A comparison of data file and storage configurations for efficient temporal access of satellite image data

A. K. Bachoo*, F. van den Bergh*, A. Gazendam†

*Remote Sensing Research Unit, Meraka Institute, CSIR, South Africa
†High Performance Computing Research Group, Meraka Institute, CSIR, South Africa

ABSTRACT

Satellite data volumes have seen a steady increase in recent years due to improvements in sensor technology and increases in data acquisition frequency. The gridded MODIS data products, spanning a region of interest of approximately 10° by 10° for a single tile, are stored as images containing almost six million pixels, with data in multiple spectral bands for each pixel. Time series analyses of a sequence of such images in order to perform automated change detection is a topic of growing importance. Traditional storage formats store such a series of images as a sequence of individual files, with each file internally storing the pixels in their spatial order. Consequently, the construction of a time series profile of a single pixel requires reading from several hundred large files, resulting in substantial performance overheads that severely constrain high-throughput analyses. We aim to minimize this performance limitation by restructuring the storage scheme for typical satellite imagery as temporal sequences in order to reduce overheads and improve throughput. Models are developed to compute the expected query time for both the time-sequential and the traditional image-based representations. These models are used to demonstrate the benefits of using a time-sequential representation. Four data structures (using the Hierarchical Data Format (HDF5), Network Common Data Format (netCDF) and a native file system approach) are implemented and compared in a series of experimental read tests to determine which format is most appropriate for implementation in the CSIR Cluster Computing Centre's facilities.

KEYWORDS: HDF5, netCDF, satellite time series storage

1 INTRODUCTION AND BACKGROUND

The need for efficient storage of sequences of satellite raster data has been growing in the scientific community for a number of years. As imaging technology improves, we are faced with an influx of huge amounts of newly acquired data, with volumes growing almost exponentially. The problem of storage is further compounded by the need to rapidly access and process subsets of the stored data. In the remote sensing field, satellite images are generally acquired periodically over time and stored in large archives spanning several terabytes in volume. The management of these sequences of images can be divided into two groups: database management systems (DBMS) and data files.

Storage of images in a DBMS involves time-consuming import and conversion of raster images into the database's internal storage format. The internal data model is generally a relational one where data is viewed as a structured table. This format is not suitable for the storage of large and complex multi-dimensional discrete data such as image sequences. Databases that provide support for large-scale image data store pixels in tables or fixed size binary large objects (BLOBs). A BLOB is a sequence of unformatted bytes and, as a result, not well-suited for compatibility with the natives types in a DBMS. In spite of these limitations, a database provides the structured query language (SQL) that simplifies data selection for complex queries. A common approach for managing a sequence of images stores the image names and metadata in a database while the actual images are stored externally on disk in its original format. Although this method can efficiently process metadata queries, it still leaves the retrieval of the bulk of the data up to the user. Scientific applications with extensive storage requirements require the integration of databases with related technologies in order to serve scientists sufficiently [1, 2, 3].

Specialized database technologies for raster storage have been proposed in several scientific papers. Reiner *et al.* show that an image tiling scheme is the most appropriate strategy for handling very large image data in a database [4]. Images are split into sub-images which are stored in the database. When an area of interest is requested, only the sub-images that contain the relevant pixel data are retrieved, resulting in substantial I/O bandwidth savings. Baumann *et al.* combine image tiling, spatial indexing and data compression into their raster database [5, 1]. Compression improves disk input/output (I/O) bandwidth efficiency by reducing the number of bytes that must be written/read to/from disk. The concept of spatial

Email: A. K. Bachoo `abachoo@csir.co.za`, F. van den Bergh `fvdbergh@csir.co.za`, A. Gazendam `agazendam@csir.co.za`

indexing allows quick retrieval of the identifier and location of a required tile. The CONCERT architecture uses variable length sequences of constant page sizes to store image tiles [6]. These page ranges allow a single linear address space to be accessed directly. Data buffering is controlled using memory mapping of disk pages.

Large n-dimensional arrays are not intrinsically supported by databases. Consequently, databases do not support the efficient mapping of a multidimensional array to 1-dimensional linear space. Hence, file storage of large satellite imagery is an attractive option for data management and retrieval. These files can be the original image format or another type, *e.g.,* netCDF. Storage of satellite images in data files means the data can be represented as huge multidimensional arrays. Scientific applications and large scale data processing benefit from a storage scheme of this sort since the data is easily accessible. The overhead incurred by using a database is avoided because the file metadata becomes the "manager". Several data files can also cumulatively store terabytes of information which makes them popular in the scientific community.

General purpose data formats for large scale storage, such as HDF5 and netCDF, are designed to be platform independent, self-describing and support the storage of large multidimensional arrays [7, 8]. They share some similarities with a database system in the sense that they have a schema for metadata and data manipulation strategies. In previous studies, the parallel implementations of HDF5 and netCDF have been compared in a series of experiments. Li *et al.* compare parallel netCDF and HDF5, concluding that parallel netCDF achieves higher parallel performance than HDF5 [9]. In contrast, other researchers show that the two file formats are, in fact, comparable in performance [10].

Several experiments have been conducted in the domain of large array storage and the optimization of the use of I/O bandwidth. The effect of array chunking on I/O performance within the context of the HDF file format has been reported by Velamparapil [11]. He shows cases where data chunking improves or degrades I/O performance. Sarawagi and Stonebraker present several techniques for efficiently organizing multidimensional arrays in POSTGRES [12]. These methods include partitioning of arrays and array duplication for different query patterns. Array duplication, with different data layout schemes, provides a general way to address most query types without impacting negatively on I/O performance, but inevitably leads to increased storage requirements. Seamons and Winslett also implement array chunking, clustering of data and clustering of different data types on disk for efficient I/O of arrays [13]. During clustering of data, arrays that are used together are placed in close proximity to each other on disk; clustering of data types works similarly. An implementation of a scientific data manager is presented by Choudhary *et al.* [14]. This system uses a database to store metadata — search patterns, access history and file offsets — and files to store the data.

2 PROPOSED TIME-SEQUENTIAL DATA STRUCTURE

Sequences of images stored in discrete files on disk in their original 2D ordering (called the "image stack" representation in the sequel) are not efficient for time series analysis due to the I/O overhead incurred when constructing a 1D profile through time. Hence, a specialized per-pixel, time sequential data model and data storage method must be implemented for improved I/O efficiency. The time series data will be stored in a large single data file.

To clarify the differences between the image stack and the time-sequential representations, a pair of equations will be presented to calculate the index of an arbitrary element within both of these representations. Let n denote the number of images in the sequence, or, equivalently, the number of time steps in the time series. Let w and h denote the width and height (in pixels) of the original 2D images. For convenience, define the total number of pixel locations, m, as $m = w \times h$.

Let $I(x, y, t)$ represent the index where the value at location (x, y) and time t can be located within a given representation. The image stack representation calculates the position of this element as

$$I_{\text{is}}(x, y, t) \triangleq (wy + x) + wht. \tag{1}$$

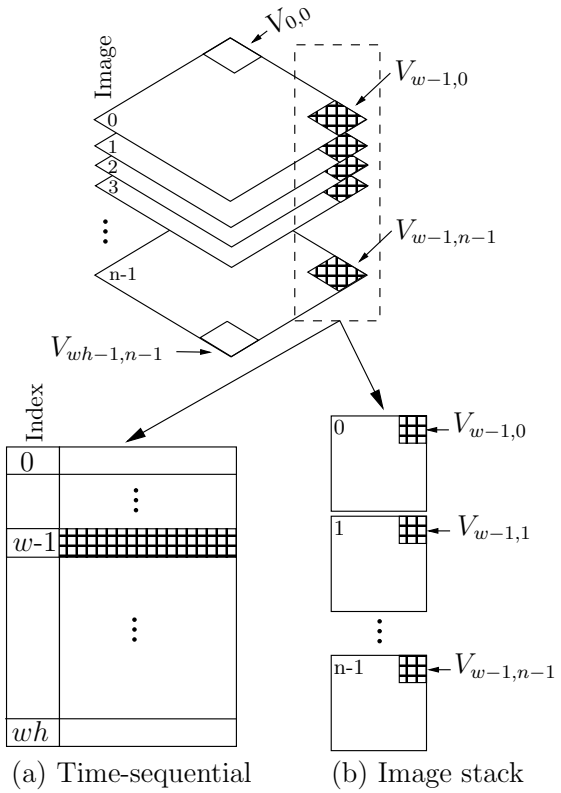
The index of an element in the time-sequential representation is computed using

$$I_{\text{ts}}(x, y, t) \triangleq (wy + x)n + t. \tag{2}$$

The term $(wy + x)$ serializes a 2D (x, y) spatial coordinate into a single contiguous index, called the *pixel index* in the sequel. Since this term is common to both Equations 1 and 2, it is convenient to refer to the value at index $I(x, y, t)$ as $V_{wy+x,t}$. Examples of the two representations are illustrated in Figure 1. In short, the time-sequential representation stores all the values belonging to a single spatial location as a contiguous block, while the image stack representation stores one time step of a sequence of spatial neighbours (one row of pixels) as a contiguous block.

A consequence of this representation is that 2-dimensional queries (*e.g.,* extracting a rectangular region on a map) will be decomposed into a set of row queries, where each query will specify a subset of the span of pixels corresponding to that row. This implies that intra-row contiguity of the data may be advantageous, since low-level read ahead and caching can be exploited.

A recent development on the mass storage market is the introduction of consumer-grade Solid State Drives (SSDs), which are potentially poised to replace mechanical hard drives in the near future. One of the most attractive features of these devices is they have very low, constant access times. This feature,



(a) Time-sequential (b) Image stack

Figure 1: Illustrative example of the differences in the ordering of the elements between the time-sequential and image stack representations

combined with their higher average throughput figures, may cause us to reexamine they way in which we structure our data.

These SSDs are typically built using banks of NAND-based flash memory, which unfortunately implies that data must be read in blocks, rather than using a pure random access addressing scheme. These blocks, called *pages*, are typically 512 [15], 2048 or 4096 bytes in size, thus resembling the disk blocks of a mechanical hard drive. This implies that an SSD disk functions just like a mechanical hard drive in that the smallest amount of data that can be retrieved with a single read is 512 (or even more) bytes.

Will a time-sequential data representation still be a good choice when an SSD is used instead of a mechanical hard drive? To answer this question, the potential impact of SSDs on data representation formats will be investigated in Section 3.

3 MODELING EXPECTED QUERY TIMES

Table 1 provides the definitions of the symbols used in equations below. Using these definitions, the approximate query time for the image stack representation is given by

$$D_{\text{is}} = \left(\left\lceil \frac{wS_t}{nS_b} \right\rceil \cdot \frac{T_b Q_h}{2} \right) n + T_r n + \left(\left\lceil \frac{Q_w S_t}{nS_b} \right\rceil \cdot \frac{T_b Q_h}{2} \right) n \quad (3)$$

This equation holds provided that the operating system read-ahead buffer is larger than two scan lines of the image, *i.e.*, $2wS_t/n < S_{ra}$. The Linux operating system kernel uses a default value of $S_{ra} = 128$kB, so it is not uncommon for the read-ahead buffer to be able to hold multiple scan lines.

Table 1: Definition of symbols

Symbol	Description
Q_w	The spatial width of a query, in pixels.
Q_h	The spatial height of a query, in pixels.
T_r	The average time required to access a given location in a file, including the time required to fill the operating system read-ahead buffer, in milliseconds.
T_b	The average time required to read a file system block of size S_b, in milliseconds.
S_b	The size of a file system block, in bytes.
S_t	The length of a single time-series, in bytes.
S_{ra}	The size of the operating system read-ahead buffer, in bytes.

Equation 3 is best understood by studying its components separately. Each time step of the time series is stored in a separate "image", thus the storage device must seek to the starting position of the query block within this image once for every time step. This yields the $T_r n$ component, which contributes one seek plus one read-ahead buffer fill operation for each step. The rest of the scan lines in this "image" block will be read in pattern where the operating system will alternately enable and disable the read-ahead after each seek operation [16, p427]. This implies that half of the lines in the query block will cause the operating system to fill the read-ahead buffer by reading wS_t/n bytes. On every other scan line the operating system would have disabled read-ahead, thus reading only $Q_w S_t/n$ bytes, rounded up to the nearest disk block. Note that the operating system read-ahead ensures that at least one scan line is always buffered, which implies that the query time is almost independent of the width of the query.

The expected query time for the time-sequential representation is modelled as

$$D_{\text{ts}} = T_r \cdot Q_h + \begin{cases} 0 & \text{if } Q_w S_t \leq S_{ra} \\ \left(\frac{Q_w S_t - S_{ra}}{S_b} \right) \cdot T_b \cdot Q_h & \text{if } Q_w S_t > S_{ra} \end{cases} \quad (4)$$

The first part of this equation can be derived in a straightforward manner: if the equivalent of a scan line of the query is smaller than the operating system read-ahead buffer, then no additional reads have to be performed, and the query time is simply equal to $T_r \cdot Q_h$. The second part of the equation just adds to this the time it would take to read any additional blocks outside of the region already covered by the read-ahead buffer.

The time required to perform a query using the image stack representation can be expressed relative to that of the same query performed using the time-

sequential representation by forming the ratio

$$D_{\text{relative}} = \frac{D_{\text{is}}}{D_{\text{ts}}} \tag{5}$$

4 STORAGE FORMATS

A number of file formats are available for multidimensional data storage. We consider HDF5, netCDF and a native file system approach for the implementation of the time-sequential data structure representation.

4.1 HDF5

HDF5 is a well-known data format for optimized and portable storage of multidimensional data. The HDF5 data model has two primary types of objects: *data sets* and *groups*. Data sets are arrays of multiple dimensions where a cell is a simple or compound HDF5 data type. Groups allow for the creation of data dependencies which can be represented as a directed graph with a stipulated entry point. Most data models (*e.g.,* TIFF and netCDF) can be represented by the HDF5 data model. Its software drivers are designed to store, retrieve and manage complex data in heterogeneous environments that are constantly evolving. At the application level, an HDF5 file is treated as a single file, but the library allows an implementation to combine multiple files from the file system transparently. Typically, metadata and raw data are stored in separate files; this makes it possible for each file to reside on different file systems.

The HDF5 data format has several key features that are advantageous for large scale data storage. The data file size and the number of objects stored can be unlimited. This is useful for storing a long sequence of large satellite images. An important aspect of data exchange is portability across different architectures. This is achieved by storing data type characteristics (size, bit order and architecture) in the file. The HDF5 data model also contains a comprehensive set of data types. As a result, external storage of raw data and the creation of new data types is well supported. The complexity of a user-defined data type does not have any limitations. This makes the data model highly generalized. Data sets may also be extended along any dimension.

Spatial selection of uniform and non-uniform array subsets is supported using set operations defined in the I/O library. Data types can also be subsetted, *e.g.*, selecting an integer from a compound type. Several performance enhancement options are provided in the HDF5 data model. Two important options are data compression and chunking. Chunking is analogous to tiling an image or array. HDF5 requires that all chunks in a data set must have the same fixed size. Chunking can also be used to solve the problems associated with random access to compressed data sets, since each chunk is compressed independently of all other chunks.

The following HDF5 structure is proposed:

1. A single root group is created, *i.e.*, the default root node.
2. Global variables are stored in the root group mentioned above. The most basic global variables are the image width and height. Other fields may also be stored, such as the image projection information.
3. Each image band, acquired over time, is represented as a 2-dimensional array data set that is a child of the root node. Hence, storing n bands will imply the creation of an HDF5 file with n data sets. Given a single 2-dimensional array data set, a column corresponds to a spatial snapshot in time. A row represents the evolution of a pixel through time. The row index is simply the *pixel index* defined earlier. In this storage scheme, band data is separated so that additional bands, if required, can be added to the file at a later stage. Each array data set is implemented as extensible (unlimited size) with chunking enabled by default.

The above storage structure improves compression in some cases since pixel values are spatially correlated within each band. Data may also be stored using a compound data type. In this case, all the band data at a pixel location for one time step is grouped into a single data element that has multiple fields. An advantage of this configuration is that only a single data set needs to be created and, thus, only a single read is required for a time series rather than n reads for n data sets.

Chunking can improve performance by reducing the number of individual read operations that have to be performed, but the following factors have to be considered:

- When a region of interest is queried, all chunks containing the pixel data will be retrieved off disk. Thus, a large performance penalty is incurred whenever a comparatively small ratio of queried pixel falls on a chunk.
- Creation of new chunks, *e.g.*, when extending an array, allocates storage for the entire chunk irrespective of whether it is completely populated or not.
- A large number of chunks will incur some overhead with respect to metadata volume, *e.g.*, the size of the B-tree used for indexing chunks.

4.2 netCDF

netCDF is a self-describing platform independent data format for exchanging scientific data. We consider only netCDF3 in this paper.

NetCDF encompasses multidimensional data in regularly spaced grids. The file has two parts — a header and array (variable) data. The header stores information dealing with dimensions, attributes, *etc.*, while the array data section contains the payload. Some limitations inherent to the netCDF format are:

- sizes larger than 2 GB (or 4 GB) are occasionally problematic to implement.

- only one dimension may be extensible.
- not suitable for hundreds or thousands of variables.
- multiple variables (arrays) can share an unlimited dimension but they must grow together.
- does not have a hierarchical group based organization.
- has a limited number of data types.

These constraints allow for a contiguous layout with very little overhead.

Variable size arrays are supported by introducing record variables that share the same unlimited dimension. The other less significant (and fixed) dimensions define the shape for one record of the variable. In our case, a variable is a sequence of images for a single band and the record is a single image band captured at a point in time. To allow the variable to grow in the unlimited direction, the fixed size records are interleaved along the unlimited dimension.

We use the netCDF 64-bit offset format for managing files sizes that are greater than 4 GB. When implementing a data cube using netCDF, the unlimited dimension must be the first dimension declared in the variable definition. In the case of managing time series image data, the unlimited dimension is time. As a result, it means that an entire image will be stored in a row. Selecting a column will produce a time series signal for a particular pixel. Unfortunately, such a layout will produce a representation that is very similar to the image stack representation introduced in Section 2, and as such is expected to be inefficient for this type of query. This problem is resolved by creating a fixed time dimension while declaring the number of pixels as an unlimited dimension. On disk, row data (a time series) is then interleaved. A spatial query, as in HDF5, will be decomposed into a set of row queries. The netCDF file is structured in the same way as the default HDF5 — we created n variables (array or data sets) for the n bands that we wish to store.

5 FILE SYSTEM DATA STRUCTURE

File systems offer a reasonably good model for storing data in the time-sequential model proposed in Section 2. Firstly, file systems have efficient mechanisms for locating a specific file. Given a file name, the file system will perform a look-up, usually in an efficient data structure such as a B-tree, to identify the first block of the file on disk. Secondly, file systems are naturally able to store data elements that may grow in size over time. This greatly simplifies the process of appending new data to the existing structure.

The file system data structure is implemented by storing each time series in a separate file. The file name is derived directly from the pixel index number, *e.g.*, the sequence $V_{k,0} \ldots V_{k,n-1}$ could be stored in a file called 'k'. One possible downside to this implementation is the large number of files that are created when the data structure contains many pixels. The 2400×2400-pixel data set used in Section 7 contains approximately 5.8 million pixels, which therefore requires 5.8 million individual files in this representation. Most file systems can not efficiently store this many files in a single directory, so a 3-level directory structure was selected to limit the number of sub-directories or files per directory to a maximum of 500.

The internal representation of each file is similar to the one used by the compound HDF5 data structure. Each time step is represented as a tuple containing all the bands; these tuples are stored sequentially inside the file.

This file system data structure trades off storage space efficiency for simplicity of implementation. Firstly, each pixel-file must consume at least one file system block, which may vary from 512 bytes to several kilobytes. If, for example, the length of the data associated with a single time series is 2512 bytes, it will consume 2560 bytes, 3072 bytes, or 4096 bytes using 512, 1024 or 2048 byte file system blocks, respectively. The unused space in each disk block is called *slack space*, and may result in substantial inefficiency.

The three-level directory structure also introduces an additional penalty. Each sub-directory is nominally stored as a special file on the file system, so to traverse a 3-level directory structure requires that at least four separate metadata blocks must be read. This increases the amount of I/O bandwidth spent to retrieve a single time series, lowering overall I/O efficiency. Since the metadata blocks are likely to be distributed over the disk, it also introduces additional seek delays. Despite all these drawbacks, the file system-based data structure is elegant in its reuse of existing implementations to solve some of the problems associated with a per-pixel representation that allows for new data to be appended at the end of the time series.

6 FILE SETUP

Default settings were used to configure the various file formats. These parameters are described in more detail in the HDF5 and netCDF reference manuals. The native file system contains binary data in multiple flat files and does not have any adjustable parameters. The data structures are all implemented on top of the Zettabyte File System (ZFS), and were accessed over a Gigabit Ethernet network using the NFS version 3 protocol.

7 MODEL VERIFICATION AND EMPIRICAL STUDY

Experiments were conducted on the *opteron* cluster in the C4 facility of the Council for Scientific and Industrial Research (CSIR). This particular cluster is composed of 46 compute nodes, with each node hosting 4 GB of RAM and four 2.6 GHz AMD Opteron cores. The nodes are connected via a non-blocking Hewlett Packard Procurve 2848 switch.

The image sequence consisted of 314 MOD09A1 data product images with dimensions 2400×2400 at 500m resolution. The data structures were populated

Table 2: Average query time (seconds) for the image stack representation storage format

Spatial subset	Model Prediction	Experimental Measurement
1×1	2.144	2.006±0.035
3×3	2.262	2.124±0.037
100×100	8.229	8.400±0.293
50×200	5.265	5.570±0.069
200×50	14.149	14.353±0.298

Table 3: Average query time (seconds) for the time-sequential representation storage format

Spatial subset	Model Prediction	Experimental Measurement
1×1	0.016	0.008 ±0.005
3×3	0.048	0.021 ±0.011
100×100	1.829	1.861 ±0.047
50×200	1.145	1.158 ±0.032
200×50	3.216	3.250 ±0.060

with image bands 0, 7 and 12, corresponding to surface reflectance (16 bits per sample), date flags (16 bits per sample) and quality flags (32 bits per sample) respectively. By analysing the 2D image representation and general types of spatial queries for the MODIS images, a set of 5 spatial access patterns were defined for experimental analysis. This resulted in block sizes of 1×1, 3×3, 100×100, 50×200, and 200×50 pixels. Given a single spatial extent as described above, the entire time series is retrieved from a data structure (314 time steps) for the given block of (x, y) coordinates. The effects of file caching are avoided by reading data only once from each position in a data structure for an experiment. We achieve this requirement by partitioning the data structure into 64 non-overlapping regions (corresponding to blocks of 300×300 pixels in image coordinates); queries within each of these blocks are also guaranteed to be non-overlapping. Each test run produces 64 timing results for each of the 5 block sizes specified above.

7.1 Comparison of spatial and time-sequential representation

The models presented in Section 3 were verified in an actual implementation, after which the models were used to analyse the expected impact of SSDs on representation preferences.

7.1.1 Model verification

To evaluate Equations 3–5, concrete values have to be selected for the various parameters. Choosing these parameters to coincide with MOD09A1 data sets yields values of $w = 2400$, $h = 2400$, $n = 314$ and $S_t = 2512$. The hardware used in this experiment was determined to have a sustained transfer rate of approximately 52 MB/s. The file system was configured with a block size of $S_b = 4096$, thereby yielding

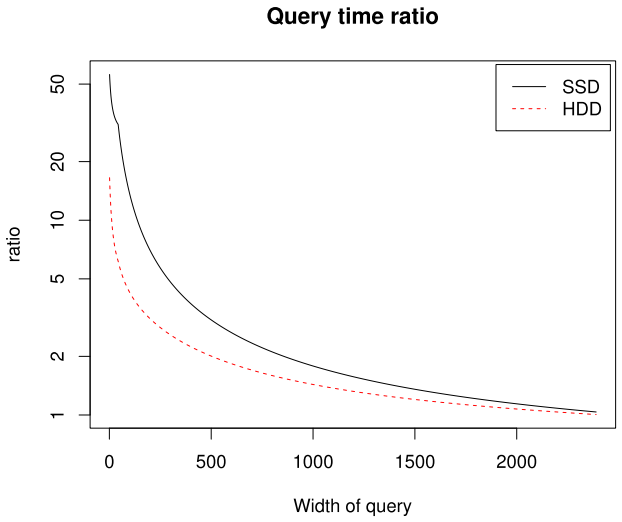


Figure 2: A plot of D_{relative} (Equation 5) over a range of query sizes.

$T_b = 0.0754$. To evaluate Equation 3 the value of T_r is set to 6.64 ms, which was obtained by measuring actual file seek operations with a spacing derived from Equation 1. Similarly, the value of T_r in Equation 2 was measured to be 16.085 ms by using the seek spacing dictated by Equation 2. These values differ because the effective seek time of a mechanical hard drive depends on the physical distance that the drive head must travel.

Tables 2 and 3 demonstrate that the models presented in Equations 3 and 4 are able to predict the expected query times reasonably well. The columns labelled "experimental measurement" represent the mean query times measured on a direct implementation of the two representations. These values were averaged over the 64 trial queries in different spatial regions of the structure. The operating system read cache was flushed between each of the 64 queries to ensure statistical independence of the values.

Having established that the models are able to predict the expected query times with a reasonable degree of accuracy, we can now simulate the expected query times of a solid state drive. Assume that our ideal SSD has a transfer rate of 200 MB/s, yielding $T_b \approx 0.0195$. Furthermore, we can estimate the value of T_r to be $T_r \approx S_{ra} T_b / S_b = 0.625$ms. This is based on the assumption that there is no delay in accessing any particular location on the SSD, and that the operating system read-ahead buffer is the same size for both mechanical hard drives and SSDs. Figure 2 illustrates the resulting values of Equation 5 for both the mechanical hard drive, and the ideal solid state drive. Note that the query height was set equal to the query width at each point on the graph. From the graph it clear that the image stack representation is slower than the time-sequential representation for all queries, but that the difference is more pronounced for smaller query sizes. The most surprising observation is that the difference between the two representations is more pronounced on the SSD than on the mechanical hard drive.

Table 4: Average query time (seconds) for image stack and time-sequential representation storage formats

Spatial subset	Per-pixel representation	Image stack
1×1	0.048±0.061	19.870±0.044
3×3	0.066±0.035	19.729±0.033
100×100	3.679±0.841	29.740±0.373
50×200	2.249±0.396	26.320±0.490
200×50	4.597±0.488	135.644±5.069

7.1.2 Experimental results

In the preceding section the performance of an ideal direct implementation of the image stack representation was analysed. In practice, the image stack is typically implemented using individual files for each of the time steps — here we investigate the real-world performance of this representation using 314 MOD09A1 images. To facilitate later comparisons, the same queries were executed on an HDF5 time-sequential data structure. The results presented in Table 1 clearly show the advantage of the time-sequential representation. Note that even in the worst case, the time-sequential representation is faster than the traditional image-based structure by a factor of 8 on the 100×100 query. This figure is almost double that of the value obtained in the more direct implementation of the image stack in Section 7.1.1; the additional delay observed here is caused by the overhead of opening all the image files repeatedly, amongst other factors.

7.2 Comparison of time-sequential data structures

In Section 7.1.2 it was established that a time-sequential representation offers a performance advantage over an image stack representation for the types of queries considered. Having decided on a time-sequential representation, it turns out that there are many different ways in which this representation can be realised. This section will investigate the relative performance of the following four time-sequential implementations :

1. An HDF5 implementation using separate data sets for each image band, with a chunk size of 1×314, denoted H5;
2. An HDF5 implementation using a compound data type to group the band data, also with a chunk size if 1×314, denoted H5C;
3. A netCDF implementation, denoted NC, and
4. A native file system implementation, as described in Section 5, denoted FS.

The underlying storage device may have an impact on the relative performance of these implementations, so four storage configurations were investigated. A network attached storage device implementing the ZFS file system on a number of hard drives in a RAID[1] configuration was selected for this experiment. Two RAID-1 configurations were tested: striping over two drives (S2), and striping over 3 drives (S3). In addition to these two RAID configurations, the ZFS on-the-fly compression option was included as another experimental parameter. To mitigate the effects of the physical location of a file on a particular drive, a second replication of each partition was created. The number of configurations in this experiment is therefore 4 file formats × 2 RAID striping options × 2 ZFS compression options × 2 replications, for a total of 32 tests.

[1]Redundant Array of Inexpensive Disks

Table 5: Raw sequential I/O throughput of the various partitions

Partition type	Throughput (MB/s)
S2 uncompressed	56.78±0.88
S2 compressed	69.06±2.49
S3 uncompressed	83.32±0.96
S3 compressed	81.39±0.30

A simple bandwidth test was performed on each of the four partitions. From Table 5 is can be seen that striping over three drives leads to better performance than striping over two drives, as could be expected. The specific file that was used to perform the bandwidth tests on (the 15 GB H5 file) appears to have been rather compressible, which is reflected in the net increase in throughput seen on the S2 compressed partition.

To simplify the presentation of the results of the query experiments performed on the four different formats, the spatial queries were grouped into *small* (1×1, 3×3) queries and *large* (100×100, 50×200, 200×50) queries. The query times were all normalised to represent the mean time to retrieve a single time series, which allowed the query times to be combined within each of the two groups. The results of the small query experiment are presented in Table 6.

The ranking in Table 6 offers few surprises, except perhaps that the FS format performed better than the H5 format. It would appear that the separate data sets used in the H5 format introduces more overhead than that incurred by the file system. These small queries are not very efficient: the fastest representation, NC, produced output corresponding to only 0.215 MB/s of the available bandwidth on the S3 compressed partition.

The results for the large query experiments are presented in Table 7. On these larger queries it becomes clear that the FS structure suffers from the considerable overhead of opening a file for retrieving each time series. Both the NC and H5C formats performed very well, with the NC format taking a small lead. The effective throughput figures are much more promising than those obtained on the small queries, with the NC format producing output at a rate of 11.97 MB/s, which corresponds to 14.7% of the maximum sequential throughput of the storage device.

Table 6: Mean query time (microseconds per time series) for small queries

	Data structure type			
Partition	FS	H5	H5C	NC
S2 uncompressed	25067 ± 6000	38249 ± 1855	15770 ± 585	14409 ± 1160
S2 compressed	18365 ± 1794	26283 ± 1013	14802 ± 461	14547 ± 1050
S3 uncompressed	20010 ± 1743	29808 ± 1082	13953 ± 741	12771 ± 819
S3 compressed	19767 ± 3050	24015 ± 484	13901 ± 360	11128 ± 1046

Table 7: Mean query time (microseconds per time series) for large queries

	Data structure type			
Partition	FS	H5	H5C	NC
S2 uncompressed	1650.4 ± 47.6	405.0 ± 21.3	246.6 ± 25.8	239.1 ± 2.9
S2 compressed	1436.4 ± 177	324.0 ± 19.5	248.8 ± 24.4	221.4 ± 2.6
S3 uncompressed	1251.9 ± 9.9	387.2 ± 18.7	232.4 ± 23.4	225.1 ± 2.0
S3 compressed	1246.5 ± 11.9	291.5 ± 20.3	218.9 ± 23.9	200.3 ± 2.2

8 CONCLUSION

We have shown that a time-sequential representation is superior to an image stack representation for storing satellite image time series data. This result was demonstrated by deriving suitable models of the expected query times for these two representations, and then confirming the model predictions with a set of experiments. The models were also used to investigate the expected impact that solid state drives will have on the choice of storage format. This comparison has shown that SSDs do not appear to behave differently from mechanical hard drives: time-sequential representations are still better than image stack representations, mostly because the image stack representation is bandwidth limited, rather than seek-time limited.

Amongst four possible options for storing a time-sequential representation, it was shown that the both the HDF5 format using a compound data type and the netCDF format are good choices. Based on these results, and the perceived bias towards the HDF format in the remote sensing community, it would be prudent to recommend the compound HDF5 representation.

Improved performance was observed on partitions that enabled the on-the-fly compression algorithms offered by ZFS. This improvement is likely due to the high degree of redundancy in the quality flag band of the MOD09A1 data. Future work will focus on strategies incorporating suitable compression algorithms into the HDF5 representation.

REFERENCES

[1] P. Baumann, E. Diedrich, C. Glock, M. Lautenschlager and F. Toussaint. "Large-scale multidimensional coverage databases". In *26th GITA Annual Conference*. 2003.

[2] J. Skiffington and K. McKelvey. "Raster in the database". In *GEOconnexion International Magazine*, pp. 22–23. 2007.

[3] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, D. DeWitt and G. Heber. "Scientific data management in the coming decade". *SIGMOD Record*, vol. 34, no. 3, pp. 34–41, 2005.

[4] B. Reiner, K. Hahn, G. Hofling and P. Baumann. "Hierarchical Storage Support and Management for Large-Scale Multidimensional Array Database Management Systems". In *Database and Expert Systems Applications : 13th International Conference*, pp. 689–700. 2002.

[5] P. Baumann, P. Furtado, R. Ritsch and N. Widmann. "The RasDaMan approach to multidimensional database management". In *Proceedings of the SAC'97*, pp. 166–173. 1997.

[6] L. Relly, H.-J. Schek, O. Henricsson and S. Nebiker. "Physical database design for raster images in CONCERT". In *Advances in spatial databases*, vol. 1262, pp. 259–279. Springer Berlin/ Heidelberg, 1997.

[7] R. Rew and G. Davis. "The Unidata netCDF: Software for scientific data access". In *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology*, pp. 33–40. 1990.

[8] C. Tan, J. Blais and D. Provins. *High Performance Computing Systems and Applications*, chap. Large imagery data structuring using hierarchical data format for parallel computing and visualization. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 2000.

[9] J. Li, W.-K. Liao, A. Choudary, R. Ross, R. Thakur, R. Latham, A. Siegel, B. Gallagher and M. Zingale. "Parallel netCDF: A high-performance scientific I/O interface". In *Supercomputing 2003*. 2003.

[10] "Parallel I/O performance study with HDF5, a scientific data package". The HDF Group, `http://hdf.ncsa.uiuc.edu/HDF5/`.

[11] G. Velamparapil. *Data management techniques to handle large data arrays in HDF*. Master's thesis, Graduate College of the University of Illinois, 1998.

[12] Sarawagi and Stonebraker. "Efficient Organization of Large Multidimensional Arrays". In *ICDE: 10th International Conference on Data Engineering*. IEEE Computer Society Technical Committee on Data Engineering, 1994.

[13] K. Seamons and M. Winslett. "An efficient abstract interface for multidimensional array I/O". In *Supercomputing 1994*, pp. 650–659. IEEE, 1994.

[14] A. Choudhary, M. Kandemir, J. No, G. Memik, X. Shen, W. Liao, H. Nagesh, S. More, V. Taylor, R. Thakur et al. "Data management for large-scale scientific computations in high performance distributed systems". *Cluster Computing*, vol. 3, no. 1, pp. 45–60, 2000.

[15] J. Kim, J. Kim, S. Noh, S. Min and Y. Cho. "A space-efficient flash translation layer for CompactFlash systems". *Consumer Electronics, IEEE Transactions on*, vol. 48, no. 2, pp. 366–375, 2002.

[16] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edn., 1992.