High Performance Access to Radio Astronomy Data: A Case Study

John F. Karpovich
James C. French
Andrew S. Grimshaw

July 11, 1994

Appeared in *Proceedings of the Seventh International Working Conference on Scientific and Statistical Database Management*, pp. 240-249, Charlottesville, VA, September 1994. Also available as University of Virginia, Department of Computer Science Technical Report CS-94-25 via the web @ ftp://ftp.cs.virginia.edu/pub/techreports/README.html.

Work partially sponsored by NSF, NASA, CESDIS and NRAO.

High Performance Access to Radio Astronomy Data: A Case Study

John F. Karpovich, James C. French, and Andrew S. Grimshaw
Department of Computer Science, University of Virginia
{jfk3w | french | grimshaw} @virginia.edu

Abstract

As CPU performance has rapidly improved, increased pressure has been placed on the performance of accessing external data in order to keep up with demand. Increasingly often the I/O subsystem and related software is unable to meet this demand and valuable CPU resources are left underutilized while users are forced to wait longer than necessary for results. This problem is especially acute for many scientific applications which use large data sets and require high performance. This paper presents our experiences with working to alleviate an I/O bottleneck in radio astronomy applications at the National Radio Astronomy Observatory (NRAO). We first present our model, the ExtensibLe File Systems model (ELFS), for improving both the performance of data access and the usability of access software. We then present the current situation at NRAO, our solution, performance results and our plans for future work.[1].

1: Introduction

Over the past decade raw CPU processing power has increased dramatically, improving by more than a factor of one hundred. At the same time the performance of disk storage devices has increased at a more modest rate, improving only about four-fold over the same period. Many programs that were once dominated by their CPU time, called *CPU-bound* programs, have now become dominated by the time spent performing I/O (i.e. they are *I/O-bound)*. For example, assuming the speedup rates given above, a program that 10 years ago spent 90% of its time using the CPU and 10% waiting for I/O would now run on a state-of-the-art computer spending only 26% of its time using the CPU and 74% of its time waiting for I/O operations. Therefore the difference in the rates of performance improvement between the processor and I/O devices has created a bottleneck for many application programs at the I/O devices, and in accordance with Amdahl's law has caused much of the increased CPU capacity to be wasted (at least from the point of view of a single program). This disparity in growth rates is likely to continue. Coupled with the increased use of parallel processing it appears that the situation will only get worse.

Over the last few years, NRAO has experienced an I/O bottleneck in running many applications. Because of this, programs spend a significant portion of their run-time performing I/O while the processor is idle. Expensive CPU resources are left underutilized and more importantly researchers are less productive because they have to wait longer for results. NRAO projects that data volume will increase significantly in the near future as new instruments come on-line. If nothing is done to alleviate the I/O bottleneck, the combination of continued unbalanced CPU speed improvements and increased data volume will cause the situation at NRAO to worsen considerably.

Since we cannot rely on new hardware developments providing an answer to the I/O bottleneck in the short-term, we propose implementing methods to better utilize the available I/O bandwidth by improving the way files are organized and accessed. Organizing files to better match access patterns can reduce the number of I/O accesses needed to fulfill a request, reduce the latency of each request, or both. For example, using an indexed file can reduce the number of I/O requests needed to retrieve a piece of data by reducing the search space of the request from the entire file to a small section of the file. Using advanced access techniques like prefetching can also significantly improve performance by better overlapping I/O and CPU usage.

It is also important that the file interface be tailored to the type of data in the file and the applications accessing it. A type specific file interface allows the user to provide information about access patterns that can be used to improve performance. For example, the user can express the intention to access every record beginning with the letter *a* or that the user intends to reuse the data repeatedly. Using this user provided information, the implementation can attempt to improve performance: in the first case, by initiating prefetching to read the data before the user needs it; in the second case, by caching the data, reducing the latency of subsequent requests for the same data. Besides potentially improving performance, a type specific interface can improve the program development process by presenting the

1. This work has been partially funded by grants NSF ASC-9201822, NASA CS-CESDIS 5908-93, and NASA NGT-50970.

file and its data in an intuitive manner, e.g. rows, records, volumes, shapes, etc., rather than unformatted bytes.

The current environment at NRAO has provided us with an ideal testing ground for our general approach to high performance file systems. We have completed the first phase of the project by finding, implementing and testing the performance of an appropriate file structure for NRAO data. The second phase of the project, applying advanced access techniques, creating a distributed version of the file structure, parallelizing the implementation and providing an improved interface, is currently underway and nearly complete. The remainder of this paper discusses our approach and the current NRAO environment in more detail and then presents the details of phase one of the project, including a brief discussion of the file structure chosen, a sketch of the implementation and performance results and observations. The final section presents our future plans for the remainder of the project and beyond.

2: Approach

Our approach to alleviating the I/O bottleneck follows the ELFS (*ExtensibLe File System*) method [1,2] first proposed by Grimshaw and Loyot. The ELFS philosophy for developing file systems has four central ideas:

- Use file structures matching data and access patterns of applications.
- Using object-oriented techniques, encapsulate the inner workings of each file type so that the application programmer can easily incorporate the appropriate file structure into a given application.
- Improve the file interface to reduce the programming burden and to allow exploitation of user knowledge to improve performance.
- Apply file type specific access software within an implementation to speed retrieval, e.g. prefetching or caching.

2.1: File Structures Matching I/O Requirements

There has been a great deal of work over the past decades in developing file structures for various uses. The database literature contains many examples of creative file structures that are well suited for particular application needs or types of data. Examples include tree-based structures such as *k-d* trees [3] and *R*-trees [4], partitioning-based structures like grid files [5] and Piecewise Linear Order-Preserving-hashing (or PLOP) files [6,7], indexed sorted files, and many others. Each has advantages and disadvantages under different access requirements and data attributes. For example, indexing schemes work well for single record retrieval when one or more of the indices can be used. Because of data locality, they also work well when a range of data is accessed along the key by which the primary file is sorted. However, a range retrieval along a key that is not the primary file's sort key may not perform well because the request will access data blocks scattered across the file.

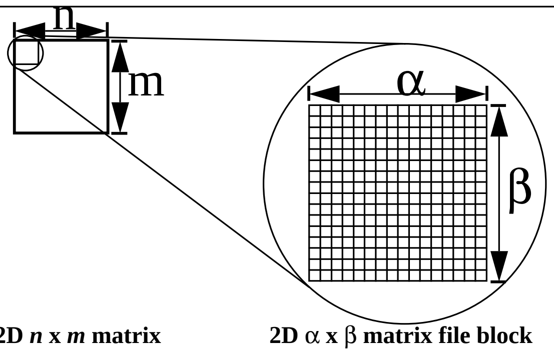


Figure 1 - Block Partitioning File Structure

For I/O performance to be maximized, it is crucial to choose file structures that best match an application's access patterns. Neither of the two most common approaches used in the scientific computing community, sequential files and relational DBMSs, match the access requirements for many applications. Sequential files are only good for applications that access data in the order in which it is stored. Using any other access pattern can be very costly. Relational DBMSs are designed to support a wide range of types of data, but rather than tailoring the file structure to the data type and access patterns, the data and access patterns must be made to fit the relational model. While the relational model is flexible enough to afford decent performance for many applications and files, their general nature often does not give the high performance necessary for scientific applications that can be achieved with file structures tailored to their needs. For example, it is easy to store a two dimensional array in a relational DBMS and perform row and column retrievals. However, using a relational DBMS will usually not lead to the best performance for such operations because the underlying file structures employed by the DBMS are designed to fit the relational model not the model of matrix data. An appropriate structure for matrix data that will be accessed by both rows and columns is a block partition structure. In a block partition structure (Figure 1), the matrix is divided into rectangular blocks and each block is stored in one data page within the file, maintaining data locality among both rows and columns. This ensures that both row and column operations can be performed efficiently.

2.2: Object-Oriented File Structures

Many of the file structures mentioned in the previous section are complex and often require a significant effort to implement them. As a result, many of these file structures have not seen widespread use. Applications that could have benefitted from a specialized file layout have been designed with simpler file schemes either due to ignorance of the existence of better schemes or due to the higher development costs of using a more advanced structure. Even when an application is developed using an advanced file structure, the implementation is often difficult to reuse and incorporate into new applications because their implementation is often intertwined with the rest of the application code. To help alleviate this problem ELFS uses an object-oriented approach to the development of new file structures. These structures encapsulate the details of the implementation while providing an interface for the user to operate on the file, such as operations to open and close the file and to insert, update, retrieve, and delete data items. The necessary file objects can then be incorporated in any application and easily extended by deriving new application-specific file classes.

2.3: Improved File Interfaces

Most file systems today fail to present an interface that reflects the type of data in the file or the file organization. As a result, many current file systems suffer from one or both of the following drawbacks: 1) the interface is difficult to use for application programmers because data is presented in a format that is not intuitive; or 2) the file system does not fully exploit the programmer's knowledge of the semantics of the data or the application domain because the interface does not allow the programmer to express this knowledge. A good example is the UNIX file system which suffers from both of these drawbacks. Unix treats all files as a stream of bytes, imposing no structure or meaning on the data. Therefore, the burden of interpretation and reformatting the data is entirely up to the user. Unix also makes no attempt to provide an interface where the user can declare properties of the file or intended access patterns to help improve performance. An example of a better interface is presented by Pane for a two dimensional matrix file [8]. The matrix file interface presents data in meaningful units, i.e. rows and columns, provides functions for manipulating the file in terms of these units, and provides a way for programmers to express how the file will be accessed, for example a method to declare the stride for upcoming accesses. This interface provides improved efficiency for developing applications software while supporting improved performance.

2.4: Type Specific Access Methods

The fourth part of our approach encompasses a set of methods that can be applied where appropriate to any file scheme to improve effective I/O bandwidth, latency, or both. These methods include intelligently prefetching data, caching data likely to be used again in the future, and parallelizing file operations and I/O related activities such as sorting. Using an object-oriented class scheme allows the implementor to choose which of the above methods are appropriate for a given file type and how they will be implemented. The details of the implementation are hidden; the user only needs to be concerned with the class interface. Selection and application of access methods can be further fine-tuned if application-specific file classes are derived from the base classes.

3: NRAO Environment

In order to effectively apply our approach to alleviate the I/O bottleneck at NRAO, we had to first understand their environment. This section describes the current environment at NRAO and the factors that will determine the success of a solution.

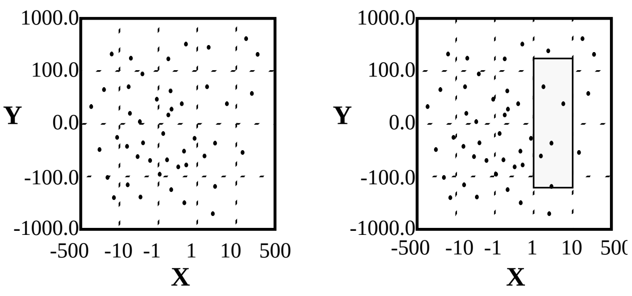
3.1: Data Volume

NRAO collects a large amount of data from its sensors. Currently, large data sets can contain up to 3-5 million measurements with each measurement ranging in size from 1 byte to 8K words of measurement data. Although in practice data sets usually do not have both a large volume of measurements and a large data size, it is not uncommon to have data sets as large as a few gigabytes. There are already plans for future systems to collect and process much larger sets of data than currently in use. For example, planning has begun on a single dish radio telescope project that could produce up to 80 gigabytes of raw data daily. In general, as new and more sophisticated astronomical instruments and methods have developed, along with increased computer power, the trend has been towards collecting and processing ever increasing amounts of data. Clearly, this trend must be considered for any long-term solutions.

3.2: Current File Structure

The standard file organization at NRAO follows the Flexible Image Transport System (FITS) format. This format organizes a file into a series of records, each of which is a tuple containing key and data elements. The key portion may be a single key component or may be composed of several different key fields. For example, radiation intensity data may be collected at a particular right ascension and declination sampled at certain times and frequencies. The data for this example would be stored as tuples with the combination of right ascension, declination, time, and frequency forming the key of the tuple and the intensity value forming the "data" part of the tuple. These records are stored one after the other in the physical file.

Currently, NRAO's data files are stored only once, sorted by one key component and no auxiliary index files are created or stored. This arrangement is due partially to limited storage capacity and partially to the programming complexity involved in index generation and maintenance. To



(*a*) Sample Data Space (*b*) Sample Sub-volume

Figure 2 - Sample Two Dimensional Data Space

access data in an order different from the sort key, the user must either accept poor performance, or sort the data beforehand using an auxiliary program and temporarily store the sorted data in another file. Clearly, the overall processing time for both of these alternatives is unattractive for large files. As discussed below, looking at data sorted by different keys occurs quite frequently and this procedure adds a significant overhead to a user's task.

3.3: Data Access Requirements

Due to the wide range of data collected and the diversity of applications that process this data, there is no single way of categorizing NRAO's data access needs. Each application has different needs, and future applications may require new access patterns, therefore flexibility is a key concern. Typically data is collected and stored once, retrieved many times, and updated occasionally (key values are rarely updated). There is little or no requirement for adding or deleting data items after a file has been created. Because of this pattern, retrieval of data is the primary performance concern at NRAO as it is the dominant operation performed after the file is created. The ability to update information is important, but data modifications are much less frequent and therefore the performance of update operations is less crucial. Often a wide range of data values are updated at once, such as when the values need to be scaled in some way. In these cases, it is acceptable to recreate the file from scratch, incurring the costs associated with doing so.

3.3.1: Data Retrieval Requirements

NRAO views the data in each file as being contained in a multidimensional data space. The number of keys determines the dimensionality of the space; each key, or axis, corresponds to a dimension in the data space. Figure 2a shows an example of a two dimensional data space where the keys are X and Y. Depending on the specific application, NRAO researchers require access to data in many different patterns. Many applications require sub-ranges of data for some key or keys. We call the data within the intersection of all sub-ranges a sub-volume of the file. Some typical sub-volumes include data in a particular time range, data within a time range for a certain frequency range and polarization values, and data generated by a specific antenna. In general, NRAO requires the ability to specify limits or boundaries on the data retrieved based on key value(s). Figure 2b shows a sample 2D data space with a highlighted sub-volume ($1 < X < 10$ and $-200.0 < Y < 200.0$). The order in which the data items are retrieved is also important. Flexibility is required to allow sorting along the different keys or iterating through the data in various ways (such as stepping through every fourth data item, etc.).

Often the same data is used by different applications, each with its own access pattern. This complicates matters since the data storage and retrieval scheme must be able to deliver acceptable performance for all applications. This is one of the major areas where NRAO's current strategy does not perform well.

3.3.2: Data Update Requirements

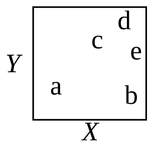
Although data is updated much less frequently than it is retrieved, the ability to modify data is still an important operation at NRAO. Updates can occur at the level of individual data values or on a much more massive scale, encompassing the entire file or a large subset of the data. Modification of an individual value may occur to correct for an error of some sort, while the modification of large chunks of data at once often occurs to scale the data values in some manner. The large majority of updates modify only the measurement data, i.e. modifications to key data is infrequent. NRAO also needs the ability to logically delete data from a file. This capability is necessary, for example, when an uncorrectable instrument error has been detected and some portion of a data set must not be used.

4: The File Structure - PLOP Files

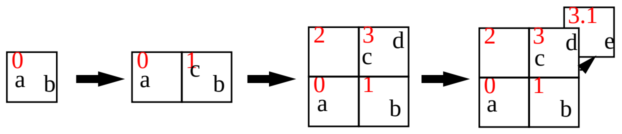
To improve NRAO's I/O performance we looked at several file structures and evaluated their suitability for NRAO's needs. Structures evaluated include indexed sorted files, R and R+ trees [4,9], quadtrees [10], *k-d* and *k-d-b* trees [3,11], grid files [5] and PLOP-hashing files[6,7]. We found that both grid and PLOP-hashing files are good candidates for NRAO data because both are specifically designed to support high performance range searches, the dominant access pattern at NRAO. Both grid files and PLOP files support range searches by physically clustering data along multiple dimensions. However, there are some important differences between them. The strategy employed by grid files for mapping logical data space to physical pages allows grid files to attain a reasonable storage utilization when the data keys are related. This is not true for PLOP files, where utilization rates can be quite poor under these conditions (as discussed in our results). However, the mapping strategy for grid files has its price. First it is a more complicated strategy that is much more difficult to implement than the PLOP file mechanism. Second, the grid file mapping requires a directory structure that may be too large to fit in memory, whereas the PLOP file does not. When the directory does not fit in memory an extra I/O operation is needed to retrieve it, making each point retrieval take 2 I/O operations for the grid file. Depending on the partitioning, PLOP files may require significantly less I/O operations for a point access on average (we believed we could get such an advantageous partitioning by collecting data distribution statistics before partitioning our files - see section 6). Another difference is that PLOP files lend themselves very well to parallel sorting (see below for more detail), which we intend to employ in the next phase of the project. Based on these factors we chose to implement PLOP files rather than grid files, though this does not indicate that we feel that either file structure is clearly superior. For a detailed account of our search for an appropriate file structure and a more in depth discussion of the other structures refer to [12].

In PLOP files a *k* dimensional data space is dynamically partitioned by splitting each dimension into a series of *slices*. The intersection of one slice from each dimension defines one logical data bucket. Each logical bucket corresponds one-to-one with a particular fixed size physical storage bucket (called a primary page) and data points are stored in the bucket that maps the corresponding subspace in each dimension. This property has several important ramifications. First, splitting a dimension creates possibly many new physical storage buckets and causes all buckets within the to-be-split slice to realign their data points. This makes minimizing the number of splits a priority to reduce the work done while splitting and to reduce the storage requirements for the PLOP file. Second, because the physical primary page can be determined uniquely from the logical bucket, a separate directory mapping a logical bucket to physical storage is not necessary. All that is required is a tree structure, called a *split tree*, which tracks the values where splits occur in each dimension. Since this structure is small it can be kept in main memory. Therefore, the primary page for a logical bucket can be accessed with one I/O operation unlike other structures that require large indexes or directories.

To help reduce the number of splits occurring in the data space, PLOP files do not require that all overflowing buckets cause a split. Rather an overflowing bucket can create a new auxiliary bucket or *extent* in a separate file and put extra data points there. Extent buckets are connected together to form a chain of data pages that all map to the same logical bucket in the data space. The decision of when to split vs when to chain extent pages is up to the implementor, but



(*a*) Sample Data Space



(*b*) Snapshots of Growing PLOP File

Figure 3 - Building a Two Dimensional PLOP File

several common strategies include splitting when an entire slice is on average 100% full or when a particular chain has grown beyond some threshold length. Figure 3 shows a sample two dimensional data space (*a*) and one possible sequence of snapshots describing the layout of the PLOP file during the insertion of the points (*b)* - the points are inserted in order a,b,c,d,e; the numbers indicate the number of the bucket; 3.1 indicates the first extent page of bucket 3.

PLOP files fit NRAO's requirements very well because they were designed specifically for multidimensional range queries, exploiting data locality along multiple keys, and reducing the search space through partitioning of the data space. With a good partitioning of the file, point accesses are efficient in the average case with a lower bound of one file I/O operation. However, the average and maximum number of accesses needed for a single item depends on the average and maximum length of extent page chains created, which in turn depends on the strategy employed while partitioning the file.

PLOP files have another property that is useful for optimizing sorted range queries, especially when parallel retrieval techniques are applied. The physical buckets for any slice along a dimension are disjoint from the physical buckets of all other slices along that dimension. Therefore sorting a query that spans several slices for the sort key can be broken into several smaller sorted queries, each working on one slice. This strategy can improve performance in two ways. First, breaking the problem into smaller pieces can improve the performance of the overall sorting computation from $O(n \log n)$ time to $O\ (n \log n/p)$ on one processor in the best case (where n is the total number of items sorted and p is the number of pieces). Second, each of the slices can be sorted in parallel on separate processors. In the ideal case, with even split sizes, homogeneous computing resources and no added overhead, sorting time could be reduced to $O(n/p \log n/p)$.

5: Implementation

Our implementation is comprised of a set of C++ classes. The main structure is a hierarchy of file classes for PLOP files: a base class for the general PLOP file (`plopFile`), a derived PLOP file class specifically for a group of data files that hold interferometry data (`IFPlopFile`), and two further derived subclasses, one each for line-spectrum and continuum interferometry data (`IFLinePlopFile` and `IFContPlopFile`). For retrieving data and testing performance, a `queryWindow` class was created which provides the interface to the user to describe queries and obtain results from the PLOP files.

The goal of the PLOP file hierarchy is to provide a framework for building high performance type-specific PLOP files and in particular PLOP files designed for NRAO's data and access requirements. We used C++ because it supports the object oriented paradigm which encourages modularization and encapsulation of functionality and, through inheritance, extensibility and code re-use.

The interface revealed to the end user consists of a handful of member functions: a constructor and destructor, and `open`, `readHeader`, `writeHeader`, `addRecord`, `reportStats` and `createFromFits` functions. These few functions allow the user to create the PLOP file and add data to it. Since our tests will not require updating records or deleting them, no update functions have been included in the interface. In the future, these functions can be added to the existing interface as needed.

Retrieval mechanisms for the PLOP files are implemented in a separate C++ class, `queryWindow`. By packaging the retrieval portion separately from the file, the user can define multiple data windows simultaneously in the same application, each related to the same file or to different files. The interface for `queryWindow` allows the user to easily specify the types of queries required by NRAO, i.e. sorted and unsorted range queries and point queries. Setting up a query is done by declaring a `queryWindow` object and then setting the range or ranges to search for each key and specifying the dimension by which to sort the resulting data. For example, to search for all data with times between 0.1 and 0.2 and U values between -1000 and +1000 sorted by time, three calls are necessary, two to set the query ranges and one to set the sort key: `set(“Time”,0.1,0.2)`, `set(“U”,-1000.0,1000.0)`, and `sortBy (“Time”,1,ASCENDING)`. The “set” functions can all take multiple ranges for a single query. The result of such a query is that all data matching any of the ranges specified is retrieved. If the user desires the data for times (0.1,0.2), (0.5,0.75), and (0.9,1.0), a query can be made by using the `set()` function three times, one for each range. Single axis value queries are formed by specifying the same value for the upper and lower part of the range, such as (0.1,0.1).

Since our short-term goal is to test the retrieval performance of our PLOP file implementation, the only retrieval command currently implemented is `countPoints()`. `CountPoints()` retrieves all of the data necessary to perform the current query, sorts it if necessary, and collects statistics about the query such as the number of pages retrieved, the number of points in the result, and the time required to complete the query. The only functionality missing that a user would need for a query is some mechanism for passing back the result data to the calling program, possibly a piece at a time. In the next phase of the project, this interface will be improved to include functions to retrieve chunks of data in various ways and will be extended to allow the user to input more information about the needs of the application. The latter will allow the `queryWindow` implementation to take advantage of this knowledge to use advanced I/O techniques like prefetching when applicable to improve retrieval performance.

6: Results

The purpose of implementing PLOP files was to improve NRAO's performance, particularly the performance of I/O requests. We selected two typical NRAO data sets and converted them to PLOP files, measuring such factors as the storage utilization, conversion time, and bucket depth (i.e. the number of physical pages - primary and extent - used to store a logical bucket). We then designed a set of data retrieval queries representative of typical NRAO requests and tested their performance. The following sections describe our experience with these test files. First, we describe the files used in more detail. Second, we present the results of the conversion process, the characteristics of the converted file and a discussion of choosing the dimensions for a PLOP file. Finally, we discuss the test queries, results, and observations about our retrieval performance.

6.1: Test Data Sets

The two data sets we selected for testing both contain interferometry data collected at NRAO's Very Large Array (VLA) facility. The VLA is composed of many individual radio antennas, each with a host of sensory devices. Groups of antennas are aimed at the same area of the sky and each antenna collects data individually. The data from each pair of antennas (called a *baseline*) are synthesized into a composite data element for each antenna pair in the array. Data of this form are collected at periodic time intervals to provide a time sequence picture of the object of interest.

The two interferometry files we converted both contain data as described above. The first file contains “line-spectrum” data which are collected for a series of polarizations and a series of frequencies within a specified *frequency band*. The keys that make each data point unique are time, baseline, polarization and frequency. In addition, there are other data values associated with each record: the source of the measurement (the astronomical object of interest), the frequency domain coordinates of the baseline (U, V, W), and the frequency band. The "data" portion of each measurement is a single complex number and weight. The second file contains "continuum" data which are also collected for a series of polarizations, but are not collected for a series of frequencies. Each baseline measurement is a composite value for a range of frequencies. Therefore the unique keys for continuum data are time, baseline and polarization. Like line-spectrum data, each continuum measurement is comprised of a complex number and weight and each record stores the frequency domain coordinates of the baseline.

These two files have some important differences. The line-spectrum data set contains records for multiple frequencies (31 frequencies for the file we used). To reduce storage space requirements for the file, NRAO typically stores all of the frequency and polarization records for the same time and baseline together as one large record. This reduces the file size by eliminating the need to duplicate the key fields for each frequency and polarization pair. The drawback of this approach is that the data for all frequencies and polarizations must be read with each record even if only a subset of these values is desired. We decided to compromise by breaking out separate polarization records while keeping all frequencies together as this provided the added ability to search based on polarization while keeping the file size close to the original size. The two files also differ in record size and number of records. The line-spectrum file contains 126,092 404 byte records (~49 megabytes) and is considered a small file by NRAO standards, while the continuum file contains 8,440,092 32 byte records (~264 megabytes) and is considered a medium sized file.

6.2: File Conversion

Before creating a PLOP file, we had to choose the dimensions (also called *axes*) of the data space. The dimensions chosen determine which keys will be able to narrow the search space and which will not. Therefore, we worked with NRAO scientists to determine the most appropriate keys to use as dimensions. Keys that are frequently used in range queries and especially those keys that frequently have a small portion of their overall range accessed are the best axis candidates, while keys that are rarely used to filter data requests are poor choices. An important consideration when choosing the axes for a PLOP file is that the more dimensions there are in the data space, the less each dimension will be split. Fewer splits within an axis lowers the precision of that axis (each slice for the dimension contains a wider range of data values), and therefore, searches will access a larger amount of unnecessary data on average. There is a fundamental trade-off between the number of axes and the precision of each axis and the choice of dimensions must balance the benefits of more axes with the decrease in average axis precision. The final decision of which axes to include must be determined by the relative importance of each access pattern and how useful each dimension is in narrowing the search space for these accesses. Our experience with this project also pointed out another important factor in determining which keys to use as axes, namely that groups of keys whose values have some relationship among them should not be used together. The reason is that the related values for the keys can cause a clustering of the points in the data space, leaving very densely populated areas and empty areas. Figure 4 shows the clustering that occurred when we used the related keys time and source as dimensions (along with 4 other keys). The table shows the number of points in each region defined by the intersection of a time and a source slice. The explanation for this behavior is that data in the line-spectrum file we used was collected by studying one source at the beginning of the time range, then switching back and forth between two sources and finally returning to the original source for the remaining time period. This shows a fundamental weakness in using PLOP files for such data.

		Source		
	Slice #	0	1	2
Time	0	15,372	702	0
	1	0	7,722	7,722
	2	0	6,318	9,128
	3	0	9,828	5,616
	4	0	7,722	7,722
	5	0	7,020	8,424
	6	1,910	8,424	5,616
	7	16,848	0	0

Figure 4 - Data Distribution in 4D Regions Defined by Time and Source Axes

For our final version of the PLOP file we used 3 dimensions: time, baseline and polarization. To attempt to produce a good partitioning of the data space our implementation prepartitioned the data space based on knowledge of characteristics of the data file and statistics about the data distribution in the source file. First, since the number of polarization values is limited and known beforehand (2 and 4 for the line-spectrum and continuum files respectively) and we know the records are distributed evenly among the values, we split the data space at each polarization value. For the remaining two axes, we collected histogram statistics for each dimension and split the data space into roughly an even number of splits for each remaining axis such that each slice along a dimension contained approximately the same number of records.

While this scheme will not produce an optimal partitioning (because the histogram statistics are collected independently for each key and do not create a full picture of the multidimensional data distribution), it did produce very

	Line-Spectrum	Continuum
Number of Records	126.092	8,440,092
Record Size (bytes)	404	32
Page Size (bytes)	4,096	4,096
Records Per Page	10	127
Splits Along Polarization Axis	2	4
Splits Along Baseline Axis	79	128
Splits Along Time Axis	80	130
Total Primary Pages	12,640	66,560
Total Extent Pages	3,168	33,320
Total Storage Used (kilobytes)	63,238	399,529
Storage Required (kilobytes) (Record Size x Number of Records)	49,747	263,753
Storage Utilization (Space Req'd/Storage Used)	78.6%	66.0%
Empty Pages	0	128
Maximum Bucket Depth	2	5
Number of Buckets w/ Depth > 2	0	20
Average Bucket Depth	1.25	1.50
Average Record Depth	1.08	1.11
Conversion Time (minutes)	8.4	87

Table 1: Conversion Results for Line-Spectrum and Continuum PLOP Files

good results. In fact, the results were much better than we could have attained with dynamic partitioning. With dynamic partitioning the data space is split as needed as more data is added (or deleted) so the order in which data is entered is crucial. Certain insertion orders can cause poor choices for split values. In particular, if data is added in sorted order along a key axis (as was the case for our line-spectrum file - it was sorted by time), the split choices can be disastrous. To see why this is true we'll assume that we chose to split the line-spectrum file dynamically by alternating splitting the time and baseline axes whenever an extent chain grows beyond some threshold depth. As data is entered in time order, both axes will split many times. The problem is that once a time slice is split, the lower valued slice will never again receive new data points (because the data being entered is sorted). With each new baseline axis split, the storage consumed by each time slice is increased. Therefore, more and more space will be allocated to time slices that have a constant number of points, lowering their data density and increasing the amount of wasted space in the file. Using a priori knowledge about the data distribution avoids problems related to insertion order.

The results of the file conversions are shown in Table 1. The most important statistics are the storage utilization, average and maximum bucket depths, and average record depths achieved. The line-spectrum file was particularly impressive, where storage utilization was almost 79%, while the average and maximum bucket depths were 1.25 and 2 respectively (storage utilization is the ratio of total storage needed for the raw data versus the total storage actually used including all overhead and wasted space). The average record depth reflects the average number of I/O operations needed to access a single record, i.e. a record stored in a primary page is at a depth of 1, a record in the first extent page a depth of 2, etc. For the line-spectrum file the average record depth of 1.08 means that 92% of all records are stored in the primary page and 8% in the first extent page).

6.3: PLOP File Query Performance

To test the performance of retrieving data from our PLOP files, we constructed a set of test queries that represent either common access patterns currently used at NRAO or interesting benchmark queries. We ran the queries a minimum of three times on both of the files and recorded the best retrieval time and other important statistics for each query, including the number of file I/O operations needed and the amount of total and useful data retrieved. All times are wall clock times encompassing the entire length of the query, including any time to calculate the pages of interest and to sort results. Therefore, all facets of the query are included in the timings. To avoid skewing of results due to file caching, large file copy jobs were run in between successive tests to clear the file cache. All tests were run on a SPARCStation IPX with 32 MB of RAM and an attached hard disk. The disk has an average rotational latency of 6.95 milliseconds and an average seek time of 9.8 milliseconds.

The measure we are most interested in is the *effective bandwidth* of a query. The effective bandwidth of a query is the amount of useful data per time unit that is retrieved from the file. This measure is more useful than total bandwidth in determining the performance of a query because it reflects not only the speed with which I/O requests can be performed, but also the accuracy of the requests (for this paper we define total bandwidth as the rate at which all data is retrieved, including overhead space and empty space due to fragmentation). For example, we may be able to achieve a high total bandwidth of, say 2 MB per second, but if the accuracy of the data retrieved is low, such as 5%, then the retrieval rate for "good" data is a more modest 100 Kilobytes/second. Raising the accuracy directly raises the effective bandwidth of a query and subsequently raises the rate at which needed data is injected into the application program.

6.3.1: Test Queries

We chose 10 queries to test our PLOP file performance:

- *1) All data, unsorted.*
- *2) All data, sorted by time.*
- *3) All data, sorted by baseline.*
- *4) Time Range (approx. 10%), all other data, unsorted.*
- *5) Time range (approx. 10%), all other data, sorted by U.*
- *6) 1 time, 1 polarization, all baselines, unsorted.*
- *7) Time range (approx. 10%), 1 polarization, all baselines, unsorted.*

#	Total Recs	# Good Records	# Primary Pages	# Extent Pages	Total Data (KB)	% Good Data	Time (secs)	Effective BW (KB/ sec)
1	126,092	126,092	12,640	3,168	63,232	78.7%	55.49	897
2	126,092	126,092	12,640	3,168	63,232	78.7%	70.5	706
3	126,092	126,092	12,640	3,168	63,232	78.7%	181.8	274
4	14,742	12,636	1,422	474	7,584	65.7%	4.10	1,216
5	14,742	12,636	1,422	474	7,584	65.7%	6.85	728
6	702	351	79	0	316	43.8%	0.271	511
7	7,371	6,318	711	237	3,792	65.7%	2.33	1,079
8	752	186	84	20	416	17.6%	1.45	51
9	15,026	4,836	1,512	360	7,488	25.5%	14.5	132
10	720	180	80	20	400	17.7%	1.49	48

Table 2: Performance for Line-Spectrum File

#	Total Recs	# Good Records	# Primary Pages	# Extent Pages	Total Data (KB)	% Good Data	Time (secs)	Effective BW (KB/ sec)
1	8,440,092	8,440,092	66,560	33,320	399,520	66.0%	375	703
2	8,440,092	8,440,092	66,560	33,320	399,520	66.0%	637	413
3	8,440,092	8,440,092	66,560	33,320	399,520	66.0%	1,382	191
4	968,220	951,320	7,680	3,344	44,096	67.4%	49.3	603
5[a]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
6	31,673	351	256	130	1,544	0.71%	1.29	8.5
7	242,055	237,830	1,920	836	11,024	67.4%	13.3	559
8	40,124	13,256	260	208	1,872	22.1%	8.0	52
9	673,924	326,172	5,200	2,596	31,184	32.7%	175	58
10	19,379	6,256	130	90	880	22.2%	6.3	31

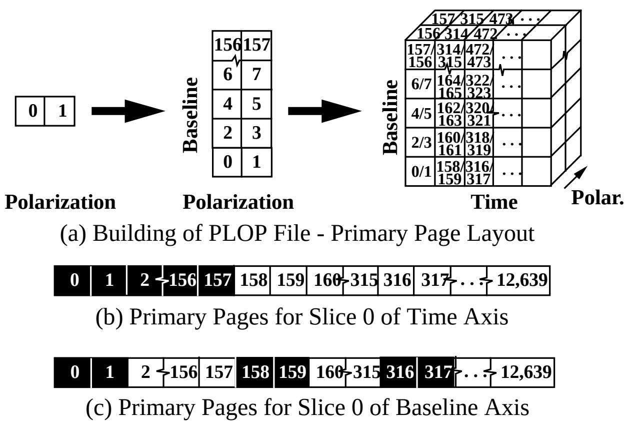
Table 3: Performance for Continuum File

a. We have not implemented an out-of-core sort. Therefore we can only sort result sets of a limited size and could not perform query 5 for the continuum file.

- *8) Time range (approx. 50%), 1 baseline, all polarizations, unsorted.*
- *9) Time range (approx. 50%), 1 antenna, all polarizations, unsorted* (A search by antenna encompasses all baselines for which the antenna is a part.
- *10) All times, 1 baseline, 1 polarization, sorted by time.*

6.3.2: Query Results and Observations

Tables 2 and 3 summarize the results of our test queries for the line-spectrum and continuum data sets, respectively. The first query retrieved the entire data set without sorting the results. The unsorted query achieved an average total bandwidth of over 1.1 megabytes per second and an effective bandwidth of almost 900 kilobytes per second for the line-spectrum file and almost 1.1 megabytes/second total and over 700 kilobytes/second effective bandwidth for the continuum file. The total bandwidth for this query was significantly below the maximum transfer rate for the disk drive (3 megabytes/second) because of the head movement required to follow chained buckets and possible head movement due to disk fragmentation. The difference between the bandwidth figures for the two files can be explained by the



(a) Building of PLOP File - Primary Page Layout

(b) Primary Pages for Slice 0 of Time Axis

(c) Primary Pages for Slice 0 of Baseline Axis

Figure 5 - Time Slice vs Baseline Slice Retrieval

difference in the average bucket depth between the two files. The line-spectrum file had an average bucket depth of about 1.25 compared to 1.50 for the continuum file. The result of this difference is that the continuum query had to follow a chain more often (once every two primary pages on average rather than once every four pages) than the query for the line-spectrum query. This caused the continuum query to have more disk head movement on average.

The second and third queries also retrieved the entire data set, but the results were sorted by time and baseline, respectively. We would expect that the sorted queries would not perform as well as the unsorted queries due to the time required to sort the results, and this was the case for both files. We would also expect that the query sorted by time would be faster than the query sorted by baseline. To see why this is the case, the procedure for retrieving data sorted along an axis key must be understood. To retrieve a range of data sorted by one of the PLOP file dimensions, the query is broken into several subqueries, one for each slice along the sort key. These queries are then processed in sequence, each retrieving, filtering and sorting one slice along the sort key. The performance of retrieving one slice of data depends on the layout of the primary pages for the slice, which is dictated by the manner that the file was originally partitioned. In both PLOP files, the polarization axis was split completely first, then the baseline dimension was completely split and finally the time dimension was split (Figure 5*a*). This order of splitting was done purposely, so that the primary pages for each time slice would all be contiguous (ignoring disk fragmentation) and therefore queries sorting by the time axis would require less head movement and perform better than the other dimensions (time was identified as the most used key for sorting by NRAO). Figures 5*b* and *c* show how retrieving a time slice requires a contiguous section of the primary file, while retrieving a baseline slice requires primary pages scattered across the file.

We would expect query 4 (10% time range, unsorted) to have performance characteristics similar to query 1 (all data unsorted), but somewhat lower effective bandwidth because the query retrieves extra unnecessary data at the boundaries of the query. For the line-spectrum file, the accuracy of the query did decrease relative to query #1, but the total bandwidth increased enough to actually increase the effective bandwidth. The cause for the unexpected increase in total bandwidth is unknown, but we suspect that it was caused by favorable scheduling by the operating system. For the continuum file, the total and effective bandwidth measures were closer to what we expected, i.e. they were close to those of query 1. Interestingly, the accuracy for the continuum query was slightly better than for query 1, rather than worse as we expected. The reason for this is that the region of data retrieved had a lower average bucket depth (~ 1.44) than the file as a whole, and therefore, retrieved fewer extent pages that were likely to be sparsely populated.

The fifth query retrieved the same data as the fourth, but sorted the data by a non-axis key, U. The result was that sorting the data increased the retrieval time by 67%. We hope that in the next phase of the project we can reduce the overall run-time of such queries by overlapping the sorting and I/O operations more effectively.

Queries 6, 7, 8, and 10 demonstrate the speed that PLOP files can afford for small, well directed queries. In query six, the reduced accuracy in the continuum file caused the query to run slower than the line-spectrum query, even though the total bandwidth was more than 50% greater and the amount of data retrieved was twelve times less. The lower accuracy for the continuum file was caused by having a larger number of records per page and larger slices (caused by the size of the file - the time slice that was searched had 128 primary pages vs 79 for the line-spectrum file).

6.3.3: PLOP Files vs Current NRAO Practices

The real test for our project is to determine the relative performance of our PLOP file implementation versus the current implementation used by NRAO. Unfortunately, collecting comparable timing information from existing NRAO applications has proved to be very difficult because the data retrieval code is intertwined with other application code.We realize that this comparison is important in determining the success of our work, so we are continuing our effort to gather this data. We are also considering a comparison against other file structures such as a sequential file (like NRAO currently uses) or an indexed sorted file.

7: Future Work

We are extending the class interface to provide functions, e.g. a `getRecords(num)` function, that return data to application programs in an intuitive and useful manner. We have already designed a distributed version of the PLOP file that will allow the file to be partitioned into several pieces and are working on its implementation. We are also working on a new parallel architecture (using the parallel object-oriented system Mentat [13]) to allow us to take advantage of the distributed file layout by allowing separate I/O workers to access and process data in parallel. The parallel architecture will also allow us to better overlap I/O and computation by implementing prefetching operations.

8: Acknowledgments

We would like to thank NRAO for partially funding this project. Thanks to Brian Glendenning, Jeff Crouse, Gareth Hunt, Alan Ferris, David Shone, and everyone else at NRAO who helped with this project.

9: References

[1] A. S. Grimshaw and E. C. Loyot, Jr., “ELFS: Object-Oriented Extensible File Systems,” University of Virginia, Computer Science TR 91-14, July 1991.

[2] John F. Karpovich, Andrew S. Grimshaw, James C. French, “Extensible File Systems (ELFS): An Object-Oriented Approach to High Performance File I/O”, to appear in the proceedings of OOPSLA94, Portland Or., October, 1994.

[3] J.L. Bentley and J.H. Friedman, “Data Structures for Range Searching”, *ACM Computing Surveys,* Vol. 11, No. 4, pp. 397-409, December 1979.

[4] A. Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching”, *Proceedings of Annual Meeting, ACM SIGMOD Record,* Vol. 14, No. 2, pp. 47-57, 1984.

[5] J. Nievergelt and H. Hinterberger, “The Grid File: An Adaptable, Symmetric Multikey File Structure”, *ACM Transactions on Database Systems*, Vol. 9, No. 1, pp. 38-71, March 1984.

[6] H. Kriegel and B. Seeger, “PLOP-Hashing: A Grid File without a Directory”, *Proceedings of the Fourth International Conference on Data Engineering*, pp. 369-376, February 1988.

[7] H. Kriegel and B. Seeger, “Techniques for Design and Implementation of Efficient Spatial Access Methods”, *Proceedings of the 14th VLDB Conference*, pp. 360-370, 1988.

[8] Brian Pane, “Efficient Manipulation of Out-of-Core Matrices”, University of Virginia, Department of Computer Science.

[9] T. Sellis, N. Roussopoulos, and C. Faloutsos, “The R+-Tree: A Dynamic Index for Multi-Dimensional Objects”, *Proceedings of the 13th VLDB Conference*, pp. 507-518, 1987.

[10] H. Samet, “The Quadtree and Related Hierarchical Data Structures”, *ACM Computing Surveys*, Vol. 16, No. 2, pp. 187-260, June 1984.

[11] J. T. Robinson, “The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes”, *ACM SIGMOD Proceedings of Annual Meeting*, pp. 10-18, 1981.

[12] John F. Karpovich, Andrew S. Grimshaw, James C. French, “Breaking the I/O Bottleneck at the National Radio Astronomy Observatory (NRAO)”, technical report, University of Virginia, Department of Computer Science, in progress.

[13] A. S. Grimshaw, E. Loyot Jr., and J. Weissman, “Mentat Programming Language (MPL) Reference Manual,” University of Virginia, Computer Science TR 91-32, 1991.