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

Quincey Koziol
The HDF Group
koziol@hdfgroup.org
Version 0.3
August 16rd, 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:

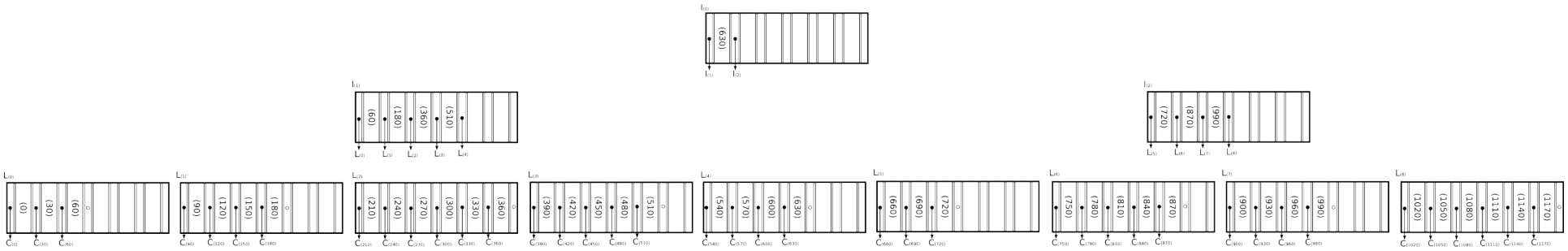

Figure 1

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). (This assumes that C(0) is the address of the chunk with elements 0-29, C(30) is the address of the chunk with elements 30-59, etc.)

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:

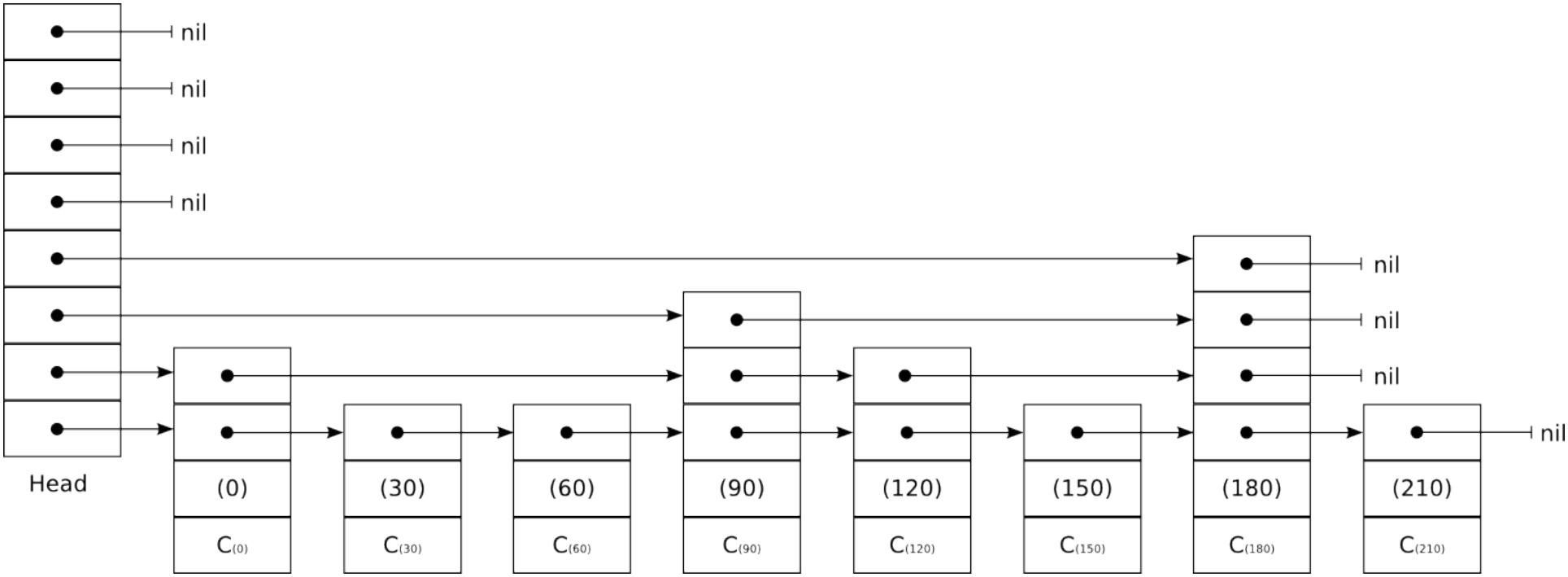

Figure 2

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:

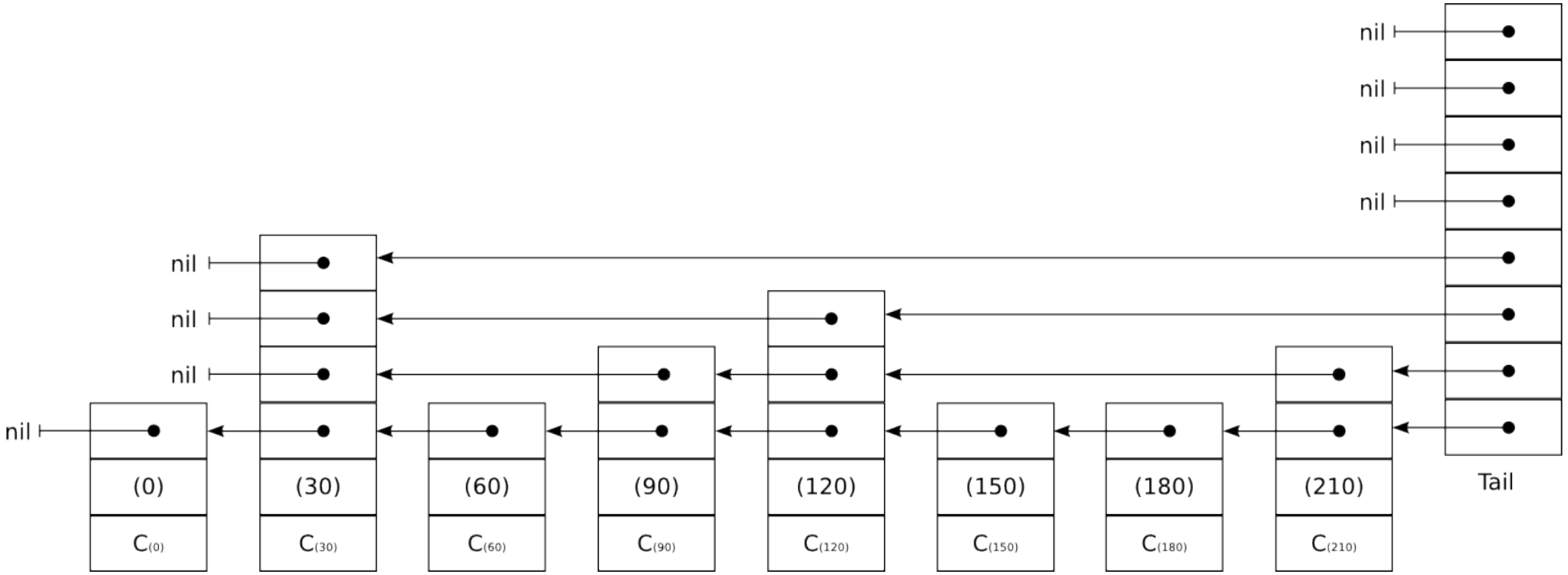

Figure 3

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:

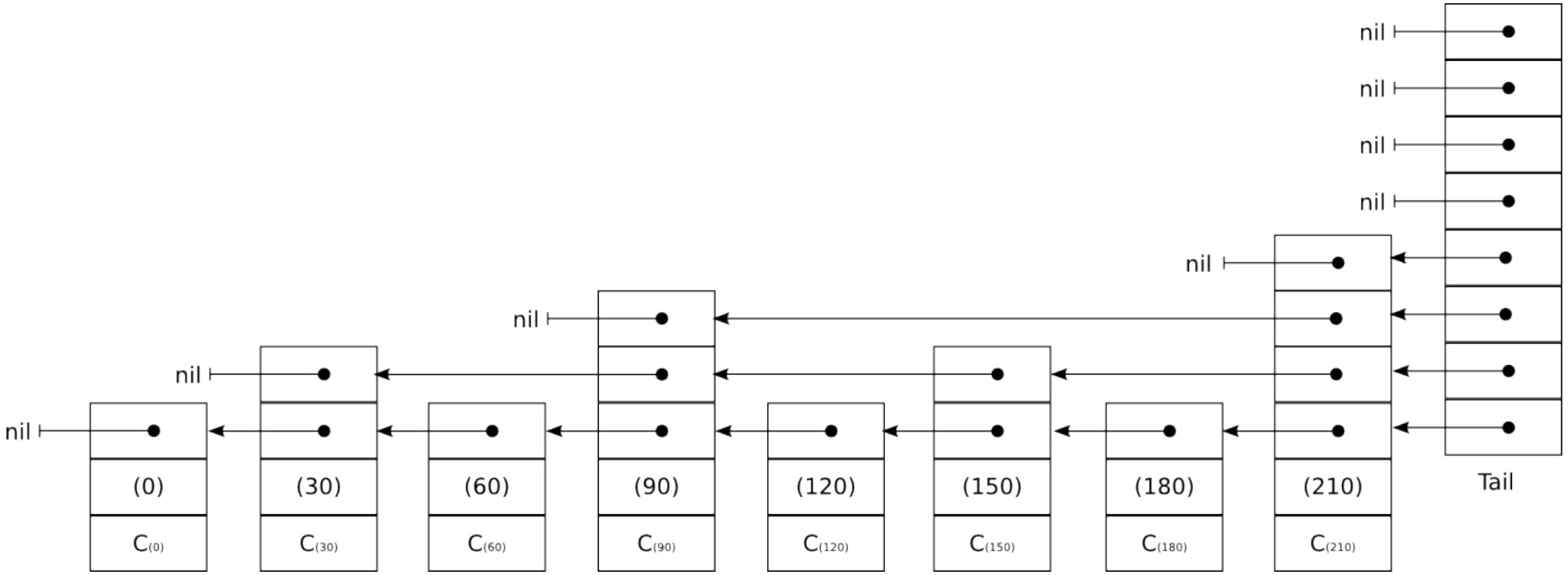

Figure 4

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:

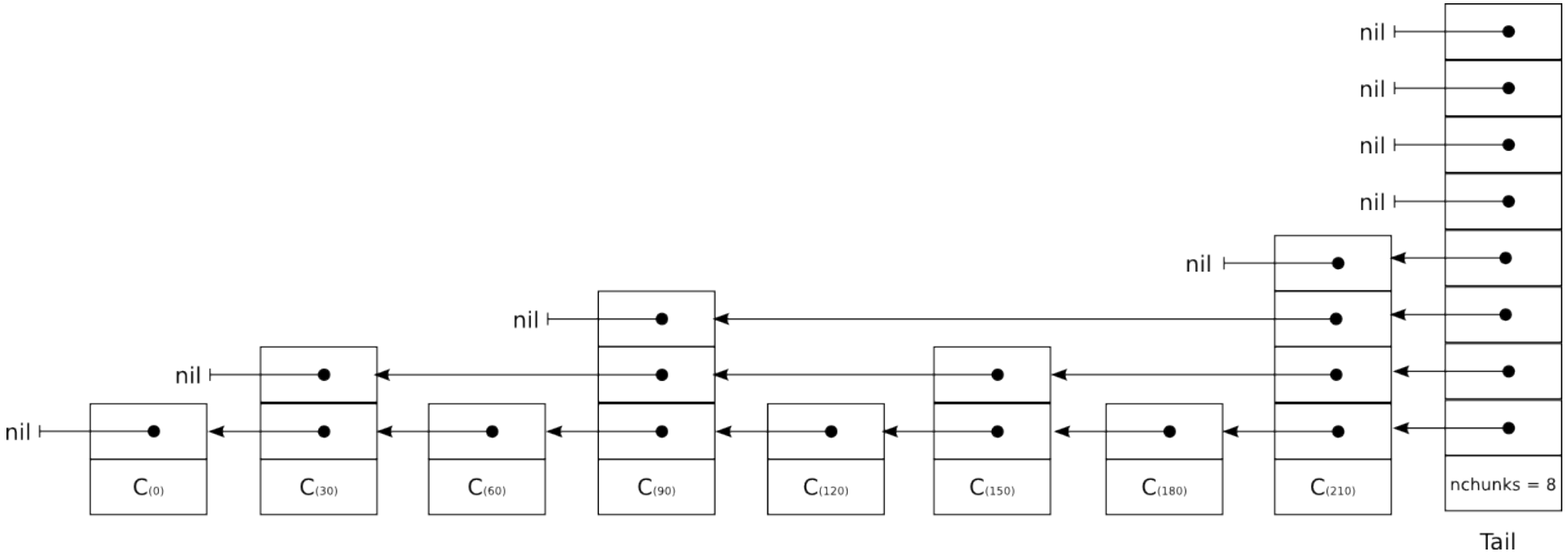

Figure 5

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:

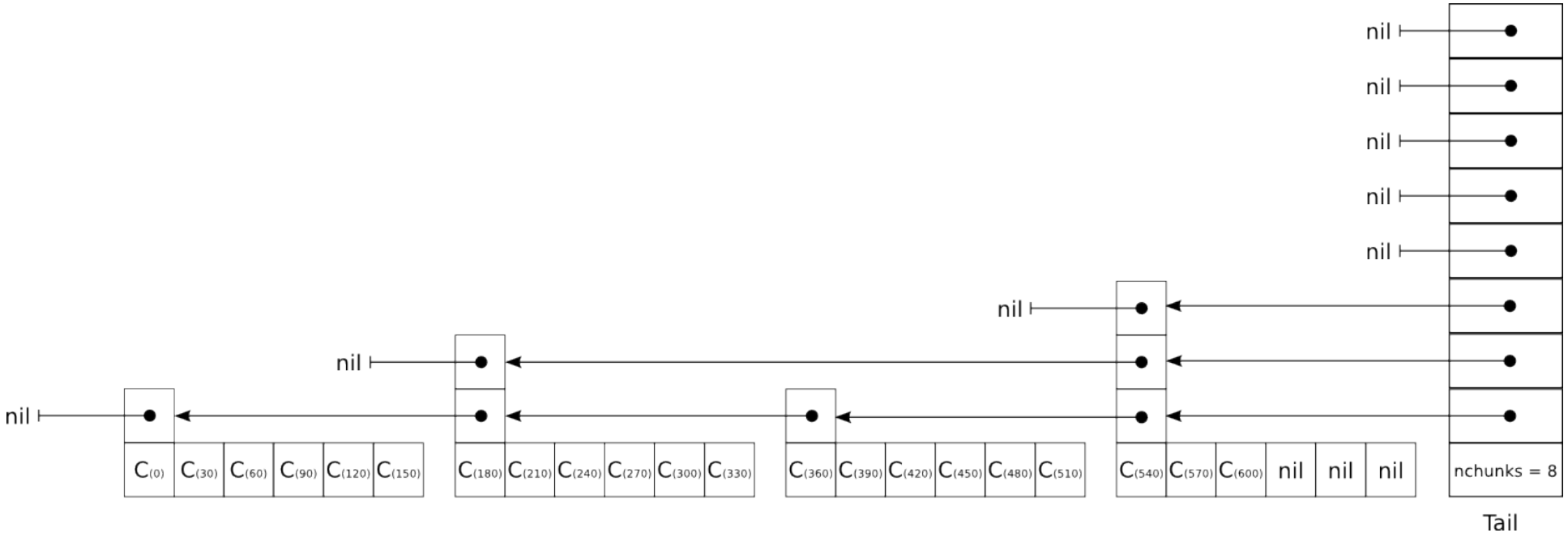

Figure 6

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:

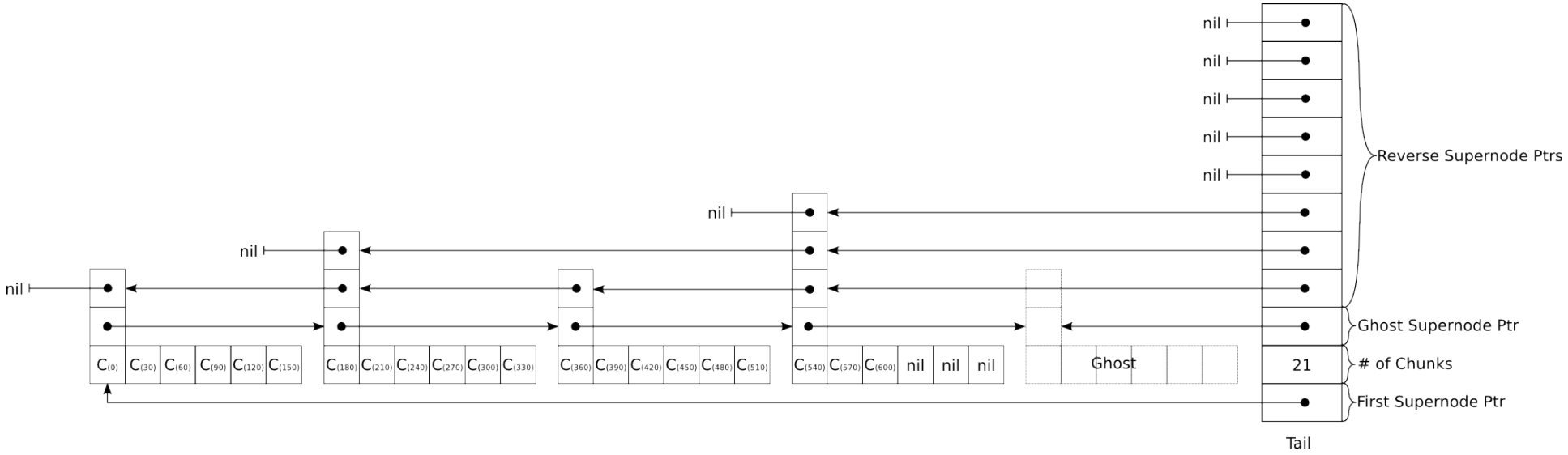

Figure 7

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:

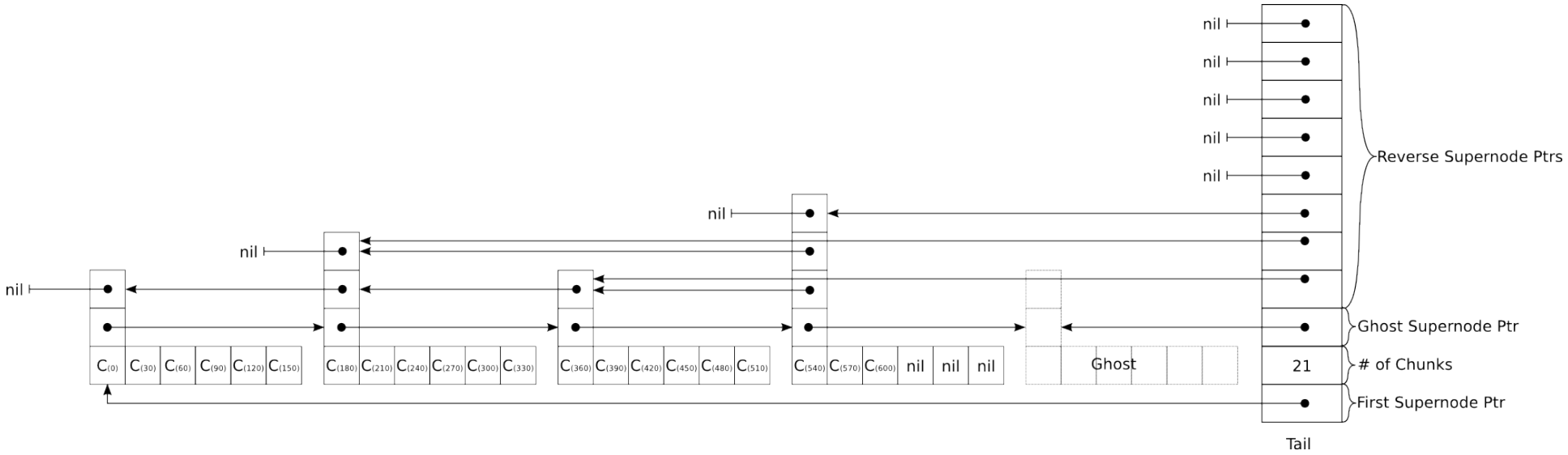

Figure 8

Figure 8 still retains redundant pointers to the supernodes, 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. Different pointer values in the tail node are redundant at different times and they are also modified to point at various internal supernodes that are the right-most supernode of that height in the skip list. Therefore the duplicated tail node pointers 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.

XIV. Maintain Pointers to Left-Most Supernodes in the Tail Node

The tail node shown in figure 8 maintains a pointer to the right-most supernode in the skip list, speeding up access to the supernodes that are to the "right" of the tallest supernode in the skip list. It would obviously be advantageous to maintain pointers to the left-most supernodes, to speed up access to supernodes on the "left" side of the highest supernode. These pointers must be added to the tail node, in order to maintain lock-free appends to the skip list for a single writer.

Retaining pointers to the left-most supernodes in the tail node means that the "first supernode pointer" in the tail node can be removed, since it duplicates the pointer to the left-most supernode at level 1 in the tail node. These changesare shown in Figure 9:

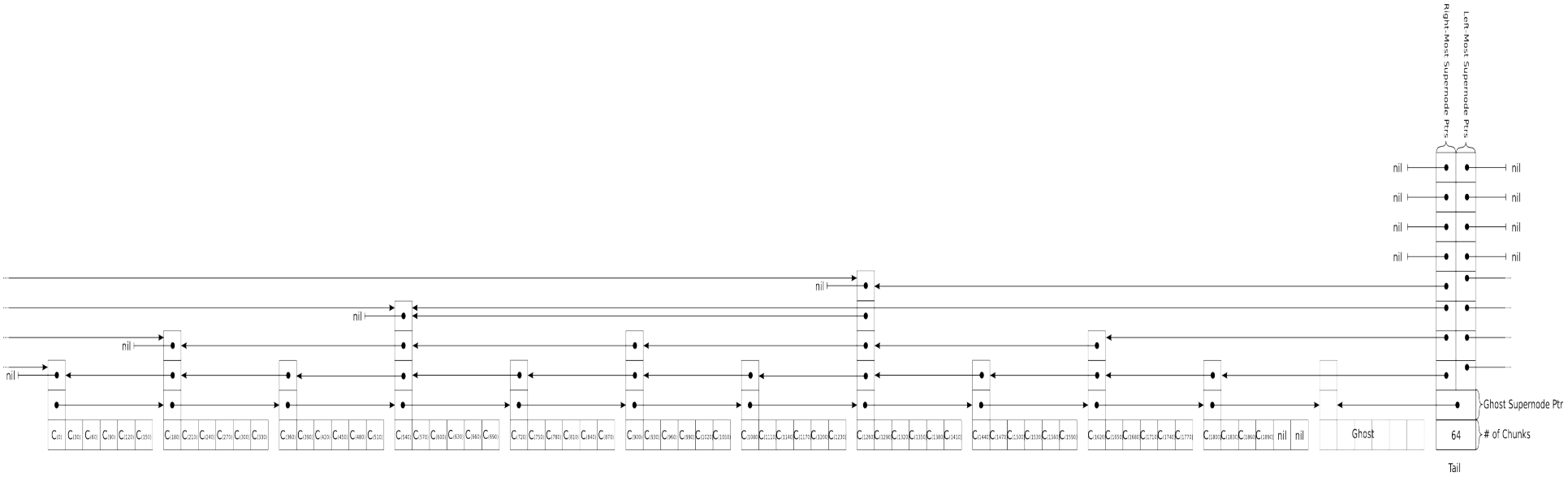

Figure 9

The number of chunks in the dataset has been increased in figure 9, to better show the right & left-most supernode pointers from the tail node.

XV. Add Forward Pointers in Supernodes

Adding pointers from the tail node to the left-most supernodes at each height in the skip list reduced the average number of I/O accesses to supernodes in the "left half" of the skip list (those supernodes before the supernode with maximum height in the skip list). If additional "forward" pointers are added to each supernode, the average number of I/O accesses can be reduced even further by choosing more efficient paths to some of the supernodes (without increasing the path to any other supernodes).

For example, in figure 9, the supernode beginning with chunk index 720 would take 3 I/O accesses to reach if only left and right-most pointers from the tail node and reverse pointers in the supernodes are used, with the following supernodes accessed: C(1260) -> C(900) -> C(720). However, if forward pointers are also stored in each supernode, only 2 I/O accesses are required to reach the C(720) supernode, with the following supernodes accessed: C(540) -> C(720) (this can be seen in figure 11).

Table 1 and figure 10 shows the average number of I/O accesses for a balanced binary tree as compared to the skip lists described here, with right-most pointers only, left and right-most pointers, and left and right-most pointers combined with "inner" pointers:

Skip List Height	Binary Tree	Right-Most Pointers Only	Left and Right-Most Pointers	Inner Pointers
5	4.161290323	2.580645161	2.096774194	1.838709677
6	5.079365079	3.063492063	2.555555556	2.142857143
7	6.05511811	3.527559055	3.031496063	2.464566929
8	7.031372549	4.015686275	3.517647059	2.788235294

Table 1

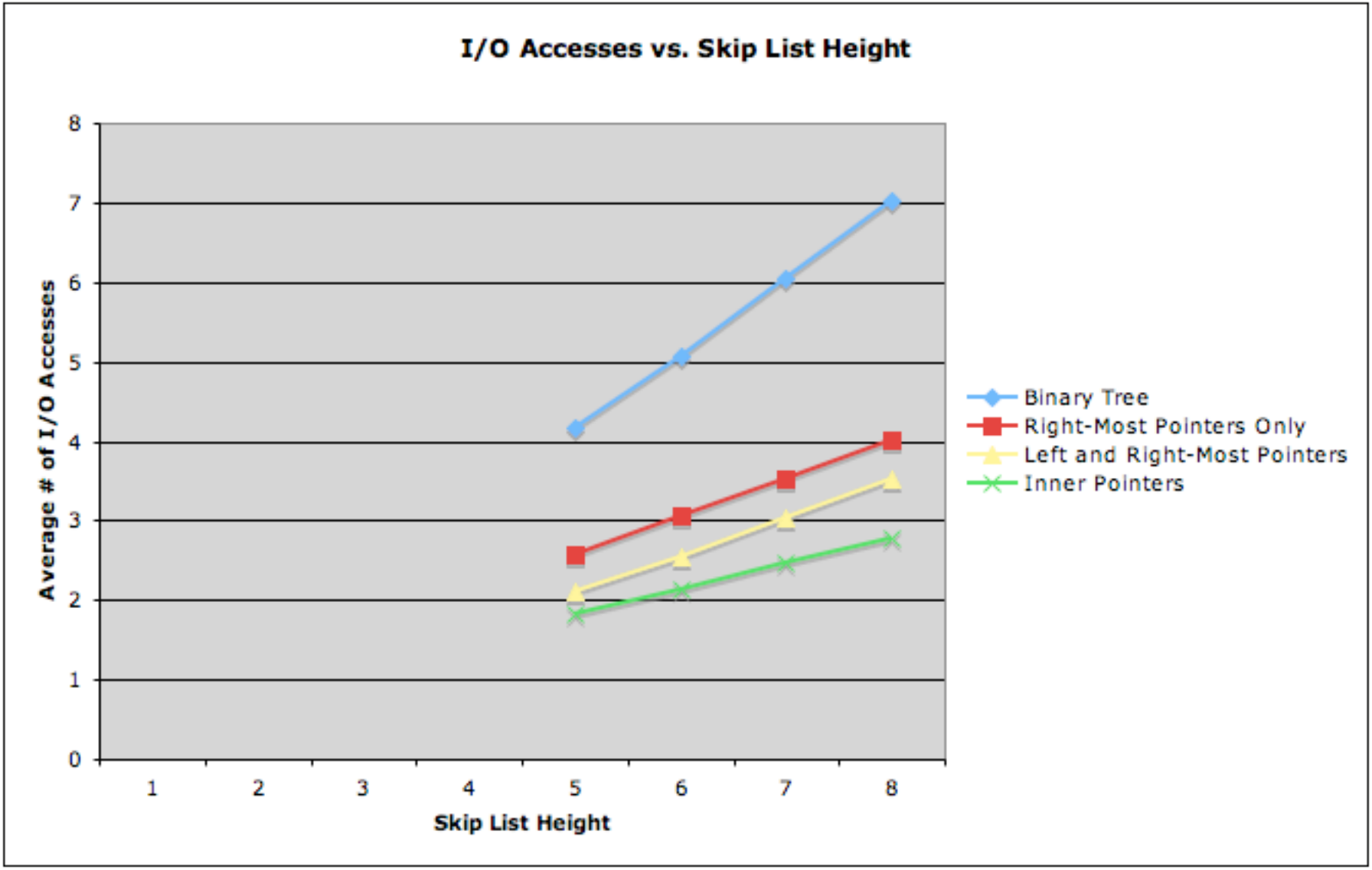
I/O Accesses vs. Skip List Height
Average # of I/O Accesses
0
1
2
3
4
5
6
7
8
1
2
3
4
5
6
7
8
Skip List Height
Binary Tree
Right-Most Pointers Only
Left and Right-Most Pointers
Inner Pointers

Figure 10

Skip lists with height < 5 aren't very interesting and determining the values for skip lists of height > 8 is fairly painful, so they are not shown. However, it's well known that the average number of I/O accesses for a lookup in a balanced binary tree increases as O(log n) and that is evident in figure 10. Using the slope of the binary tree in figure 10 as a guide, it's evident that the slopes of the lines for the skip list data structures are proportional to O(log n) for lookups also, although they all manage to increase at a slower rate. Particularly, the skip list with "inner" supernode pointers as well as left and right-most "external" supernode pointers has a much lower slope than the balanced binary tree, proportional to ~O(log n)/3 (the other two forms of skip lists increase proportional to ~O(log n)/2).

However, including forward pointers at heights >1 to the supernodes without modifying existing supernodes when a new supernode is appended to the skip list requires keeping additional ghost supernodes. A ghost supernode at each height currently used in the skip list needs to be tracked in the tail node, and is pointed to by the right-most supernode greater than or equal to that height in the skip list (not the right-most supernode of a particular height). (Actually, a ghost supernode of <max. height> + 1 needs to be tracked, since the right-most supernode of <max. height> will need to point to it).

These changes are shown in Figure 11:

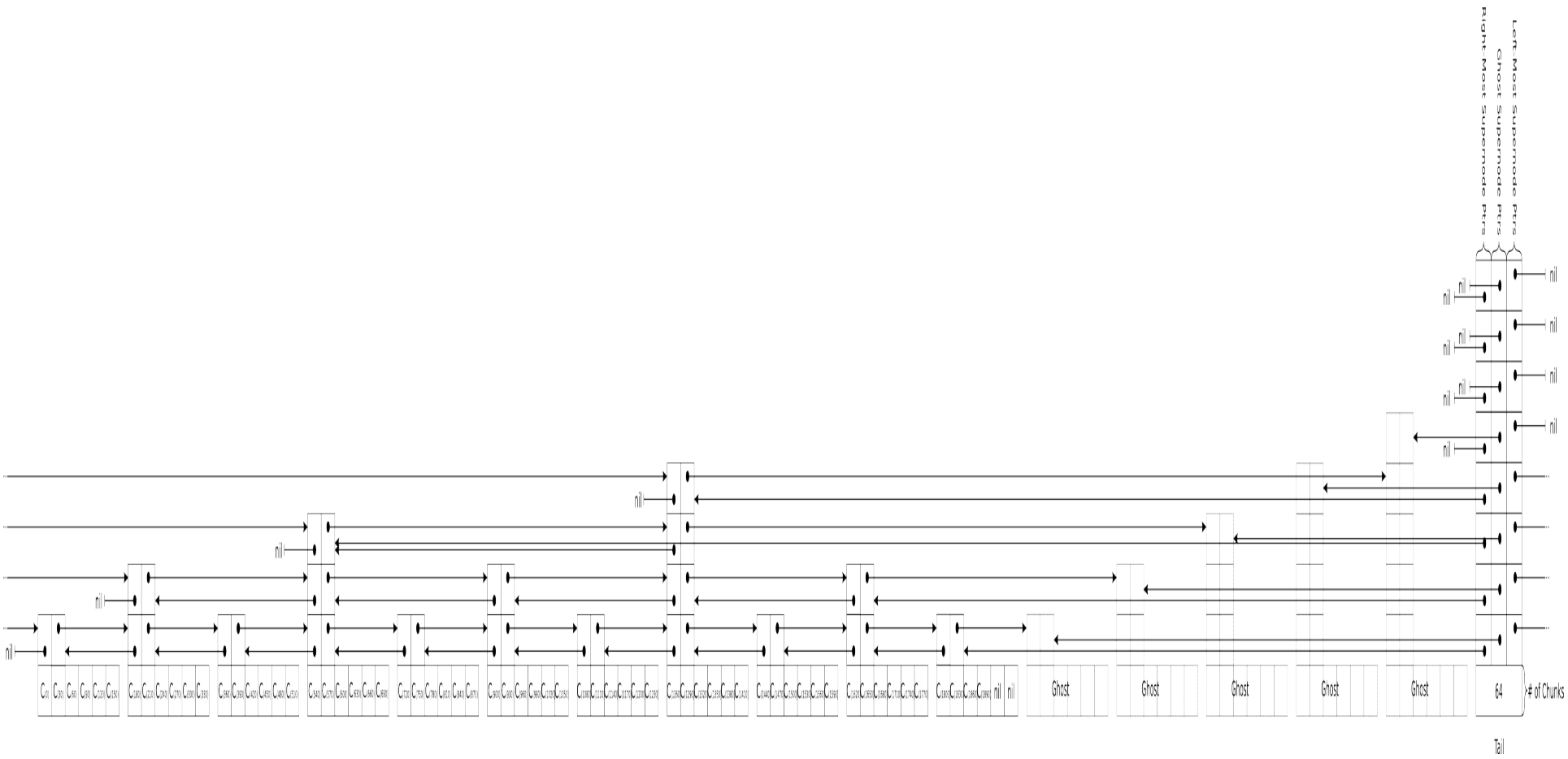

Figure 11

As mentioned above, figure 11 shows the height 4 supernode (which starts with chunk index 1260) with its forward pointer at height 4 pointing to the height 5 ghost supernode, since that will be the next supernode in the skip list at that height. The height 4 supernode also has its forward pointer at height 3 pointing to the height 3 ghost supernode, since that will be the next supernode in the skip list of that height (it's the right-most supernode greater than or equal to height 3).

Adding these forward pointers and additional ghost supernodes increases the size of the data structure, but the reduction in average number of I/O accesses for lookups makes the extra space worthwhile. Particularly, the number of ghost supernodes only increases as log(n) of the number of supernodes and is an increasingly small percent of the overall space for the data structure as the number of nodes increases.

XVI. Create "Indexnodes" for "Finished" Supernodes on Left Half of Skip List

After contemplating the improvements for adding forward pointers to supernodes in the previous section and thinking about why the average lookup time improved, it seems to be due to the additional pointers to more supernodes. To take this a step further, we create "indexnodes" pointing to all the supernodes on the "left half" of the skip list, which are "finished" (all ghost supernodes pointed to have been filled in).

A indexnode is just an array of pointers to supernodes and is pointed to by the left-most pointer at each height in the tail node. This makes accessing all supernodes in the left half of the skip list take constant time of 2 I/O accesses: one for the indexnode and one for the supernode desired. Forward and reverse pointers in supernodes on the left half of the skip list are no longer needed, since all the supernodes can be interated in forward or reverse order in constant time also.

This is shown in figure 12:

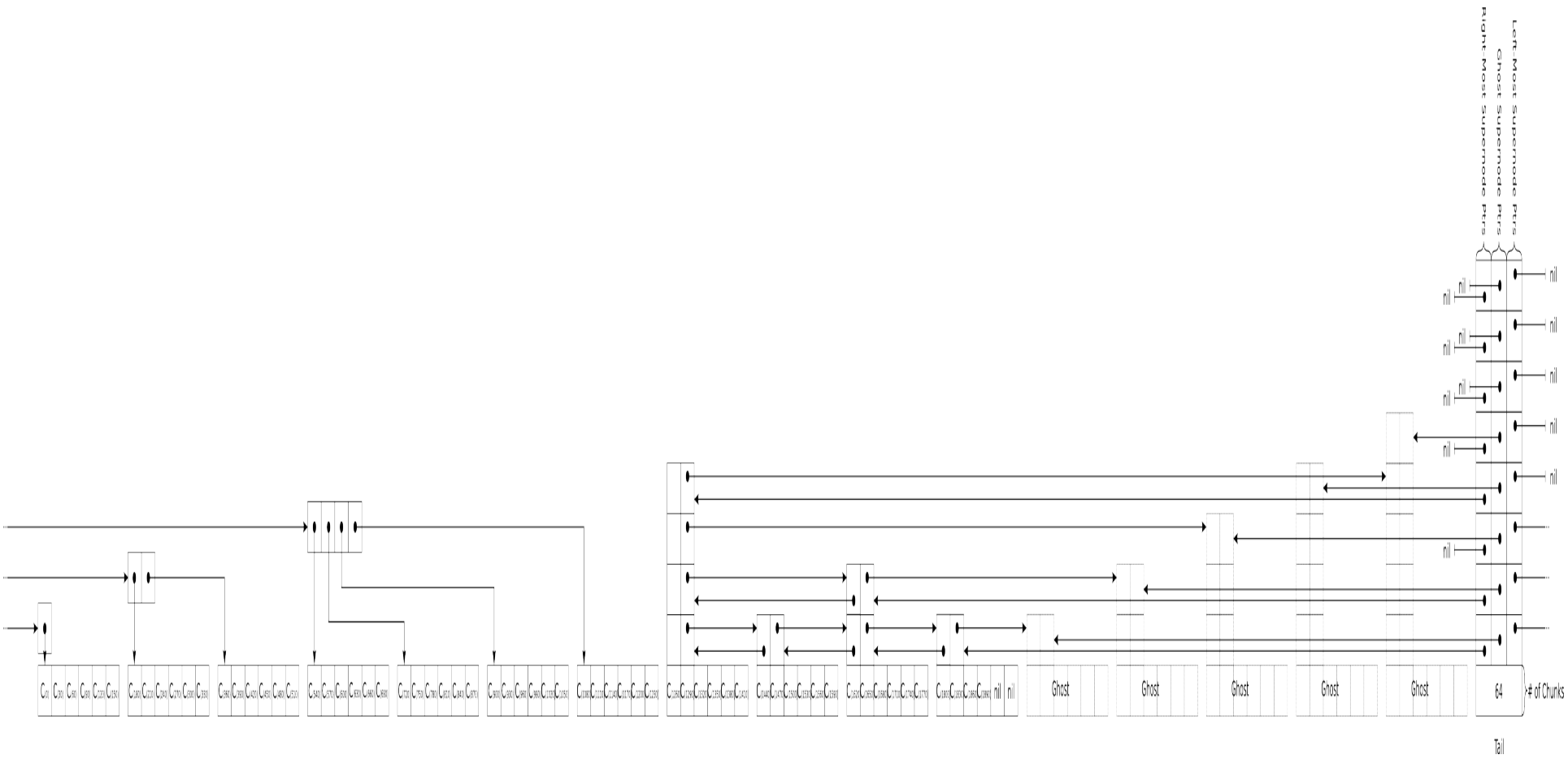

Figure 12

There would be a number of problems if this particular form of the data structure were to be actually implemented. Fortunately, we can take this "arrayification" further and avoid this awkward data structure, in the next sections.

XVII. Introduction to Resizable Array Data Structure

Looking at the structure created in the previous step (figure 12), it is beginning to resemble a data structure for storing a dynamic array, particularly one which uses the classic solution of making each portion of the array twice as large as the previous portion in the array. Brodnik, et al's paper "Resizable Arrays in Optimal Time and Space" addresses this exact problem, showing how to implement a dynamic array with O(1) append, shrink and lookup operations, along with optimal metadata overhead for indexing the array elements.

An example of a resizable array structure is shown in figure 13, with data blocks starting at 16 elements in size:

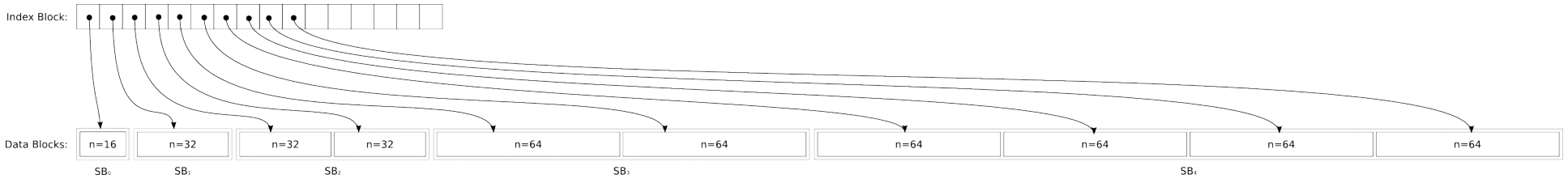

Figure 13

For indexing chunks in HDF5 datasets, the array elements in the data blocks will be pointers to the chunks on disk.

As mentioned in the paper, the resizable array "super blocks" are not actually constructs in the data structure, they are just shown to conceptually group the data blocks that the index block points to. The super blocks help show the pattern of changes to the data blocks: first the size of the data block doubles, then the number of data blocks of that size doubles, then the size doubles again, then the number of data blocks of that size doubles, etc.

One aspect of the resizable array described in the paper is not desirable for storing an index of chunks in an HDF5 file: the index block starts with a small number of pointers to data blocks and changes in size (by doubling) when more space for data block pointers is needed. This constant increase in size will cause the index block to move in the file, making it difficult for to implement a single writer/multiple reader solution. This is addressed in the next section.

XVIII. Combine Skip List and Resizable Array

In order to eliminate changing the index block's size, we can combine the tail node and indexnodes shown in figure 12 with the index and data blocks in figure 13. The tail node from figure 12 becomes the index block in figure 13, with pointers to explicit super block arrays instead of the indexnodes. This is shown in figure 14:

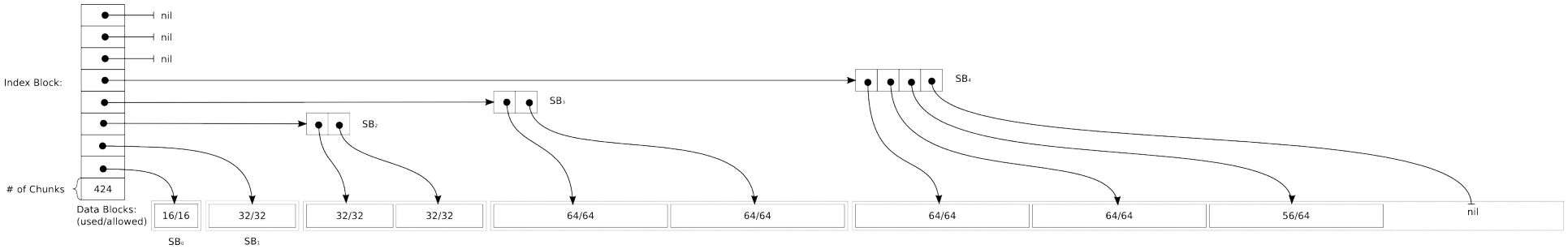

Figure 14

Several things should be noted about the data structure in figure 14:

- The super blocks for the first two data blocks are omitted and the pointers in the index block point directly to the data blocks.
- Constant time (i.e. O(1)) lookup, append and shrink are still in effect. The number of I/O accesses to find any chunk address is always 2 or 3: index block -> super block -> data block (the super block access is not required for the first two pointers in the index block, since the pointer in the index block points directly to the data block for the row). In actual operation, the index block will certainly be "hot" enough to stay in the HDF5 metadata cache, along with many/most of the super blocks, making the average number of I/O accesses to retrieve a chunk address between 1 and 2, in all likelihood.
- Super blocks/data blocks which are not yet needed are not allocated and have a 'nil' pointer in the index block/super block (respectively) that refers to them. As with the skip list structure however, if the unlimited dimension of the HDF5 dataset is extended and no data is written to the dataset elements covered by the dimension, the resizable array data structure will be "semi-sparse" like the skip list data structure.
- No ghost nodes/supernodes are needed.
- As with the skip list's tail node, the height of the index block is constant and its height can be computed at creation so that resizing it is unneccesary. This can be done by computing the maximum number of chunks possible for the file's address range (typically 64-bits) and deriving the maximum number of data/index blocks needed to index that many blocks. For example, if there are 16 chunks in the initial data block, a chunk size of 2048 (with a single unlimited dimension), and a 4-byte (integer) data element size, maximum height of the index block needed to store 1.84E+19 bytes of elements (which is the maximum offset possible in a 64-bit file) is 47. (See this spreadsheet for investigating other parameter values).
- One thing to note from the spreadsheet is that the extra space needed to locate a chunk in the index data structure tends to 0 as the number of chunks increases (seen in the "Metadata Space per HDF5 Chunk" column), because the metadata overhead per chunk goes quickly to just the amount of space needed to store the pointer to the chunk. Intuitively, this corresponds well with the data structure behaving (for lookups, etc.) like an array of chunk pointers: it should have space requirements similar to an array of chunk pointers.

Single writer/multiple reader access is preserved during append operations according to the following algorithm:

- If there's room for the new chunk's address in the last/right-most data block, the data block is updated with the chunk addres and then serialized and written to disk in one I/O operation. Then the # of chunks in the index block is updated, serialized and written to disk in one I/O operation.
- If the right-most data block is full and the right-most superblock has room for more data block pointers, a new data block is allocated and initialized with the new chunk address, then serialized and written to disk in one I/O operation. Then the super block is updated with the pointer to the new data block, serialized and written to disk in one I/O operation. Then the number of chunks in the index block is updated and it's serialized and written to disk in one I/O operation.
- If both the right-most data block and the right-most super block are full, both a new data block and super block to point to it are allocated, updated, serialized and written to disk, as above. Then the index block is updated with the pointer to the new super block and number of chunks, serialized and written to disk, as above.

At any point during the operations above, if a reading process accesses the data structure, it always sees valid and initialized information (although care must be taken not to access chunk addresses beyond the limit given in the index block). This is slightly more complicated than the original sequence of operations for the reverse skip list, but follows the same pattern. The constant time operations should easily compensate for the slight additional algorithmic complexity in updating the data structure on disk.

XIX. Hoist More Information Into the Index Block

There still appears to be room for some optimizations to the data structure shown in figure 14:

- The size of data blocks gets fairly large in short order if a "reasonable" number of "chunks per data block" is chosen - it might be worthwhile to compress the data blocks on disk. This doesn't improve the number of I/O accesses, but does reduce the chunk index metadata overhead. (This can be experimented with in the spreadsheet by adjusting the "Data Block Compression Factor" parameter)
- The size of super blocks is fairly small (for "reasonable" numbers of "chunks per data block"). Either a smaller "chunks per data block" value could be chosen, causing the size of the super blocks to more closely track the size of the data blocks, or the super block "expansion factor" could be increased from 2 to 4 or larger. (Note: I don't have a good feel for the effect of changing the data block or super block expansion factors does for the data structure's space, it's pretty obvious that the O(1) access will be preserved though. These effects can be experimented with in the spreadsheet by adjusting the "Data Block Expansion Factor" and "Super Block Expansion Factor" parameters)
- It is possible to eliminate very small super blocks (from previous point) by "hoisting" more of the data block pointers in the index block, when the number of data blocks in a super block is below some threshhold. This would also reduce the number of I/O accesses required to retrieve a chunk address for data blocks directly pointed to by the index block to only 2. It appears that this is worthwhile, particularly if an application developer knows that the dimension size for a dataset is very likely to be less than a particular size. In that case, super block creation can be delayed until the unlimited dimension's size is greater than that value, without very much increase in the size of the index block. (This can be experimented with in the spreadsheet by adjusting the "Min. # of Data Blocks per Super Block" parameter)
- One could even take the optimization in the previous point a step further and allow the index block to store a certain (small, probably) number of pointers to chunks directly in the index block itself. This would reduce the number of I/O accesses required to retrieve the chunk address for chunks directly pointed to by the index block to only 1. Since this also only implies a small overhead for the index block, it also appears to be

worthwhile to implement. (This can be experimented with in the spreadsheet by adjusting the "# of Chunk Pointers Directly Stored in Index Block" parameter)

Note that implementing the last two optimizations above is roughly equivalent to the heuristics for storing "tiny", "normal" and "huge" objects in the fractal heap code.

The last two optimizations are shown in figure 15, with the "Min. # of Data Blocks per Super Block" parameter set to 4 and the "# of Chunk Pointers Directly Stored in Index Block" parameter parameter set to 4 also:

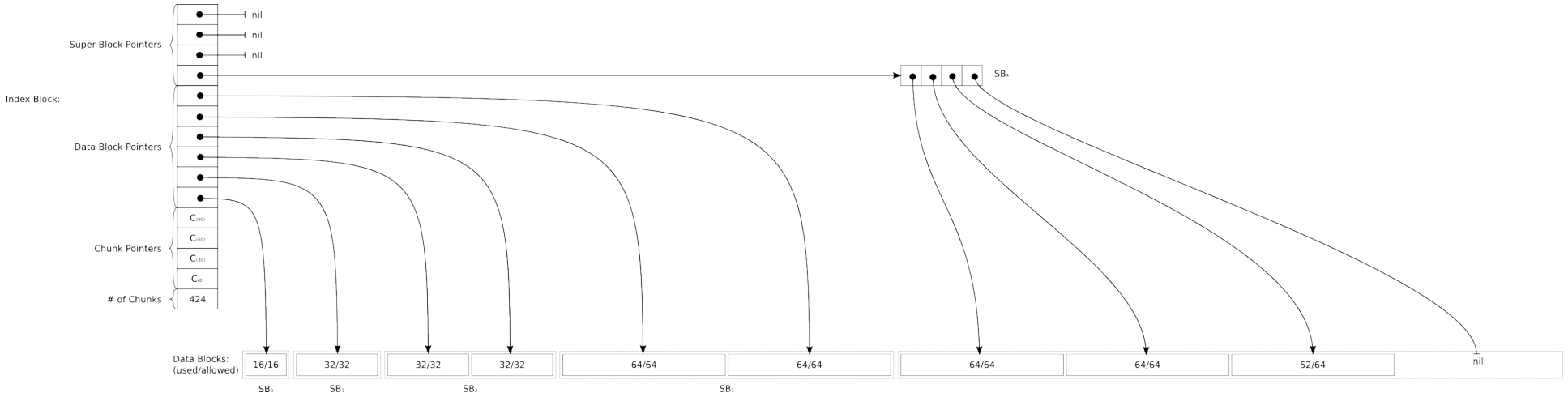

Figure 15

XX. Add Configuration Block for Constant Data Structure Values

The final change to the resizable array structure described above is to add a "configuration block" to the data structure (this could have been added to the data structure in an earlier step as well). There are several constant values associated with the data structure described here and the configuration block is added to hold this information. It holds a pointer to the index block along with the height of the index block, the number of chunk addresses per data block unit, the number of chunk pointers in the index block, the min. number of data block pointers in a super block, the size of the chunks, etc.

This is shown in figure 16:

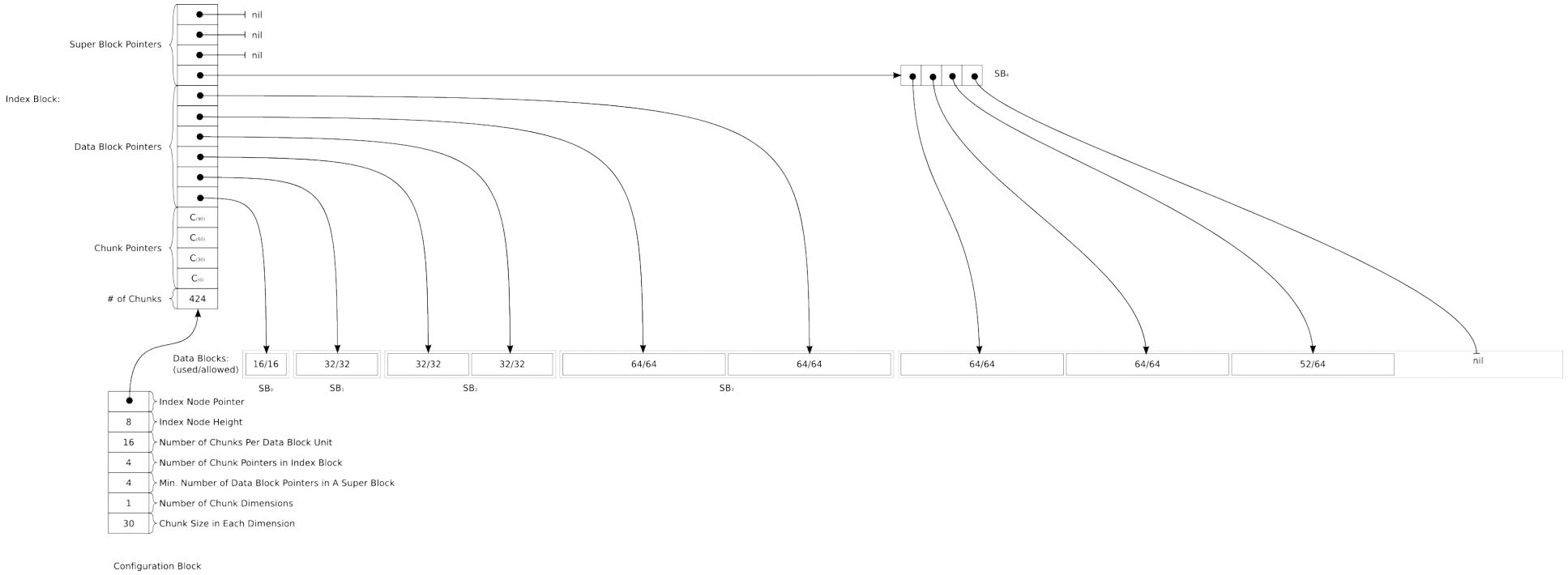

Figure 16

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

XXI. 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 17 shows a 3-D dataset with chunk sizes of (30, 25, 40), maximum dimension sizes of (unlimited, 50, 80) and current dimension sizes of (3180, 50, 80):

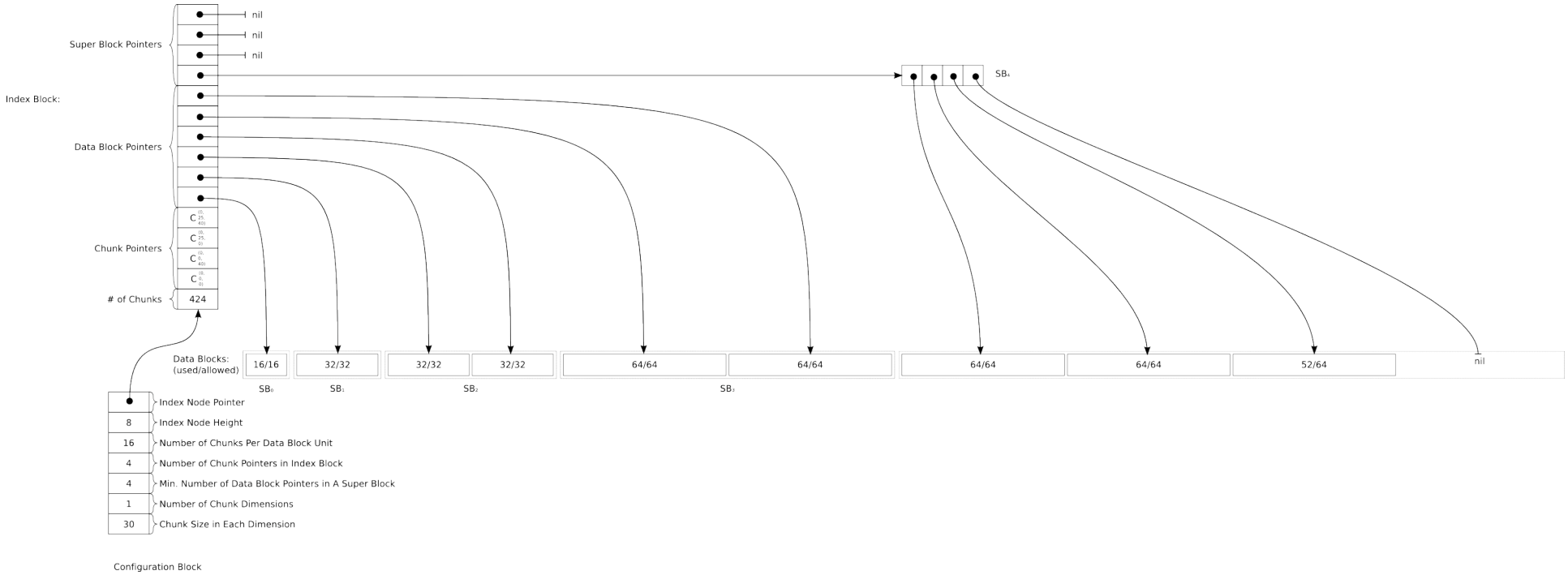

Figure 17

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 17 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.