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

Asheer Bachoo[1], Frans van den Bergh[2], Albert Gazendam[3]

[1]Remote Sensing Research Unit, Meraka Institute, CSIR, South Africa, abachoo@csir.co.za

[2]Remote Sensing Research Unit, Meraka Institute, CSIR, South Africa, fvdbergh@csir.co.za

[3]High Performance Computing Research Group, Meraka Institute, CSIR, South Africa, agazendam@csir.co.za

## Abstract

*Recent improvements in sensor technology, together with increases in data acquisition frequency, have resulted in a surge in satellite data volume. A single tile from the gridded MODIS products, spanning a region of interest of approximately 10° by 10°, is stored as an image containing close to six million pixels, with data in multiple spectral bands for each pixel. Time series analyses of sequences 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. To construct a time series of a single pixel through time using such traditional storage solutions would require accessing hundreds of very large files, resulting in significant overheads which limit high-throughput analyses. We aim to reduce this bottleneck by restructuring the storage scheme for typical satellite imagery as temporal sequences in order to reduce overheads and improve throughput. 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.*

## 1 Introduction and background

The management of sequences of very large images can be divided into two groups: database management systems (DBMS) and data files. Using a DBMS, images are imported into and stored using the database internal format. The DBMS usually has a relational database model where data is



*ISBN 978-0-620-42117-1*

perceived as a structured table. This format is not suitable for the storage of large and complex multidimensional discrete data such as image sequences. The data model implies that pixels are stored in tables or fixed size binary large objects (BLOBs). BLOBs are unstructured sequences of bytes and, hence, not easily compatible with some of a DBMS's native types. User defined procedures for data analysis suffer the same fate (Abiteboul et al., 2005). Database and related technologies must be combined in order to serve the scientific world since support for efficient multidimensional data retrieval is limited (Baumann et al., 2003; Skiffington and McKelvey, 2007; Gray et al., 2005).

Techniques for raster data storage and efficient access in databases have been presented in the literature. Reiner *et al.* show that a tiling scheme is the most effective strategy for handling large image data in a database (Reiner et al., 2002). Data is split up into subimages and when a region of interest is requested, only the relevant tiles are accessed, resulting in significant I/O bandwith savings. Baumann *et al.* incorporate tiling, spatial indexing and compression strategies into their raster database (Baumann et al., 1997, 2003). Compression improves disk I/O bandwidth efficiency, while spatial indexing allows quick retrieval of the identifier and location of a required tile. The CONCERT architecture uses arbitrary length sequences of fixed page sizes to store image data (tiles) (Relly et al., 1997). These page ranges allow a single linear address space to be accessed directly. Data buffering is controlled using memory mapping of disk pages.

Databases do not intrinsically support large *n*-dimensional arrays, nor do they support efficient mapping of them to 1-dimensional space. Hence, flat file storage of large satellite imagery is an attractive option for data management and retrieval. The overhead incurred by using a database is avoided because the file metadata becomes the "manager". Several data files can cummulatively store terabytes of information which makes them popular in the scientific community.

Data formats such as HDF5[1] and netCDF are platform independent, self-describing and support the storage of multidimensional arrays (Rew and Davis, 1990; Tan et al., 2000). They can be viewed as database systems since they have a schema for metadata and data manipulation strategies. Previous comparisons between HDF5 and netCDF have looked at their parallel implementations: Li *et al.* compare parallel netCDF and HDF5 in a series of tests, concluding that parallel netCDF achieves higher parallel performance than HDF5 (Li et al., 2003). In contrast, other researchers show that the two file formats are, in fact, comparable in performance (Chilan et al., 2006).

Several experiments have been conducted in the domain of large array storage and its optimized I/O. Array chunking and its effects on I/O performance within the context of the HDF file format is reported in (Velamparapil, 1998). Sarawagi and Stonebraker (1994) describe methods for efficient organization of multidimensional arrays in POSTGRES. These methods include partitioning of

1 The HDF Group, http://hdf.ncsa.uiuc.edu/HDF5/

arrays and array duplication for different query patterns. Seamons and Winslett (1994) also implement array chunking, interleaving of data (clustering) and interleaving of different data types on disk for efficient I/O of arrays. An implementation of a scientific data manager is presented in (Choudary et al., 2000). 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 are not efficient for time series analysis due to the I/O overhead incurred when constucting 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.

Figure *1* illustrates the way data will be structured in the proposed time-sequential representation. Each spatial pixel coordinate *(x,y)* is mapped to a unique number $i = y \times C + x$, where *C* is the number of columns in the original 2-dimensional image. The entire time series at that coordinate is then stored as a row in the new table, as shown in Figure 1, where the columns represent the time dimension, and the row index corresponds to the pixel identifier *i*. Since the original 2-dimensional grid has effectively been serialised, 2-dimensional queries (e.g., extracting a rectangular region on a map) will now be decomposed into a set of row queries in the new table. Pixels that were horizontal neighbours in the spatial representation are now consecutive rows in the serialised representation, which implies that the contiguity of rows of pixels in the spatial representation is preserved. This allows operating system level read-ahead and caching to be exploited.

| | $t_0$ | $t_1$ | ... | $t_n$ |
|---|---|---|---|---|
| $p_0$ | $v_{0,t_0}$ | $v_{0,t_1}$ | | $v_{0,t_n}$ |
| $p_1$ | $v_{1,t_0}$ | $v_{1,t_1}$ | | $v_{1,t_n}$ |
| ... | | | | |
| $p_m$ | $v_{m,t_0}$ | $v_{m,t_1}$ | | $v_{m,t_n}$ |

Figure 1: Storing time series data seqentially per-pixel

## 3 Data structures

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 per-pixel data structure representation.

### 3.1 HDF5

The HDF5 data model consists of two primary types of objects: *datasets* and *groups*. Datasets are arrays of multiple dimensions where a cell is a simple or compound HDF5 data type. Groups facilitate the creation of data dependencies. The HDF5 data format supports unlimited file sizes and an unlimited number of objects, highly generalized data types, spatial set operations, performance options (e.g., chunking, compression and data shuffling), parallel I/O and unlimited dimension sizes. The following HDF5 structure is proposed:

1. There will be just the default root group i.e., just the root node.
2. Global variables, such as image height and width and projection information, will be stored in the header.
3. Each image band, captured 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* datasets. Band data is separated so that additional bands, if required, can be added to the file at a later stage. Arrays will be implemented as extendible (unlimited size). These arrays will be chunked.

Alternatively, all the bands at a single pixel location for a single timestep can be grouped as one element using an HDF5 compound data type. This results in a data structure having the same structure described above except that it will contain just a single dataset.

### 3.2 netCDF

NetCDF encompasses multidimensional data in regularly spaced grids. Only netCDF version 3 is considered in this paper, since netCDF version 4 is similar to HDF5. Some limitations inherent to the netCDF format are : i) sizes larger than 4GB are difficult to handle; ii) only one dimension may be unlimited in size and iii) limited number of datatypes. The strength of netCDF lies in its contiguous layout and its single header file, which means there is little overhead in the data management. Variable size arrays in netCDF are supported by introducing record variables. In our case, a variable is a sequence of time series profiles and the record is a single time series signal. To allow the variable to grow in the unlimited direction, the fixed size records are interleaved along the unlimited dimension. The netCDF3 64-bit offset was enabled to allow for file sizes greater than 4GB. The file is structured in the same way as the HDF5 - *n* variables (array or datasets) are created for the *n* bands that we wish to store. A spatial block query, as in HDF5, will be decomposed into a set of row queries.

### 3.3 Filesystem Data Structures

A file system provides an ideal mechanism to store time series data in a per-pixel fashion: simply store the entire time series associated with a given *(x,y)* coordinate in a separate file. An interface was developed to map a pixel coordinate to its corresponding pixel identifier, which is translated to a filename; this method leaves the bulk of the management of the data storage to the operating system. Since the data structure is expected to contain on the order of millions of files (each representing an entire time series at a given location), a three-level directory structure was created to avoid the expected performance degradation that a filesystem experiences when too many files are created in a single directory. Like with the compound datatype HDF5 data structure, the internal format of each pixel-file was a band-interleaved representation.

This type of data structure has several disadvantages: fixed size operating system disk blocks result in a significant amount of wasted disk space (slack space), a file has to be opened (and closed again) for every location read, and the three-level directory structure implies that at least four filesystem metadata reads must be performed to read each file. On balance, the strengths of this approach are its relative simplicity, good portability and the ease with which new data can be appended.

### 3.4 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.

## 4 Experimental results

Experiments were conducted on the CSIR's C4 cluster. A set of 314 MOD09A1 data product images were used in these experiments. Bands 0, 7 and 12 were imported into the data structures, corresponding to surface reflectance (16 bits per sample), date flags (16 bits per sample) and quality flags (32 bits per sample) respectively, all at 500m resolution. Five spatial access patterns, with respect to the 2D image representation, were considered for experimental analysis, resulting 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. To avoid the effects of file caching, each location in a given data structure is only read once in each experiment. This is achieved 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 thus produces 64 timing results for each of the 5 block sizes specified above.

### 4.1 Comparison of spatial and time-sequential representations

A performance baseline was established by performing the time series queries on the traditional image-based format. This approach involves opening each of the 314 files for every timestep of every query. 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 15.

Table 1: Mean query time (seconds) using a time-sequential data structure versus the original image format

| | **Data structure type** | |
|---|---|---|
| **Spatial subset** | **Time sequential** | **Original images** |
| 1×1 | 0.048 ± 0.066 | 33.524 ± 22.732 |
| 3×3 | 0.057 ± 0.070 | 31.174 ± 6.941 |
| 100×100 | 3.852 ± 0.595 | 131.070 ± 50.068 |
| 50×200 | 2.327 ± 0.369 | 188.956 ± 9.048 |
| 200×50 | 9.802 ± 2.637 | 149.638 ± 4.348 |

### 4.2 Comparison of time-sequential data structures

Having established the benefit of a time-sequential representation over an image-based representation, we now investigate the relative performance of four time-sequential formats. Four data structures are created and stored on a RAID[2] storage system. A stripe of 2 and 3 disks denoted S2 and S3 are implemented using the ZFS. ZFS offers on-the-fly data compression, so partitions with and without the compression were included. A second replication of each partition was created to measure the impact of a data structure's physical location on the disks. The four data structures are: an HDF5 implementation using separate datasets for each image band (H5); HDF5 using a compound data type for storing band data (H5_C); the netCDF format (NC) and the native filesystem data structure (FS). From empirical tests, the HD5 chunk size is set to 1×314. Effectively, a total of 32 data structure/partition combinations were created : 4 data structure types × 2 RAID striping options × 2 compression options × 2 replications. When reading netCDF and

2 Redundant Array of Independent Disks. A RAID system uses 2 or more disks simultaneously to improve I/O performance.

HDF5 data structures, file handles were kept open during all the queries i.e., the data structures were only opened once. The mean throughput of each of the partition types is listed in Table 2.

Table 2: 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 |

To reduce the volume of data, the spatial queries were grouped in *small* (1×1, 3×3) and *large* (100×100, 50×200, 200×50) queries. Within each of these groups, the queries times of the components were averaged and normalised to represent the time required to retrieve a single time series. The results of the small queries experiment are presented in Table 3. Despite all the arguments against the FS data structure implementation, it performed better than the H5 data structure on these small queries. Note that the NC data structure offered the best performance, regardless of the partition type. Even on the fastest partition type, effective NC I/O throughput is only 0.215 MB/s, or 0.264% of the available sequential I/O throughput, which highlights the inefficiency of such small read requests.

The results for the large queries are presented in Table 4. On the larger reads, the overheads of the FS data structure (opening a file for every pixel read) becomes the dominating factor, causing it to finish last in this experiment. The NC data structure still produced the best overall results, although the difference between the NC and H5_C data structures is comparatively small. Effective I/O throughput with the NC data structure on the compressed S3 partition rises to 11.97 MB/s, or 14.7% of the available sequential I/O throughput.

Table 3: Mean query time (microseconds per time series) for small queries

| | **Data structure type** | | | |
|---|---|---|---|---|
| **Partition type** | **FS** | **H5** | **H5_C** | **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 4: Mean query time (microseconds per time series) for large queries

| | Data structure type | | | |
|---|---|---|---|---|
| **Partition type** | **FS** | **H5** | **H5_C** | **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 |

# 5 Conclusion

The NC data structure provides the highest achievable throughput for both small and large queries. The H5_C format provides similar performance but is ranked second. The RAID options had a predictable result: S3 performed better than S2 on both the raw throughput tests as well as the data structure query tests, which indicates that network bandwidth is not yet a limiting factor. Owing to the high compressibility of the quality flag band data, the compressed partitions performed better than their uncompressed counterparts, providing additional proof that network bandwidth is still adequate. Future work will focus on improved compression strategies, since compression appears to improve performance without additional investment in hardware.

# References


Chilan, CM, Yang, M, Cheng, A and Arber, L 2006, *Parallel I/O performance study with HDF5, a scientific data package*, The HDF Group, viewed February 2008, <http://hdf.ncsa.uiuc.edu/HDF5/>.

Abiteboul, S, Agrawal, R, Bernstein, B, Carey, M, Ceri, S, Croft, B, DeWitt, D, Franklin, M, Molina, HG, Awlick, DG, Gray, J, Haas, L, Halevy, A, Hellerstein, J, Ioannidis, Y, Kersten, M, Pazzani, M, Lesk, M, Maier, D, Naughton, J, Schek, H, Sellis, T, Silberschatz, A, Stonebraker, M, Snodgrass, R, Ullman, J, Weikum, G, Widom, J and Zdonik, S 2005, 'The Lowell database research self-assessment', *Communications of the ACM*, vol. 48, no. 5, pp. 111–118.

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

Baumann, P, Diedrich, E, Glock, C, Lautenschlager, M and Toussaint, F 2003, 'Large-scale multidimensional coverage databases', in *26th GITA Annual Conference*.

Choudary, A, Kandemir, M, No, J, Memik, G, Shen, X, Liao, W, Nagesh, H, More, S, Taylor, V, Thakur, R and Stevens, R 2000, 'Data management for large-scale scientific computations in high performance distributed systems', *Cluster Computing*, vol. 1, pp. 45–60.

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

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

Reiner, B, Hahn, K, Hofling, G and Baumann, P 2002, '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.

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

Rew, R and Davis, G 1990, '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.

Sarawagi, S and Stonebraker, M 1994, 'Efficient organization of large multidimensional arrays', in *ICDE: 10th International Conference on Data Engineering*, IEEE Computer Society Technical Committee on Data Engineering.

Seamons, KE and Winslett, M 1994, 'An efficient abstract interface for multidimensional array I/O', in *Supercomputing 1994*, pp. 650–659.

Skiffington, J and McKelvey, K 2007, 'Raster in the database', in *GEOconnexion International Magazine*, pp. 22–23.

Tan, CJ, Blais, JAR and Provins, DA 2000, 'Large imagery data structuring using hierarchical data format for parallel computing and visualization', in *High Performance Computing Systems and Applications*, Kluwer Academic Publishers.

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