Optimizing Tertiary Storage Organization and Access for Spatio-Temporal Datasets

(Published in the Proceedings of the NASA Goddard Conference on Mass Storage Systems, March 1995.)

Ling Tony Chen, Doron Rotem, Arie Shoshani
Lawrence Berkeley Laboratory
Berkeley CA 94720

Bob Drach, Meridith Keating, Steve Louis
Lawrence Livermore National Laboratory
Livermore CA 94550

Abstract

We address in this paper data management techniques for efficiently retrieving requested subsets of large datasets stored on mass storage devices. This problem represents a major bottleneck that can negate the benefits of fast networks, because the time to access a subset from a large dataset stored on a mass storage system is much greater that the time to transmit that subset over a network. This paper focuses on very large spatial and temporal datasets generated by simulation programs in the area of climate modeling, but the techniques developed can be applied to other applications that deal with large multidimensional datasets. The main requirement we have addressed is the efficient access of subsets of information contained within much larger datasets, for the purpose of analysis and interactive visualization. We have developed data partitioning techniques that partition datasets into "clusters" based on analysis of data access patterns and storage device characteristics. The goal is to minimize the number of clusters read from mass storage systems when subsets are requested. We emphasize in this paper proposed enhancements to current storage server protocols to permit control over physical placement of data on storage devices. We also discuss in some detail the aspects of the interface between the application programs and the mass storage system, as well as a workbench to help scientists to design the best re-organization of a dataset for anticipated access patterns.

1. Introduction

Large-scale scientific simulations, experiments, and observational projects, generate large multidimensional datasets and then store them temporarily or permanently in an archival mass storage system (MSS) until it is required to retrieve them for analysis or visualization. For example, a single dataset (usually a collection of time-history output) from a climate model simulation may produce from one to twenty gigabytes of data. Typically, this dataset is stored on up to one hundred magnetic tapes, cartridges, or optical disks. These kinds of tertiary devices (i.e., one level below magnetic disk), even if robotically controlled, are relatively slow. Taking into account the time it takes to load, search, read, rewind, and unload a large number of cartridges, it can take many hours to retrieve a subset of interest from a large dataset.

An important aspect of a scientific investigation is to efficiently access the relevant subsets of information contained within much larger datasets for analysis and interactive visualization. Naturally, the data access depends on the method used for the initial storage of this dataset. Because a dataset is typically stored on tertiary storage systems in the order it is produced and not by the order in which it will be retrieved, a large portion of the dataset needs to be read in order to extract the desired subset. This leads to long delays (30 minutes to several hours is common) depending on the size of the dataset, the speed of the device used, and the usage load on the mass storage system.

The main concept we pursue here is that datasets should be organized on tertiary storage reflecting the way they are going to be accessed (i.e. anticipated queries) rather than the way they were generated or collected. We show that in order to have an effective use of the tertiary storage we need to enhance current storage server protocols to permit control over physical placement of data on the storage devices. In addition, these protocols need to be enhanced to support multiple file reads in a single request. We emphasize in this paper the aspects of the storage server interfaces and protocols, as well as simulation and experimental results of the effects of dataset organization for anticipated access patterns.

In order to have a practical and realistic environment, we choose to focus on developing efficient storage and retrieval of climate modeling data generated by the Program for Climate Model Diagnosis and Intercomparison (PCMDI). PCMDI was established at Lawrence Livermore National Laboratory (LLNL) to mount a sustained program of analysis and experimentation with climate models, in cooperation with the international climate modeling community [1]. To date, PCMDI has generated over one terabyte of data, mainly consisting of very large, spatio-temporal, multidimensional data arrays.

A similar situation exists with many scientific application areas. For example, the Earth Observing System (EOS) currently being developed by NASA [2], is expected to produce very large datasets (100s of gigabytes each). The total amount of data that will be generated is expected to reach several petabytes, and thus will reside mostly on tertiary storage devices. Such datasets are usually abstracted into so called "browse sets" that are small enough to be stored on disk (using coarser granularity and/or summarization, such as monthly averages). Users typically explore the browse sets at first, and eventually focus on a subset of the dataset they are interested in. We address here this last step of extracting the desired subsets from datasets that are large enough to be typically stored on tape.

Future hardware technology developments will certainly help the situation. Data transfer rates are likely to increase by as much as an order of magnitude as will tape and cartridge capacities. However, new supercomputers and massively parallel processor technologies will outstrip this capacity by allowing scientists to calculate ever finer resolutions and more time steps, and thus generating much more data. Because most of the data generated by models and experiments will still be required to reside on tertiary devices, and because it will usually be the case that only a subset of that data is of immediate interest, effective management of very large scientific datasets will be an ongoing concern. However, there is an additional benefit to our approach. Even if we accept the premise that users will be tolerant of long delays (i.e. placing orders that take several hours or overnight to fill), it is still in the best interest of mass storage facilities to be able to process requests more efficiently, by avoiding to read data not needed. This translates into savings on the hardware needed to support an average access load.

It is not realistic to expect commercial database systems to add efficient support for various types of tertiary storage soon. But even if such capabilities existed, we advocate an approach that the mass storage service should be outside the data management system, and that various software systems (including future data management systems) will interface to this service through a standardized protocol. The IEEE is actively pursuing such standard protocols [3] and many commercially available storage system vendors have stated that they will help develop and support this standards effort for a variety of tertiary devices. Another advantage to our approach is that existing software applications, such as analysis and visualization software, can interface directly to the mass storage service. For efficiency reasons, many applications use specialized internal data formats and often prefer to interface to files directly rather than use a data management system.

In section 2, we describe in some detail our approach and the components modules necessary to support it. Section 3 describes the storage system interface design for both the write and the read processes. Section 4 contains simulation and experimental results, and section 5 describes a workbench that was designed to help scientists in selecting the organization of datasets that best suits their anticipated access patterns.

2. Technical Approach

The goal is to read as little data as possible from the MSS in order to satisfy the subset request. For example, for geological fault studies the most likely access pattern is regional (in terms of spatial coordinates) over extended time periods. For this application, the dataset should be partitioned and stored as regional "bins" or "clusters" over time, as opposed to the traditional way of storing data globally for one time slice. In general, the portions of a dataset that satisfy a query may be scattered over different parts of the dataset, or even on multiple volumes. For example, typical climate simulation programs generate multiple files, each for a period of 5 days for all variables of the dataset. Thus, for a query that requests a single variable (say "precipitation") for a specific month at ground level, the relevant parts of the dataset reside on 6 files (each for a 5 day period). These files may be stored on multiple volumes. Further, only a subset of each file is needed since we are only interested in a single variable and only at ground level. If we collected all the parts relevant to a query and put them into a single file, then we would have the ideal cluster for that query. Of course, the problem is one of striking a balance between the requirements of all queries, and designing clusters that will be as close as possible to the ideal cluster of each query. This idea is a common methodology used in data management systems (called "physical database design"). However, such methods have not been applied or investigated much in the context of mass storage systems.

In the past two years we have investigated and developed partitioning algorithms for a specific application: simulation data generated by climate models. In that context, we have identified specific access patterns which we characterized as query types. In general, the problem is one of finding the best compromise in how to store the data for conflicting access patterns. We have shown that although the general problem is NP-complete, it is possible to develop effective approximate solutions using dynamic programming techniques [4]. We only discussed below the methodology of our approach and the software modules needed to support our approach. The details of the algorithms used are discussed in [4].

2.1 Functional description of the components

We need to address the components necessary for both writing the reorganized dataset and reading the desired subset. The first component, which we call the "data allocation and storage management" is responsible for determining how to reorganize a dataset into multiple "clusters", and for writing the clusters into the mass storage system in the desired order. The parts of the dataset that go into a single cluster may be originally stored in a single file or in multiple files. The second component, which we call "data assembly and access management" is responsible for accessing the clusters that contain relevant data for the requested subset, and for assembling the desired subset from these clusters. One consequence of this component is that analysis and visualization programs are handed the desired subset, and no longer need to perform the extraction of the subset from the file. The details of the two components are shown in Figures 1 and 2.

On the left of Figure 1, the Data Allocation Analyzer is shown. It accepts specifications of access patterns for analysis and visualization programs, and parameters describing the archival storage device characteristics. This module selects an optimal solution for a given dataset and produces an Allocation Directory that describes how the multidimensional dataset should be partitioned and stored.

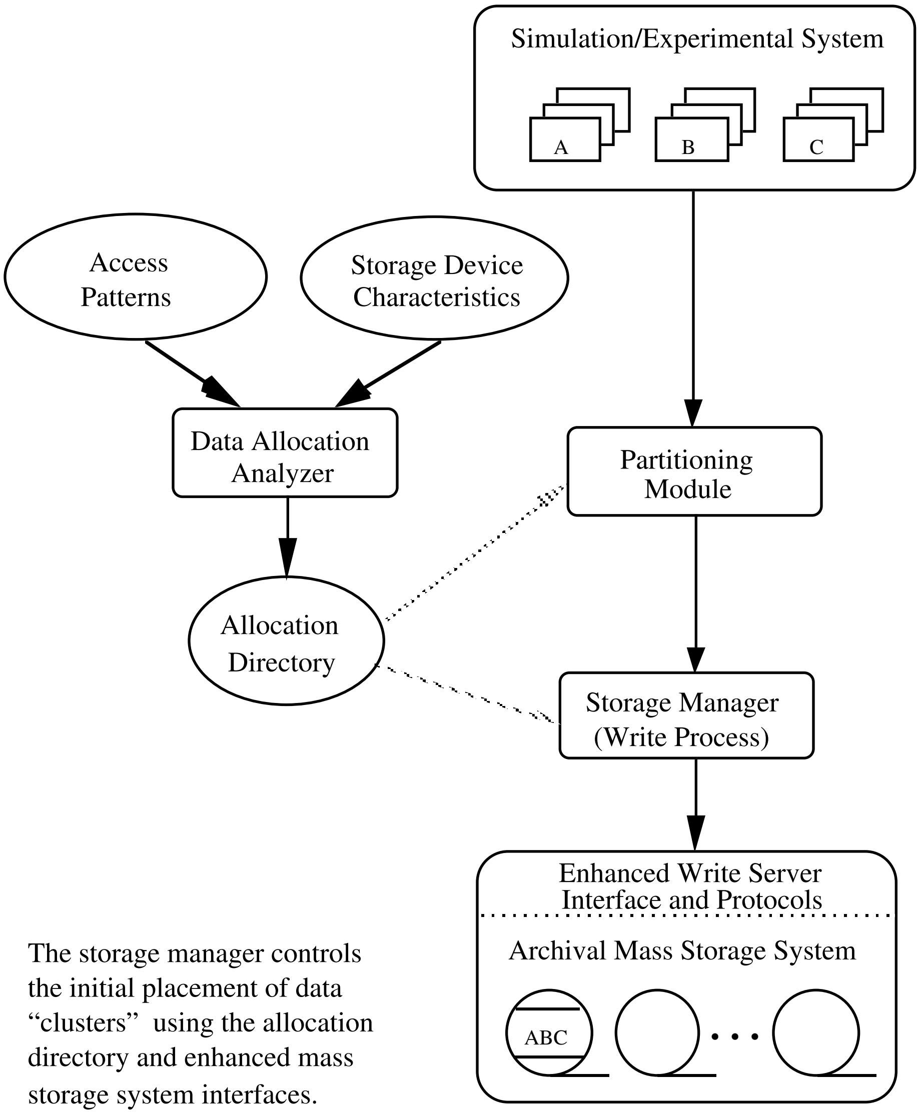

Figure 1: Data allocation and storage management details

The Allocation Directory is used by the File Partitioning Module. This module accepts a multidimensional dataset, and reorganizes it into "clusters" that may be stored in consecutive archival storage allocation spaces by the mass storage system. Each cluster is then stored as a single file, which in most tertiary storage devices today is the basic unit of retrieval (that is, partial file reads are not possible). The resulting clusters are passed on to the Storage Manager Write Process. In order for the Storage Manager Write Process to have control over the physical placement of clusters on the mass storage system, enhancements to the protocol that defines the interface to the archival mass storage system were developed. Unlike most current implementations that do not permit control over the direct physical placement of data on archival storage, the enhanced protocol permits forcing of "clusters" to be placed adjacent to each other so that reading adjacent "clusters" can be handled more efficiently. Accordingly, the software implementing the mass storage system's bitfile server and storage servers, needs to be enhanced as well. More details on the modified protocols are given Section 3.

In Figure 2, we show the details of reading subsets from the mass storage system.

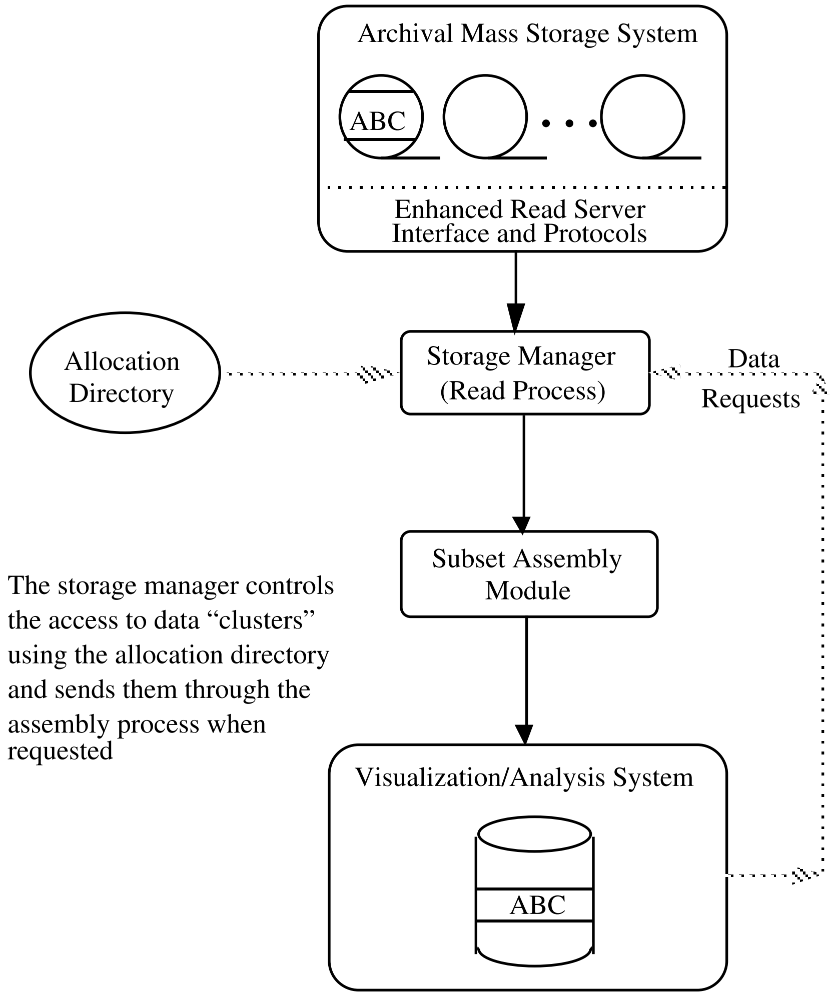

Figure 2: Data assembly and access management details

Upon request for a data subset, the Storage Manager Read Process uses the Allocation Directory to determine the "clusters" that need to be retrieved. Thus, reading of large files for each subset can be avoided. Here again, the bitfile server and storage server of the mass storage system needs to be extended to support enhanced read protocols (see Section 3 for details). Once the clusters are read from the mass storage system, they are passed on to the Subset Assembly Module. Ideally, the requested data subset resides in a single cluster (especially for queries that have been favored by the partitioning algorithm). But, in general, multiple clusters will have to be retrieved to satisfy a subset request, where only part of each cluster may be needed. Still, the total amount of data read will typically be much smaller than the entire dataset. The Subset Assembly Module is responsible for accepting multiple clusters, selecting the appropriate parts from each, assembling the parts selected into a single multidimensional subset, and passing the result to the analysis and visualization programs.

2.3 Characterization of datasets ,queries, and hardware

The typical dataset in climate modeling applications is not composed of just a single multidimensional file for several variables, but rather a collection of multidimensional files, each for a subset of the variables. The granularity of the spatial and temporal dimensions are common to all variables, but some variables may contain only a subset of these dimensions. For example, a typical dataset may have 192 points on the X dimension, 96 points on the Y dimension, 19 points on the Z dimension (i.e. 19 elevations), and 1488 points on the T (time) dimension covering one year ((12 months) x (31 days/month) x (4 samples/day)). This dataset may contain a "temperature" variable for all X,Y,Z,T and a "precipitation" variable for X,Y,T only. For the "precipitation" variable, the Z dimension is implicitly defined at the ground level. A typical dataset may have close to a hundred of such variables, each using a different subset of all the dimensions. Thus, the characterization of a dataset involves a description of each the above dimensions and for each variable the dimensions that apply to it.

The characterization of queries required extensive interaction with the scientists using the data. After studying the information provided by scientists, we have chosen to characterize "query types", rather than single queries. A query type is a description for a collection of queries that can be described jointly. For example, a typical query type might be "all queries that request all X,Y (spatial) points, for a particular Z (height) one month at a time over some fixed subset of the variables". Thus, assuming that the dataset covers 2 years and 20 height levels, the above query type represents a set of 480 queries (24 months X 20 heights). It was determined that providing query types is more natural for these applications. Further, the query type captures a large number of example queries, and thus permits better analysis of usage patterns.

Each query type is defined as a request for a multidimensional subset of a set of variables, where the multidimensional subset must be the same for all variables of the query type. A query type is defined by selecting one of the following 4 parameters for each dimension:

1) *All*: if the entire dimension is requested by the query type.

2) *One*(coordinate): if exactly one point (the coordinate element) of the dimension is requested.

3) *Any:* if one value along the dimension is requested for this query type. Note that it is assumed that any one of the values within this dimension is equally likely to occur.

4) *Range*(low,high): if a contiguous range that starts at low and ends at high of the dimension is requested.

All variables in our application are defined over a subset of the following seven dimensions: X(longitude), Y(latitude), Z(height), Sample, Day, Month, Year. Note that the Time dimension has been split into 4 dimensions that specify the sample within a day, the day within a month, the month within a year, and the year. The splitting of the time dimension makes it possible to specify "strides" in the time domain, such as "summer months of each year", the "first day of each month", etc. Some variables may not have all dimensions defined. For example, "precipitation" is defined at ground level only, and has no height (Z) dimension.

It has been determined that for our application this query type definition encompasses almost all possible queries that users would want in this application area. It was observed (and verified with climatologists) that the One and Range parameters are not used as often as the All and Any parameters. An example of a query type specification is given below:

Temperature, Pressure: All X, All Y, One(Z,0), All S, All D, Range(M,6-8), Any Y

This query type specifies that temperatures and pressures are requested over all X,Y positions, for Height 0 (ground level), but only for sample points and days in the summer months for a single year. A query belonging to this query type can be specified for any year. Thus, if the dataset is over 20 years, this query type represents 20 possible queries, each being a subset of the multidimensional space.

The characterization of the tertiary storage devices should accommodate various types of devices. We identified the following 5 parameters that are needed to characterize any tertiary storage device for the purpose of determining the optimal partition:

1) M (MegaBytes): the capacity of each tape.

2) R (MB/second): sustained transfer rate, excluding any overhead for starting and stopping.

3) Ts(x) (seconds): fast forward seek function. A mapping function between the distance of the forward seek and the time it takes. For example, if it takes 10 seconds to initialize a seek, and 20MB/s thereafter, the seek function is: $Ts(x) = 10 + (x/20)$. In cases where it is difficult to determine the constant value, the seek function is simply x divided by the seek speed.

4) Tm (seconds): mount time. The time it takes to change a cartridge up to the point where we can read the first byte out of the new cartridge. This time includes: unload previous tape, eject previous tape, robot time to place previous tape back on shelf, robot time to retrieve new tape from shelf, mount new tape, setup tape to be ready to read the first byte.

5) FO (bytes): extra File Overhead. This is the overhead (in bytes) involved in breaking one long file into two shorter files. If retrieving the long file takes T2 seconds, and retrieving the two consecutive shorter files requires T1 seconds, then the file overhead $FO = (T1-T2)*R$, where R is the transfer rate defined in point 2 above.

This five parameter model has proven to be sufficient to describe most removable media systems such as robotic tape libraries or optical disk juke boxes. In the latter case, we adjust the seek time component of our cost function to zero as it is negligible compared to the time it takes to dismount and mount a new platter. Measurements of these parameters for two robotic tape systems are given in the next section.

3. The Storage System Interface Design

The developmental and operational site for our work is the National Storage Laboratory (NSL), an industry-led collaborative project [5] housed in the National Energy Research Supercomputer Center (NERSC) at LLNL. The system integrator for the National Storage Laboratory is the IBM Federal Sector Division in Houston. Many aspects of our work complement the goals of the National Storage Laboratory.

The NSL provides two important functions: a site where experiments can be performed with a variety of storage devices, and facilities necessary to support storage and access of partitioned datasets. The first mass storage software developed at the NSL was an enhanced version of UniTree, a system originally written in the 1980s at LLNL and later commercially marketed by DISCOS, OpenVision, and T-mass. The enhanced NSL system, called NSL-UniTree, features network-attached storage, dynamic storage hierarchies, layered access to storage-system services, and new storage-system management capabilities [6]. A commercial version of NSL-UniTree was announced late in 1992 by IBM U.S. Federal. Work on a new storage software system, called the High Performance Storage System (HPSS) [7], is also in progress at the NSL. A central technical goal of the HPSS effort is to move large data files at high speed, using parallel I/O between storage devices and massively parallel computers. HPSS also seeks to increase the efficiency of scientific and commercial data management applications by providing an extensible set of service quality attributes that can be applied to storage resources and devices.

NSL-UniTree manages data using a typical hierarchical file system approach compatible with widely used operating systems such as UNIX. Access to the file system is provided via standard FTP and NFS file transfer protocols, or via a file-oriented client application programming interface (API). The initial interfaces developed for HPSS also support FTP, NFS, and a Client API. However, large scientific datasets are more efficiently viewed as a collection of related objects, rather than as a single large file or multiple independent files. To provide better access to such datasets, we have developed a specification for a more suitable interface between mass storage systems and application software to provide better control over data storage organization and placement for data management clients, such as the data partitioner and subset assembler discussed here. Though modifications to existing interfaces will also be required for HPSS, these will be much smaller in scope because of the system's better ability to classify data by service quality.

3.1 The "write process"

Enhancements necessary to support this work were designed for an experimental version of NSL-UniTree. For the Storage Manager Write Module to provide new attributes related to physical placement of clusters on the mass storage system, modifications to the NSL-UniTree file transfer protocols were developed. Unlike most current implementations that do not permit control over the direct physical placement of data on archival storage, the modified protocol provides a space allocation scheme for storing related data "clusters" and the piece-wise writing and reading of these "clusters".

We designed a functional interface between the data partitioning engine and the mass storage system that provides the ability to control allocation of space and physical placement of data. The approach taken is to define several "Class of Service" (COS) attributes associated with clusters and cluster sets and provide them to the storage system via a modified FTP interface. These cluster-related COS attributes consist of:

1) a cluster set ID.
2) a cluster sequence number.
3) a frequency of use parameter.
4) a boundary break efficiency.

The cluster sequence number identifies the linear relationship between the clusters, and in effect tells the storage system the desired order for storing the clusters. The frequency of use parameter indicates the desirability of storing a cluster close to the beginning of a tape (or to a suitable dismount area) to avoid seek overhead. The boundary break efficiency is a measure of how desirable it is that a cluster stays adjacent to its predecessor. This attribute is used to determine whether separation of two clusters across tape volumes should be avoided if possible.

The COS attributes for a set of clusters are provided to the storage system prior to delivery of the individual data clusters. In reality, they are treated as the initial cluster in a set. This permits the storage system to assign the the rest of the clusters to tape volumes for the desired tertiary storage device. We call an ordered collection of clusters assigned to a single physical volume a "bundle". Thus, the last step of the partitioning process, (i.e. the bundling of clusters such that a bundle can fit on one physical volume), is done by the storage system. This was considered necessary for situations where precise storage system parameters may not be known to the partitioning engine. In addition, this provides the storage system management with an ability to override the partitioning engine's decisions in order to prevent storage system overload or wasted space on physical volumes.

The interface design for NSL-UniTree is shown in Figure 3. As can be seen, this mechanism allows the partitioning engine to determine what it perceives to be optimal data layout for a given device destination. However, it gives final control to the storage system. Data is transferred to the storage system in cluster sets with the COS attributes sent as the first cluster. There is no strict requirement that all clusters be contained within a cluster set, but a containment relationship is necessary if the benefits of data association are to be realized. Clusters that are provided to the storage system independently will be stored individually (i.e., as normal files).

So as not to constrain the partitioning engine unnecessarily, no limits on cluster set size have been established. This, however, means that a cluster set might be larger than the available storage system disk cache. To prevent cache overflow, the clusters are organized into sets whose sizes are manageable by the storage system.

Once the cluster set attributes are available in the storage system, the bundling function is called to assign clusters into bundles for internal use by the storage system's migration process. The migration server builds a bundle table and iteratively examines that table to confirm existence of complete bundles. To provide the necessary inter-cluster cohesion, bundles are migrated to tertiary storage levels only when complete. Note that bundles may not necessarily represent an entire cluster set. This depends on the storage characteristics of the targeted tertiary volume.

To ensure clusters are stored without unnecessary volume breaks (which would result in an additional media mount penalty for the reading process), the migration server checks destination volume space availability prior to actual bundle migration. If there is insufficient space to store the entire bundle on the destination volume, the bundle migration will be deferred for a finite period that can be set by storage system management policy. The idea behind this deferral is to allow non-bundled files to be migrated first in the hope that this results in the mounting of a new (i.e., empty) volume. Storage system management can override deferral in situations where the system's cache space is in danger of exhaustion. In some storage systems, a full disk cache is a fatal condition.

3.2 The "read process"

We wish to support an object view for application programs where the objects are multidimensional datasets and requests can be made for subsets of these datasets for a single variable at a time. The function of the "subset assembler" is to take such a request from the application, figure out what clusters to request from the mass storage system (if more than one is needed to satisfy the request), and assemble the relevant parts of each cluster into a single multidimensional file to be returned to the application. Consequently, the application programs do not need tp deal with the data assembling details.

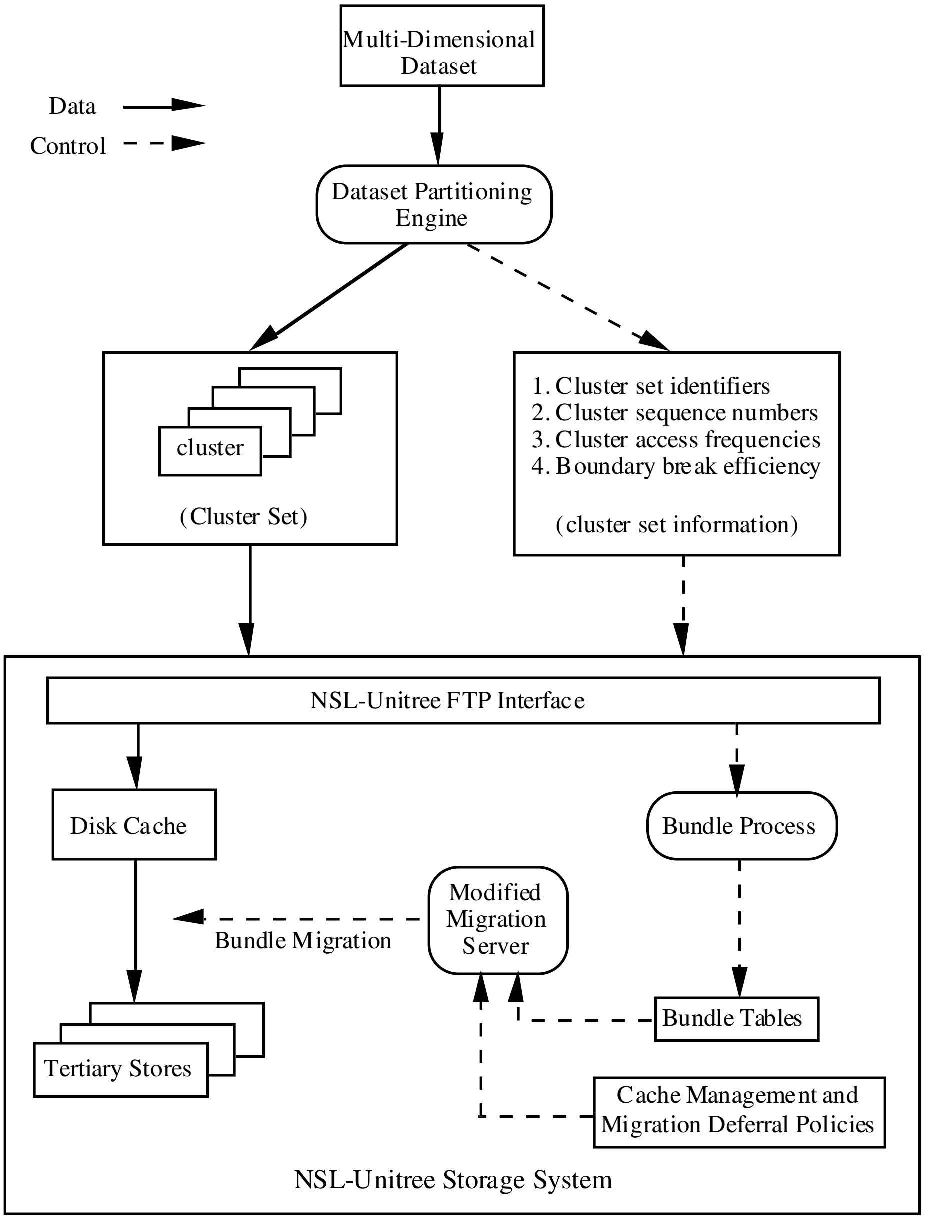

Figure 3: NSL-UniTree enhancement implementation

One of the more difficult tasks that must be accomplished to achieve the subset assembly is the selection of relevant parts from each cluster and the transposition of the dimensions for those parts into the order desired by the application. We decided to take advantage of existing software to perform this function. Currently, the climate modeling files are stored in DRS format [9]. Each file is linearized on the dimensions, and there is a description file associated with each data file. A DRS library exists that performs selection of a part of a multidimensional file, transposing it as desired. The DRS library tries to allocate adequate memory to perform this function in memory. However, when the file is too large to fit in memory, a buffer management algorithm is used to optimize the use of available buffers for files residing on disk. One of the advantages of dealing with clusters, which are relatively small files, is that it is more likely that enough memory can be allocated for these files, and thus the selection and transposition operations can run efficiently.

The Storage Manager Read Module supports efficient reading of clusters. The desired interface to the mass storage system for the read process is one which supports a single request for multiple files corresponding to the set of clusters that the subset assembler needs. When only a single cluster is needed to satisfy the subset request, the read module needs to mount the proper volume, position to the cluster and read only that cluster. When multiple clusters are needed to satisfy a subset request, the read module needs to ensure reading of clusters in such a way that no unnecessary rewind of the volume take place. Thus, the storage system management should treat this request as a request for an unordered set of files; that is, disregard the specific order that the files were mentioned in the request. It should read the files in the order that is internally most efficient, depending on what volumes are mounted at the time of the request and the order of files on the volume. Abilities to support this type of optimization are supported in most modern storage systems, including NSL-UniTree. This capability was recently tested at the NSL with an Ampex DST robotic tape system. We make use of the DRS library and the NSL-UniTree storage management systems as shown in Figure 4.

As can be seen from Figure 4, the subset assembler accepts a request from the application program using a specially designed API language. This request typically consists of the name of a dataset, a variable, and range specifications for each of the dimensions that the variable is defined on. The subset assembler consults the Allocation Directory, and identifies which clusters are needed to satisfy the request. It then issues a request to NSL-Unitree to stage the files representing these clusters into the cache.

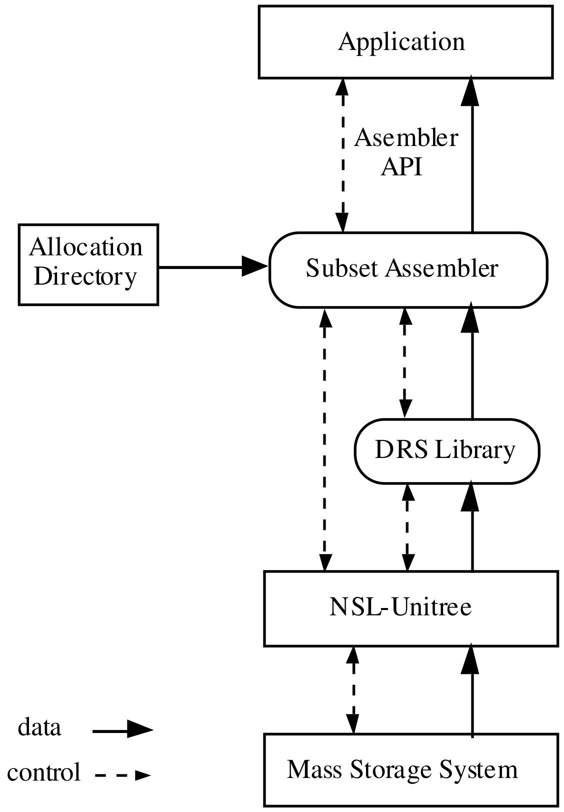

Figure 4: Subset assembler and its interface to NSL-Unitree and DRS library

At this point the assembler does not know in which order files (clusters) will be staged. NSL-Unitree supports a "check status" function where the requesting program can check if a single file was already staged to disk. Therefore, the assembler needs to issue a series of checks to see which of its requested files has been staged. Once a positive response is received for a specific file, the assembler issues a request to the DRS library to read the relevant portion of the file and to transpose the data into the desired order. The information for composing the DRS request is derived from the original application request as well as information from the allocation directory on how each cluster is structured.

The subset assembler places the data which are returned as a result of the DRS request into a pre-allocated buffer space that is large enough to hold the entire subset requested by the application. It then repeats the above process, one cluster at a time, until all clusters have been read. For most visualization applications this subset is small enough to fit in memory. Otherwise, it will be stored on the user's workstation disk. There is the potential of performing these reads in parallel, but we have not concentrated on parallel disk reads, since the major bottleneck for the application is reading files off robotic tape storage.

We plan to partition a commonly used dataset and use this subset assembly process for visualization applications in the summer of 1995. We have performed several experiments and simulations to verify the expected benefits of partitioning a dataset into clusters. These were performed directly on the device, and do not involve the read process described above. We expect the read process to add negligible overhead to the measurements made, since the main component of the response time is reading data from tertiary storage. This needs to be verified through direct use of the subset assembler. The results of the experiments and simulations that were performed so far are given in Section 4.

3.3 HPSS interface implications

We plan to extend our work to the NSL's HPSS storage system as well. There are significant differences between NSL-UniTree and HPSS that may improve the effectiveness of the current mass storage system interface design.

HPSS already implements a COS capability [8] that defines a set of performance and service quality attributes assigned to files and affects their underlying storage resources. COS definitions are designed to help provide appropriate service levels as requested (or demanded) by HPSS clients. In the current HPSS implementation, COS is a data structure associated with a "bitfile", a logical abstraction of a file as managed by an HPSS Bitfile Server. The storage resources used to store the bitfile's data are provided by HPSS Storage Server objects. These Storage Server objects (virtual volumes, storage maps, and storage segments) are associated with a COS-related data structure called a "storage class" that identifies device-dependent characteristics of the type of storage provided by the objects.

Using an HPSS Client API library that currently mirrors POSIX.1 specifications, applications such as the read and write processes described here can specify an existing COS identifier for a file, or fill in COS "hint" and "priority" structures to describe desired (or required) service quality attributes for the file. User-specified priorities which may be NONE, LOW, DESIRABLE, HIGHLY_DESIRABLE, or REQUIRED, affect how the hints will be treated by the Bitfile Server. Using these capabilities, it should be a straightforward process to provide to HPSS with the COS attributes generated by the dataset partitioning engine, and to have these attributes interpreted properly by the servers.

Creating a new file through the Client API is currently performed through an "hpss_Open" call whose input parameters can include pointers to the hint and priority structures. The Bitfile Server will be required to locate an appropriate type of storage if these structures are provided by an application. On the other hand, if null pointers are passed, the Bitfile Server will be free to use a default COS definition for the new bitfile. The hpss_Open call returns a pointer to the COS definition actually used by the Bitfile Server so that applications may monitor the level of service they receive.

The Storage Server's storage segment object is the conventional method for obtaining and accessing "internal" storage resources. Clients of the Storage Server (this would normally be the Bitfile Server, but could be a data management application authorized to use Storage Server APIs) are presented with storage segments of specific storage class, with address spaces from 0 to N-1 (where N is the byte length of the segment). The Bitfile Server, or other client, provides a storage class identifier and an allocation length during creation of new storage segments. During the Storage Server's allocation of new physical space, only storage maps that have proper storage class are searched. To ensure locating free space of the appropriate type, the storage class should represent a service conforming to the client-specified COS hints and priorities passed to the Bitfile Server when creating new files.

The COS capability in HPSS was designed to be extensible, and additional attributes can be implemented to exert greater influence over server actions for data cluster placement and migration operations. A stated goal for future releases of HPSS is better integration with large data management systems and COS attributes for controlling placement or collocation of related data on physical media in HPSS. This will allow HPSS to be easily used by the read and write processes described here.

Unlike NSL-UniTree, the separation of the Bitfile Server, Storage Server, and Migration/Purge Server in HPSS is clearly defined. Modular APIs exist at each server interface, providing different implementation choices using HPSS. One approach is to expand the Bitfile Server's present COS definition to match the requirements of the data partitioning engine and assembly process. The current bundling algorithm could be implemented as added code in the Bitfile Server to ensure inter-cluster cohesion is maintainable. Another approach is to write a new application that could entirely replace the Bitfile Server. This application, if appropriately authorized, could access the internal storage class metadata structures of the Storage Server, and would also be able to provide application-specific COS attributes to interested storage consumers. As a Bitfile Server replacement, it would be able to create its own mappings of COS to storage class, thus explicitly controlling the actions of the Storage Server, and therefore able to enforce external partitioning decisions at the internal device level.

4. Simulation and Experimental Results

4.1 Measurements on hardware characteristics

We have performed detailed timing measurements on an Exabyte Carousel Tape System as well as an Ampex D2 Tape Library System, to validate our hardware model and also collect the appropriate parameters for the model. The results of our experiments are shown in Table1.

Note that the file overhead for the Ampex system is quite large (11 seconds, which is the time needed to transfer 141 MB) due to our lack of control over the behavior of the robotic system. In contrast, the Exabyte has a relatively small file overhead. In order to remain device independent, we treat each device as a black box and measure its performance. Our first experimental results from the Ampex were both larger than expected, and less consistent from file to file. We speculated that the speed of the device might be high enough that a degree of inertia might be present in hardware, or that it might be presumed in software, such that consecutive file reads at high speeds were unattainable in our environment.

To test this hypothesis, we inserted a small pad file (1 KB) between each of the larger test files. This resulted in much faster, and much more consistent read times for the test files. Consequently, the average file overhead decreased from 11 to 3.25 seconds.

The effects of file overhead on query response times are discussed in the Section 4.3 below.

	Capacity (MB)	Transfer Rate (MB/s)	Seek Speed (MB/s)	Mounting (Seconds)	File Overhead (seconds)	File Overhead (MB)
Exabyte	4500	0.265	31.25	315	0.24	0.064
Ampex (high FO)	25000	12.864	503.32	39	11.0	141.5
Ampex (low FO)	25000	12.864	503.32	39	3.25	41.8

Table 1: Measurements of hardware characteristics

4.2 Response Time Results

Based on these hardware measurements, we were able to apply our partitioning algorithms to an actual PCMDI dataset. We could then compare the response times of queries before and after we apply the partitioning method. The partitioning method puts query types into groups, such that all query types in each group have at least one variable in common. Obviously, there is no need to consider the effect of a query type on another if they access no variables in common. This simplifies the presentation of possible solutions to the designer. The query types defined by the designer for the actual dataset are shown in Table 2 along with the amount of data that each needs to access.

query type	variable and dimensions specification	Megabytes requested
	query types for group 1	
1	U,V,W for any month at ground level	76.2
2	U,V,W for any month at all height levels	1447.5
3	U,V,W for any day at any height level	1.6
4	U,V,W for any month at any level for any Y	3.0
5	U,V,W for any year at all levels for range of Y	2171.3
	query types for group 2	
6	T for any month for all for all X,Y,Z	482.5
7	T for a range of 3 months for any height level	76.2
8	T for any sample on any height for a range of Y	0.07
	query types for group 3	
9	A cloud variable for any month for all X,Y	50.8

Table 2: Query types and amount of data requested

Note that groups 1 and 2 have 5 and 3 query types, respectively, and group 3 has only one query type. It turns out that groups with only one query type are quite common, since for some variables there is only one predominant type of access. It is always possible to find an optimal solution for a query type belonging to such a group, since there are no potential conflicting query types. Therefore, we show here only one such representative group.

Note that the amount of data requested varies from less than a megabyte to over 2 gigabytes. Queries requesting small amounts of data are typically for visualization purposes, while those requesting large amounts of data are typically for summarization and post processing. Queries for intermediate amounts of data are typically needed for analysis, such as Fourier transformation. The post processing type queries are less important to optimize since they take so long that users might as well wait for an overnight processing.

Tables 3 and 4 show the response time for the 9 query types of Table 2. The response times are expressed in minutes, where "original" and "new" refer to the times before and after partitioning, respectively. The new timings on the Ampex were actual measured times on the real system, while all other times are calculated times based on the measured hardware model. It was impractical to run the queries before partitioning because their response time takes several hours. We also show the optimal times, which are calculated assuming that all the information to answer the query is in a single file, and that the file is positioned at the beginning of a tape. Thus, the optimal time is equal to the time to mount a single tape, plus the time to read the first file.

Query type	Megabytes requested	Optimal	Original	New	Ratio
1	76.2	6.45	174.97	6.45	27.13
2	1447.5	92.85	174.97	92.85	1.88
3	1.6	1.72	30.55	4.08	7.48
4	3.0	7.35	174.97	92.85	1.88
5	2171.3	274.27	2142.60	1118.72	1.92
6	482.5	32.01	174.97	40.83	4.28
7	76.2	6.45	535.65	6.45	83.05
8	.07	3.25	30.55	3.25	9.40
9	50.8	8.44	237.30	8.44	28.12

Table 3: Exabyte carousel response times (in minutes)

Query type	Megabytes requested	Optimal	Original	New	Ratio
1	76.2	0.75	9.80	2.73	3.59
2	1447.5	2.52	9.80	2.73	3.59
3	1.6	0.65	1.60	2.73	0.59
4	3.0	0.65	9.80	2.73	3.59
5	2171.3	3.47	117.00	29.07	4.03
6	482.5	1.27	9.80	10.67	0.92
7	76.2	0.75	29.30	1.90	15.42
8	.07	0.65	1.60	1.90	0.84
9	50.8	0.72	9.80	0.72	13.61

Table 4: Ampex D2 tape system response times (in minutes)

The dataset we experimented with, contained 57 variables (each defined over all or a subset of the seven dimensions X,Y,Z,S,D,M,Y) and 62 query types. These query types were derived after extensive interviews with scientists interested in this dataset. The query types were partitioned into groups as explained above, and each group was analyzed separately. The tables only show the query types that access the wind velocity vector U,V,W, the temperature T, and one cloud variable that had only a single query type associated with it. However, they are representative of the response times for other query types as well. In practice, most of the variables are accessed by a single query type, and only a few variables are accessed by 2-5 query types.

The original layout of the dataset was one where all the variables for all X,Y, and Z for a period of 5 days was stored in a single file. Files were stored one after another according to time, until the next file would not fit on the same tape. At this point a new tape was used and the process continued until all the data was stored. This storage method represents the natural order that data was generated by the climate simulation program, which, in general, is a poor organization for typical access patterns. The original response times were calculated on the basis of this actual storage of the dataset.

We note that all the new response times on the Exabyte are better than the original times. For the Ampex tape system, the improvements in response time are not as dramatic as in the case of the Exabyte tape system. The main reason for this is that the Ampex has a much larger file overhead than the Exabyte tape system. The large file overhead resulted in larger files being created by the partitioning algorithm. In the case of variables U,V,W, each file corresponds to one month of data for all X,Y, and Z, which comes to roughly 1.5 GB of data per file. And in the case of variable T, each file corresponds to roughly two Z levels of data for all X,Y, and T, which comes to roughly 700 MB of data per file. The consequence is that queries that ask for a small amount of data end up reading a single large file to answer the query. This is especially obvious for query types 3 and 8, where the result is that the response time is even slower than with the original data organization. As will be seen in the next section, the improvement of the file overhead from 11 seconds to 3.25 seconds made a significant difference in the results. In addition, our algorithm can take advantage of partial file reads (no file overhead) to further improve performance.

As expected, the partitioning for the cloud variable achieved optimal time as it was tuned for the single query type accessing it.

4.3 Effects of the file overhead

The experiments with the Ampex robotic system were performed after it was connected recently to NSL-UniTree. As was mentioned above, the file overhead was found to be quite large, about 11 seconds. Consequently, the size of each cluster was relatively large (about 200 MB), and the total number of clusters for this dataset was only 245 for a one year dataset. For the lower file overhead (about 3.3 seconds), which was obtained later, the size of each cluster was about 100 MB and the number of clusters about doubled. As discussed below, the lower file overhead improved the solution results significantly. In general, when the file overhead is small, and the number of clusters is larger, the response times tend to be shorter, because less unnecessary data is read for a given requested subset of the dataset.

To understand the effect of the file overhead better, we performed a simulation for the same set of query types, assuming the lower file overhead of 3.25 seconds and no file overhead at all. The effect of no file overhead can be achieved if the tape system permits partial file reads; that is, the system can seek to a position on the tape and read precisely the number of bytes requested. Thus, we can take the set of files (clusters) assigned to a tape and store them consecutively as a single physical file. This is indeed a feature that the Ampex system is capable of, and it will be exploited in the NSL-HPSS implementation (mentioned in Section 3.3).

The simulation results (in minutes) for the Ampex system are shown in Table 5, along with the original and measured response time that were shown in Table 4 for comparison purposes.

Query Type	Original	High FO	Low FO	No FO	High FO ratio	Low FO ratio	No FO ratio
1	9.80	2.73	1.58	0.75	3.59	6.20	13.07
2	9.80	2.73	2.68	2.53	3.59	3.66	3.87
3	1.60	2.73	1.58	0.75	0.59	1.01	13.07
4	9.80	2.73	2.68	2.53	3.59	3.66	3.87
5	117.00	29.07	25.70	24.60	4.02	4.55	4.76
6	9.80	10.67	8.88	8.19	0.92	1.10	1.20
7	29.30	1.90	0.86	0.75	15.42	34.07	39.07
8	1.60	1.90	0.86	0.69	0.84	1.86	2.32
9	9.80	0.72	0.72	0.72	13.61	13.61	13.61

Table 5: Effects of various file overheads on response time for the Ampex D2

As can be seen, the new ratios improved for all query types, some by a factor of 4, and the response times for all query types are better than the original times in both the case of the low file overhead (low FO) and the case of no file overhead (no FO). These simulation results show that systems that permit partial file reads perform better than systems that do not support that. But, even if partial file reads are not available, the gains that can be obtained by the partitioning algorithms are still very significant, especially in cases that the file overhead is low.

Our experiments confirmed, as expected that the lower the file overhead the better the results. However, we have learned from the experiments and simulations that even with a large file overhead the overall improvement of the partitioning algorithm is very significant. As was shown in Table 4, 6 of the 9 queries improved by a factor of 3.5 to 15, at the cost of 3 queries degrading by a small factor of less than 2. We also note that the lower file overhead of 3.25 seconds is still large compared to the Exabyte system. The file overhead is a function of the software that controls the tape system, and that it should be reduced to the extent possible to accommodate scientific applications. Of course, the best devices for this purpose are those that support partial file reads.

5. The Dataset Partitioning Workbench

Naturally, the scientist who will be using the dataset (or a database designer acting on the behalf of multiple scientists) is in the best position to know what are the typical queries that will be used and the frequency of their use. The dataset partitioner should optimize the reorganization of the dataset according to this information. Initially, we thought that letting the scientist specify a weight for each query will be a reasonable way of representing the relative importance and/or the frequency of use of each query. However, we found out that assigning weights is not a meaningful task for the scientists, and was confusing in practice.

Depending on the weights assigned to queries, different partitioning solutions can be found. The choice of a partitioning solution is very important since the process of partitioning and reorganizing a large dataset is a costly one. Thus, it was important from a practical point of view that we develop methods for facilitating the process for selecting a solution based on intuitive measures rather than query weights. The result is an interactive "partitioning workbench". The goal of the workbench is to help the scientists see the trade-offs between possible solutions based on actual times that each query will take under a given solution.

In presenting the solutions to the scientist, we assume that all queries that belong to the same query type will have the same response time. In reality the response times to queries that belong to the same query type may vary somewhat depending on where the data relevant to the query is located on tape. This approximation is reasonable since each query that belongs to the same query type needs to access the same amount of data.

We observed that in order to make the estimates on actual times meaningful it is necessary to present them relative to the best possible time for each query. The best possible time (optimal time), is calculated assuming that all the information to answer the query is in a single file, and that the file is positioned at the beginning of a tape. Thus, the optimal time is equal to the time to mount a single tape, plus the time to read the first file. For example, if the best possible time for a query is 30 minutes, and the solution chosen results in 33 minutes, there is little to gain by trying to further optimize for that query.

We produce the multiple solutions presented to the designer as follows. We start by assigning all query types the same weight, and generate multiple possible solutions based on the permutation of the dimensions. Each solution consists of the estimates on actual response times for all query types, as well as the optimal response times. The solutions are ordered according to the overall response time relative to best possible times. The best solution is presented to the user as "option 1" as shown in the left part of Figure 5 (which is actually in color, but shown in black and white here). It shows an actual partitioning of a dataset for the Ampex robotic tape system. Six query types are considered for this group of query types (group 2) where the narrow bars shows the optimal times, and the thick bars the actual times for each of the six query types. As can be seen this option achieves optimal times for the first four query types, at the expense of the last two. Note that the user can control the scale of the display for better viewing of details.

At this point, the user can select any query type to be viewed on the right part of the screen for other possible options. For example, the scientist may want to optimize query type 5, and see what the effect will be on the other query types. The scientist selects query type 5 (by clicking on it) and all possible solution options are then displayed, as shown on the right part of the screen of Figure 5. As can be seen, there are three options that achieve optimal time for query type 5: options 2, 4, and 5. The scientist can now select any of these options (by clicking on the desired option) to see the effect on other query types. Suppose that option 2 was selected. The result is shown in Figure 6. As can be seen in the left part of the figure, the effect of selecting option 2 is that query type 5 achieves optimal time at the expense of slowing down query types 1, 2, and 4. This may be a preferred solution if indeed query type 5 is particularly important to optimize for.

The scientist can continue to explore other options and settle on the most desirable option for his/her needs. This form of interaction is much more meaningful to scientists than assigning weights. Seeing the trade-offs in terms of actual times makes the choice of a solution a more straightforward process. In some cases there may be a serious conflict between query types, in that selecting a solution that favors one query type may disfavor another, and vice versa. When such conflicts arise the scientist needs to make a difficult choice or resort to some duplication of data. We have not addressed so far the possibility of data duplication. It is the subject to future research.

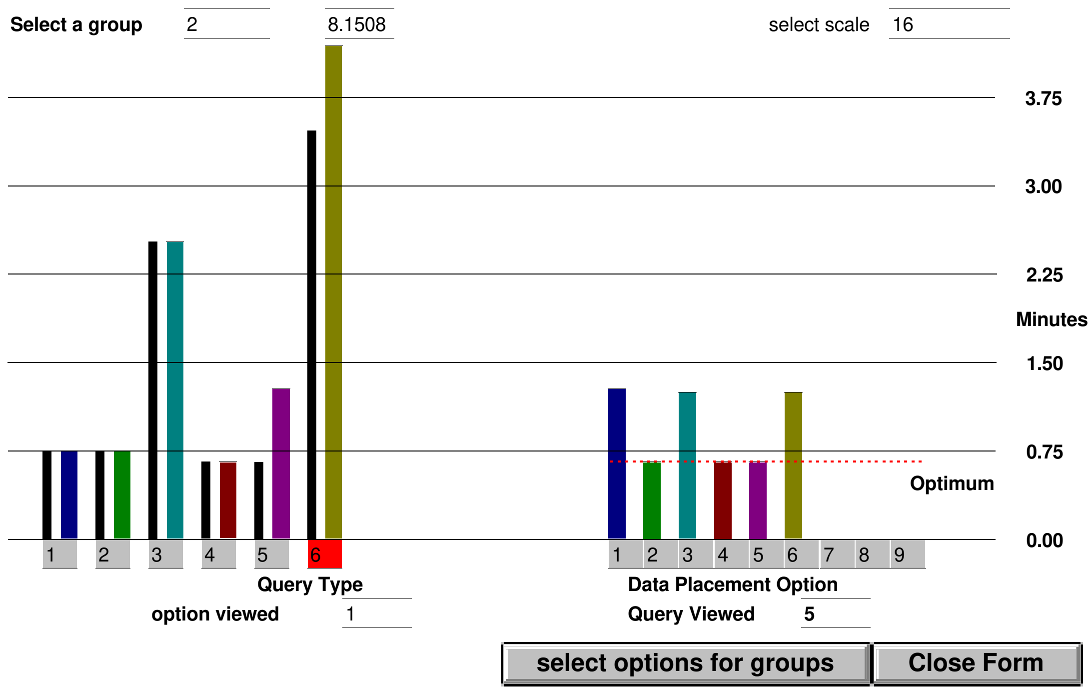

Figure 5: Display of a solution option for a group of query types

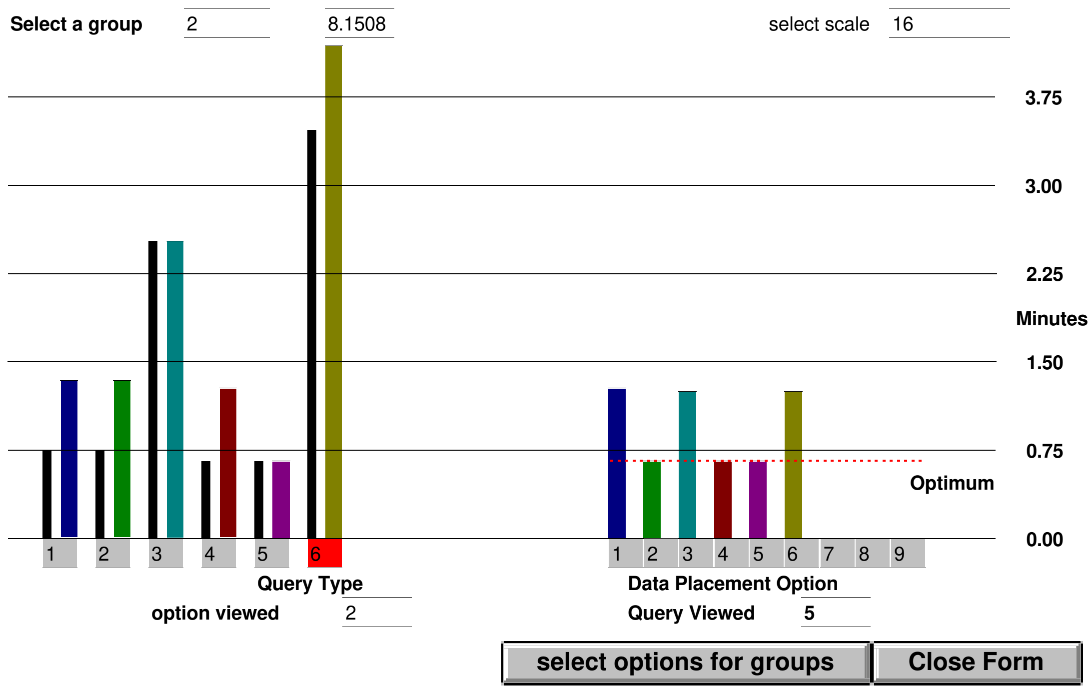

Figure 6: Display of a solution that optimizes query type 5

Once a desired solution is selected, the restructuring of the dataset into multiple clusters and their layout on tape volumes is generated by the workbench. This information is used by the write process to partition the dataset accordingly and to store the clusters on tapes.

The front end of the workbench was implemented using a relational database system (ACCESS) on a PC Windows platform that presents the user with a GUI. It is used to store up to 25 tables that contain the dataset information, the hardware characteristics, query types, and the various partitioning solutions calculated. The backend of the workbench is a collection of C++ programs that calculate the partitioning solutions, and the estimated response times. In addition to screens for selecting a solution discussed above, there are various screens developed for the scientists to enter or modify the information on datasets, hardware characteristics and query types.

6. Concluding remarks

Scientific application that access very large datasets face a major bottleneck when they need to access subsets of very large datasets from tertiary storage. This state of affairs has been the main reason that scientific analysis is not currently performed on an ad-hoc basis. Analysis cannot be spontaneous if a request for an interesting subset takes hours. Further, if the mass storage system is sharable by many users, the access of unnecessary data when subsets are requested, reduces the efficiency of such systems. As a result, supporting a certain user load requires additional physical devices such as read channels or larger robotic systems.

The approach we have taken is common in data management systems. To achieve higher efficiency of data access from disks, data is often clustered according to its expected use. We have chosen to work closely with a specific application area (climate modeling) where this problem is affecting the productivity and quality of the analysis. This gave us a realistic framework to understand the nature of spatio-temporal datasets and typical access to such datasets in modeling applications.

The results so far confirm the benefits of this approach. In realistic examples, it was possible to pinpoint typical access patterns and restructure the datasets to fit the intended use of the data. We found it useful to provide users with estimates of response times for making partitioning choices. Once a partitioning choice is made, users can estimate the response time to ad-hoc queries, and decide whether they want to wait for a response. Because many of the requests are for on-line visualization, the size of the subsets requested are small, and thus only a few clusters need to be accessed. In such cases, response time improved by a factor of 10-100.

There are many directions that one can take at this point. One is to consider the benefits of duplicating some of the data. In some applications, a small percentage of duplication can dramatically improve global response time when query types have inherent data partitioning conflicts. Another area is to consider more general access patterns and query types.

There is also the question of how generic such algorithms can be. We think that it will be necessary to specialize on application domains. However, selecting broad categories of data types, such as spatio-temporal data, or sequence data (e.g., time series), can make such techniques generally useful. The reason that we chose to concentrate on the spatio-temporal domain is that many disciplines are in this domain (geology, earth science, environmental sciences, etc.) and that spatio-temporal datasets tend to be very large (simulation data, satellite data, etc.).

Finally, it is worth mentioning that tape striping techniques are being investigated to mitigate the slow response time of accessing data from tape systems (see, for example [10]). In this approach no knowledge of access patterns is used; rather it is intended to take advantage of multiple tapes that are synchronized to be read in parallel. Striped tape systems will complement our partitioning techniques, in that clusters could now be spread over multiple tapes for parallel reads. The main gains provided by partitioning will continue to be important when we use such systems because they reduce the data that needs to be read for a given request.

Acknowledgment

This work was performed by the Lawrence Livermore National Laboratory under contract W-7405-Eng-48 and by Lawrence Berkeley Laboratory under contract DE-AC-76SF00098, and was supported by the Office of Scientific Computing of the Office of Energy Research, U.S. Department of Energy.

References

[1] Gates, W., Potter, G., Phillips, T., Cess, R., An Overview of Ongoing Studies in Climate Model Diagnosis and Intercomparison, Energy Sciences Supercomputing 1990, UCRL-53916, pages 14-18.

[2] EOS Reference Handbook, NASA document NP-202, March 1993.

[3] Coleman, S., Miller, S., Eds., Mass Storage System Reference Model, Version 4, IEEE Technical Committee on Mass Storage Systems and Technology, May 1990.

[4] L.T. Chen, et al, "Efficient Organization and Access of Multi-Dimensional Datasets on Tertiary Storage Systems", To appear in a special issue on Scientific Databases, Information Systems Journal, Pergammon Press, Spring 1995.

[5] R. Coyne, and R. Watson, The National Storage Laboratory: Overview and Status, *Proc. Thirteenth IEEE Symposium on Mass Storage Systems*, Annecy, France, June 13-16, 1994.

[6] S. Coleman, R. Watson, and R. Coyne, The Emerging Storage Management Paradigm, *Proc. Twelfth IEEE Symposium on Mass Storage Systems*, Monterey, CA, April 26-29, 1993.

[7] D. Teaff, R. Coyne, and R. Watson, The Architecture of the High Performance Storage System, *Fourth NASA GSFC Conference on Mass Storage Systems and Technologies*, College Park, MD, March 28-30, 1995.

[8] S. Louis, and D. Teaff, Class of Service in the High Performance Storage System, *Second International Conference on Open Distributed Processing*, Brisbane, Australia, February 21-24, 1995.

[9] R. Drach, and R. Mobley, *DRS User's Guide,* UCRL-MA-110369, LLNL, March, 1994.

[10] Drapeau, A., Katz, R., Striped Tape Arrays, Twelfth IEEE Symposium on Mass Storage Systems, 1993, pages 257- 265.