Improving Access to Multi-dimensional Self-describing Scientific Datasets*

Beomseok Nam and Alan Sussman
UMIACS and Dept. of Computer Science
University of Maryland
College Park, MD 20742
{bsnam,als}@cs.umd.edu

Abstract

Applications that query into very large multi-dimensional datasets are becoming more common. Many self-describing scientific data file formats have also emerged, which have structural metadata to help navigate the multi-dimensional arrays that are stored in the files. The files may also contain application-specific semantic metadata. In this paper, we discuss efficient methods for performing searches for subsets of multi-dimensional data objects, using semantic information to build multi-dimensional indexes, and group data items into properly sized chunks to maximize disk I/O bandwidth. This work is the first step in the design and implementation of a generic indexing library that will work with various high-dimension scientific data file formats containing semantic information about the stored data. To validate the approach, we have implemented indexing structures for NASA remote sensing data stored in the HDF format with a specific schema (HDF-EOS), and show the performance improvements that are gained from indexing the datasets, compared to using the existing HDF library for accessing the data.

1 Introduction

As the size of data files produced in many scientific areas continues to grow, currently it is not hard to find databases containing terabytes of data, and petabytes or exabytes of data will soon be common. In order to store and process such large datasets, we have been developing middleware systems such as the Active Data Repository(ADR) [12] and DataCutter [4]. Scientific datasets can be stored and processed on a cluster or parallel machine with ADR or across a distributed set of machines (the Grid) with DataCutter. These systems support efficient subsetting and processing of large dataset by integrating application-specific processing into the storage manager.

Another issue related to dealing with large datasets is that, to help in navigating through large scientific datasets, many self-describing scientific data file formats have been developed, such as Flexible Image Transport System(FITS) [7], Planetary Data System(PDS) [1], Network Common Data Format(NetCDF) [17], and Hierarchical Data Format(HDF) [15]. Self-describing data formats contain structural metadata, which is used by a corresponding runtime library to navigate through the file to improve I/O performance, by allowing for direct access (once the metadata is read) to particular datasets within a file, or to parts of the dataset. Files in these self-describing formats also may contain application-specific metadata, which provides semantic information about the contents of the file [8]. XML is an example of a text-based self-describing data format that is widely used. In this paper, we address the problem of improving the performance of accessing subsets of data stored in scientific self-describing data formats, using spatial indexing techniques.

The contents of scientific data files typically are a collection of multi-dimensional arrays, which we will refer to as datasets, along with the corresponding metadata. Since a dataset can be sparse, it is often useful to organize the raw data as chunks of some fixed size and allocate storage on demand, instead of organizing the data as one contiguous sequence within the file. If no data is written to a chunk then that chunk will not be allocated on disk. In addition to storage savings from allocating space for chunks as needed, data chunking makes it possible to extend the size of a dataset as required by an application and may also improve I/O performance, as we will show in the experimental results later in the paper.

Dataset can be spread over disks discontinuously in chunked storage layout, hence the file should have indexing structures such as B-trees to locate the file address of each chunks. However B-tree is not capable of handling multi-

*This research was supported by the National Science Foundation under Grants #EIA-0121161, #ACI-9619020 (UC Subcontract #10152408), and #ACI-9982087, and Lawrence Livermore National Laboratory under Grants #B500288

dimensional information. And none of self-describing data file formats support the multi-dimensional indexing feature to the best of our knowledge.

In the past couple of decades, much research has been done to create high-dimensional indexing structures that satisfy range queries and nearest neighbor queries efficiently, such as R-trees [9] and its descendants, including R+-trees [19], and R*-trees [2], SS-trees [22], SR-tree [11], and X-trees [3].

With these multi-dimensional indexing trees, we are designing a generic indexing library that stores semantic indexing information into any kind of self-describing scientific data format. For this purpose, it is impractical to modify the internal structures of various data formats; instead we create separate index files for the data structures stored in a file that uses a self-describing format. The index structures that are used depend on the semantics of the stored scientific data objects. For example, some data objects may require a one dimensional index based on temporal values, while other data objects may require a 2D or 3D spatial index. Such semantic information about what data fields to use for indexing is required both to create index files and to generate functions that allow for performing multi-dimensional *range queries* using the index. A range query specifies the data to be retrieved as a bounding rectangle in a multi-dimensional coordinate space (i.e. the one used to build the index).

As the first step with this approach, and in order to show that indexing improves I/O performance, we have designed an indexing library that creates an R*-tree for datasets stored in HDF4 or HDF5 formats. We concentrate on improving the performance of reading data from the datasets, since many datasets are never modified once created (i.e. datasets acquired from sensors or produced by simulations do not change once acquired).

The rest of the paper is organized as follows. Section 2 defines the terminology for data chunking, and discusses performance issues related to chunking. In Section 3 we present the design of a generic spatial indexing library and show the performance improvements provided by data chunking and indexing for range queries. We conclude in Section 4.

2 Data Chunking

Data chunking partitions a dataset into coarse-grained blocks to reduce disk access time when accessing large amounts of data in a file. Most self describing scientific data formats store data as multi-dimensional arrays, to ease access from within scientific programs. Scientific applications access multi-dimensional arrays with various access patterns. Some applications read sub-arrays in row major order, or in column major order. Others read sub-arrays specified as regular sections [10]. Scientific data format libraries support reading sub-arrays with various access patterns, but most of them do not show good I/O performance along every dimension. Only a few libraries, which support data chunking, achieve similar performance for any kind of access pattern. For datasets consisting of data arrays, each data chunk can be viewed as a sub-array within the dataset. The order of data accesses into a multi-dimensional array critically affects the I/O performance. To achieve maximum I/O performance by minimizing disk seek operations, each chunk should be a single contiguous sequence in the file. We use the term *physical chunk* to refer to a sub-array that is a physically contiguous single sequence within a file on disk. Depending on the data access pattern, physical chunking can provide much higher I/O performance than a contiguous, row or column major, ordering of the array elements [21, 18]. A *logical chunk*, on the other hand, is a conceptual partitioning of a dataset on disk. A multi-dimensional dataset can be partitioned into logical chunks whether it is a single contiguous array or a physically chunked array. When a dataset is stored as a single array on disk, disk seek operations are required to access each row of a logical chunk. On the other hand, when a dataset is partitioned and ordered as physical chunks, the layout of the physical chunking can also be viewed as the logical chunking. However, logical chunking does not necessarily have to use the same partition as physical chunking, (i.e. a logical chunk in a physically chunked dataset can contain several physical chunks, and could even be a subset of a physical chunk). Logical chunking by itself does not improve I/O performance, but is necessary to create an index into the data, as we will discuss in Section 3.

2.1 Case study - HDF

Hierarchical Data Format: Hierarchical Data Format (HDF) is a self-describing scientific data file format and runtime library developed at the National Center for Supercomputing Applications (NCSA) to store and serve heterogeneous scientific data. A file stored in HDF contains supporting metadata that describes the content of the file in detail, including information for each multi-dimensional array stored, such as the file offset, array size and the data type of array elements. HDF also allows application-specific metadata to be stored. Thus, the metadata within a file make HDF an essentially machine independent format. The most recent version of HDF is HDF5. Although HDF5 was designed to overcome some deficiencies of the older HDF4, HDF5 has a totally different internal representation of data objects from previous HDF versions. HDF4 is backward compatible with previous versions HDF, but HDF5 is not compatible with HDF4 [15].

Data chunking in HDF: To improve I/O performance, HDF supports two different storage layouts. The default storage layout is a contiguous layout, in which the elements of a multi-dimensional array are stored in either row-major order or column-major order. The second choice is a chunked layout, in which data is stored as physical chunks, as described above, with each chunk stored in row-major or column-major order.

Data chunking also works well with data set compression, hence HDF also provides a compressed chunked layout. When an application requests a subset of a compressed dataset stored with a chunked layout, it is not necessary to decompress the entire data set, but only the chunks that overlap with the requested subset [8]. However, we will not discuss data compression issues further in this paper.

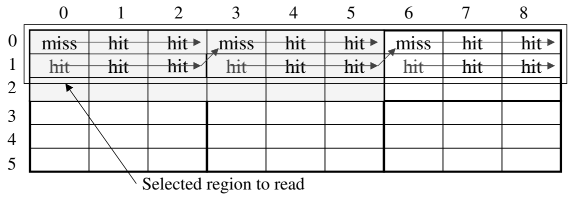
In HDF5, a chunked layout has other advantages over a contiguous layout. In particular, a chunked storage layout allows extending the size of a stored multi-dimensional array in any dimension, not just the slowest varying array dimension (outermost in row-major order, innermost in column-major order). In addition, disk space for a chunk does not have to be allocated on disk until data is written into that chunk, which can decrease disk storage requirements. HDF4, on the other hand, provides only some of the advantages of a chunked layout. In HDF4, extending the size of an array dimension is allowed only for the slowest varying dimension, but not for any other dimensions.

In accessing a subset of a large dataset, data chunking reduces expensive disk seek times and improves overall I/O performance by taking advantage of spatial locality in any of dimensions [21]. If an application accesses a data element in a large dataset, there is a high probability that it will also access nearby elements, in all dimensions for a multi-dimensional array dataset. On the other hand, the contiguous storage layout can exploit spatial locality only in the dimension that varies fastest in storage order. (i.e. the innermost dimension for row-major order and the outermost dimension for column-major order).

However, a chunked layout does not always provide better performance than a contiguous layout. One case in which data chunking may hurt I/O performance occurs when the size of a chunk is very large and the region selected to read is smaller than the size of a chunk, causing unnecessary data to be read from disk, since disk I/O is always done in units of complete chunks.

Potential problems with HDF data chunking: Both the HDF4 and HDF5 libraries cache data in a *data chunk cache* to improve I/O performance. However, the functions that read datasets in both libraries are designed as if the size of the data chunk cache is infinite, potentially causing significant performance problems. Because the read functions in the HDF libraries read arrays in row major (or column major) order, whether the array has a chunked layout or contiguous layout, that ordering does not match the ordering of data with a chunked storage layout, potentially leading to many data chunk cache misses.

(a) H5Dread. A chunked layout can reduce performance. Suppose each chunk contains a subarray of 3x3 elements, and the size of each chunk is 512K bytes. Since the default size of the data chunk cache is 1MB, the cache can only store two chunks. Reading element (0,6) evicts the first chunk from the cache, which contains element (1,0), thereby causing a cache miss.



(b) H5Xread. The H5Xread function reads data elements in chunk order to minimize cache misses. Therefore the same chunk on disk will not be read multiple times.

Figure 1. *The ordering problem for H5Dread with a chunked layout*

Suppose we want to read two rows of a dataset stored with a chunked layout. The standard HDF library read function, H5Dread, reads the data in row major order, as shown in Figure 1(a). When the first row of the array is read, all array elements in the chunks that contain the first row are cached in the data chunk cache, along with the rest of the chunks. When the next row is read, the library searches in the cache, but will not find the chunk needed, because the default chunk cache size is 1MB. If the total size of the chunks that contain one row of a dataset is greater than 1MB, the data chunk cache will not be able to hold all the chunks and will evict the chunks in the cache using an LRU replacement policy. Therefore when the library reads the second row of array elements, the first element in the second row will not be found in the cache, as shown in Figure 1(a). So the chunk containing that array element must be read from disk again, and the same problem will occur for all other data chunks both in that row and in subsequent rows.

The HDF library developers have recognized this problem, and warn of severe performance penalties in the HDF User's Guide [14]. Their solution to the problem is to add a function to the HDF5 API that increases the size of the data chunk cache, placing the burden of selecting the appropriate data chunk cache size on the application developer. We now propose another solution.

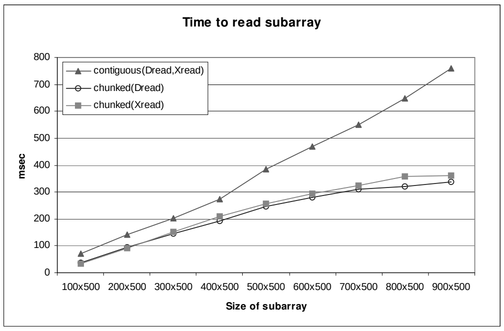
H5Xread: We have added new functionality to the HDF5 library, in the form of a function called H5Xread with the same interface as H5Dread, to read multi-dimensional array datasets from disk in the same order they are stored with a chunked storage layout. Such a strategy avoids unnecessary cache misses and reading the same chunk from disk multiple times. After chunks are retrieved from disk, they are reorganized in memory to produce the desired contiguous array layout. For arrays stored with a contiguous layout, H5Xread reads the data from disk in the same order as H5Dread. Figure 1 shows the difference in data accesses between the H5Xread and H5Dread functions. The array read function in the HDF4 library has the same performance problem as H5Dread, and the same functionality as in H5Xread can be implemented for that library.

Performance evaluation: We now present the results of a performance evaluation of the standard HDF5 dataset read function, H5Dread, with our H5Xread function, for chunked storage layouts. In the experiment, we partitioned a two-dimensional 64MB dataset, containing an array of 4000x1000 elements, each of which is 16 bytes. The array was partitioned into 160 KB logical chunks, each of which contains 100x100 elements. For the chunked layout, we made the physical chunk size the same as the logical chunk size. The 64MB dataset is large enough for this performance evaluation because it will show the performance differences between the access functions. The experiments were run on a SunBlade 100 workstation with a 500MHz Sparcv9 processor, 256MB memory, and a 7200RPM IDE disk with a seek time of 9ms.

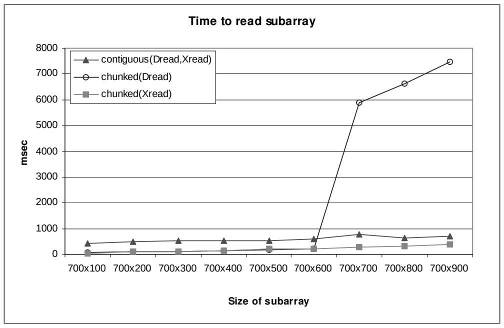
Figure 2 shows the time to read two different shaped subarrays from the dataset. We measured the wall clock time, varying the number of rows read in in Figure 2(a), and varying the number of columns read in Figure 2(b).

Figure 2(a) shows that the chunked storage layout provides better I/O performance than the contiguous layout in most cases. The performance gap between the chunked layout and the contiguous layout increases as the number of rows increases. This is because as the size of a column grows, even more disk seek operation are needed for the contiguous layout than for the chunked layout.

For these experiments, we used the HDF library default sized data chunk cache of 1MB, so the chunk cache holds six of the 160KB chunks. In Figure 2(b), when the number of columns in the selected subarray is less than or equal to 600, the H5Xread function shows similar performance to that of H5Dread for a chunked layout, but as the number of columns increases, so that the size of each row increases, the cache fills up before reading an entire row - in this experiment when the row size reaches 700 elements, at which point the H5Dread function suffers from many cache misses, while H5Xread continues to provide stable I/O performance. The performance difference can be a large factor, here up to a factor of 9, as is seen in the right side of the figure.



(a) The selected region has a fixed number of columns, and the number of rows increases.



(b) The selected region has a fixed number of rows, and the number of columns increases.

Figure 2. *Time to read selected regions of the dataset*

The H5Xread functionality also provides stable performance characteristics for higher dimensional datasets. We have evaluated performance for three-dimensional datasets, and the results are essentially the same as those for the two-dimensional experiments, meaning that the H5Xread function with a chunked layout provides better performance than H5Dread with either a chunked or a contiguous layout.

3 Spatial Indexing

A common type of retrieval pattern on multi-dimensional datasets is spatial range queries, which read a subset of the multi-dimensional array within a given range of values for each of several dimensions (e.g., three-dimensional space or time). If an application has to scan the entire index space (the metadata) for the entire dataset, performing a spatial range query could be a very expensive operation.

A large number of indexing techniques have been proposed to improve the performance of range queries and nearest neighbor queries for multi-dimensional datasets. Techniques for speeding up searches into high-dimensional datasets have been researched extensively [5, 6]. The most common multi-dimensional indexing structure, the R-tree, is a height-balanced tree similar to the well-known B-tree [9]. When point data is inserted into a leaf node of an R-tree, the minimum bounding boxes of the internal nodes are enlarged to cover the child nodes, sometimes requiring that internal nodes be split to maintain the balance criteria. For a given multi-dimensional range query, a search into an R-tree traverses all nodes in the tree with minimum bounding boxes that overlap the range. The R*-tree is an optimized R-tree extensions that minimizes overlap of nodes [2].

The goal of using a spatial index is to avoid searching all the elements in a multi-dimensional dataset to perform a spatial range query. If the dataset is partitioned into coarse-grained chunks, and the bounding box for each chunk (i.e. the minimum and maximum values for each dimension, for example, spatial and temporal coordinates) is placed in an index structure, not all elements within the dataset must be searched, but only elements in the chunks with bounding boxes that overlap the query range. This effect reduces the amount of data retrieved from disk, and should improve query response time.

3.1 Multi-dimensional scientific data formats

We have been designing and implementing a generic indexing library for various multi-dimensional scientific data formats using an R*-tree. The R*-tree provides better performance and storage utilization than an R-tree, especially for high-dimensional data. Figure 3 shows the design of the indexing library. A new multi-dimensional scientific data format can utilize the services of the indexing library by implementing three functions that (1) create an index file, (2) search the index file for a range query, and (3) read a subset of the dataset using the information returned from searching the index. The generic indexing library provides an API for these functions. The functions must pass in as parameters the dataset name, the dataset size, the type of the data stored in the dataset, the number of dimensions for the multi-dimensional dataset, data chunking information, and an index file name. With this information, the indexing library can create the index in the index file, which can then be used to search the dataset via range queries, and also read subsets of the dataset using file location information acquired from the index. Once the R*-tree index is created and stored on disk, applications can use that index as long as the dataset is not updated. In order to read data files in a specific scientific data format, the indexing library read function must call a read function from the particular scientific data format library. The name of the read function, and some additional information about various parameters, must be obtained from the scientific data format library. Future work on the generic indexing library will concentrate on automatically extracting that information from the metadata within the self-describing data file to automate generation of an index, and we also plan to automate the process of generating the range query functions (index search and data read) for a scientific data format library.

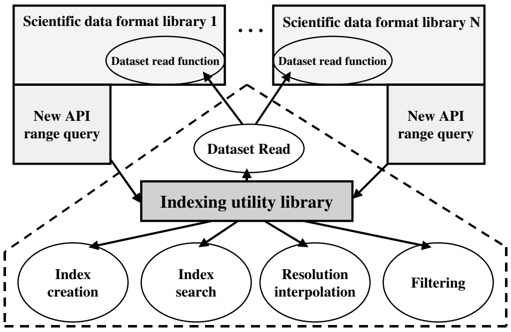


Figure 3. *Generic spatial indexing library*

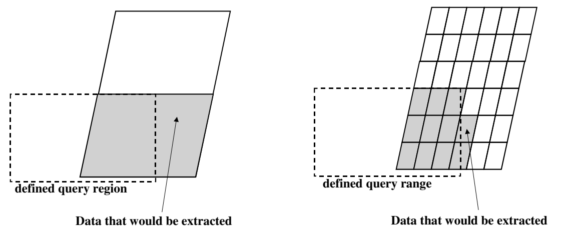
The generic indexing library has an index creation module, an index searching module, a resolution interpolation module, and a filtering module. Multi-dimensional datasets, in particular ones with spatial and/or temporal dimensions, may contain data elements at different granularities. For example, multiple sensors on the same orbiting satellite may have different resolutions. Hence some sensor datasets may have arrays that are several times larger than the corresponding geographic datasets that allow for determining the spatio-temporal locations of the data elements. The generic indexing library addresses this problem by providing an interpolation mechanism, which we do not discuss further. The last function that the indexing utility library provides is data filtering. Because data is stored as chunks, a range query can return all the chunks that overlap the given range query. However, not all data elements in those chunks will overlap the query range, so the library supports data filtering to return only those data elements that fall within the query range. If the application can accept extra elements (i.e. perform its own filtering), the library can also return the unfiltered chunks.

3.2 Case study: HDF-EOS

NASA's Earth Observing System Data and Information System (EOSDIS) is a system that acquires, stores, and distributes sensor data acquired from orbiting satellites. HDF has been selected as the standard data format by the EOSDIS project, and a metadata schema has been specified to store Earth Observing System (EOS) data. In addition, a library has been implemented on top of the HDF library, called HDF-EOS, to extend the capabilities of the HDF library to allow for the construction of special data structures, called *grids*, *swaths*, and *points*. We focus on swaths, because that is the way most HDF-EOS data is stored [16]. A grid structure is produced using a projection operation via a given mathematical transformation between the rows and columns of an array and the latitude/longitude information stored with the EOS data, and is used to store the results of such projection operations. The latitude/longitude information for each array element can be computed based on the array offset using map projections such as Mercator or Goode. A point structure is a table that contains data records taken at irregular time intervals and across scattered geographic locations. A swath structure is based on the way a typical satellite sensor acquires data, whereby an instrument takes a series of scans perpendicular to the ground track of the satellite as it moves along that ground track.

The HDF-EOS library has versions both for HDF4 and HDF5, called HDF-EOS4 and HDF-EOS5. Despite HDF4 and HDF5 being quite different data formats, the HDF-EOS4 and HDF-EOS5 libraries have essentially the same basic features for the HDF-EOS data structures. One of the differences between the two versions is that HDF-EOS4 supports data chunking only for grid structures, and not for point or swath structure. HDF-EOS5 supports data chunking for swath structures. Both versions of the HDF-EOS library allow a user to specify a range query, by specifying the data to retrieve as a box in latitude and longitude. Once a query region is defined, by the *defboxregion()* function, the user reads the data from that query region with an *extractregion*() function.

In an HDF-EOS swath structure, the latitude, longitude, and temporal information for the dataset is stored as separate arrays from the sensor value arrays. To retrieve the geographic information for a data element in a sensor value array, the elements in the geographic datasets that have the same offsets as the sensor element must be retrieved. The HDF-EOS library does not support spatial indexing structures. To read the sensor values that fall within a query range, the *defboxregion* function must scan every geographic dataset to obtain the location(s) of the region



(a) EOS range query

(b) Range query with data chunking

Figure 4. *Data read for a range query*

within the file, because the geographic information for the EOS datasets is not evenly distributed through the spatial domain (i.e. it has *spatial irregularity*) [20]. Once a region is defined with the *defboxregion* function, the corresponding *extractregion* function can be called to read the desired sensor data from the file. It is an expensive operation to scan all elements in a geographic dataset, so HDF-EOS provides several approximation options. First, an application can retrieve the set of scanlines that have any single element that overlaps the query range. In this *any-point* mode, all geographic data must still be searched. Second, if the mid-point of a scanline overlaps the query range, that scanline can be read in *mid-point* mode. In this mode, the *defboxregion* function reads only one column of the geographic dataset (the one for the middle element in the scanline). Finally, if both end points of a scanline overlap the query range, the entire scanline will be read in *end-point* mode. Mid-point and end-point selection are much faster than any-point selection, but there is a tradeoff between response time and accuracy in retrieving the desired data.

For our indexing library, creating the index requires reading all the geographic information for a swath to obtain minimum and maximum location (latitude/longitude) values for each logical/physical chunk.

For reading subsets of a dataset using the indexing library, all elements in chunks that intersect the query range are read, while the HDF-EOS library returns all elements in any scanline that overlaps the query range. Therefore the number of elements read by the two libraries may be different. Figure 4 illustrates a simple example. The range query functions return the query result in the form of a one dimensional array of data elements, but with EOS data each element in the array is associated with two-dimensional geographic coordinate information (latitude and longitude). However, some of the returned elements may not be in the query range, but the application cannot determine which elements should be discarded without the geographic coordinate information. Therefore the range query function in the indexing library returns the geographic information corresponding to each sensor value. Using this geographic information, applications can filter out sensor values that do not overlap the query range. If the chunk size is large, the R*-tree search may end up reading more unnecessary data than the HDF-EOS *extractregion* function, but it is much more likely that the HDF-EOS function will read more unnecessary data.

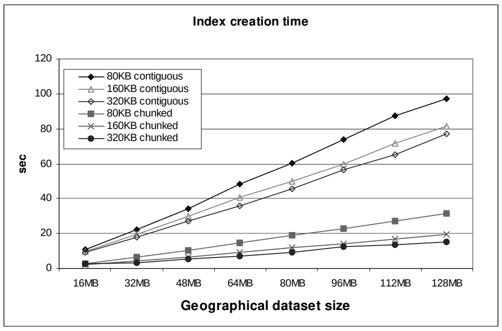
3.3 Performance evaluation

We evaluate performance for reading HDF-EOS data via range queries. We have implemented versions of the HDF-EOS4 and HDF-EOS5 range query APIs that call the indexing library. The test datasets range in size from 16MB with 30 chunks, to 128MB with 800 chunks. In our experiments we used H5Xread, described in Section 2.1, to read data from the file, instead of the HDF library H5Dread, since it provides better performance.

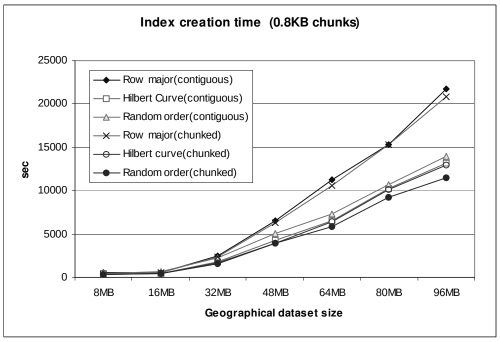
In the experiments, we have measured both the time to create an R*-tree index file for various numbers of chunks, and the time to perform range queries using the index. For range query performance, we have measured the time to read a subarray for three different shapes of the selected region within a two-dimensional array. The first query selects a region that spans many columns, but relatively few rows. For this kind of query, the HDF-EOS defboxregion function reads the data that exactly matches the query range in any-point or mid-point mode. However, our indexing library read function may read extra elements that are not in query range, but are in chunks that overlap the query range. The second query selects a mostly square region from the 2D array. The third query selects a region that spans many rows, but relatively few columns. For the second and third query, HDF-EOS library will read many more elements than our indexing library will. Since the HDF-EOS library reads all the elements in any scanline (row) that overlaps the query region,

All the results presented measure elapsed wall clock time. The size of the test dataset for measuring range query is 4000x1000, with each logical or physical chunk containing 100x100 elements of type double, for a total of 80KB per chunk. For measuring R*-tree index creation time, we created logical and physical chunk sizes of 0.8KB, 80KB, 160KB, and 320KB. Because the HDF-EOS4 library does not support data chunking, we measured performance only with a contiguous storage layout, and partitioned the arrays into logical chunks for indexing. The number of array elements requested for the first query is 200x900, for the second query 1000x500, and for the third query 2000x200. We ran the experiments on the same SunBlade 100 used for the data chunking experiments in Section 2.1.

Figure 5(a) shows the time to create the R*-tree index file



(a) Varying the chunk size and the data layout



(b) Varying the tree insertion ordering, with small chunks

Figure 5. *Time to create an index file, varying the chunk size and the data layout*

for various dataset and chunk sizes. The figure shows that the time to create the index depends linearly on the number of chunks, which is determined by the chunk size for a fixed size dataset. The question to answer then, is what is the best chunk size? There is a tradeoff between index creation time and disk access time for range queries. When the chunk size is small, the number of chunks is large and it takes a long time to create the index, as seen in Figure 5(a). On the other hand, a small chunk size will causes range queries to read less extra data. When the chunk size is large, so the number of chunks is small, index creation does not take long but it is necessary to read those large data chunks from disk and filter out unnecessary data elements, which is an expensive operation. The most important decision criterion is that the index will be used for all searches, but once an index file is created it will not be changed unless the dataset is updated. Although the index is not likely to change often, the time to create the index file should not be ignored. For example, when the number of chunks becomes very large, for example 50,000, it takes several hours to create the index file on the experimental machine. Most of time to create the index file is spent building the R*-tree, performing operations to maintain the desired tree properties. However when the number of chunks is small, reading the geographic dataset to produce the key values to insert into the R*-tree becomes a large fraction of the time to build the tree. Also, as shown in Figure 5(a), it is faster to create the index for a chunked layout compared to a contiguous layout, because reading the geographic dataset is more expensive for the logical chunks in the contiguous layout.

For the experimental dataset, and for 800 chunks, the R*-tree library[1] created a 73KB index file, while the size of the dataset is 128MB. Also, an HDF file can contain several swath structures, each with its own latitude, longitude and time information, and a swath can contain several multi-dimensional datasets with sensor values. An index is therefore needed for each swath, not for every dataset. Therefore, the index file does not require a significant amount of disk storage compared to the size of the dataset it is indexing.

For a given set of keys (hyper-rectangles), the structure of an R*-tree is nondeterministic because changing the order keys are inserted into the tree can change the tree structure. To build better trees to achieve better query performance, the R*-tree insertion algorithm removes and re-inserts keys from a node with more than the allowed maximum number of keys (an *overflowing* node), to keep the bounding rectangle for the node small [2]. As shown in Figure 5(b), we evaluated the the tree creation algorithm using different key insertion orderings, looking at a row major ordering, Hilbert curve ordering, and a random ordering of the keys. A Hilbert curve is a space filling curve that visits every point in a rectangular grid, preserves spatial locality in multiple dimensions [13]. The results show that the Hilbert ordering reduces the time needed to create the R*-tree. The random ordering shows similar performance to the Hilbert ordering for a chunked layout, however for a contiguous layout the random ordering performs worse than the Hilbert ordering, because it performs more disk seek operations.

We also measured the time to perform a range query with the trees built from the three different insertion orderings. Those results showed that all of the orderings produced trees that took about the same amount of time to search, and searches are very fast (at most a couple of msec), taking much less than the time required to read the data specified by the results of the R*-tree search.

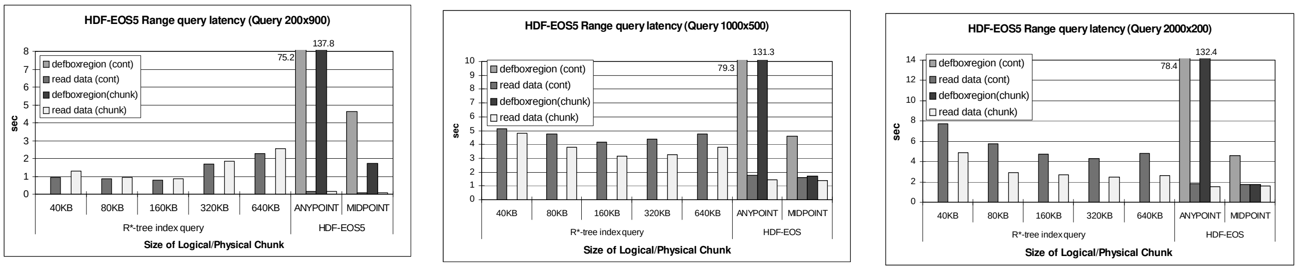
Because the performance results were very similar for both the HDF-EOS5 and HDF-EOS4 libraries, we only show results for the HDF-EOS5 library. Figure 6 shows the time to read a subset of the dataset for the three queries, using both the indexing library range query function and the HDF-EOS5 standard range query functions. The time for the indexing library includes both searching into the R*-tree and reading the geographic and sensor value data from disk. As we described earlier, the HDF-EOS library has two separate functions to perform a range query, so there are two bars in the graph for each data layout (contiguous and chunked).

For a single query, the extractregion functions in HDF-EOS5 and HDF-EOS4 read only a subset of the sensor value dataset. But the corresponding defboxregion functions read every element in the geographic datasets to determine the file location information for the requested region in *any-point* mode, and read either one or two columns of the geographic data in *mid-point* mode or *end-point* mode, respectively. For the three queries in the experiments, the HDF-EOS defboxregion function returns an empty region in *end-point* mode, so no results are shown.

The indexing library range query function reads the R*-tree index file (if the index has not already been read into memory), and the chunks of the sensor value and geographic dataset returned by the R*-tree search. The geographic data can be used to filter the sensor data that is returned, but does not lie in the query range. As seen in Figure 6, the time to perform the extractregion operation in the HDF library is less than the indexing library query time in most cases, but that is because the extractregion function only reads data from the sensor value dataset, and does not read the geographic information. The location information to determine which sensor values to read is computed by the defboxregion function, and when we look at the time to execute that function, we see that using the index library to perform the range query provides enormous performance benefits. Comparing the time to read the data in HDF-EOS5 *any-point* mode to that of the indexing library, for the 200x900 query the time for the indexing library was less than 3% of the HDF-EOS5 defboxregion time, for the 1000x500 query the indexing library time was less than 5% of the defboxregion time, and for the 2000x200 query the indexing library time was less than 7% of the defboxregion time. If the HDF library is used to select a region in *mid-point* mode, the performance is about the same or somewhat worse than that of the indexing library, but the indexing library should return a better approximation to the data that actually falls within the query range. Also, the indexing library is guaranteed to return all data elements that fall within the query region, but the HDF-EOS library in mid-point mode will not return a scanline with a mid-point element that does not fall within the query range.

As Figure 6(a) show, the performance of the indexing library for reading a region with many columns and relatively few rows decreases as the chunk size grows, because the indexing library range query function reads more unneeded

[1] We employ the HnRStar library, version 1.0 available at *http://research.nii.ac.jp/~katayama/homepage/research/srtree*

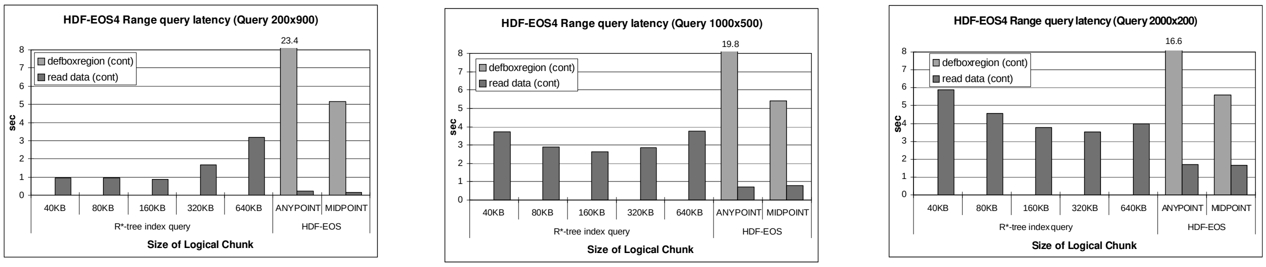


(a) 200x900 query

(b) 1000x500 query

(c) 2000x200 query

Figure 6. *Time for range queries with HDF-EOS5*



(a) 200x900 query

(b) 1000x500 query

(c) 2000x200 query

Figure 7. *Time for range query with HDF-EOS4*

data from disk. For this kind of query, the contiguous layout performs best because it does not cause many disk seeks, so gives about the same or even slightly better I/O performance than a chunked layout. The defboxregion function reads the geographic dataset one scanline (row) at a time, and that is very inefficient, especially when the dataset is stored with a chunked layout, since it will cause many disk seek operations to read each scanline. Therefore defining a region takes much longer with a chunked layout than a contiguous layout for this type of range query. For this type of query, the number of extracted elements is the same for both the indexing library range query function and the HDF-EOS query function.

We see from Figure 6(b) that for the second query that covers a mostly square region, the performance of the HDF-EOS extractregion function is worse than for the first query with many columns and few rows, because extractregion reads the entire scanline for every one that overlaps the query range, not just the elements in the query range.

For the third query with many rows and few columns, we see from Figure 6(c) that as the chunk size grows the time to read data for the indexing library decreases, because of fewer disk seek operations. In the best case, even though the indexing library function must also read the geographic dataset, which is done by the defboxregion function in the HDF-EOS library, the indexing library function takes about the same time as extractregion for a chunked layout. This is because extractregion reads a large amount of unneeded data, as was the case for the second query. For the third query, the amount of unneeded data read by extractregion is even larger than for the second query.

Even though the amount of unneeded data read by the indexing library is usually less than for the HDF-EOS library, it is still necessary to filter the unneeded data. When the size of the chunks is small, filtering is not expensive, but the R*-tree search time will be long because of the large number of leaf nodes in the tree. However, R*-tree search time is very small compared to the time to read the datasets from disk.

In our experiments, as the chunk size grows larger, performance decreases because the indexing library reads extra data that is outside of the query range. And if the chunk size is too small, performance also decreases because of additional disk seeks. However, overall the indexing library shows much higher performance than HDF-EOS *anypoint* mode, and better performance than *mid-point* mode for many queries, despite the indexing library performing the filtering needed to remove unnecessary data using geographic information, which is not provided by the HDF-EOS library.

4 Conclusion

We have shown that I/O performance can be improved with the use of both semantic indexing structures and data chunking, for navigating through multi-dimensional self-describing scientific datasets. However, for many scientific data formats no semantic indexing library has yet been developed. In the near future, we plan to extend our generic indexing library to work with various other self-describing scientific data formats, such as netCDF [17]. Our ultimate goal is to automate the process of generating indexing structures for various self-describing scientific datasets, using meta-information that can be automatically extracted from the metadata within the datasets, augmented with semantic information provided by developers or users of the scientific data file formats.

References

[1] Planetary data system data preparation workbook. Technical Report JPL D-7669, Part I, Jet Propulsion Laboratory, California Institute of Technology, February 1995.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD90)*, pages 322–331, May 1990.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.

[4] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[5] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces – index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, Sept. 2001.

[6] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, Sept. 2001.

[7] E. W. G. D. C. Wells and R. H. Harten. FITS: A flexible image transport system. *Astronomy and Astrophysics Supplement Series*, 44:363–370, 1981.

[8] M. Folk. A white paper HDF as an archive format: Issues and recommendations. http://hdf.ncsa.uiuc.edu/archive/hdfasarchivefmt.htm, January 1998.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD'84*, pages 47–57. ACM Press, May 1984.

[10] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[11] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD97)*, pages 369–380, May 1997.

[12] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying very large multi-dimensional datasets in ADR. In *Proceedings of the 1999 ACM/IEEE SC99 Conference*. ACM Press, Nov. 1999.

[13] B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, January/February 2001.

[14] NCSA, University of Illinois, Urbana-Champaign. *HDF User's Guide, Version 4.1r5*, November 2001.

[15] NCSA, University of Illinois, Urbana-Champaign. *Introduction to HDF5*, April 2001.

[16] Raytheon Systems Co., Upper Marlboro, MD. *HDF-EOS Users Guide for the ECS Project, Volume I: Overview and Example*.

[17] R. Rew, G. Davis, and S. Emmerson. NetCDF User's Guide: An Interface for Data Access, version 2.3, 1993.

[18] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings Supercomputing '94*, pages 650–659. IEEE Computer Society Press, Nov. 1994.

[19] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *The VLDB Journal*, pages 507–518, 1987.

[20] C. T. Shock, C. Chang, B. Moon, A. Acharya, L. Davis, J. Saltz, and A. Sussman. The design and evaluation of a high-performance earth science database. *Parallel Computing*, 24(1):65–90, Jan. 1998.

[21] M. S. Sunita Sarawagi. Efficient organization of large multidimensional arrays. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 328–336. IEEE Computer Society, February 1994.

[22] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, New Orleans, U.S.A., 1996.