Breaking the I/O Bottleneck at the National Radio Astronomy Observatory (NRAO)

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

September, 1993

Available as University of Virginia, Department of Computer Science Technical Report CS-94-37 via the web @ ftp://ftp.cs.virginia.edu/pub/techreports/README.html.

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

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
THORNTON HALL
CHARLOTTESVILLE, VIRGINIA 22903-2442
(804) 982-2200 FAX: (804) 982-2214

Breaking the I/O Bottleneck at the National Radio Astronomy Observatory (NRAO)

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

Department of Computer Science
University of Virginia
Charlottesville, VA
{jfk3w | grimshaw | french} @virginia.edu

1.0 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 bandwidth of disk storage devices has increased at a more modest rate, improving only about four-fold over the same time period. Programs that were once dominated by their CPU time, called *CPU-bound* programs, have now become dominated by the time spent waiting for 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 looks like it will continue for some time, suggesting 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 I/O bottleneck, 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 it appears that new hardware developments will not provide an answer to the I/O bottleneck in the short-term, we propose implementing methods that will utilize the available I/O bandwidth better by improving the way files are organized and accessed. Organizing files to better match access patterns can reduce the number of I/O accesses necessary to fulfill a request, reduce the average 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. For example, suppose an application processes a series of data items, and each item requires one time unit for I/O and one time unit for processing by the CPU. Without prefetching it would take two time units to process each data item. With prefetching it would still take two time units for the first item to be processed, but all subsequent items would require only one time unit to process (the I/O time is pipelined away).

To fully utilize the file structures and access methods discussed above, it is 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, the implementation can initiate prefetching to read the required data before the user needs to use it; in the second case, the implementation can cache 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 file and its data in an intuitive manner that matches the specific file type. For instance, data can be presented to the user in meaningful terms, such as rows, records, volumes, shapes, etc., rather than as unformatted bytes (as is the case for Unix files).

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.

2.0 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:

- Find and use file structures that match the data and access patterns of particular applications or sets of applications more closely than file structures currently used.
- 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 system 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. Based on the file type some or all of the following techniques may be applied: prefetching, caching, and parallel access to different parts of the data set.

2.1 File Structures Matching I/O Requirements

There has been a great deal of work over the past two or three 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], primary files with secondary indexes, and many others. Each of these file layouts 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.

For I/O performance to be maximized, it is crucial to choose file structures that best match an application's access requirements. Neither of the two most common approaches used in the scientific 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. For example, retrieving the data sorted in an order other than the normal

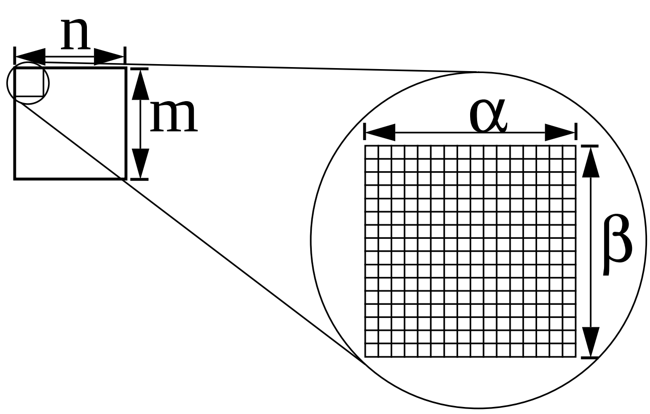


Figure 1 - Block Partitioning File Structure

sequential order may require multiple passes through the entire file. 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. Each block is stored in one data page within the file, maintaining data locality among both rows and columns. The advantage of this structure over a row or column major structure is that both row and column operations can be done efficiently.

2.2 Object-Oriented File Structures

Many of the file structures discussed above 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. To help alleviate this problem the ELFS method uses an object-oriented approach to the development of new file structures. These structures

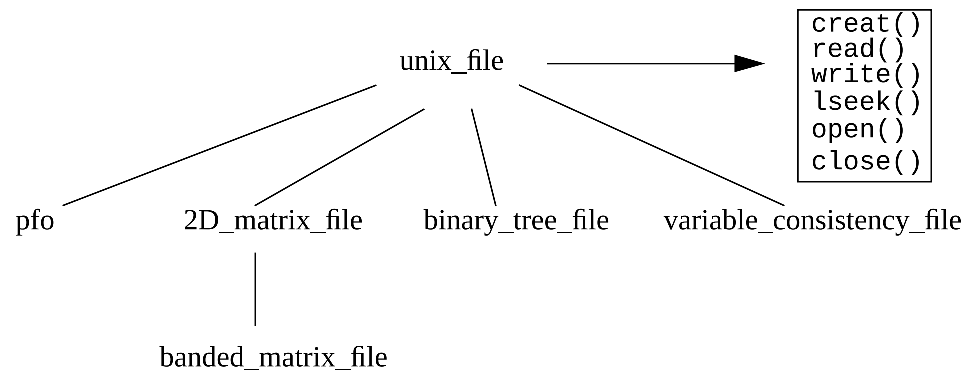


Figure 2 - A Sample ELFS Class Hierarchy

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 either deriving new application-specific file classes or extending the class interface of existing file classes. Figure 2 shows a sample file class hierarchy derived from a basic UNIX file object [1][1].

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 of this problem is the UNIX file system which suffers from both of these drawbacks. Unix treats all files as simply 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, such as 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 upcom-

1. Figure 2 is reproduced without permission from page 4 of [1]

ing accesses. This interface provides both improved efficiency for developing applications software and improved performance.

2.4 Type Specific Access Methods

The fourth part of our approach, type specific access methods, encompasses a set of orthogonal methods that can be applied where appropriate to any file scheme to improve I/O bandwidth, latency, or both. These methods include selectively and intelligently prefetching data, caching data likely to be used again in the near future, and parallelizing file operations and other 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.

Prefetching

Prefetching data involves guessing ahead and starting I/O operations for data that is likely to be requested in the near future. The simplest prefetching strategy is to read the next block of data in the file after the last one read. Called *sequential prefetching*, this method is employed in many database and file systems, such as IBM's DB2, and works well for retrievals where reads will be sequential. For access patterns that do not behave this way, prefetching can actually be detrimental as resources, including buffer space, I/O device bandwidth, and server bandwidth, are all used for reading in bad guesses. The key is to guess correctly as often as possible. One promising approach to achieving better guesses is to allow the user to specify intentions for how the file is going to be accessed. These intentions can be used to determine whether prefetching should be employed and if so, which data is likely to be needed next. This should help reduce bad guesses and wasted resources. As an example, if a user can specify a stride or some function that describes an upcoming series of accesses, the file object can read ahead to reduce or ideally eliminate the read latency.

File Caching

File caching is another popular method for reducing I/O latency and increasing I/O bandwidth. File caching relies on temporal locality of data references, i.e. that the data from a given reference is likely to be used again in the near future. File caching exploits this property by keeping recent requests around in local memory so if they are needed again, the request can be satisfied from memory instead requiring an I/O operation. This approach is applied throughout the computer storage hierarchy and is particularly success-

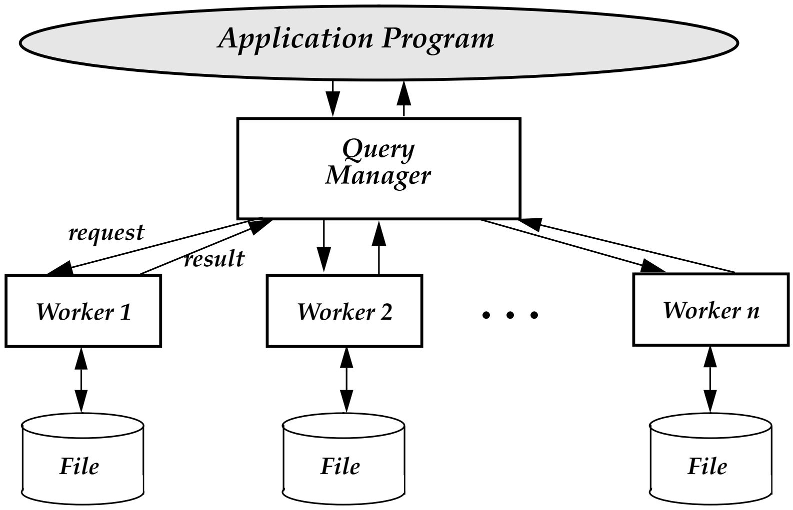


Figure 3 - Multiple I/O Servers on a Single Task

ful when accesses display significant temporal locality. However, in applications where data files or subsets of data files too large to fit into local memory are read, this scheme does not help and may take up valuable memory and processor resources from other parts of the application unnecessarily. Like prefetching, the user often knows the access patterns of the application best and so it seems reasonable that providing an interface to express the user's intentions will lead to better overall use of resources.

Multiple I/O Threads

Typical applications have one thread of control and therefore completely stop and wait when an I/O operation is pending. A better model is to try to overlap computation as much as possible with file I/O activity by splitting I/O operations and application code into separate threads of control that can execute in parallel. This allows the application program to potentially continue computation after issuing an I/O request, while at the same time the I/O request is being satisfied. Since some I/O related operations, such as sorting data, are CPU-intense themselves, this idea can be further applied to splitting the I/O operations into several pieces, each working in parallel to satisfy a given request. The overall effect is that the bottleneck resource, namely the I/O device hardware, is better utilized by keeping it busy as often as possible. Figure 3 shows a sample request split among several I/O servers working together.

Implementing the tailored file structures or type specific access methods in isolation will help to reduce the overall latency of an I/O request. Combining the two components will provide better performance than either alone. The addition of the file object model provides a cohesive framework for using various file structures in different situations. Dense and sparse 2D matrix file classes have been implemented using this approach [8]. This work demonstrated good performance characteristics and we believe reduced the cognitive burden for application programmers.

3.0 NRAO Data Requirements and Factors For Success

In order to effectively apply our approach to alleviate the I/O bottleneck at NRAO, we must first understand their environment. The following 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 involve 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 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

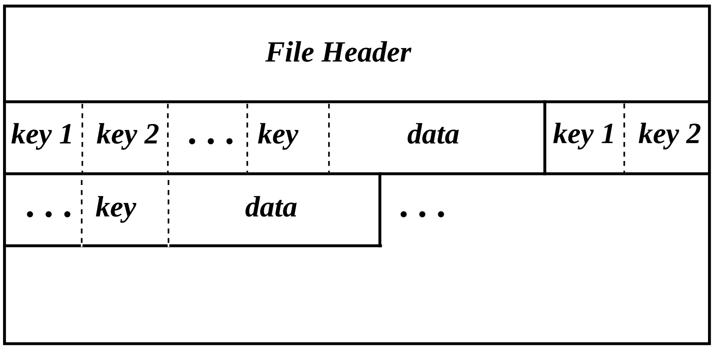


Figure 4 - FITS File Layout

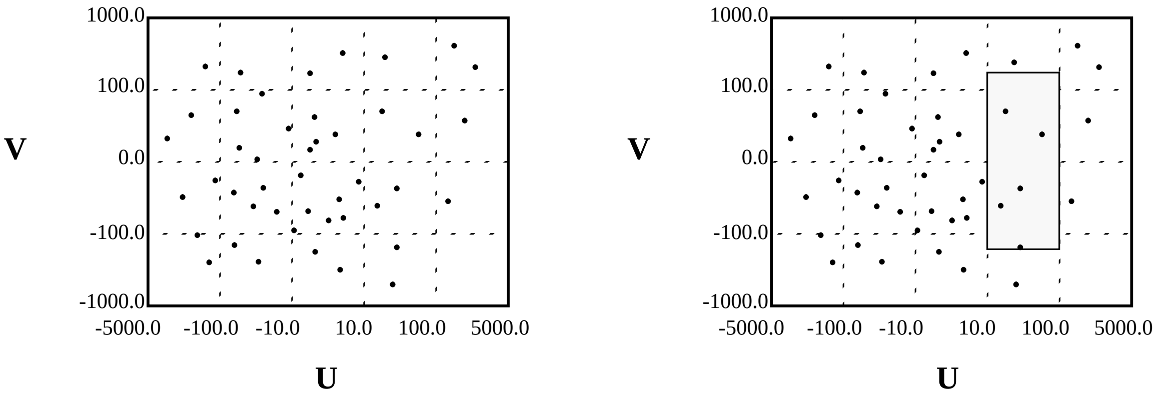
value forming the “data” part of the tuple. These records are stored one after the other in the physical file (Figure 4).

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 access data in an order different from the sort key, the user must either accept the poor performance caused by many I/O operations, 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 adds a significant overhead to a user’s task.

3.3 Data Access Requirements

Once data has been collected and stored in a file, the data must be accessible to users and application programmers for retrieval and update. 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. The following sections describe the relative mix and importance of retrieval, update, deletion, and insertion operations for NRAO.

For any data item there are four main operations that may be required: retrieval, update, insertion, and deletion. At NRAO data is commonly collected and stored once, retrieved potentially 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.



(*a*) (*b*)

Figure 5 - Sample Two Dimensional Data Space

Retrieval of data is the primary performance concern at NRAO. 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. Therefore the performance of update operations is less crucial. It is often the case that 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 inevitable costs associated with doing so.

3.3.1 Data Retrieval Requirements

In general, 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 4*a* shows an example of a two dimensional data space where the keys are U and V (frequency domain coordinates). 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 example typical sub-volumes include data in a particular time range, data within a time range for a certain frequency ranges 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 4b our example 2D data space with a highlighted sub-volume (10.0 ≤ U ≤ 100.0 and -200.0 ≤ V ≤ 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.).

Programs differ in how many times they use the data. Most NRAO applications have a set access pattern that is known at the time the program is written. Some programs, usually user interactive applications, will not be able to predetermine the access requirements, or may only be able to narrow the range of operations allowed. If knowledge of the program's access requirements can be exploited, then it is possible that significantly more intelligent I/O processing can be performed and higher overall performance achieved.

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 (i.e. storing only one copy of the file without indexes) 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.

3.4 Factors For An Effective Solution

The key factors that define an acceptable solution to NRAO's I/O needs include:

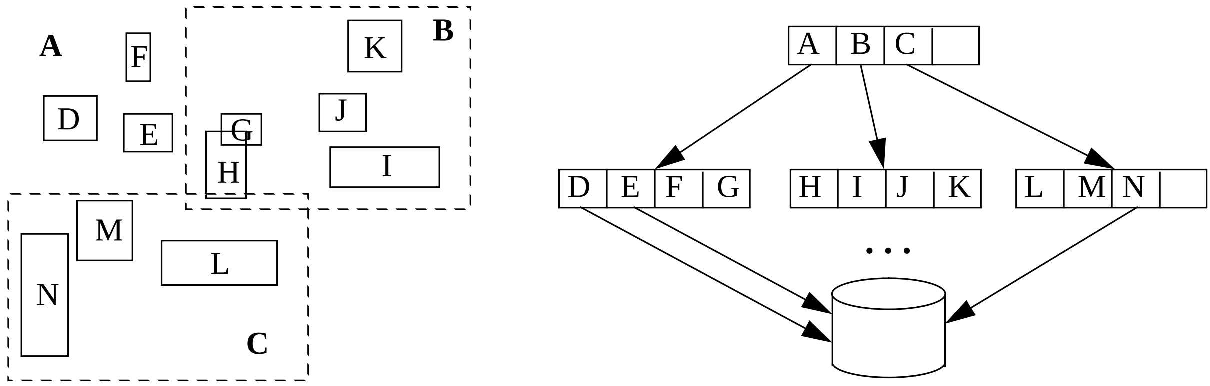
- *Increased data retrieval rates on average*. The main objective of the work detailed here is to help alleviate NRAO's I/O bottleneck, so that NRAO can perform its research mission more effectively and can utilize its computing resources more fully
- *Scalability.* In order for the solution to remain viable, the solution adapted now must be able to handle the larger data volumes projected in the near future. Degradation of performance must be graceful as data volumes increase.
- *Flexibility to access data in different manners.* In order to be useful for a significant portion of NRAO's current and future systems, the solution must be flexible enough to support a fair range of data types and access methods effectively.

- *Acceptable performance of update and logic delete operations*. Obviously, NRAO would like updates to perform as quickly as possible, but due to the relatively lower volume of such requests it is acceptable that performance of update operations be less than that of retrieval operations. However, the level of performance degradation cannot be “too severe”.
- *No dramatic increases in storage space*. Due to the cost of storage devices, it is important that any solution not require a drastic increase in the amount of storage needed for files. An acceptable yardstick is that space requirements should not increase by more than 2 or 3 fold on average.
- *No increased programming burden for NRAO application software developers*. A solution should not significantly increase the burden on programmers to use files. Our goal is too actually decrease the burden on programmers through object-oriented techniques.

4.0 The Search for an Appropriate File Structure

Choosing a file structure that matches the data access requirements of an application or group of applications can dramatically improve performance. An appropriate file structure will reduce the overall number of I/O operations necessary to perform requests, increase the *effective* I/O bandwidth (i.e. the bandwidth of *useful* data accessed), or both. The central idea to most of the structures we considered is adjusting the physical locality of records stored in the file to the needs of the applications using them. If groups of records that are likely to be accessed together are physically stored close to each other, the number of I/O operations can be reduced (fewer blocks of data will be needed) and the average I/O operation time can also be decreased (less overall head movement). An indexed sorted file scheme provides a simple example. Suppose we wish to access a range of data along the sort key. First, using the index will (potentially) reduce the number of I/O operations necessary by eliminating all of the file blocks not within the specified range (this is a simplification as we have ignored the overhead of using the index). Second, since the file is sorted along the key we are interested in, consecutive data blocks are physically consecutive on the disk. Therefore, the disk head will need to move less distance for each block requested (this is also a simplification: disk fragmentation may cause disk blocks to be physically split up. However, on average head movement should be reduced).

To improve NRAO’s I/O performance we looked at several file structures and evaluated their suitability based on the criteria discussed in section 3.4. Structures evaluated include indexed sorted files, R [4]and R+ trees [9], quadtrees[10], *k-d* and *k-d-b* [11] trees, grid files [5] and PLOP-hashing files [6,7]. We



(*a*) 2D objects (*b*) R-tree representation

Figure 6 - Sample Two dimensional R-Tree

found that both grid and PLOP-hashing files are good candidates and chose PLOP files for our implementation.

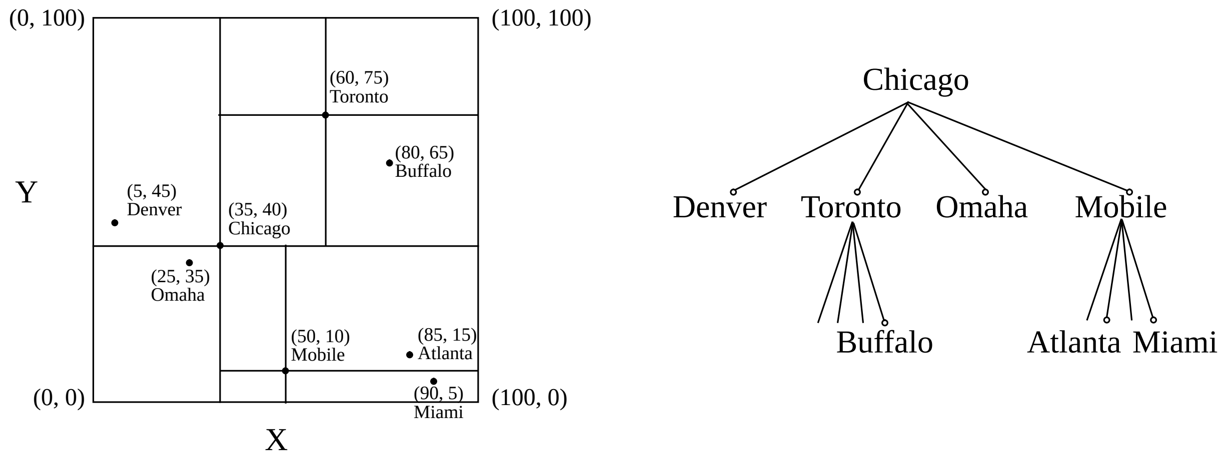
Indexed Sorted Files

Indexed sorted files offer good retrieval performance for single point and range queries when an index and sorting order can be exploited; update, insert and delete operations are straightforward and perform well; they are relatively easy to implement. However, indexed sorted files do not perform well for retrievals that cannot exploit the indexing and sorting scheme, such as retrieving a range of data to be sorted by a key other than the file's primary sort key. Since NRAO applications often need to access data sorted in different ways, it is clear that although sorted indexed files can improve performance in certain cases, they are not flexible enough to handle a large portion of NRAO's needs.

R and R+-Trees

R trees [4] are height-balanced trees that can be used as a file indexing scheme similar to B-trees [12]. The tree consists of leaf nodes containing k-dimensional data objects and intermediate nodes which define a bounding hyperrectangle for all of their child nodes and pointers to the child nodes. Figure 6 shows a set of two dimensional data objects and the corresponding R-tree representation[2]. R+-trees [9] have the same basic structure, but improve retrieval performance by using different algorithms to insert, update and retrieve data objects. The main change is in allowing data objects to be split up to avoid overlap of intermediate nodes. This improves a search for objects within a *k*-dimensional hyperrectangle by reducing the number of paths in the tree that need to be searched. Both R and R+ trees work well for files that consist of

2. Figure 6 is reproduced without permission from Figures 3.1 and 3.2 on page 510 of [9]. Minor modifications have been made.



(*a*) 2D partitioned data space (*b*) Point quadtree representation

Figure 7 - Sample Two dimensional Point Quadtree

spatial data objects, such as map regions in a geo-data application or two and three dimensional objects frequently found in CAD applications. However, NRAO does not use this type of data and therefore the R and R+-tree approaches are not appropriate for our work. If NRAO does collect spatial data in the future, this approach should be reevaluated.

Quadtrees

Quadtrees [10] are a class of hierarchical data structures that are based on the principle of recursive decomposition of space. Point quadtrees successively break up a k-dimensional data space into 2^k pieces surrounding data points in the database. The pieces break the data space into regions whose values in each dimension are either greater than or less than the value in data point around which the split occurred. The data space partitioning is tracked using a 2^k-ary tree. For example, in a two dimensional data space on the Cartesian plane, a split around a point A will break the data space into $2^2 = 4$ regions corresponding to the NW, NE, SW and SE quadrants surrounding the split point. Figure 7 shows a sample partitioning of a 2D data space containing city locations and its quadtree representation[3]. Point quadtrees where at most one data point is allowed in each region have performance characteristics similar to simple indexed files. Point queries are efficient if the tree is kept balanced ($O(\log n)$, base 2^k). However, since there is no guarantee of physical locality of “close together” points in the file, range queries using point quadtrees suffer the same drawback as regular indexed files - data may be scattered all over the file. Range queries corresponding to

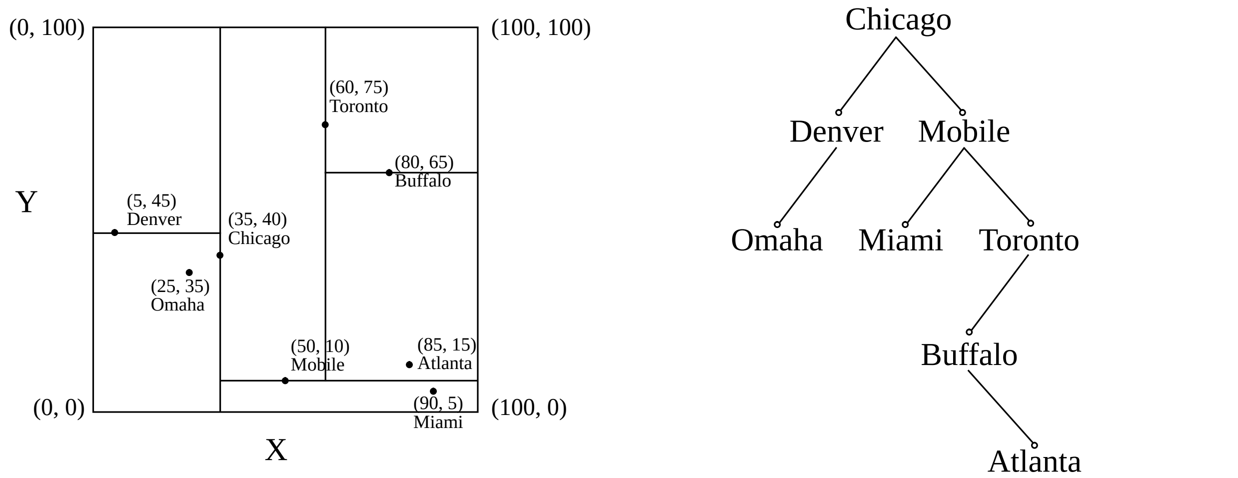
3. Figure 7 is reproduced without permission from Figure 2 on page 191 of [10]. Minor modifications have been made.

a small fraction of the index space may still perform well (relative to no index or simple index schemes), but large range queries will likely require most or all of the data file to be accessed.

If the simple quadtree approach is extended to allow multiple points per region (e.g. the number of records that will fit in a disk block), then performance for range queries can be improved through increased data locality. Compared to the single sort key approach, physically grouping data together along several keys trades-off the lesser performance of queries along the primary key for improved performance for queries along any of the other keys. The idea of trading off the performance of one query type for several other query types is not unique to quadtrees; in fact, all of the remaining files structures discussed employ this concept.

Overall, the point quadtree (with multiple points per region) has several favorable properties for NRAO's environment. The ability to define which keys are of importance and to have relatively good performance for queries using any of these keys fits NRAO's requirement for range retrieval along different keys very well. The storage space required (ignoring internal fragmentation, if any) should not exceed twice the normal non-indexed file size and will be significantly less if the size of the key data is small relative to the size of the rest of the data in a record. Average retrieval performance across all queries should be improved (though some queries may be slower). Insert, update, and delete operations will perform fairly well ($O(\log n)$) in the average case.

However, quadtrees suffer from some drawbacks as well. First, the tree must be kept balanced if retrieval performance is to be optimized. Doing so causes insert, update and deletion times to be increased in the worst case and adds complexity to their implementation. Second, as with all indexed approaches, searching the index adds file accesses to requests, decreasing performance. Third, the splitting scheme employed by quadtrees, i.e. that each split partitions every dimension into 2 parts is somewhat inflexible. This scheme forces each dimension to be split the same number of times. For spatial data, such as data on a Cartesian coordinate system, this may not cause any problems. However, when used with other types of data, forcing each dimension to be split equal number of times may prove to be restricting. For example, consider a typical NRAO file where the keys include time, antenna pair (*baseline*), U, V & W values, and polarization type. Both polarization and baseline values are finite and have a limited range (roughly 4000 values for baseline and 1-4 values for polarization). Quadtrees are not flexible enough to allow polarization to be a key dimension for any but the most trivial file sizes since there could be at most four splits before all possible values for polarization were exhausted. The same problem may occur for baseline in very large



(*a*) 2D partitioned data space (*b*) *k-d* tree representation

Figure 8 - Sample Two Dimensional *k-d* Tree

files. Due to the drawbacks listed above and the availability of file structures that solve some of them, quadtrees were not considered the best file structure for our work.

K-d and K-d-b Trees

A close relative to the quadtree is the *k-d* tree [3] (and its balanced tree version the *k-d-b* tree [11]). Like the quadtree, the *k-d* tree is a hierarchical data structure that recursively partitions the data space. Unlike the quadtree, the *k-d* tree is a binary tree and breaks the k-dimensional data space into two pieces during each split rather than 2^k pieces (Figure 8[4]). On the negative side, the *k-d* tree is a “taller” tree than a quadtree (when k > 1) meaning that queries will need to access more index nodes during a search. For point queries, the retrieval time is still *O*(log n), but with a log base of two rather than 2^k. On the positive side, there is no restriction on the number of splits each dimension must have or their order. Thus a *k-d* tree can be used for a wider range of access patterns and file types than the quadtree. For example, the typical NRAO file discussed earlier may split the *polarization* dimension twice, *time* five times and *baseline* three times.

With the exception of those areas just discussed, the *k-d* tree has very similar characteristics to the quadtree. On the plus side: 1) *k-d* trees offer flexible definition of the key structure; 2) storage requirements should not exceed twice the original, non-indexed file size; 3) if multiple points are allowed in the leafs of the tree, data locality along all dimensions is improved for range queries; and 4) average insert, update and delete operation perform acceptably. On the negative side: 1) accessing data still requires stepping through

4. Figure 8 is reproduced without permission from Figure 40 on page 231 of [10]. Minor modifications have been made.

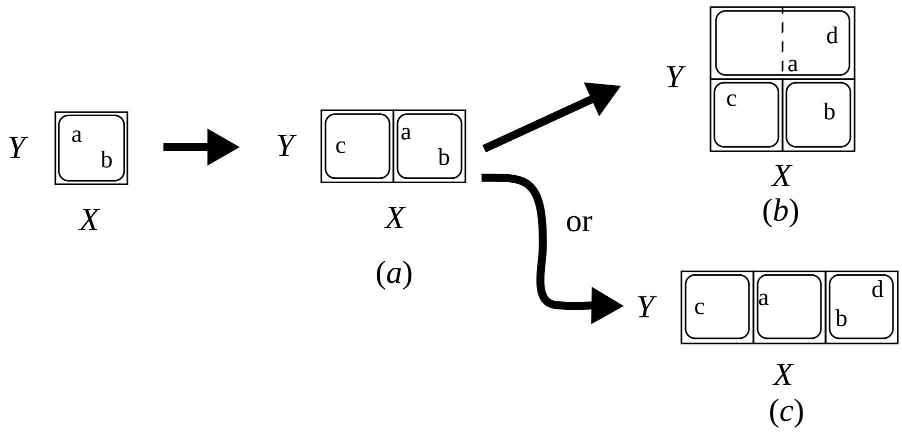
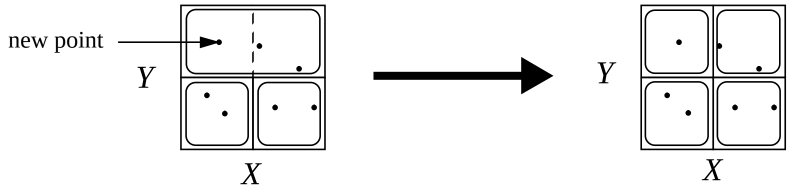


Figure 9 - Building a Two Dimensional Grid File
(assume each bucket can hold up to 2 data points)

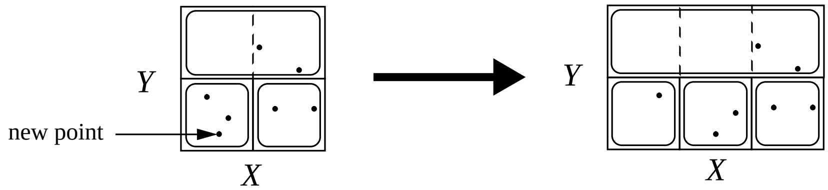
a multi-level index structure, which adds overhead; and 2) balancing the tree is required, increasing the implementation complexity and time required for the insert, update and delete operations.

Grid Files

The basic building block of a grid file is a *bucket* which is a block of storage (e.g. a disk block) where some number of data records can be stored. A grid file starts out with one empty bucket and data is inserted into the bucket until it overflows. When overflow occurs, the data space is logically split in two along one of the *k* dimensions, a new bucket is created, and the data points in the overflowed bucket are realigned to reflect the new partitioning (Figure 9*a*). When another bucket overflows, the data space is again partitioned along one of the dimensions. Figure 9*b* shows a split along a different dimension than the first split and Figure 8*c* shows a split along the same dimension as the first split. In order to save space and reduce the amount of realigning necessary when a bucket overflows, each split is a *logical* partition of the data file. New physical buckets for all of the new logical regions are not allocated immediately, but rather are added only when needed. Figure 9*b* demonstrates how only one new physical bucket is created during a split that created two new logical regions (the rounded rectangles represent physical buckets). A directory structure is stored in a separate file to keep track of the current mapping between logical regions and physical buckets. When further overflows occur, the directory is first checked to see if the overflowed physical bucket contains more than one logical region. If so, the combined region is split into two smaller regions along one of the split values that already exists, and one part is mapped to a new physical bucket (Figure 10*a*). Otherwise a new split is created for the data space (Figure 10*b*). A similar process is followed for deletes,



(*a*) Overflow in a multiply mapped bucket



(*b*) Overflow in a singly mapped bucket

Figure 10 - Bucket Overflow in a Two Dimensional Grid File

where empty or near empty buckets are merged together or two adjacent splits, also called *slices*, are combined removing the logical split.

Grid files match NRAO's requirements very well.The structure of the file is tailored to providing good performance for range queries using multiple keys by 1) ensuring data locality within each bucket and 2) reducing the search space through the use of the directory. Like all index structures, the directory adds overhead to each access. Unlike *k-d* trees and quadtrees, the grid file directory is of depth one. Therefore each point access requires exactly two I/O operations, one to access the directory and one to access the main data file. While insert, update, and delete operations are more complex than in any of the other file structures discussed so far, they are still possible and perform fairly well in the average case. A split in a large file can be a significant amount of work, but should occur infrequently.

The grid file is very flexible about the number of keys (dimensions) and the order in which splits occur, allowing the general file structure to be useful for a range of specific data files. Grid files are designed to allow different algorithms to be used in the split and merge decisions, i.e. when to merge, which dimension to split or merge, and around which value to split or merge. This flexibility allows the implementor to decide whether a more accurate, more computationally expensive strategy is needed or a simpler but possibly less accurate strategy is sufficient. Strategies for choosing the split dimension include round-robin, weighted round-robin, random, or a more complex strategy, including strategies that adapt to the data inserted. Strategies for where to split include (roughly in decreasing order of accuracy) using the median value in the to-be-split slice, the mean value of the slice, the median or mean value of the bucket being split, the mid-point in the current slice's range, etc. A common strategy is to choose the dimension to split in a round-robin fashion and to choose the value to split around based on the median value in the entire slice of the chosen dimension. Note that using the median value of the whole slice may produce better split results than, say, using the mean value, but it is computationally very expensive since all points in the slice must be accessed to make the decision. Using the median split strategy we can guarantee that the grid file will not be more than twice the size of the original file (ignoring the size of the directory). This is because a new bucket of storage is allocated only when an existing bucket is more than 100% full and since only one new equal sized bucket is added during the split, the utilization after splitting must be greater than 50%.

Piecewise Linear Order-Preserving Hashing (PLOP) Files

PLOP files are very similar to grid 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. Data points are stored in the bucket that has corresponding values in each dimension. However, unlike grid files, each logical bucket corresponds one-to-one with a particular physical storage bucket (also called a primary bucket or page). 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, the need for the grid file directory is eliminated because the physical primary page can be determined uniquely from the logical region. A separate tree structure, called a *split tree*, is needed to keep track of the splits for each dimension, but this structure is very small relative to the file and is assumed to be able to fit in local memory.

To help reduce the number of splits occurring in the data space (to conserve storage), PLOP files do not require that all overflowing buckets cause a split. Rather the overflowing bucket can instead create a new auxiliary bucket or *extent* in a separate file and can put extra data points there. Extent buckets can be connected together to form a chain of data pages that all map to the same subregion in the data space. The decision of when to split vs when to chain extent pages is up to the implementor, but 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, such as six. Figure 11 shows a sample data space and two possible ways of building a PLOP file, one splitting whenever a slice is greater than 100% full, the other splitting when the

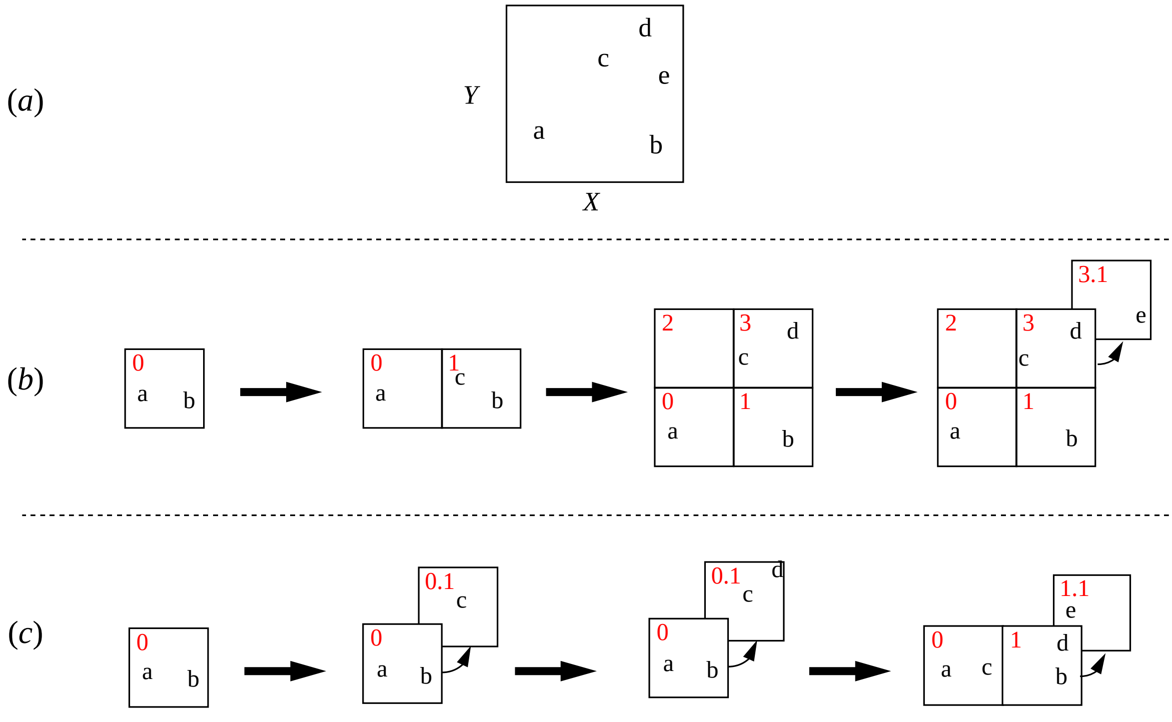


Figure 11 - Building a Two Dimensional PLOP File

depth of the chain is greater than two (the points are inserted in the sequence a, b, c, d, e). The final transition in Figure 11*c* warrants some explanation. The addition of *e* would create a new extent, the second, for region 0. Therefore, the strategy determines a split is necessary. We choose the X dimension at random and split at the median point which causes two points to be put into the left split (region 0) and three points into the right split (region 1). Finally, region 0's extent page is reclaimed and an extent is created for region 1 to hold its extra point. As demonstrated in Figure 11, PLOP file partitioning depends heavily on the split strategy employed. Using different split strategies the same insertion sequence of data can yield significantly different data partitions and storage utilizations. The final state of Figure 11*b* requires more space (5 pages), but has greater precision (more precise partition values and hence more precise searches) than Figure 11*c*.

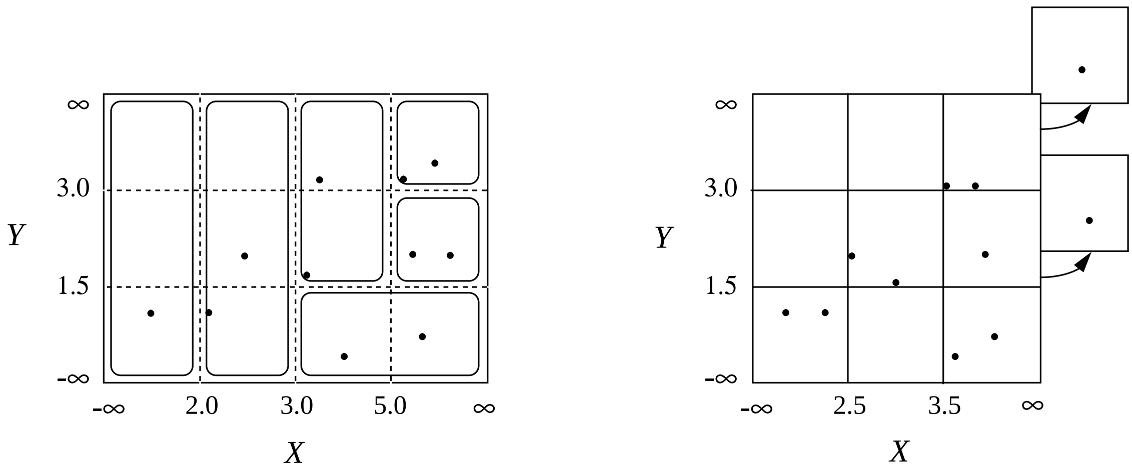
Like grid files, PLOP files fit NRAO's requirements very well and for the same reasons. PLOP files are tailored to multidimensional range queries, exploiting data locality along multiple keys, and reducing the search space through partitioning of the data space. Point accesses are efficient in the average case, though the upper bound on the number of accesses needed to access a single item depends on the split strategy.

Grid Files vs PLOP Files

Grid files [5] and PLOP files [6,7] both eliminate the hierarchical index structure used in *k-d* trees and quadtrees by imposing more structure on the data. Thus both the need for a logarithmic number of accesses to an index file and the complexity of balancing a tree structure are eliminated (although new complexities in properly maintaining the file structure are introduced). Grid and PLOP files are similar in their general approach, both are based on partitioning the k-dimensional data space into k-dimensional subregions where all points within that region are stored. By storing points within a subregion together, both grid and PLOP files support increased data locality along all dimensions, thus facilitating range queries. In general both grid files and PLOP files are very good candidates for NRAO's environment. In fact, they are the two most appropriate files structures we have studied. However, there are several properties that differentiate them, including point access performance, storage utilization, ease of implementation and parallel sorting characteristics.

Grid files require a fixed two I/O operations to access a single bucket and therefore have a predictable single bucket access performance. On the other hand, PLOP files require a varying number of operations for a single bucket access and as a result have less predictable performance. However, eliminating the directory allows PLOP files to access a single bucket in as little as one I/O operation. This means that PLOP files can achieve average bucket depths (the number of chained pages in a bucket) of less then two and can therefore potentially enjoy a performance advantage on average over grid files for point accesses and range queries. Kriegel and Seeger [6] compared the average relative performance of grid and PLOP files (measured as the number of I/O operations necessary to fulfill a request) for point queries and several types of range queries. They found PLOP files to be superior for all queries for all sample data sets tested, which included data sets with uniform and non-uniform distributions. This study provides some concrete examples that PLOP files can achieve low average bucket depth and superior performance over a range of data distributions and is generally encouraging for PLOP files. However, this study does not prove conclusively the superiority of PLOP files for all data distributions and insertion patterns.

A fundamental difference between grid files and PLOP files is how each handles allocating storage. Grid files attempt to keep the physical storage requirements low by grouping adjacent logical regions together in one physical bucket, while PLOP files do it by trying to reduce the number of logical regions (and hence physical buckets) by allowing crowded regions to use more space as needed. These different



(*a*) Grid File (6 Physical Pages Needed) (*b*) PLOP File (11 Physical Pages Needed)

Both files were created by inserting the following points in order: (1,1), (2,1) (2.5,2), (3,1.5), (3.5,3), (4,0.5), (5,3), (5.5,2), (6,0.75), (7,4), (8,2)

Figure 12 - Inserting a Sorted List Into Grid and PLOP Files

approaches yield different relative storage utilizations under different circumstances. It appears that the insertion pattern is particularly important. In the same study where Kriegel and Seeger compared the performance of range and point accesses, they compared the relative storage requirements of grid and PLOP files [6]. They found that using random insertion of points in the test data sets resulted in PLOP files having better storage utilization in all of their test cases. While it appears that PLOP files perform somewhat better than grid files for random insertion patterns, there is reason to believe that PLOP files may not perform as well with some non-random insertion patterns. An example of a realistic non-random insertion pattern is inserting points from a previously sorted data set. Figure 12 compares the final layout of a grid file and a PLOP file after inserting a series of points sorted by their X value in ascending order. For both files the split axis was alternated between the X and Y dimensions, starting with the X axis, and both chose the split value as the median value of the slice where the split was to occur. The PLOP file was split when an insertion caused a bucket depth of greater than two (i.e. a second extent page would be needed). Figure 12*a* demonstrates how the grid file can adapt to this insertion pattern by mapping sparsely populated areas of the partitioned space to one physical bucket. Figure 12*b* shows how the PLOP file does not adapt well to this particular insertion pattern. Once a split occurs along the X dimension, no points will ever be added to slices with lesser X values. Whenever the space is partitioned along the Y axis, all of the new buckets are wasted except the one for the largest valued X slice (the right-most in the figure). Overall, it appears that while PLOP files may marginally outperform grid files in storage utilization for certain insertion patterns, grid files are more likely to perform well for a wide range of insertion patterns and seem to offer a better worst case utilization.

Another difference between grid and PLOP files is their ease of implementation. Both file structures have complex algorithms for implementing their dynamic insert, update and delete routines. The actual complexity of the implementation differs within each file structure depending on the split and merge strategies used. However, because grid files allow physical regions to be mapped to different size and shape logical regions, the insert, update, and delete operations are more complex than those of PLOP files. Since the mapping between logical regions and physical buckets is one way (from logical to physical only), determining whether to split a full bucket or simply break up a physical bucket that spans multiple logical regions involves searching the directory entries of adjacent logical regions in all dimensions. Determining how to split up a multiply mapped physical bucket can also be complex. Similarly, determining when and how to merge during a delete operation is difficult. PLOP files do not require any directory searches or the subsequent directory updates. In fact, once a split is determined to be necessary, the split process for PLOP files is conceptually straightforward: create the new physical buckets, go through each bucket along the split slice moving the appropriate data to the newly created buckets, and add an entry into that dimension's split tree.

For retrievals, PLOP files are again easier to implement. There is no need for an intermediate access to the directory since there is no directory for PLOP files. Retrieving all points in a bucket is simple, by following the chained pointers. Due to the many-to-one relationship between logical and physical buckets, efficiently implementing range searches for grid files requires additional software to track which logical regions have already been searched. For example, suppose three adjacent logical buckets in a grid file are mapped to the same physical bucket (such as in the left-most column of Figure 11*a*) and we issue a range query that encompasses this entire region. The grid file implementation must be able to track which physical buckets have been searched to avoid excess I/O operations and duplicate data. Overall, PLOP file I/O algorithms are conceptually simpler and easier to implement.

PLOP files have the property that each physical page maps to exactly one logical bucket. This means that the physical buckets for any slice along a dimension are disjoint from the physical buckets of all other slices in that dimension. Therefore sorting the data of a query that spans several slices for the sort key can be broken into multiple smaller sorted queries, each working on one slice. This strategy can improve performance in two ways. First, breaking the problem into smaller chunks improves the performance of the overall sorting computation. Sorting has a $O(n \log n)$ time complexity, where n is the number of items sorted. Assume a query that retrieves 2^{20} items can be broken into 16 equal sized disjoint sets. The combined time to sort all the disjoint sets is $16(2^{16} \log 2^{16}) = 2^{24}$ time units as compared to $2^{20} \log 2^{20} = 1.25 \times 2^{24}$ (a difference of 25%). Second, each of the slices could 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 $O(n/p \log n/p)$ where p is the number of slices along the sort axis that are spanned by the query. Grid files do not have this property as the boundaries for physical buckets can span multiple slices. It is therefore difficult to separate the data into disjoint sets along the sort key.

After weighing the differences between grid and PLOP files, we decided to use the PLOP file structure for our NRAO test implementation. This does not imply that grid files are not acceptable for NRAO's environment, but rather that PLOP files appear to be better. The results of Kriegel and Seeger encouraged us that we could achieve better retrieval performance using PLOP files. The ease of implementation of PLOP files was also an attractive feature as was the ability to implement parallel sorting algorithms easily. The main drawback of PLOP files, that of potentially poor data space partitioning and low storage utilization for non-random insertion patterns (as in Figure 12), could be a serious problem in NRAO's environment because it is likely that non-random insertion patterns, especially sorted data sets, will be used in the creation of PLOP files. We felt this problem could be alleviated by collecting statistics about the data set and pre-partitioning the PLOP file based on these statistics. Since the partitioning is based on statistics about the entire file, the partitioning will not be biased by the insertion order, and should be better than a dynamic partitioning strategy. The remainder of the paper will describe our PLOP file implementation, the results we achieved, and our plans for future work.

5.0 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`). Figure 13 shows the relationship between these classes. 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. Several additional classes were implemented to support the PLOP file hierarchy, including classes implementing individual PLOP file pages and interferometry

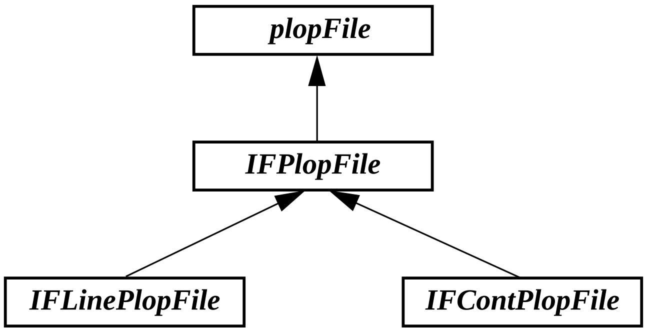


Figure 13 - PLOP File Class Hierarchy

records (`plopPage` and `IFRecord`, respectively), file statistics for calculating split values (`IFFileStats`), and trees and lists of split values.

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 the object oriented paradigm because it supports modularization and encapsulation of functionality and, through inheritance, extensibility and code re-use. The purpose of the base `plopFile` class is to define those objects and functionality that are common to all PLOP files. The `IFPlopFile` subclass contains additional objects and implements most of the functionality for the interferometry data files we tested. Finally, the type-specific PLOP file classes, `IFLinePlopFile` and `IFContPlopFile` define and implement the remaining file-specific functionality.

Figure 14 shows the essential details of the `IFLinePlopFile` class definition, including the parts derived from its ancestors and the virtual functions that were redefined. For brevity some details have been omitted. 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 easily be added to the existing interface as needed.

Retrieval mechanisms for the PLOP files are implemented in a separate C++ class, `queryMgr`. By packaging the retrieval portion separately from the file object, the user can define multiple queries simultaneously in the same application, each related to the same or to different PLOP file objects. To perform a query, the user first specifies the query using the `queryDesc` class and then tells the PLOP file object that a new query is being started (via a call to the `newQuery()` member function). The user is returned a queryMgr object that is used to request the records that satisfy the query. The interface for `queryDesc`

Derived From plopFile:

protected:
```
plopPage        *currentPrimPage;
plopPage        *currentExtPage;
int             primaryFd,
                hdrFd, extFd;
char            *fileNamePrim,
                *fileNameExt;
int             numAxes;
int             numKeys;
keyDesc         *keys;
int             pgSize, recSize,
                recsPerPage;
int             pageHdrSize;
long            totalRecs;
long            totalExtents,
                nextExtentPgNum;
long            totalPrimPages;

// slice trees for each dimension
sliceTreeNode   **axisSlice;

// # slices for each axis
int             *numAxisSlices;

// # points in each slice of each axis
long            **numPointsInSlice;

void createPlopFiles();
```
public:
```
plopFile(char* primNm, char* extNm);
~plopFile();
```

Derived From IFPlopFile:

protected:
```
int sortPageList(pageInfo**, IFRecord**, int, int);
void readFitsWritePlop(IFFitsFile&);
void createSplits(IFFitsFile&);
void initPlopFile();
calcPageNumber(int*, IFRecord);
calcPageNumber(int*, float*);
calcPageNumber(int*);
```
public:
```
int             numFrequencies;
float           baseFreq;
float           freqDelta;
date            observeDate;
float           IFVal, RAVal, DecVal;

addRecord(IFRecord*, pageInfo*);
void reportStats();
int open();
int createFromFits(int, IFFitsFile&);
```

IFLinePlopFile Specific:

New:
private:
```
int                 numFreqSel;
frequencySelector   *freqSel;
```

Redefined IFPlopFile virtual functions:
protected:
```
void copyFitsInfo(IFFitsFile&);
void setupKey(IFFitsFile&);
```
public:
```
int readHeader();
int writeHeader();
```

Figure 14 - IFLinePlopFile Class Definition

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 `queryDesc` object and then setting the range or ranges to search for each dimension 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 the time dimension, two `queryDesc` "set" calls are necessary, `set("Time", 0.1, 0.2)` and `set("U", -1000.0, 1000.0)`, and one call is necessary to set the primary sort dimension, `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).

```
public:                                                  public (cont.):
// constructor and destructor
queryDesc(IFPlopFile*);                                  // set functions
~queryWindow();                                          void set(int key, int lower, int upper);
                                                         void set(char* keyName, int lower, int upper);
// reset functions                                       void set(int key, float lower, float upper);
void resetAll();                                         void set(char* keyName, float lower, float upper);
void reset(int key);                                     void set(int key, polarization lower, polarization upper);
void reset(char* keyName);                               void set(char* keyName, polarization lower,
                                                                  polarization upper);
// sorting functions                                     void set(int key, Time lower, Time upper);
void sortBy(int key, int sortNumber, int sortDir);       void set(char* keyName, Time lower, Time upper);
void sortBy(char* keyName, int sortNumber, int sortDir);          . . .
```

Figure 15 - QueryDesc Class Interface

The only retrieval command currently implemented is `getRecords(int)`. `GetRecords()` returns the specified number of records requested up to the amount remaining in the query. If the query is sorted the records are returned in sorted order, otherwise they are returned in an unspecified order.

6.0 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. 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 PLOP file 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 data value for each antenna pair in the array. By correlating the data in this manner, the array can act like a single very large antenna instead of a group of smaller antennas. 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 correlated 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 range or *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. 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.

Although these two files appear to be very similar there are some important differences. First, 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. For our line-spectrum file, the storage space required to break the file into individual frequency records is 338% more than grouping all frequencies into one record. We therefore decided to keep all frequencies together. The ratio of data bytes to key bytes in the line-spectrum records is fairly high, 372 data bytes vs 32 key bytes, or 11.6 times. The layout of the continuum file is different. Since there is no frequency key, there is only one measurement per record. Consequently, the continuum file has a very low data-to-key storage ratio of 0.6 (12 data vs 20 key bytes).

The two files differ in one other important aspect, size. The line-spectrum file is considered to be a small file by NRAO standards, while the continuum is a medium sized file. The line-spectrum file contains 126,092 records (in the “grouped frequency” record format described above), each 404 bytes in length (~ 51 million bytes). The continuum file contains 8,440,092 records, each 32 bytes in length (~ 270 million bytes).

6.2 File Conversion

Before creating a PLOP file, the dimensions (also called *axes*) of the data space must be chosen. The dimensions chosen determine which keys will be able to narrow the search space when data is accessed and which dimensions will not. Therefore, choosing the dimensions must be driven by data access requirements. 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 for a dimension. 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 the 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. The precision of each dimension is also effected by the split strategy employed during creation of a PLOP file. Round-robin split strategies spread out the available number of splits evenly among the dimensions, while weighted schemes give more precision to some dimensions at the expense of other dimensions.

NRAO's typical access patterns for line-spectrum files primarily use seven different keys: *time*, *baseline*, *polarization*, *frequency*, *source*, *U*, and *V*. Of these *time* is the most frequently used in range queries, but the other axes are all used regularly in various applications. Due to the increased storage cost associated with storing a separate record for each frequency (as discussed in section 6.1), we decided not to use frequency as an axis. We tried three strategies for defining the dimensions of the data space.

Our initial PLOP file conversion used all six of the remaining keys (after frequency was eliminated), creating a six-dimensional data space. Our implementation automatically pre-partitioned the data space using the following strategy. First, since the number of polarizations is limited (a maximum of four discrete values), known beforehand (two in this case), and the records are distributed evenly among the polarization values, we split the file along each polarization value. Similarly, there are usually a limited number of sources (three in this file) and we know the values beforehand. However, there is no guarantee that the data points are evenly distributed among the different sources. To keep the partitioning scheme reasonably simple, we ignored this last fact during our test and allocated a split for each source value, realizing that a poor distribution among the sources would waste a significant amount of space (this did not turn out to be the case, as the distribution of data among the sources was fairly even). For the remaining four axes, we collected histogram statistics for each dimension independent of the other dimensions. We then split the data space roughly an equal number of times in each dimension using the histogram statistics to determine split values such that each slice along a dimension had approximately the same number of points. Since the histogram statistics are collected independently for each dimension this partitioning is not optimal because it fails to look at the entire multi-dimensional distribution and therefore ignores inter-dimensional relationships. This scheme should, however, give better results than dynamic partitioning. The size of the final data space was set to be approximately the size of the data space needed if all buckets were 100% full.

A full description of the partitioning process for the line-spectrum file follows. First, the total size (number of buckets) of a 100% full data space was determined. Each bucket contains 10 records and there are 126,092 records, so 12,610 buckets are needed if each bucket is on average 100% full. Next we allocated the polarization partitions (2) and then the source partitions (3), cutting the data space into 2x3 = 6 separate regions. We estimated how many partitions would be necessary to evenly distribute the remaining splits among the remaining dimension as

$$\sqrt[D-d]{\frac{B}{\prod_{1}^{d} N_d}}$$

where D is the number of dimensions in the data space, d is the number of dimensions whose splits have already been allocated, N_d is the number of splits allocated to dimension d (the dimensions in this example are numbered as follows, polarization - 1, source - 2, U - 3, V - 4, baseline - 5, time - 6), and B is the total number of buckets desired (12,610 in this case). For the U dimension, the number of splits was the fourth root of [12,610 / 2x3] (6, rounded down), while for V, the number of splits was the cube root of [12,610 / 2x3x6] (7, rounded down). The number of splits for baseline and time were determined similarly to be 7 and 8 respectively, rounding up for the final dimension. Thus, we created a final 2×3×6×7×7×8 six-dimensional data space comprised of 14,112 primary buckets.

To convert the line-spectrum file to a PLOP file, we prepartitioned the space as described above, creating all of the empty primary pages. We added the data points without using any dynamic splitting strategy to further partition the data space. Using this conversion strategy resulted in poor storage utilization and bucket depth. In fact, approximately two-thirds of the primary buckets were empty, the maximum chain

.

	Source			
Time	*Slice #*	0	1	2
	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

Table 1: Data Distribution in 4D Regions Defined by Time and Source Axes

bucket depth was 29, and the storage utilization was 52%.[5] One observation is that if we had used a Grid file implementation, it is likely that this problem would not have occurred.

After collecting distribution statistics for each pair of axes, it was discovered that there was a poor distribution of data for source and time pairs. Table 1 shows the distribution of points to each of the four-dimensional regions defined by the intersection of slices from the source and time axes. If the data were evenly distributed, there should be approximately 5,250 points in each region. Almost all of the regions in Table 1 are either significantly over- or under-populated, resulting in deep bucket depths or wasted space, respectively. In fact, each zero value in the table means that a 2×6×7×7 space (588 pages) is completely empty. This problem arose because there is an inter-relationship between the time and source axes. Frequently, one particular source is observed for a period of time, then another source is observed for a while, and so on. This explains the distribution in Table 1. Source 0 was observed first for almost the entire range of times in the first time slice, then sources 1 and 2 were observed, alternating between then during the next few time slices. Finally, source 0 was studied again at the end of the file's time range.

To attempt to correct this poor partitioning, we chose to eliminate the source axis, reducing the data space to five dimensions. This action, of course, eliminates source as a key that can reduce the search space for a query. We prepartitioned the file as before, resulting in a 2×8×9×9×10 five dimensional data space (12,960 subregions). The storage utilization and bucket depth statistics were better than the previous case,

5. Storage utilization for PLOP files is the percentage of space taken up by actual data records out of all the space allocated for the file. The total storage allocated to the file includes all primary and extent pages and the header file. Therefore, the utilization rate includes PLOP file overhead and internal fragmentation within data pages. Utilization = (record size * number of records)/total space allocated.

but were still fairly poor. Storage utilization rose to 56%, empty buckets dropped to 60%, and the maximum bucket depth dropped to 19. The average bucket depth when only non-empty buckets were considered was 4.50 and the average depth when considering all buckets was 2.36. These numbers were not acceptable from either a storage viewpoint or a retrieval performance standpoint.

Analyzing the data distribution pattern revealed that there is a relationship between the U and V axes as well as a relationship between U, V pairs and baselines. This time we decided to eliminate the U and V axes, again reducing the number of axes that can be used in range queries to reduce the search space. The resulting data space was a 2×79×80 three-dimensional data space (12,640 subregions) comprised of polarization, baseline, and time axes. The results of the file conversion for this strategy were very good. Storage utilization rose to almost 79%, empty pages were eliminated completely, and average bucket depth was reduced to approximately 1.25. In addition, the maximum bucket depth was reduced to 2, with 92% of all points stored in the primary page and 8% stored in the extent page (an average point depth of 1.08). These results mean that for this particular file we were able to reduce the average number of accesses for a point query to 1.08 indicating that we would expect better average case performance for point using this file over a similar grid file while maintaining the same worst case performance (2 I/O operations).

Since the continuum file is very similar in layout, we used the same three axes as the final line-spectrum file. The continuum file was prepartitioned into a 4×128×130 (66,560 subregions) three-dimension space. Conversion of the file showed that the pre-partitioning for this file also worked well. Storage utilization was 66%, there were only 128 empty pages (0.2%), average bucket depth was 1.50, and average point depth was 1.11. While the maximum bucket depth was five, depths of greater than two were very rare (only 20 out of 66,560 subregions or 0.03%). The conversion results for both files are shown in Table 2.

The conversion process brought out several important aspects of PLOP files. First, the selection of axes for PLOP files is crucial for good performance and storage utilization. PLOP files work best with axes that are completely independent of one another. Conversely, they do not handle interrelationships between axes very well, often forcing large portions of the data space to be empty or near empty, while other portions are overpopulated. Second, prepartitioning the data space based on fairly simple statistics and a simple heuristic can provide good results even for insertion patterns that would not work well with dynamic partitioning. Because of prepartitioning the converted line-spectrum file had very good partitioning despite being inserted sorted by the time axes. This insertion pattern would have yielded very poor partitioning for

	Line-Spectrum	Continuum
Number of Records	126,092	8,440,092
Record Size (bytes)	400	32
Page Size (bytes)	4,096	4,096
Records Per Page	10	127
Number of Splits Along Polarization Axis	2	4
Number of Splits Along Baseline Axis	79	128
Number of Splits Along Time Axis	80	130
Total Primary Pages	12,640	66,560
Total Extent Pages	3,168	33,320
Header File Size (kilobytes)	5.5	8.7
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 Point Depth	1.08	1.11
Conversion Time (minutes)	8.4	87

Table 2: Conversion Results for Line-Spectrum and Continuum PLOP Files

dynamic partitioning schemes. Third, PLOP files can be created for real files that have excellent performance indicators (average and worst case bucket depth) *and* high storage utilization.

The conversion process involves a large amount of work, including gathering the histogram statistics needed for the partitioning scheme, actually partitioning the data space, and converting each record from the source file format (a FITS file in both cases) to the PLOP format. Therefore, we implemented several optimizations during the conversion process, including reading and buffering of large chunks of source records and ordering of record insertions to minimize file I/O operations to the destination PLOP file. The line-spectrum conversion took approximately 8.4 minutes on a SparcStation IPX using an attached disk while the continuum file took about 1 hour and 27 minutes on the same hardware. These results seem to indicate that conversion time not only increases with file size, but also increases significantly with the number of records converted.

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 for the VLA files, 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 timing information including any software overhead associated with the query. To avoid skewing of results due to file caching, large file copy jobs were run in between successive tests to clear the file cache of any relevant data. All tests were run on a single Sun SPARCStation IPX with 32 megabytes 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 can be 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 (total bandwidth is the rate at which all data, including overhead space and empty space due to fragmentation and non-full pages, is retrieved). For example, we may be able to achieve a high total bandwidth of, say 2 megabytes 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. Increasing effective bandwidth effectively reduces the amount of time spent performing I/O which is exactly what is needed to alleviate an I/O bottleneck.

6.3.1 Test Queries

The twelve queries we chose to test our PLOP file performance and the rationale for the choice are now presented.

- *1) All data, unsorted* - this is a benchmark test to determine the full file retrieval rate. Since all data is desired, the difference between the total and effective bandwidth for this query should reflect only the amount of space used as page overhead and space wasted due to internal fragmentation.
- *2) All data, sorted by time* - this is a common query at NRAO and when compared with the unsorted version (query #1), this query should give a good indication of the performance of sorting for our PLOP files.
- *3) All data, sorted by baseline* - this query was chosen to compare the performance of sorting by different keys. When the PLOP files were created, the time axis was split last. Consequently, all of the

pages for a time slice are potentially consecutive (depending on disk fragmentation) and therefore we would expect sorting by baseline to potentially be slower than sorting by time due to increased disk head movement.

- *4) Time Range (approx. 10%), all other data, unsorted* - benchmark query to compare retrieval time for range query vs all data (query #1).
- *5) Time range (approx. 10%), all other data, sorted by U* - benchmark query to compare sorting time for non-axis sort key unsorted retrieval time (query #4). The exact same range was used as query #4.
- *6) 1 time, 1 polarization, all baselines, unsorted* - this is a common query at NRAO to gather all of the baseline data for a particular time and polarization. Since the ranges for this query narrow the search space considerably, we would expect very fast retrieval times.
- *7) Time range (approx. 10%), 1 polarization, all baselines, unsorted* - a common NRAO access pattern. For multi-source files, this query is similar to accessing all the data for a single source and polarization (often a table mapping source to time range is kept).
- *8) Time range (approx. 50%), 1 baseline, all polarizations, unsorted* - common NRAO query, used for aggregating a set of measurements for a single baseline.
- *9) Time range (approx. 50%), antenna #1, all polarizations, unsorted* - a common NRAO query that is often used to correct or inspect the data from an instrument that may have been malfunctioning during a certain period of time. Searching for one antenna's data means searching for all baselines of which the antenna is a part. Baseline values are stored as a single number with the lower numbered antenna being used as the high-order byte and the higher numbered antenna being used as the low-order byte. For example, the baseline 3-5 (i.e. the baseline comprised of antennas 3 and 5) is stored as 3x256 + 5 = 773. To retrieve data for an antenna, it is necessary to search the range with the antenna as the lower numbered antenna and those ranges with the antenna as the higher numbered antenna. Since antenna #1 is the lowest numbered antenna, to retrieve all of its data, only the range 257-284 needs to be searched (there are only 27 antennas). Therefore, we would expect antenna #1 to have the best retrieval time of all similar antenna searches.
- *10) Time range (approx. 50%), antenna #14, all polarizations, unsorted* - same as query 9, except the antenna searched for is in the middle of the range of antennas. This query will require 14 ranges to be searched, the 13 ranges with antenna 14 as the higher numbered antenna, and the one range with antenna 14 as the lower numbered antenna. This query should take longer than query 9, but less time than query 11.

#	Total Records Retrieved	# Good Records	% Good Records	# Primary Pages	# Extent Pages	Total Data Retrieved (KBytes)	% Good Data (Accuracy)	Time (seconds)	Total BW (KBytes/ sec)	Effective BW (KBytes/ sec)
1	126,092	126,092	100%	12,640	3,168	63,232	78.7%	55.49	1,139	897
2	126,092	126,092	100%	12,640	3,168	63,232	78.7%	70.5	897	706
3	126,092	126,092	100%	12,640	3,168	63,232	78.7%	181.8	348	274
4	14,742	12,636	85.7%	1,422	474	7,584	65.7%	4.10	1,850	1,216
5	14,742	12,636	85.7%	1,422	474	7,584	65.7%	6.85	1,107	728
6	702	351	50%	79	0	316	43.8%	0.271	1,166	511
7	7,371	6,318	85.7%	711	237	3,792	65.7%	2.33	1,627	1,079
8	752	186	24.7%	84	20	416	17.6%	1.45	287	51
9	5,072	4,836	95.3%	504	120	2,496	76.4%	4.03	619	473
10	15,026	4,836	32.2%	1,512	360	7,488	25.5%	14.5	516	132
11	21,778	4,836	22.2%	2,184	528	10,848	17.6%	20.3	534	94
12	720	180	25%	80	20	400	17.7%	1.49	268	48

Table 3 - Query Performance for Line-Spectrum PLOP File

- *11) Time range (approx. 50%), antenna 27, all polarizations, unsorted* - same as queries 9 and 10, except for the largest numbered antenna, #27.
- *12) All times, 1 baseline, 1 polarization, sorted by time* - a common NRAO query. This query gives a time sequence picture of all the data collected from a single baseline and polarization.

6.3.2 Query Results and Observations

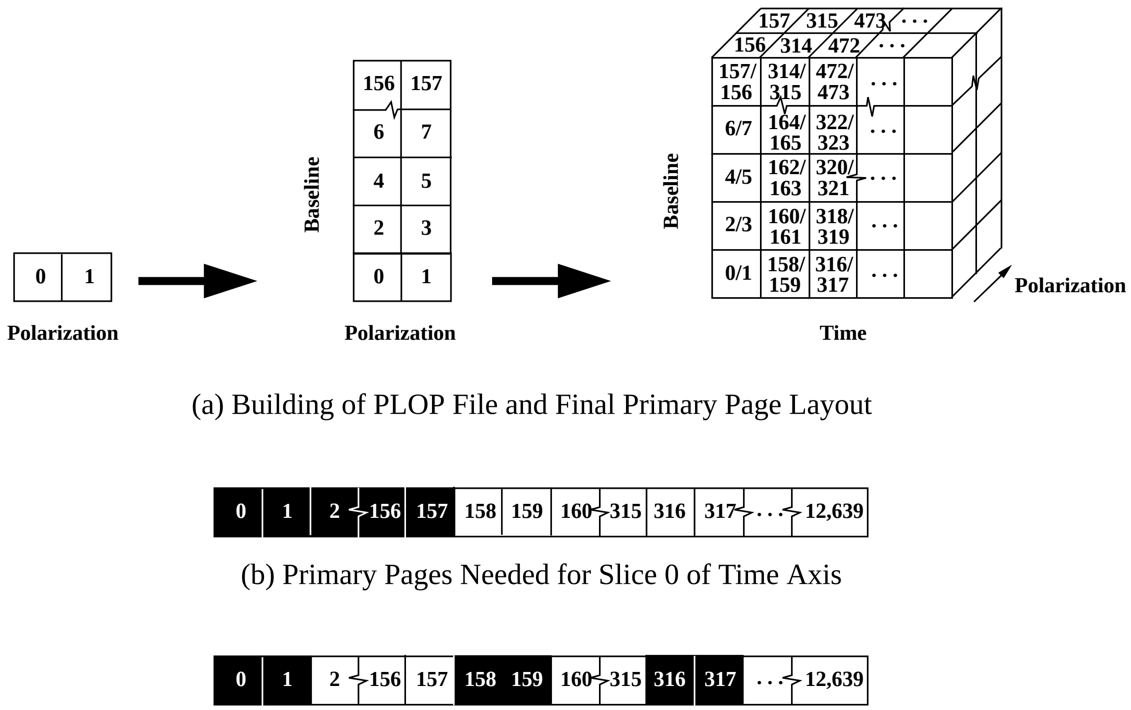
Tables 3 and 4 summarize the results of the tests for the line-spectrum and continuum data sets, respectively. Since we have not implemented an out-of-core sorting algorithm, query 5 could not be run for the continuum file as the resultant data was too large to sort in local memory. We would expect similar results to those achieved for the line-spectrum file scaled to the size of the continuum file.

#	Total Records Retrieved	# Good Records	% Good Records	# Primary Pages	# Extent Pages	Total Data Retrieved (KBytes)	% Good Data (Accuracy)	Time (seconds)	Total BW (KBytes/ sec)	Effective BW (KBytes/ sec)
1	8,440,092	8,440,092	100%	66,560	33,320	399,520	66.0%	375	1,065	703
2	8,440,092	8,440,092	100%	66,560	33,320	399,520	66.0%	637	627	413
3	8,440,092	8,440,092	100%	66,560	33,320	399,520	66.0%	1,382	289	191
4	968,220	951,320	98.3%	7,680	3,344	44,096	67.4%	49.3	894	603
5	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
6	31,673	351	1.1%	256	130	1,544	0.71%	1.29	1,197	8.5
7	242,055	237,830	98.3%	1,920	836	11,024	67.4%	13.3	829	559
8	40,124	13,256	33.0%	260	208	1,872	22.1%	8.0	234	52
9	329,316	326,428	99.1%	2,600	1,184	15,136	67.4%	28.5	531	358
10	673,924	326,172	48.4%	5,200	2,596	31,184	32.7%	175	178	58
11	693,528	0	0%	6,240	1,652	31,568	0%	194	163	0
12	19,379	6,256	32.3%	130	90	880	22.2%	6.3	140	31

Table 4 - Query Performance for Continuum PLOP File

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, possible head movement due to disk fragmentation, and program code and operating system overhead. The difference between the bandwidth figures for the two files can be explained by the 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, especially on a non- or lightly fragmented disk (which we believe our disk was).

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



(a) Building of PLOP File and Final Primary Page Layout

(b) Primary Pages Needed for Slice 0 of Time Axis

(c) Primary Pages Needed for Slice 0 of Baseline Axis

Figure 16 - Retrieval of a Time Slice vs a Baseline Slice

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 of the query. 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 16*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 16*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 or favorable disk/file placement. For the continuum file, the total and effective bandwidth measures were closer to what we expected, that is, 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 exact same data as the fourth query, but sorted the data by a non-axis key, U. The result was that sorting the data increased the retrieval time by 67%. It is our 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 six, seven, eight and twelve 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, smaller record sizes and by having larger slices (caused by the size of the file - the time slice that was searched had 128 primary pages rather than 79 for the line-spectrum file). The accuracy for this query could be increased by reducing the page size, which would reduce the number records per page and force the file to be split more often, reducing the range of each slice. For example, if the continuum file page size were reduced to 1K, only 31 records would fit on a page and, using the same splitting scheme described earlier, would result in a $4 \times 260 \times 262$ data space. The single time, single polarization query would require 260 1K pages instead of 128 4K pages, nearly 50% less data. Of course, this requires more I/O operations and potentially more disk seeks and the overall performance of reducing the page size depends on the actual amount that the disk head must move and the speed of the disk relative to the CPU. This illustrates the inherent trade-off between the precision of each axis and the number of I/O operations necessary to access a range of data.

Queries 9, 10, & 11 illustrate the effect the baseline numbering scheme has on queries using antenna numbers. For both files, the query for antenna #1 (query 9) had the highest accuracy and effective bandwidth and shortest run-time as expected. The higher precision of the continuum file's baseline axis helped to increase the accuracy for the first two antenna queries, reducing the amount of unnecessary data retrieved. The results of the last antenna query for the continuum file indicate that there was no data for antenna #27 in the file, but we would expect that the continuum file would have higher accuracy for this query as well.

6.3.3 Comparison of PLOP Files and Current NRAO Practices

The real test for our approach would be to determine the performance of our PLOP file implementation versus that of NRAO applications using their current file structures and retrieval methods. 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.

7.0 Summary

We feel that the standard paradigm of treating files as untyped, unstructured byte streams is an outdated concept. Rather files should be treated as persistent objects. By developing file objects and applying object-oriented techniques, we feel that high performance I/O can be achieved while better supporting ease-of-use and easing the development and maintenance of I/O related code. This is the ELFS philosophy. In this paper we have presented our experience in applying the ELFS approach to the specific problem of providing high performance multidimensional range search capabilities for radio astronomy data. We developed a base class, `plopFile`, which is implemented using a file structure designed to support multidimensional range searches, the PLOP-hashing file. We extended the base class functionality for interferometry data (`IFPlopFile`) and further extended it for two specific types of interferometry data (`IFLinePlopFile` and `IFContPlopFile`). Due to the complexity of implementing PLOP files, development of the base plopFile class, the `IFPlopFile` derived class and the `IFLinePlopFile` class took a significant amount of work. However, the development of the second derived class, `IFContPlopFile`, took very little effort. Further, we feel that extending the base classes for other applications will be straightforward and relatively easy.

The resulting plopFile class hierarchy contains an intuitive interface that presents the data in a meaningful way to the programmer. The interface also allows the programmer to specify a query ahead of the time when the data will actually be needed. By decoupling the description of the query from the retrieval of results, later parallel implementations can attempt to asynchronously prefetch data to better overlap computation and I/O. To test the performance of our PLOP file objects we ran a series of real world and benchmark queries on real NRAO data files. The results are very encouraging. We believe that we have produced a high performance, extensible multidimensional range searching object that can be extended for use in many applications.

8.0 Recent and Future Work

We have recently 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, 14]) 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.

9.0 Acknowledgments

We would like to thank NRAO for the partial funding of the project which helped make this project possible. We would like to especially thank several individuals at NRAO whose knowledge of NRAO's environment and help with gathering needed data files and statistics was crucial to this project. Thanks to Brian Glendenning, Jeff Crouse, Gareth Hunt, Alan Farris, David Shone, and everyone else at NRAO who helped with this project.

10.0 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] C. J. Date, *An Introduction to Database Systems*, Vol. 1, Fourth Edition, pp. 63-64,Addison-Wesley, 1986.

[13] A. S. Grimshaw, “Easy to Use Object-Oriented Parallel Programming with Mentat”, *IEEE Computer*, pp. 39-51, May, 1993.

[14] For more information about Mentat look at the Mentat home page on the WorldWide Web. Use Mosaic to view http://uvacs.cs.virginia.edu/~mentat.