Using Skip Lists for Indexing Chunk Locations of HDF5 Datasets with One Unlimited Dimension

Quincey Koziol
The HDF Group
koziol@hdfgroup.org
Version 0.2
August 3rd, 2007

I. This Document's Intended Audience

This document is written for the HDF5 development team, and outside developers who are familiar with HDF5's current data model. Certain commonly used terms in HDF5 are not explained in detail, please see the HDF5 documentation at http://www.hdfgroup.org/HDF5 for more information about HDF5.

II. Current Implementation of Indexing HDF5 Datasets with Unlimited Dimensions

HDF5 datasets with unlimited dimensions are required to be stored in "chunked" form. To retrieve elements in the dataset, the chunk containing those elements must be located and accessed. The locations of chunks for a dataset are stored in a B-tree data structure, which maps the index of the chunk to the file offset where the chunk's elements are stored. This diagram shows an example of a B-tree that maps chunk indices to file offsets for a dataset with one dimension and a chunk size of 30 elements:

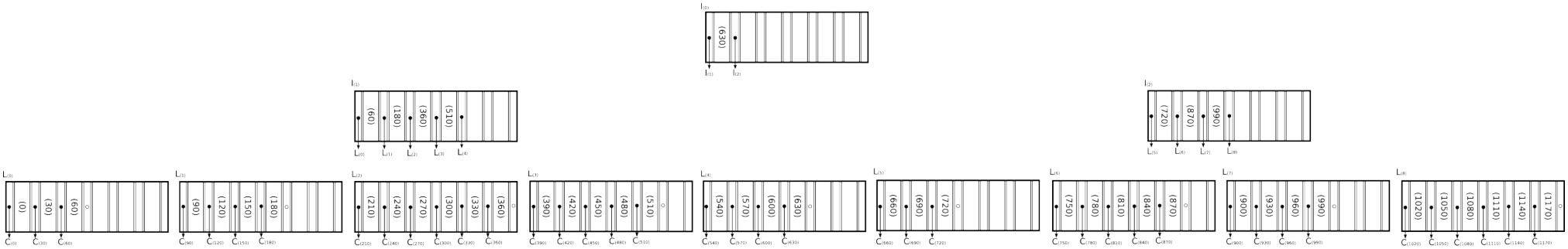

Figure 1

The algorithms for operating on these B-trees follow the normal rules for searching, inserting and deleting records in B-trees. For example, to locate the chunk containing element 108, the following nodes/chunks are accessed: I(0) -> I(1) -> L(1) -> C(90).

III. Drawbacks With Current Implementation

Because HDF5 datasets only allow their dimensions to be increased or decreased at their upper bound, a B-tree used to index chunks for a 1-D dataset will only ever insert or remove records from its right-most node. This negates most of the advantage of using a complicated data structure like a B-tree to index the chunks.

Additionally, for applications which wish to rapidly append records to the dataset, having to traverse and/or update multiple B-tree nodes for each record insertion imposes an additional performance penalty. This is especially a drawback when splitting one or more B-tree nodes is required to accomodate new record for a chunk.

This is illustrated in figure 1 if a new record for chunk index 1200 is inserted in the B-tree. When this occurs, leaf L(8) will need to be split and internal node I(2) will need to be updated with the information for the new leaf that will be added to the B-tree. After enough other new records are added (1230, 1260, etc.), not only will the current right-most leaf node need to be split again, but internal node I(2) will need to be split and the root node I(0) will need to be updated also. Something to note from these insertions (appends, really) is that only the right-most nodes at each level of the tree are involved in the operation, which gives an intuitive sense that the data structure isn't operating to its fullest potential and leads toward the final solution described below.

IV. How to Fix the Problems with B-tree Indices?

This document describes how to replace the B-trees used to index chunks for datasets with only one unlimited dimension with a specialized variation of a skip list. The final skip list solution will be built up in steps below, starting with a "classic" skip list structure and describing each change from the previous step.

V. Classic Skip Lists

The classic form of a skip list is a described as a "probabilistic" data structure that provides an alternative data structure to balanced binary trees and has the same O(log n) lookup, insertion and deletion time for operations. They are described by William Pugh in his paper: "Skip lists: a probabilistic alternative to balanced trees".

Figure 2 shows a classic probabilistic skip list, used to index chunks for a chunked dataset with one unlimited dimension:

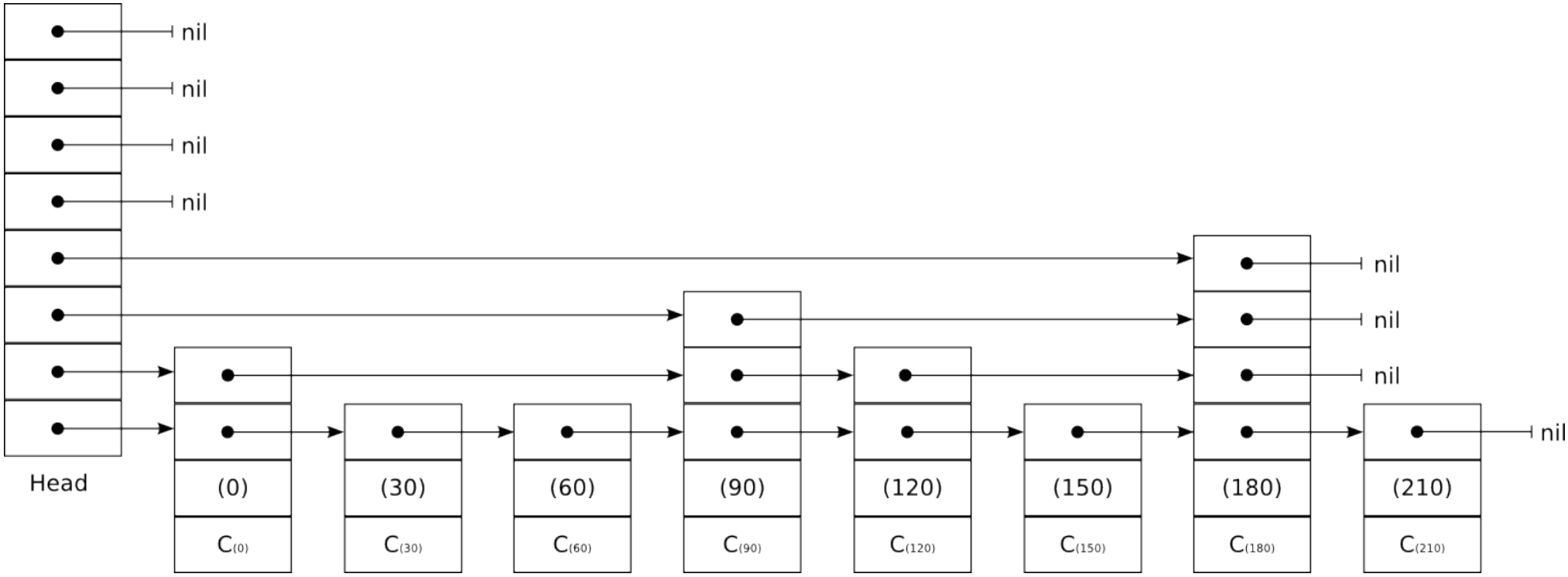

Figure 2

Note that the probabilistic nature of the skip list is shown by the number of "forward" pointers for each node (the "height" of the node) being randomly chosen, according to a power of 2 distribution (as described in the paper referenced above).

VI. Reversing the Skip List Pointers

The first place to improve on the structure described in Figure 2 is the way that internal nodes in the data structure would need to be updated with the address of a new node that is appended to the list of nodes when the dataset's dimension is increased and a new chunk needs to be added to the dataset.

Addressing this problem requires that the internal pointers for the nodes be reversed in direction and the "head" node with forward pointers replaced with a "tail" node with reverse pointers. This is shown in Figure 3:

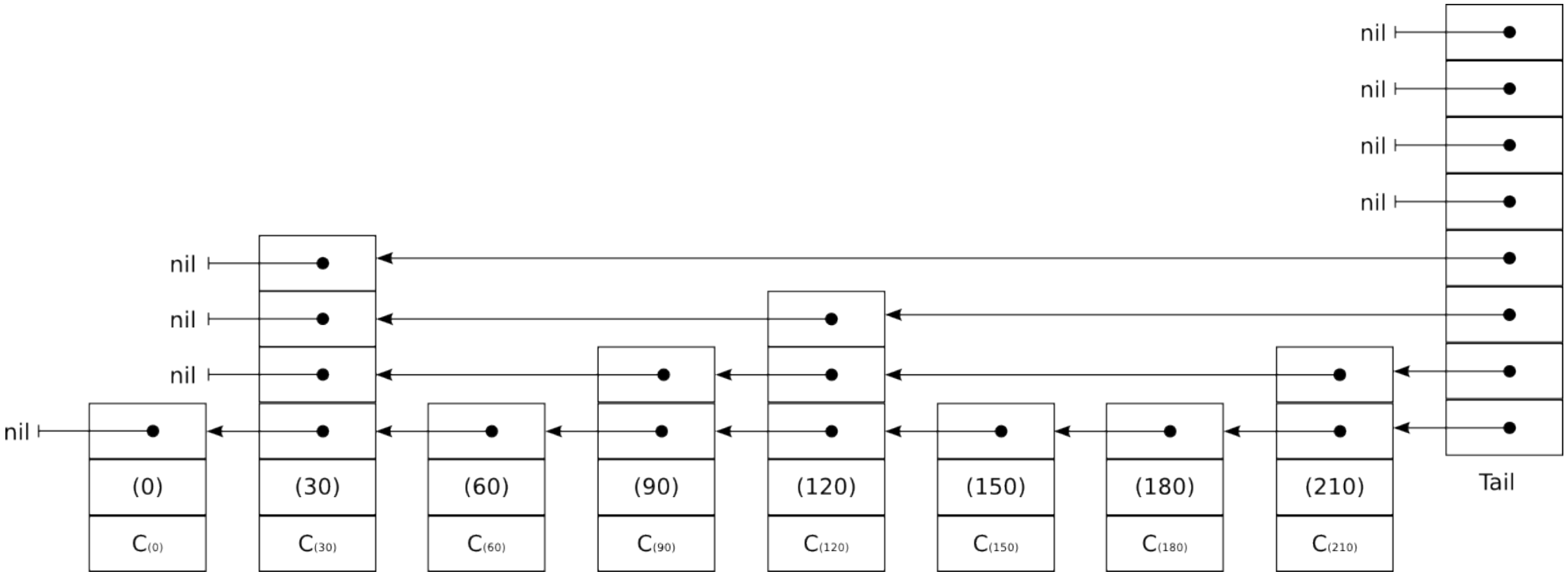

Figure 3

As you can see from Figure 3, when the size of the dimension is increased enough that a new node is appended to the end of "reversed" skip list (just before the "tail" node), only the tail node needs to be updated with information about the new node, and the new node can get the addresses of the internal nodes that it needs from the tail node, without accessing or updating those internal nodes. This eliminates one of the major drawbacks of the B-tree index: modifying multiple internal nodes when adding new records.

VII. Sidebar: Reversed Skip Lists Can Be "Lock-Free" For A Single Writer and Multiple Readers

An important point to note at this step is that a reversed skip list can have a node appended by a single writing process without blocking or confusing reading processes, as long as the tail node can be updated on disk atomically. This can be done simply, as follows:

1. A new skip list node is allocated and initialized with the proper information to point to internal nodes, etc. This has no affect on readers, aside from there being some space that they won't know how to reach.
2. The new node is written to disk. Again, this does not affect readers, since they can't reach the node, due to all copies (in a reader's memory and in the file) of the tail node not referencing it yet.
3. The writer process's in-memory copy of the tail node is updated to point to the new node and a serialized form of the updated tail node is composed in a single, contiguous buffer in memory. This can't affect reading processes because the file hasn't been modified.
4. The serialized buffer for the tail node is written out in a single I/O operation by the writer process. As long as the file system that the HDF5 file is stored on provides an atomic I/O guarantee for single I/O operations (usually the case for POSIX-compliant file systems), any readers that access the tail node will either get a copy of the tail node information before it is updated and thus not "see" the new node while they hold the tail node in memory, or the reader will get a copy of the tail node after the access and "see" the skip list with the new node, which is OK since that node has already been written to disk with correct information.

Additionally, at any point in time, a reader process can re-read the tail node and safely get an updated view of the reversed skip list, since the tail node is guaranteed to always point at existing, initialized, and constant skip list nodes. (Again, assuming POSIX-compliant atomic single I/O operations)

VIII. Making the Reversed Skip List Deterministic

Nodes are never inserted in the middle of the skip list created, so it is very easy to move away from probabilistic heights for nodes and determine the optimal height of a new node to append to the list. The optimal height should be the height that makes the internal node pointers form the equivalent of a balanced binary tree. An optimal deterministic reversed skip list is shown in figure 4:

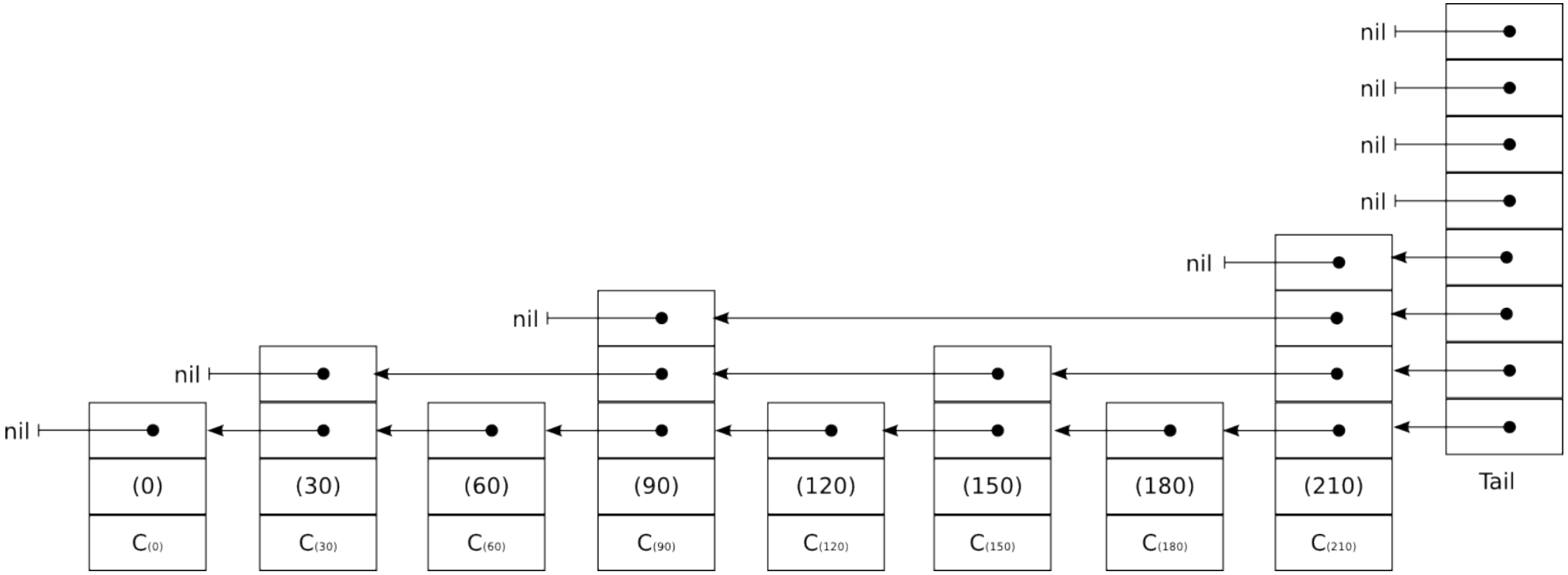

Figure 4

IX. Eliminating "Probing" the Node Keys

The canonical algorithm for locating a node in a skip list involves "probing" the key value of the node pointed to from the current node at the current height of the lookup algorithm's operation, in order to determine if the lookup algorithm should advance to the node probed at the current height, or if the lookup algorithm should "drop down" a level in the current node (when the key from the probed node is further in the list than the key being searched for) and make a probe at that height.

However, if the chunk size is fixed and all the skip list nodes needed to cover the current dimension size are present, we can compute the nodes present in the skip list and their "implied" chunk index, based on the dimension size of the dataset. Therefore, the chunk index values do not need to be stored in

each node and the nodes don't need to be "probed" - the precise sequence of nodes to look up a particular chunk's address can be computed, based on the dimension size. Creating all the nodes to cover the current dimension size is also a requirement for maintaining the deterministic form of the skip list, since the node heights are pre-determined by their location in the skip list.

This change requires keeping the total number of nodes in the tail node, but overall makes the nodes + tail node total size smaller (although that's a fairly minimal aspect of this change). This "implicitly keyed" form of a deterministic reverse skip list, shown in figure 5:

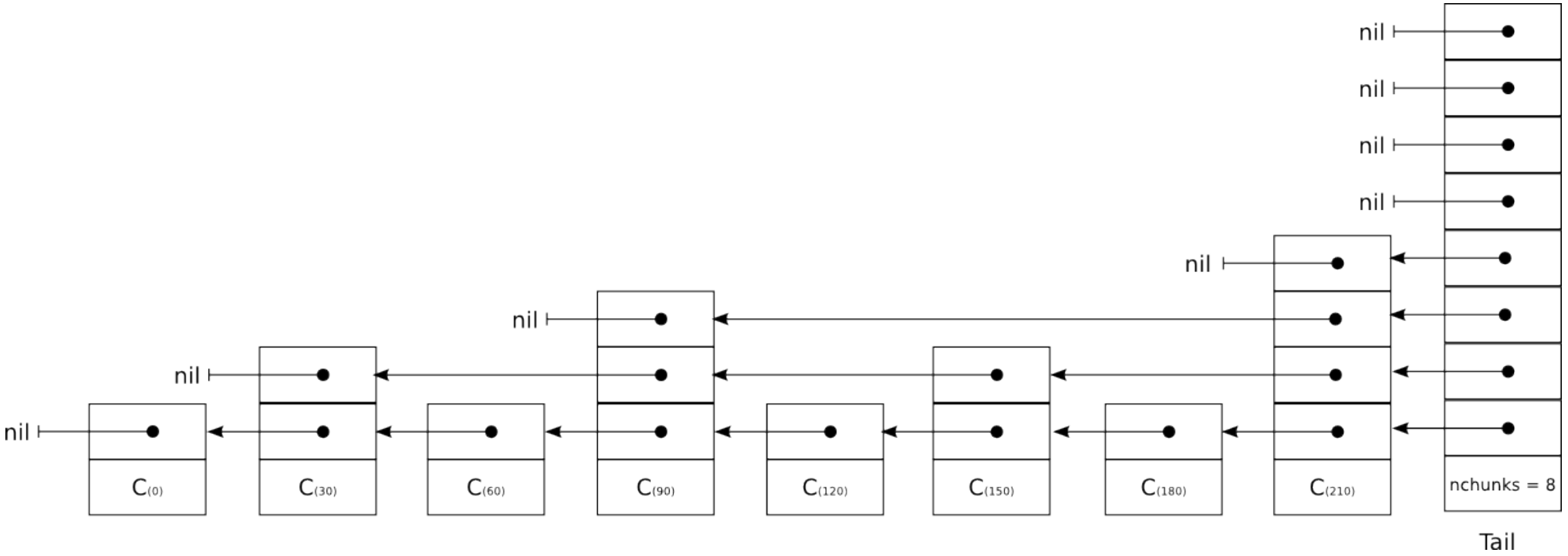

Figure 5

This fits well with the way HDF5 datasets can be operated on, since the HDF5 library guarantees that chunk sizes are fixed and only modifies the upper bound of the dimension size. This will require that the HDF5 library immediately create skip list nodes (but not necessarily allocating raw data storage for the chunks themselves) when the dataset's dimension is extended however.

Requiring that the nodes be created for chunks that don't exist yet seems a small price to pay for the other benefits gained to this point (and is mitigated by the "supernode" idea in the next section), although it does incur a possible performance and size penalty. This should be monitored by the application developer, especially if the chunk size is small and the dimension is extended to be very large without writing data to the elements covered. However, the primary use case for datasets with one unlimited dimension is for applications to extend the dimension size by exactly the number of elements to add to the dataset and then immediately write those elements, not for applications to extend the dimension by a large amount and then write data to only a small number of the elements that are between the previous and current values of the dimension size.

Because the space for storing the chunk doesn't need to be allocated yet (as long as the address of the chunk in nodes which are "unwritten" is set to a known invalid value) there should be minimal space overhead in the file. Later, when a chunk is allocated and data is written to that chunk, the node an be updated with the address of the new chunk. Since this update can be a single atomic I/O operation on the node, the single writer/multiple reader "lock-free" status is preserved still: either a reading process gets a copy of the node with the invalid chunk address and can't "see" the data for the chunk (and will assume it contains the "fill value" for the dataset), or it retrieves a copy of the node after the chunk address was updated to point to a valid, initialized chunk of raw data and has access to the elements written. (Again, this is not the primary use case).

X. Sidebar: Implementation Note

It's implied that the application developer would not need to instruct the HDF5 library when to use a B-tree and when to use a skip list. The HDF5 library would automatically choose to use a skip list index when the dataset has only one unlimited dimension and would use a B-tree for datasets with more than one unlimited dimension (which require insertions, not just appends).

However, creating the nodes for unwritten chunks make the data structure "semi-sparse" - the nodes to reference the chunks are allocated and initialized, but the storage for the actual chunk data isn't allocated yet. This is different from the B-tree structure currently used to index HDF5 datasets, which allows for "fully sparse" structures - both B-tree nodes that aren't needed yet and chunks that aren't written to are unallocated in the file. It's possible that applications which need fully sparse indexing for datasets with one unlimited dimension will need to have a mechanism for overriding the default choice of the HDF5 library to use a skip list index and instead request that a B-tree index be used for their datasets, but since this isn't the primary use case, implementing this optional behavior will be deferred until requested.

XI. Combining Small Nodes into "Supernodes"

As described up to this point, each node in the skip list is very small, probably on the order of tens of bytes. Performing such small I/O operations to access those nodes will be very inefficient, especially on parallel file systems, which are oriented toward much larger I/O operations. To correct this problem, we combine multiple small nodes into a single larger "supernode", which contains an array of chunk addresses along with the pointers to other skip list supernodes. This is shown in figure 6, with more chunks added to the dataset, to show the supernode structure links more effectively:

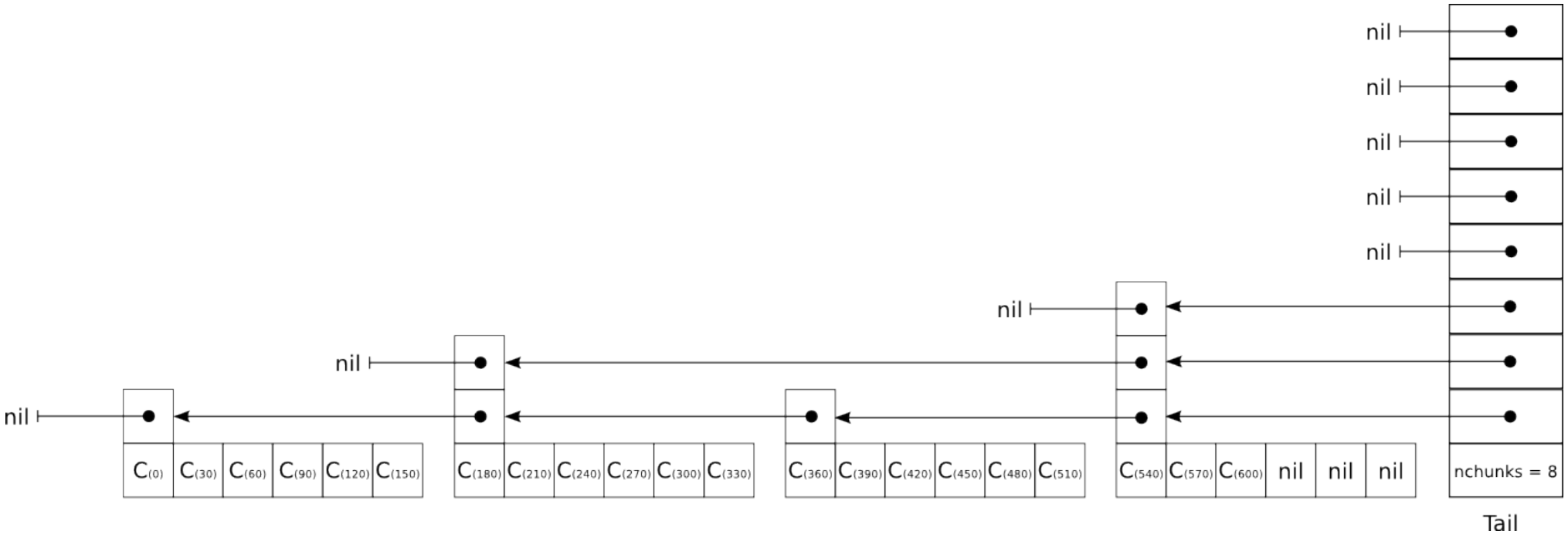

Figure 6

The supernodes refer to each other with the implicitly-keyed, deterministic reverse skip list structure described above, but internally just store a sorted array of chunk addresses that can be accessed in O(1) time, thus they have a similar conceptual structure to the nodes in a B-tree. The size of all the supernodes is constant and should be able to be set by the application developer (again, similar to the way the HDF5 library allows the size of a B-tree's nodes to be chosen by the application developer).

As before, the single writer/multiple reader aspect of the data structure is maintained when defining a new chunk address inside an existing supernode, according to the following algorithm:

- When the size of the dimension is increased and a new chunk is added to the end of the skip list, the chunk is allocated (and raw data is written to it) and the new chunk address is added to the last supernode in the writer's memory.

- Then, the last supernode is serialized into a buffer and written out in a single I/O operation. The reader processes may or may not get the new supernode when they access the skip list, but since the number of chunks in the tail node has not been updated, readers will not attempt to access the newly allocated chunk in the supernode (making the "nil" values in the supernodes redundant, technically).
- After the supernode update has completed, the tail node is updated, serialized and written to the file in a single I/O operation, as before, allowing readers to "see" the new chunk because the number of chunks has been increased.

If it is necessary to allocate a new supernode, the set of operations is identical to the procedure described earlier for appending a new "small" node to the reverse skip list.

Note that it takes the same number of I/O accesses to update the skip list when using supernodes instead of smaller nodes: 2 - one to update the "small" node/supernode and one to update the tail node. (Technically, there's 3 I/O accesses in both schemes if you include the I/O access for writing the raw data to the chunk).

XII. Use Ghost Supernode to Allow Forward Iteration Without Updating Existing Supernodes

Although the variation of a skip list described to this point is very efficient when appending elements to the dataset (taking O(1) time) and retains O(log n) time for lookups as well, it is not very efficient for accessing all the elements in the dataset from the lowest value of the dimension index to the largest (i.e. a forward iteration over the elements). With the scheme described up to this point, a forward iteration operates in ~O(log n) time, due to the lack of forward pointers in the supernodes.

Having a forward pointer between supernodes would seem to imply that a previously written supernode would need to be updated with the address of a newly created supernode, violating the goal that existing supernodes are constant and also creating an additional I/O operation. However, if we create an extra, unused, "ghost" supernode for each skip list, we can use the following algorithm to set the forward pointers in supernodes without requiring modifying existing supernodes or performing extra I/O operations:

- When a new supernode needs to be created, use the space reserved by the current "ghost" supernode for the new supernode.
- Allocate space in the file for a new "ghost" supernode, with the height of its reverse pointers set appropriately for it's future position in the reverse deterministic skip list.
- Set the forward pointer in the new supernode to the address of the newly allocated "ghost" supernode.
- Update the "ghost" supernode pointer in the tail node with the address of the newly allocated "ghost" supernode.

Then, when the new "used" supernode is serialized and written to the file, it will already have the correct address of the next supernode in the skip list and won't need to be updated if/when that "ghost" supernode is used to store chunk addresses. Forward iterations through the list of supernodes in the skip list will stop when a supernode's forward pointer equals the address of the "ghost" supernode from the tail node (as opposed to stopping when the "nil" address value is reached, as is the case for reverse iteration).

The address of the first supernode in the reverse skip list and the address of the ghost supernode will need to be stored in the tail node. All other operations for operating on chunk addresses, supernodes and the tail node are performed as described previously. Note that the newly allocated "ghost" supernode does not need to have any data written to it, its space is just maintained in the bookkeeping of the skip list.

Adding a "ghost" supernode to the skip list is shown in figure 7:

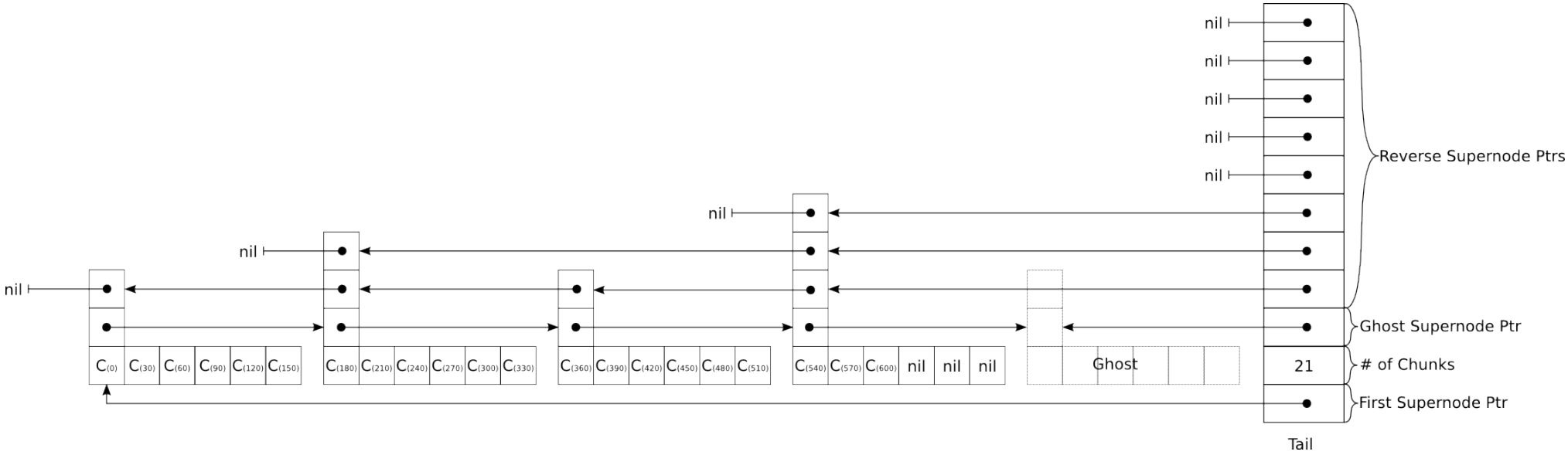

Figure 7

XIII. Maintain Pointers to Right-Most Supernodes in the Tail Node

In figure 7, it's obvious that the reverse pointers in the tail node store redundant pointers to the last "real" supernode in the skip list (of height 3). It would be advantageous to allow the tail node to point to the right-most supernode of a given height instead of allowing a higher internal supernode to occlude those previous nodes. This would allow some internal supernodes to be reached more quickly than traversing to the occluding supernode and then determining the address of the desired supernode.

Having the tail node's reverse pointers retain the address of the right-most nodes is shown in figure 8:

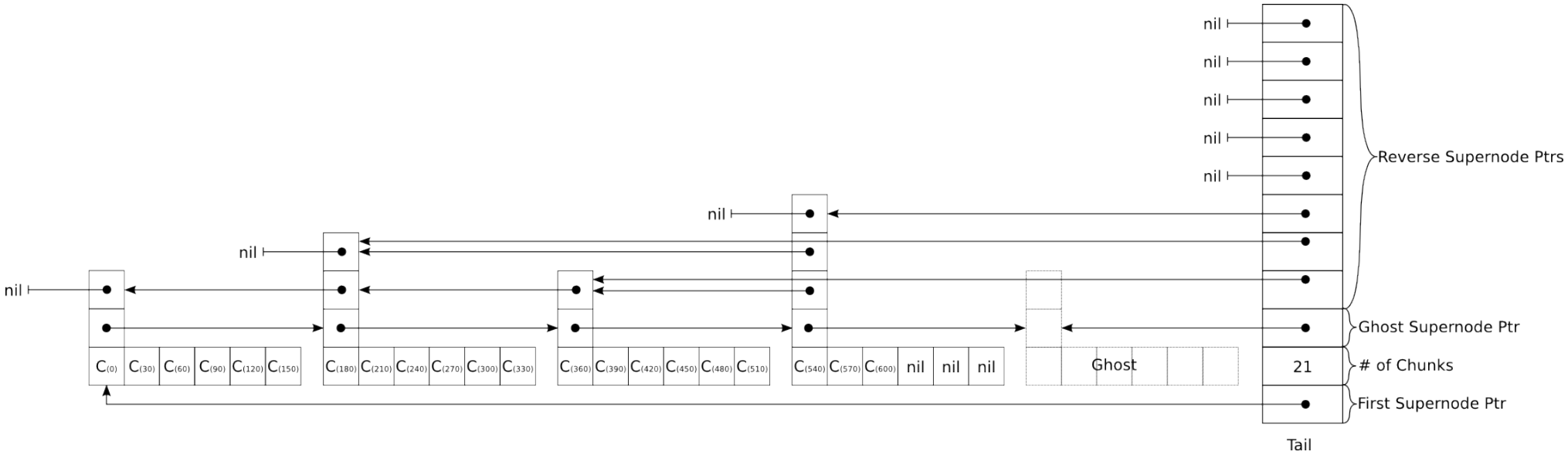

Figure 8

Figure 8 still retains redundant pointers to the supernodes from the tail node, but the internal pointers from other supernodes allow the lookup operation in the future when the dataset's dimension size is increased and must remain. Which pointer values in the tail node are redundant is dynamic and the redundant ones are also modified over time to point at various internal supernodes and can't be removed either. So, the currently duplicated pointers to other supernodes in the data stucture are now required, in one way or another.

This change to the data structure means that it's always possible to reach the right-most node of a given height in the skip list. It may be useful to re-introduce a "head" node into the data structure, with pointers to the left-most node at each level of the skip list, in order to reduce the number of supernode I/O operations when performing a lookup operation. However, this would require extra I/O operations for likely a minimal gain and is not pursued further at this time. Possibly if the unlimited dimension was converted to a fixed dimension at a certain point, it would be worthwhile to add a head node at that point.

XIV. Eliminate Top-most Reverse Pointer in Supernodes with Height >1

It can be seen (in figures 5 & 8, for example) that the top-most reverse pointer in supernodes of height >1 is either the 'nil' value or is redundant (a previous, higher supernode would be chosen to reach the supernode required for lookup operations) and can be eliminated. The single reverse pointer in supernodes of height 1 is also redundant for lookups, but allows efficient reverse iteration and seems to be worthwhile to leave in the data structure.

This doesn't affect any of the operations on the skip list, but would either allow more chunk addresses to be stored in a supernode (if the supernode size was held constant) or supernode sizes to be smaller (if the number of chunk addresses was held constant).

Eliminating the top-most reverse pointer in supernodes whose height is >1 is shown in figure 9:

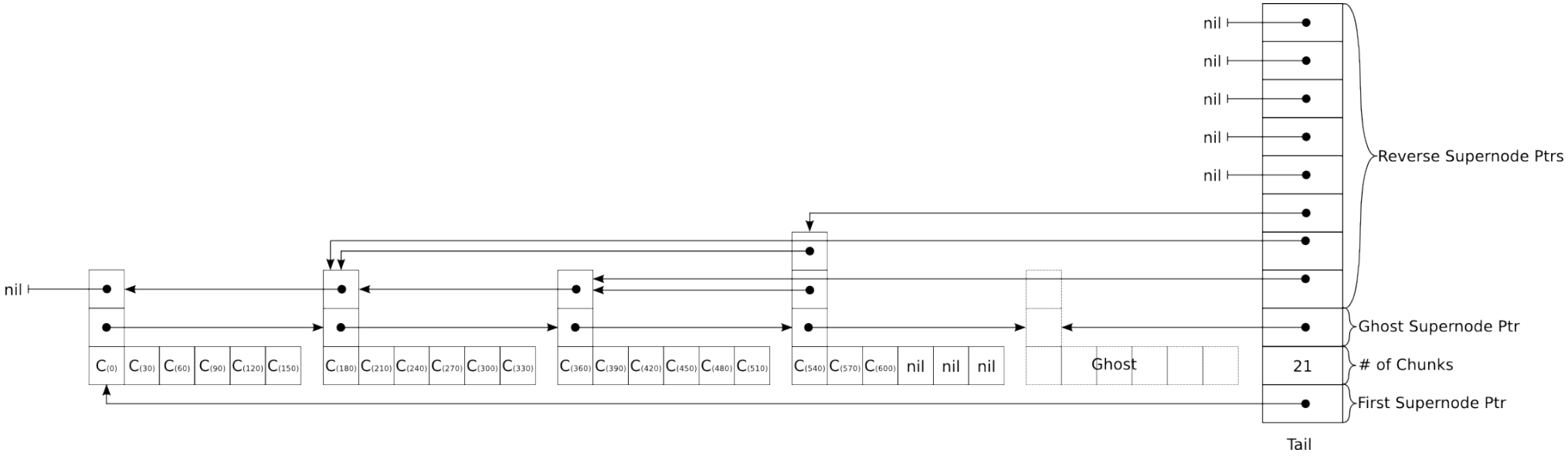

Figure 9

XV. Add Configuration Node for Constant Data Structure Values

The final change to the skip list structure described here is to add a "configuration node" to the data structure (this could have been added to the skip list in one of the earlier steps as well). There are several constant values associated with the skip list data structure described here and the configuration node is added to hold this information. It holds a pointer to the tail node along with the height of the tail list node, the number of chunk addresses per

supernode, the size of the chunks, etc. The pointer to the first supernode is also moved from the tail node to the configuration node, since it is constant also.

This is shown in figure 10:

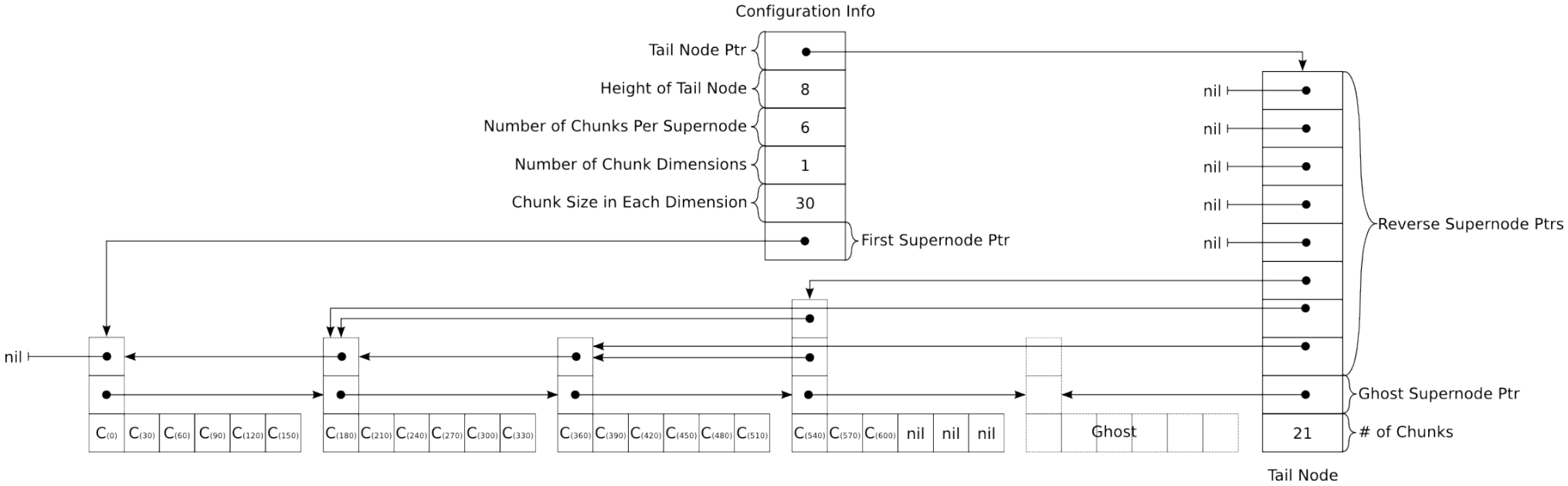

Figure 10

The configuration node information will be stored in an HDF5 dataset's object header as a new form of the "layout" message. The tail node and supernodes will be added to the HDF5 file format as "infrastructure components", similar to the existing B-trees and heaps used in the file.

XVI. Sidebar: Indexing of Chunks for Datasets with >1 Dimension, But Only 1 Unlimited Dimension

Although all the figures and text in this document describe one dimensional datasets, this is done just for expositional simplicity and it's important to note that all the work described here can be applied to datasets with >1 dimensions, as long as there is only one *unlimited* dimension for the dataset.

However, when the chunk size of non-unlimited dimensions is smaller than the maximum size of that dimension, it is necessary to store pointers to more than one chunk for each increment of the chunk size in the unlimited dimension.

For example, defining a 3-D dataset with two fixed-size dimensions and one unlimited dimension would allow appending fixed-size 2-D "slices" along the unlimited dimension of the dataset. Figure 11 shows a 3-D dataset with chunk sizes of (30, 25, 40), maximum dimension sizes of (unlimited, 50, 80) and current dimension sizes of (450, 50, 80):

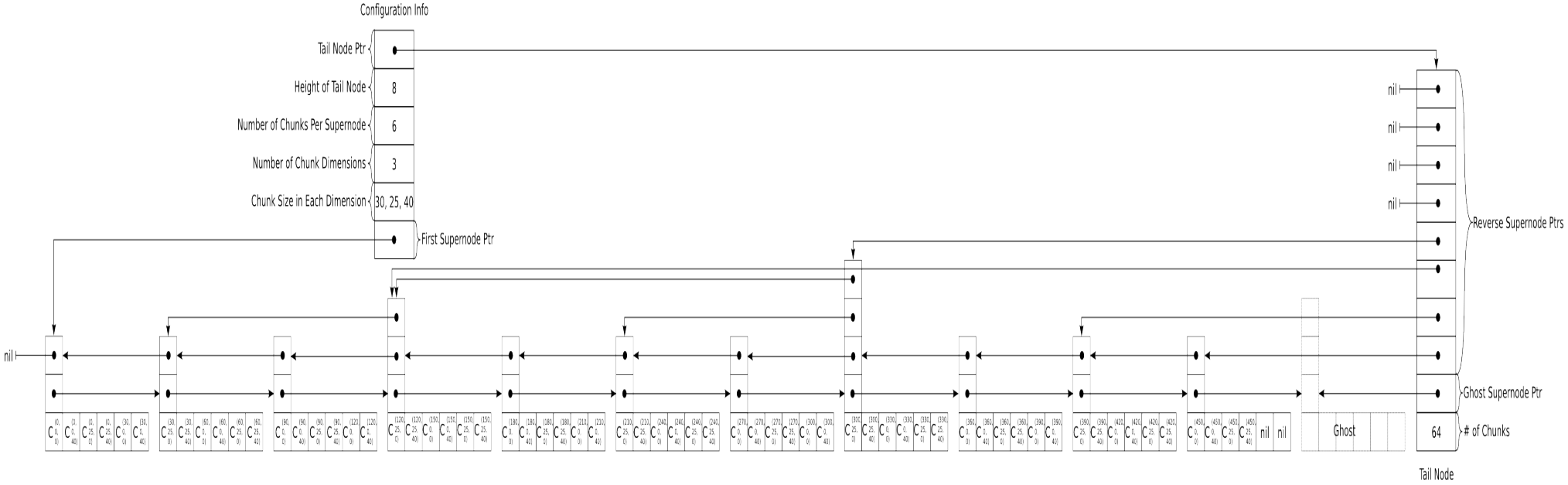

Figure 11

It should be noted that room for all the chunk addresses needed for fixed-size dimensions (if there is more than one chunk needed in those dimensions) must be allocated for each chunk increment in the unlimited dimension, to avoid inserting chunk addresses into the skip list later. This is shown in figure 11 with groups of four chunks in the skip list (two in each fixed-size dimension) for each chunk increment in the unlimited dimension.

In practice, this will probably not make much difference, since it's very unlikely that the "slices" of elements appended to the unlimited dimension would be sparsely populated.