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

Purpose

This document describes the design and semantics of 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").

The intended audience for this whitepaper is anyone who needs a broad overview of how the SWMR feature will work inside the library. This could range from new library developers and test engineers, who need to quickly get up to speed on SWMR, to project managers and potential funding sources, who need to get a handle on the complexity of the feature.

The document begins with a description of the problem, continues with an overview of the limitations of the feature at this time, then moves on to a general solution to the SWMR problem, and finally concludes with a description of the layout, behavior, and potential SWMR issues of the data structures that are needed to support the initial implementation of SWMR in HDF5 1.10.

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. The level of detail is "mid-level"; somewhere between a high-level cartoon and a low-level implementation plan. The former risks being too glib to really give a good picture of how complicated the SWMR feature is under the hood and the technical challenges that must be overcome. The latter would include too much detail that would risk obscuring the overall ideas behind our solutions to the SWMR problem.

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

Currently, without using file locking or centralized control of read and write operations, it is not possible to safely read and write concurrently to/from an HDF5 file from multiple processes. The reason for this is that an HDF5 file is not a simple flat binary file, but instead utilizes complicated data structures for internal indexing and data organization. When created or modified, these HDF5 data and file objects are cached and subsequently flushed to disk via a modified LRU algorithm. In a single-process situation, the combination of the written-out file objects and the process' in-memory state provide a consistent and complete picture of the HDF5 file. When a second process opens such a partially-flushed file, it will be missing the un-flushed in-memory state of the first (writer) process. If the reader encounters a file object that stores an offset to another, un-flushed, file object, this would result in either returning garbage data to the reader or attempting to construct an HDF5 file object from garbage, which would

most likely crash the reader. This is a limitation, not just of HDF5, but of any file format that is internally more complicated than a simple bucket of bytes.

The general solution to this problem is to modify the writer's metadata cache and file data structure code 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.

SWMR Semantics

The semantics of SWMR operations are:

- 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 non-SWMR 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".
- Non-SWMR readers will not be able to open a file opened for SWMR writing.

Scope and Limitations

Due to the complexity of the SWMR feature and the technical challenges involved in the implementation, development is proceeding in several phases. The project is currently on phase I, in which the only allowed operation is appending to previously-created datasets. Phase I also has a few other characteristics, some of which are elaborated on later in this section:

- 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 also be created as a part of this phase's work. The content of future phases is out of scope for this document, but the general idea is to extend SWMR operations to object creation and then object rename/deletion.

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. All other HDF5 datatypes, including compound, array, enumeration, and fixed-length strings, are simply stored as contiguous bytes in the dataset and there are no interactions with other objects in the file.

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. 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 with some form of flush dependency.

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

Compatibility

Files created under SWMR will probably not be compatible with versions of HDF5 before 1.10.0. This is because avoiding torn writes (defined below) under SWMR operation requires metadata cache objects to contain a checksum field and the chunk index data structure in HDF5 1.8.x and earlier is a version 1 B-tree, which does not store a checksum. Efficient SWMR operation may also require improved chunk index structures and superblock extensions that are not present in HDF5 1.8.

The lack of a checksum could potentially be ameliorated in a somewhat hacky way by appending a checksum to the B-tree nodes under SWMR writing since a reader using HDF5 1.8 would not notice the checksum. This would feature would contain a lot of caveats and sharp edges, though, and should probably not be implemented unless 1.8 compatibility without a repack step were a strong requirement of SWMR.

I/O stack requirements

For SWMR to work properly, the I/O stack on the system where the writer resides, from the write API call to the storage medium, must provide two guarantees:

- **Ordering: Write operation ordering must be preserved.**

  If this were not preserved, the flush dependency logic could be overridden and metadata objects could be written to the file out of order.

- **Atomicity: Write operations should be atomic at the write function call level.**

  If this were not preserved, incomplete file objects could be encountered by a reader. A read that encounters both old and new data due to broken atomicity is called a *torn write*.

The entire I/O stack must be considered when determining whether ordering and atomicity are considered since caching and other optimizations that break these characteristics can occur at any level.

Unfortunately, many I/O stacks will fail one or both of these requirements. For example, the Linux kernel is only atomic at the page level with respect to writes so most file systems fail the atomicity requirement on that OS. Also, the caching layer can break ordering on many network file systems.

To mitigate this, the HDF5 library will use the file object's checksum field to ensure that a valid file object has been read from the disk. When the checksum does not correctly reflect the data read from the disk, the library will retry the read according to a yet-to-be-determined policy. This will mitigate both the ordering and atomicity problems.

Additional Notes

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.

The SWMR feature is intended to be a part of the future HDF5 1.10 release. SWMR functionality will not be available in the 1.8 versions of HDF5 due to missing required file format changes and a SWMR-unsafe chunk index data structure.

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, time 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:

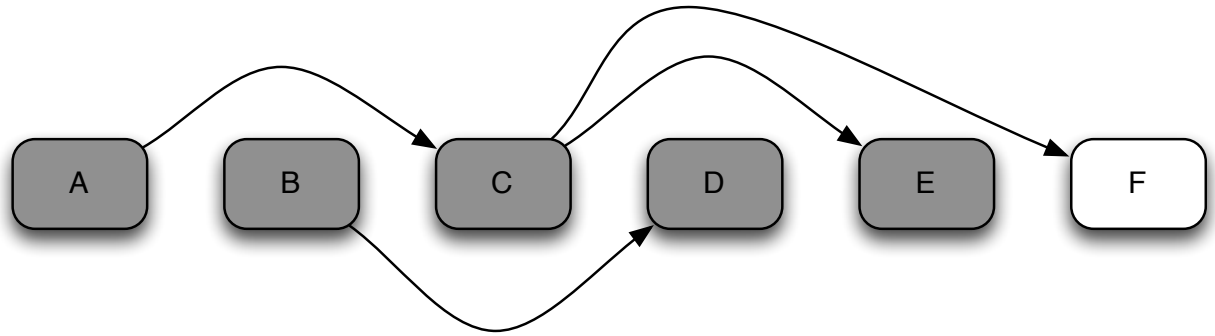

Figure 1 – Representative flush dependencies between metadata cache objects.

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.

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 continguous or compact datasets[1], or via the chunk index in chunked datasets. Since the SWMR prototype is only concerned with extendable datasets, which must be chunked, we ignore the simpler forms.

When appending to datasets, the only file data structures that are created or modified are the object header, which stores dataset size information, and the index that the library uses to quickly locate a particular chunk in the HDF5 file. There are four of these, shown in the table below.

Table 1 - Chunk indexing data structures.

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!). **NOTE: This data structure will probably not be usable under SWMR semantics due to a lack of checksums.**
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.

The chunk index is where most of this version of SWMR's complexity lies since the indexes can have fairly complicated internal structures, with many flush dependencies. The index can be created lazily or can have some initial structure set up at dataset creation time . When created lazily, 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.

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

Chunk Proxies

Chunk proxies are metadata cache objects that represent chunks in the chunk cache. If SWMR writes are enabled, a chunk proxy is created in the metadata cache whenever a chunk is created. This proxy object acts as a cross-cache dependency between a metadata cache object and a chunk cache object. Like any other flush dependency, this prevents the parent (B-tree node, etc.) from being written to disk before the child (chunk, via the proxy). Without chunk proxies, metadata objects that refer to non-existent storage could be written to disk, causing reader errors.

Chunk Index: B-Tree (version 1)

NOTE

The version 1 B-tree nodes may not be suitable for use in SWMR writers. The nodes lack a stored checksum field, so it is impossible for a reader to check for torn writes, which can occur on I/O stacks that are not atomic at the write call level[2].

Overview

The data structure is a relatively straightforward implementation of a standard B^+-tree data structure, with the addition of 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[3]. The keys are (essentially) the coordinates of the element in the chunk closest to the origin 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 0^{th} chunk[4]. Each node also includes pointers to its left and right siblings in the level[5]. 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.

---

[2] As mentioned previously, many I/O stacks do *not* meet this criterion.

[3] 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).

[4] 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.

[5] The "undefined address" is stored in sibling pointers of nodes that are on the edge of the tree.

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 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.

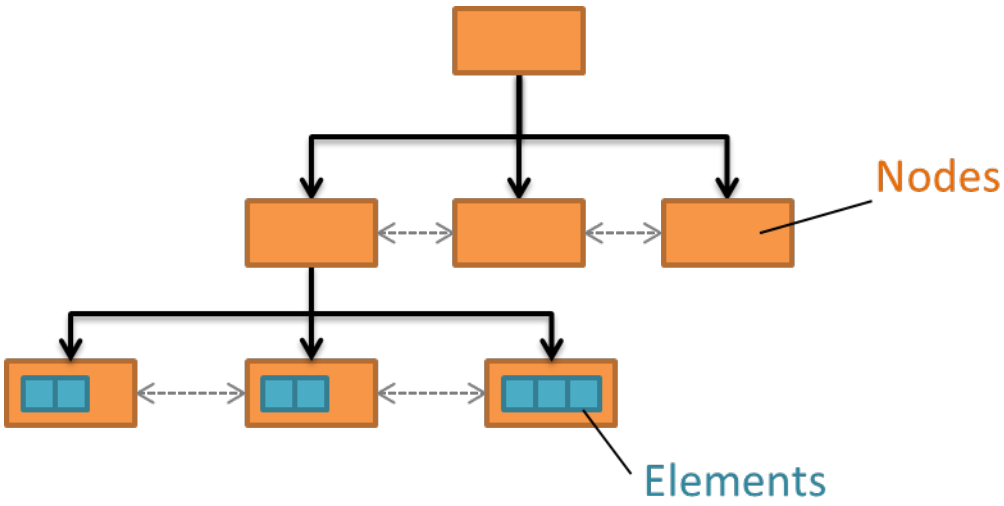

Figure 2 – Version 1 B-tree structure. Not all child nodes are shown to keep the figure uncluttered.

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.

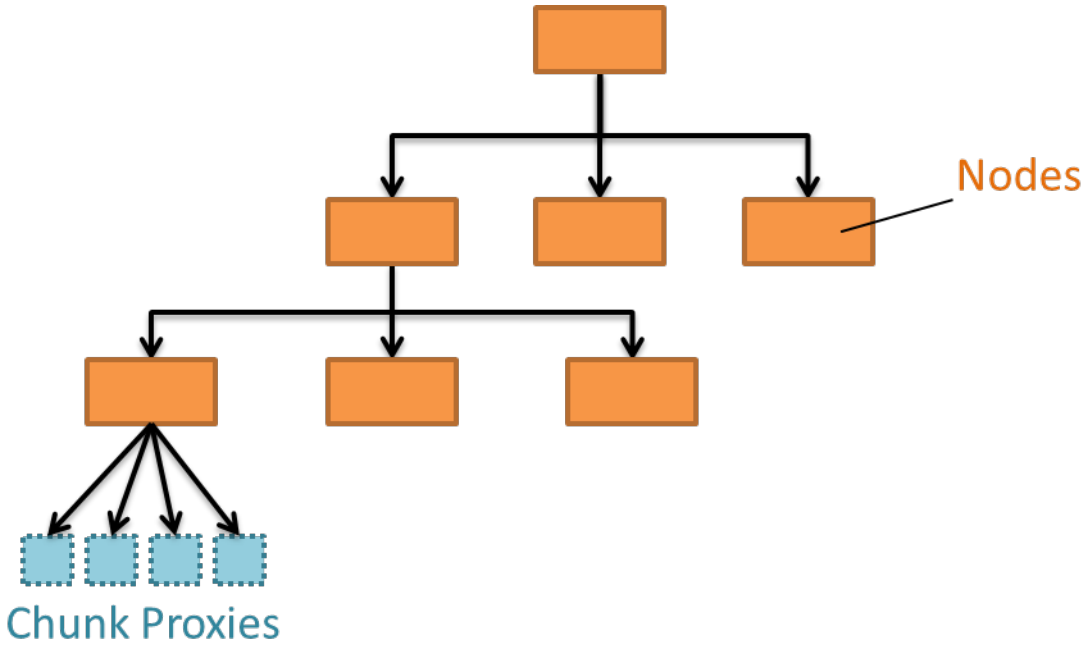

Figure 3 - Version 1 B-tree flush dependencies. Not all child nodes are shown to keep the figure uncluttered. The base of the arrow is the parent and the point is the child.

Table 2: 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.

Chunk Index: B-Tree (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, although it stores the number of child records for

each internal node, making it technically a "counted B-tree"[6]. Note that this differs from the version 1 B-tree subtype, as the version 2 implementation attempts to rebalance records in 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.

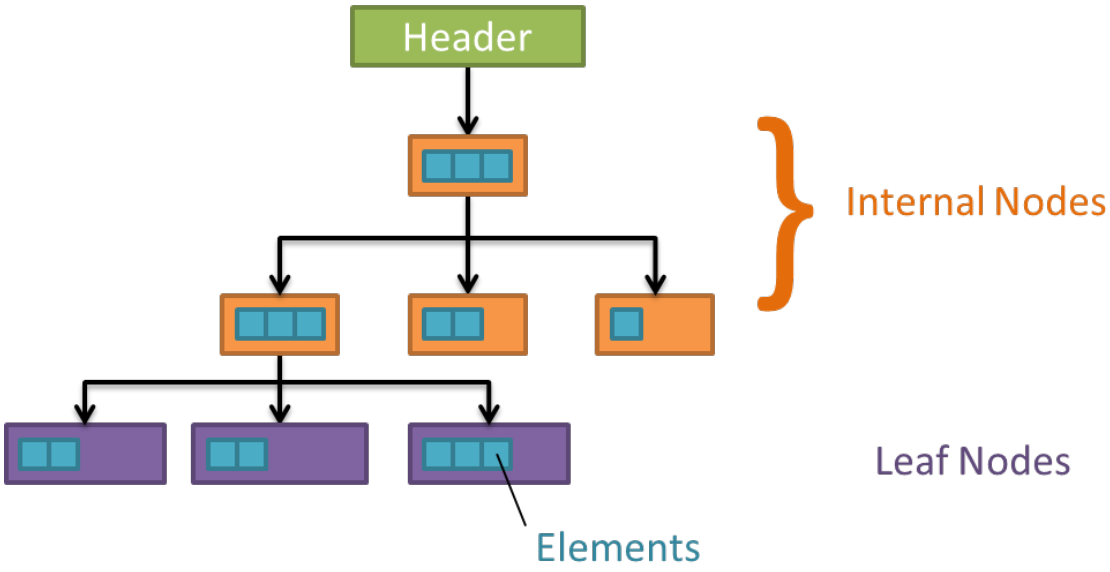

Figure 4 - Version 2 B-tree structure. Not all child nodes are shown to keep the figure uncluttered.

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

Table 3 - Differences between the version 1 and 2 B-trees

Characteristic	version 1	version 2
B-tree sub-type	B^+-tree	Counted 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

[6] http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html

	Symbol table node (when used for indexing links in a group)	Internal Nodes Leaf Nodes
Keys	Offsets of chunk in all dims	Offsets of chunk in all dims
Values	File offset of chunk data Size of chunk Filter mask	File offset of chunk data Chunk size (if filtered) Filter mask (if filtered)
Sibling node pointers?	Yes	No
Data stored in internal nodes?	No	Yes
Records rebalanced on inserts/deletes?	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.

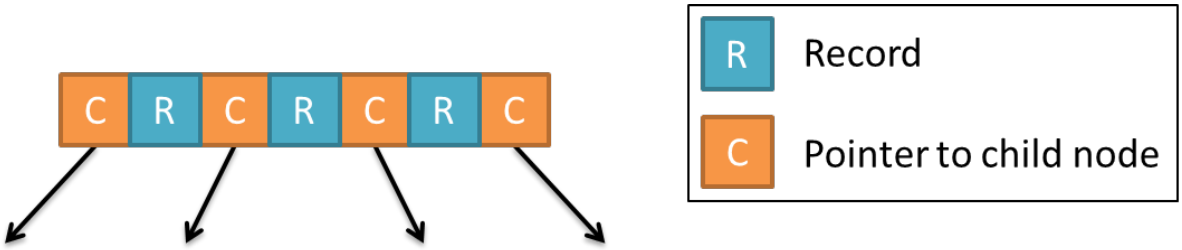

Figure 5 - Records and children in version 2 B-tree nodes.

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 locate the n^{th} record in the structure in O(log N) time.

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 space usage. 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 4: Parent-->Child flush dependencies between version 2 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

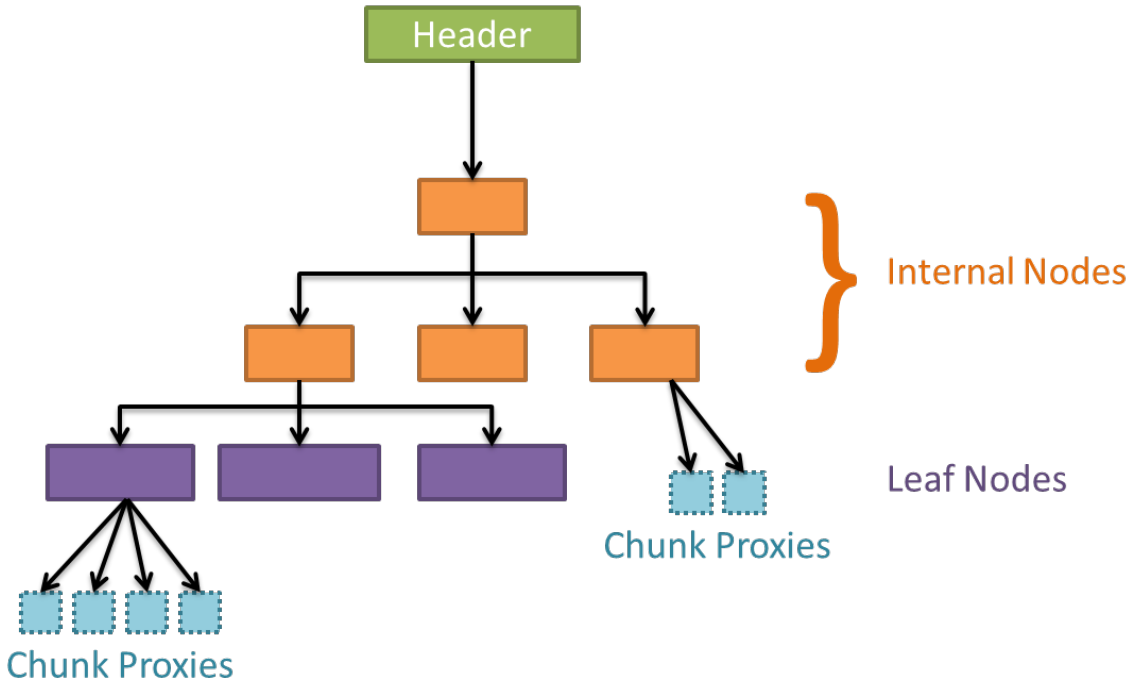

Figure 6 - Version 2 B-tree flush dependencies. Not all child nodes are shown to keep the figure uncluttered. The base of the arrow is the parent and the point is the child.

Chunk Index: 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 dataset's 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 depth 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 14^{th} 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 4153^{rd} element in this super block, which is stored on the 5^{th} 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 4294967295^{th}.

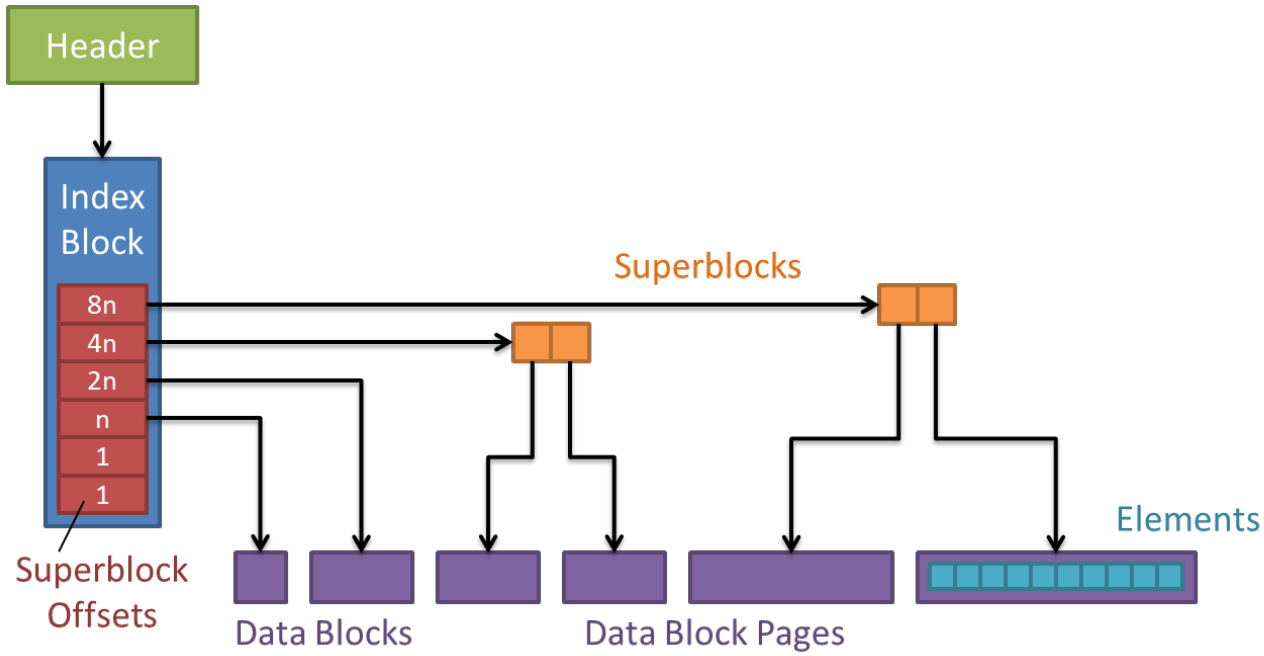

Figure 7 - Extensible array structure.

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 5 and expressed graphically as arrows in figure 6 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.

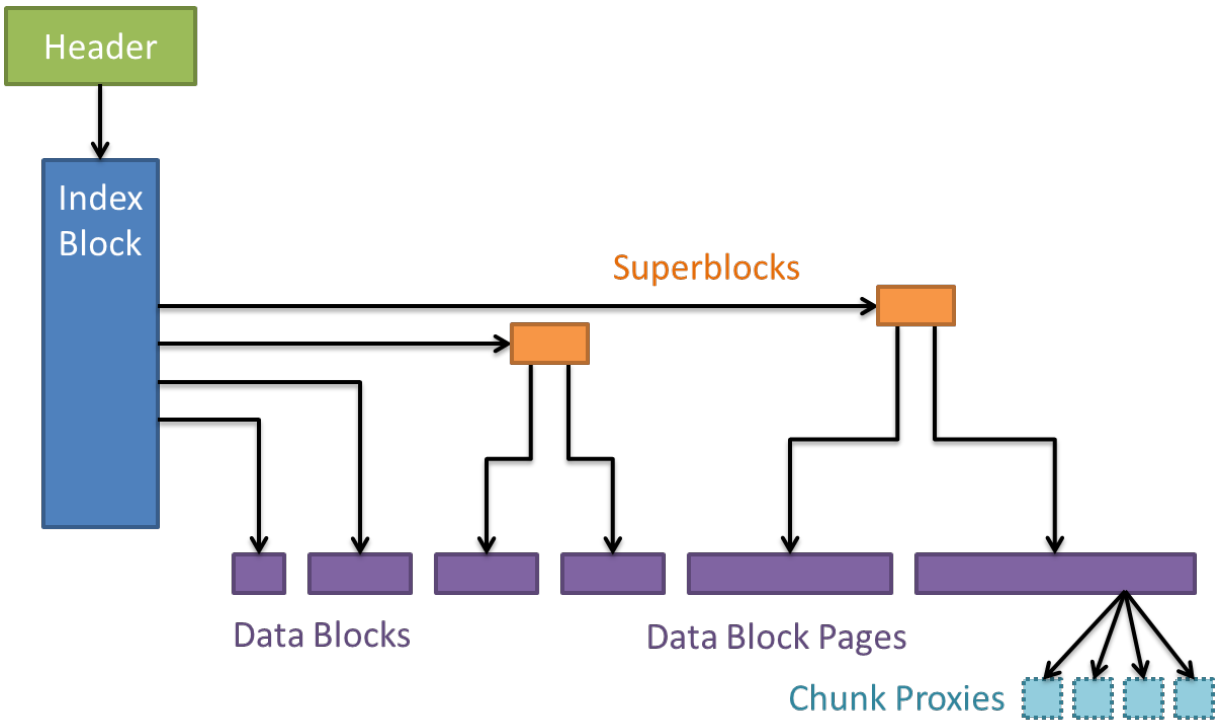

Figure 8 - Extensible array flush dependencies. The base of the arrow is the parent and the point is the child. The figure only shows chunk proxies depending on the data block pages, even though the index block and data blocks are also parents to chunk proxies.

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 5: 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

Chunk Index: Fixed Array

NOTE: The flush dependencies and testing are currently not complete for this data structure in the prototype.

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 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[7] 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[8]. 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.

[7] 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.

[8] 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.

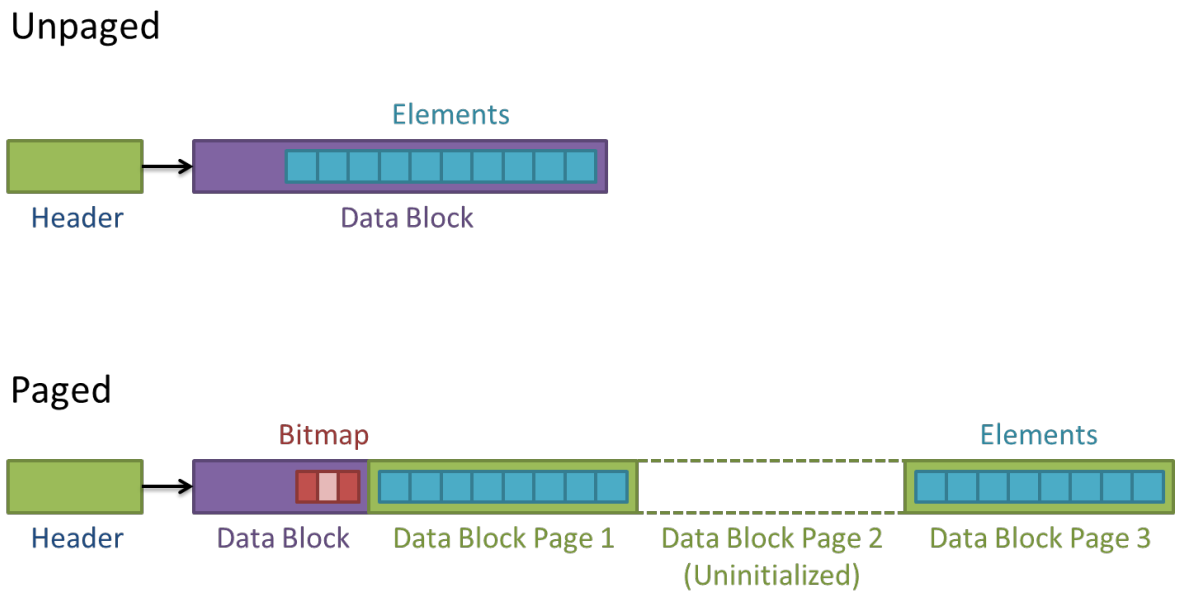

Figure 9: Fixed array structure.

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 6: 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

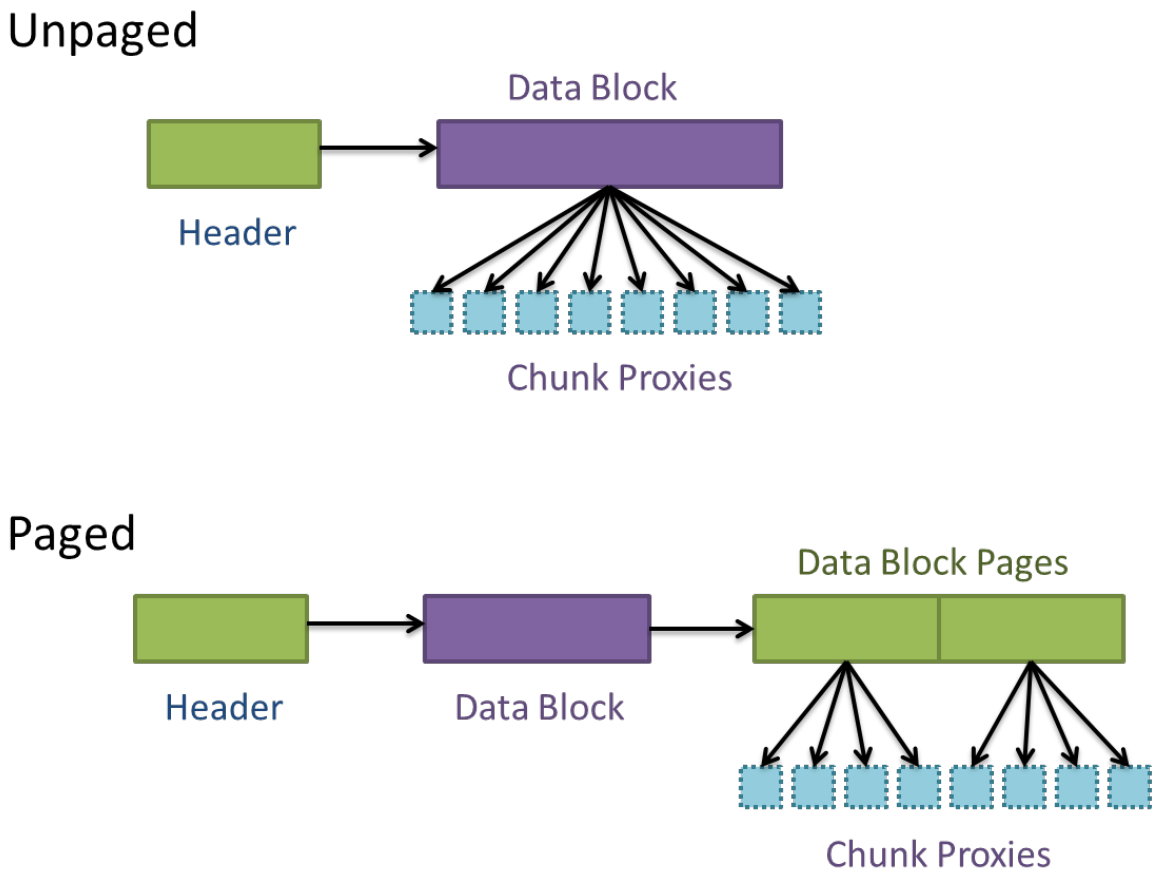

Figure 10: Flush dependencies between fixed array metadata cache objects for both paged and unpaged fixed arrays. The base of the arrow is the parent and the point is the child.

Revision History

June 5, 2013	Version 1 created from prior individual data structure documents.
June 14, 2013	Version 2 added figures, introduction, major text changes. Intro part is very rough and unfinished.
June 25, 2013	Version 3 incorporated feedback from Quincey and Neil.
June 26, 2013	Version 4 rewrote the introduction.
June 30, 2013	Version 5 tidied the figure/table captions and a little cleanup.