the arrow is the parent and the point is the child.

References

Revision History

June 5, 2013:	Version 1 created from prior individual data structure documents.
June 14, 2008:	Version 2 added figures, introduction, major text changes. Intro part is very rough and unfinished.