EFFICIENT ORGANIZATION AND ACCESS OF MULTI-DIMENSIONAL DATASETS ON TERTIARY STORAGE SYSTEMS

L.T. Chen[1], R. Drach[2], M. Keating[2], S. Louis[2], D. Rotem[1], A. Shoshani[1]

[1]Lawrence Berkeley Laboratory, Berkeley CA 94720

[2]Lawrence Livermore Laboratory, Livermore CA 94550

(Published in Information Systems Journal, April, 1995)

Abstract — This paper addresses the problem of urgently needed data management techniques for efficiently retrieving requested subsets of large datasets from mass storage devices. This problem is especially critical for scientific investigators who need ready access to the large volume of data generated by large-scale supercomputer simulations and physical experiments as well as the automated collection of observations by monitoring devices and satellites. This problem also negates 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 than the time to transmit that subset over a fast network. This paper focuses on very large spatial and temporal datasets generated by simulation of climate models, but the techniques described here are applicable to any large multidimensional grid data. The main requirement is to efficiently access relevant information contained within much larger datasets for analysis and interactive visualization. Although these problems are now becoming more widely recognized, the problem persists because the access speed of robotic storage devices continues to be the bottleneck. To address this problem, we have developed algorithms for partitioning the original datasets into "clusters" based on analysis of data access patterns and storage device characteristics. Further, we have designed enhancements to current storage server protocols to permit control over physical placement of data on storage devices. We describe in this paper the approach we have taken, the partitioning algorithms, and simulation and experimental results that show 1 to 2 orders of magnitude in access improvements for predicted query types. We further describe the design and implementation of improvements to a specific storage management system, UniTree, which are necessary to support the enhanced protocols. In addition, we describe the development of a partitioning workbench to help scientists select the preferred solutions.

1. INTRODUCTION

Scientists working with spatio-temporal data do not naturally think of their data in terms of files or collections of files, but rather in terms of basic abstractions such as spatial and temporal variables, multidimensional arrays, and images. This work is directed toward providing support for such abstractions within the context of current hierarchical mass storage systems. One of the most critical issues for scientific investigators is the increased volume of data generated by large-scale supercomputer simulations and physical experiments. In addition, the automated collection of observations by monitoring devices and satellites produce vast data at increasingly faster rates. These large datasets have, in some cases, led to unreasonably long delays in data analysis. In these situations, the speed of supercomputers is no longer an issue; instead, it is the ability to quickly select subsets of interest from the large datasets that has become the major bottleneck.

To address this need, we have developed algorithms for partitioning datasets into "clusters" based on anticipated data access patterns and storage device characteristics, as well as enhancements to current storage server protocols to permit control over physical placement of data on storage devices. The access patterns considered in this paper are range specifications in the multidimensional space. The techniques developed can be applied to any multidimensional datasets, although our emphasis and example applications is on spatio-temporal datasets.

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.

The main requirement is to efficiently access relevant information contained within much larger datasets for analysis and interactive visualization. Although the initial focus of this work is on spatial and temporal data, our results can be applied to other kinds of multidimensional grid datasets.

The developmental and operational site for our work is the National Storage Laboratory, an industry-led collaborative project [2] housed in the National Energy Research Supercomputer Center (NERSC) at LLNL. The system integrator for the National Storage Laboratory, IBM Federal Sector Division in Houston, has projects already in place that are investigating improved access interface and data reorganization techniques for atmospheric modelers at NCAR [3]. Many aspects of our work complement the goals of the National Storage Laboratory.

2. BACKGROUND

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 until it is required to retrieve them for analysis or visualization as shown in Figure 1. 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 (current IBM 3480 tape cartridge technology, used in the storage systems at LLNL, allows 200-250 megabytes per cartridge). 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.

This inefficiency generally requires that the entire set of original data be retrieved and downloaded to a disk cache for the researcher to analyze or interactively visualize a subset of the data. 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.

A similar situation exists with many scientific application areas. For example, the Earth Observing System (EOS) currently being developed by NASA [3,4], 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.

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 [6] 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

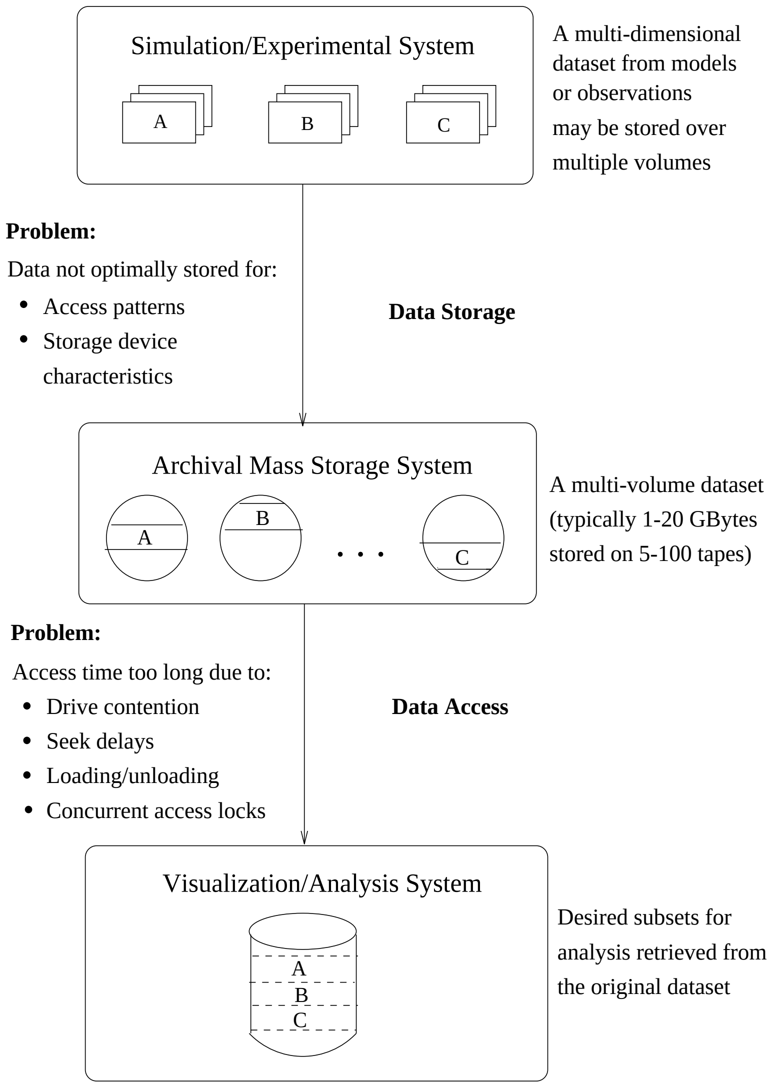
Simulation/Experimental System
A
B
C
A multi-dimensional dataset from models or observations may be stored over multiple volumes
Problem:
Data not optimally stored for:
• Access patterns
• Storage device characteristics
Data Storage
Archival Mass Storage System
A
B
. . .
C
A multi-volume dataset (typically 1-20 GBytes stored on 5-100 tapes)
Problem:
Access time too long due to:
• Drive contention
• Seek delays
• Loading/unloading
• Concurrent access locks
Data Access
Visualization/Analysis System
A
B
C
Desired subsets for analysis retrieved from the original dataset

Fig. 1: Current Situation

formats and often prefer to interface to files directly rather than use a data management system.

3. APPROACH

As mentioned above, the main problem we address is the slow access of small subsets from a large dataset in archival storage needed for visualization and analysis. As can be seen in Figure 1, this problem has a storage organization component and an access component. 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. We show schematically in Figure 1 that the desired subset (which consists of pieces A, B, C which belong to a single dataset) is scattered over multiple volumes of the Mass Storage System.

We address the above problem by developing algorithms and software that facilitate the partitioning of a large dataset into multiple "clusters" that reflect their expected access. For example, if many desired subsets consist of certain variables over a period of a year, then reorganizing and partitioning the data such that the corresponding variables are stored as "yearly clusters" in contiguous storage locations will facilitate efficiently reading the desired data. 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.

The term "partitioning algorithm" is used to indicate that as a result of the algorithm a dataset will be partitioned (or restructured) into many such clusters. The term "cluster" is used to convey the idea that all the data that satisfy a query should ideally reside in a single cluster. The goal is to minimize the amount of storage that has to be read when a subset of the data is needed.

The way that the above techniques interact with the existing software is shown schematically in Figure 2. The same basic system components shown in Figure 1 also exist in Figure 2, along with additional components. The component labeled "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 (as mentioned above, a typical climate modeling dataset is stored in multiple files, each containing 5 days worth of data). The component labeled "Data Assembly and Access Management" is responsible for accessing the clusters, and for assembling the desired subset from clusters (rather than reading the dataset). 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. Note that the schematic illustration in the Archival Mass Storage System is intended to show that the desired cluster "ABC" may be stored in contiguous storage space for efficiency as a result of the allocation analysis. The details of the two components are shown in Figures 3 and 4.

On the left of Figure 3, 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 and produces an Allocation Directory that describes how the multidimensional datasets should be partitioned and stored.

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. 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 7.

In Figure 4, we show the details of reading subsets from the mass storage system. 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. 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

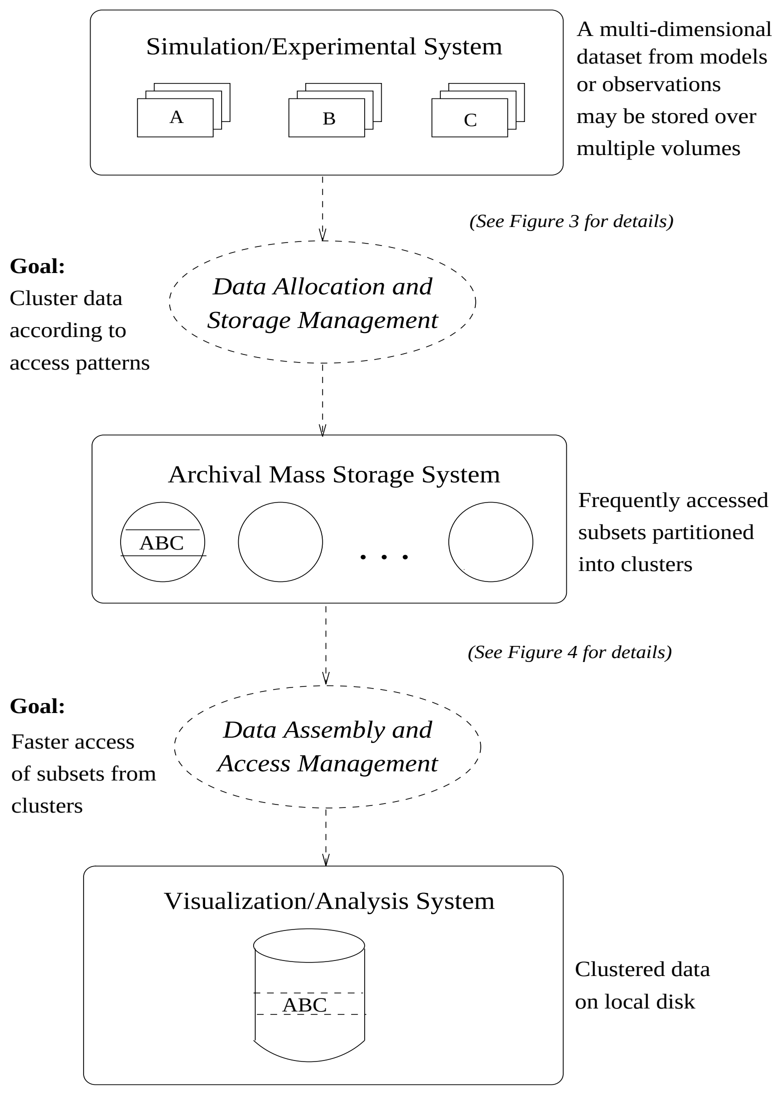
Simulation/Experimental System
A
B
C
A multi-dimensional dataset from models or observations may be stored over multiple volumes
(See Figure 3 for details)
Goal: Cluster data according to access patterns
Data Allocation and Storage Management
Archival Mass Storage System
ABC
. . .
Frequently accessed subsets partitioned into clusters
(See Figure 4 for details)
Goal: Faster access of subsets from clusters
Data Assembly and Access Management
Visualization/Analysis System
ABC
Clustered data on local disk

Fig. 2: Areas of Improvements

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.

Next, we discuss some details of the partitioning and subset assembly processes, as well as the management of the allocation directory and associated metadata.

3.1. The Partitioning Process

The characterization of access patterns of the analysis and visualization programs is essential for the organization of data to achieve high access efficiency. Of course, there may be conflicting access patterns. Thus, an analysis of the access patterns is needed to determine the optimal partitioning and allocation of clusters on archival storage. The partitioning algorithms use a model of the access patterns as well as a model of the physical device characteristics. The specific techniques used for determining the optimal allocation are given in Section 4.

In an environment of typical mass storage systems we find a multi-level hierarchy consisting of memory, magnetic disks, and robotic devices for tapes and optical disks. Each level in the hierarchy may serve as a cache for the next level. As a by-product of our partitioning algorithms,

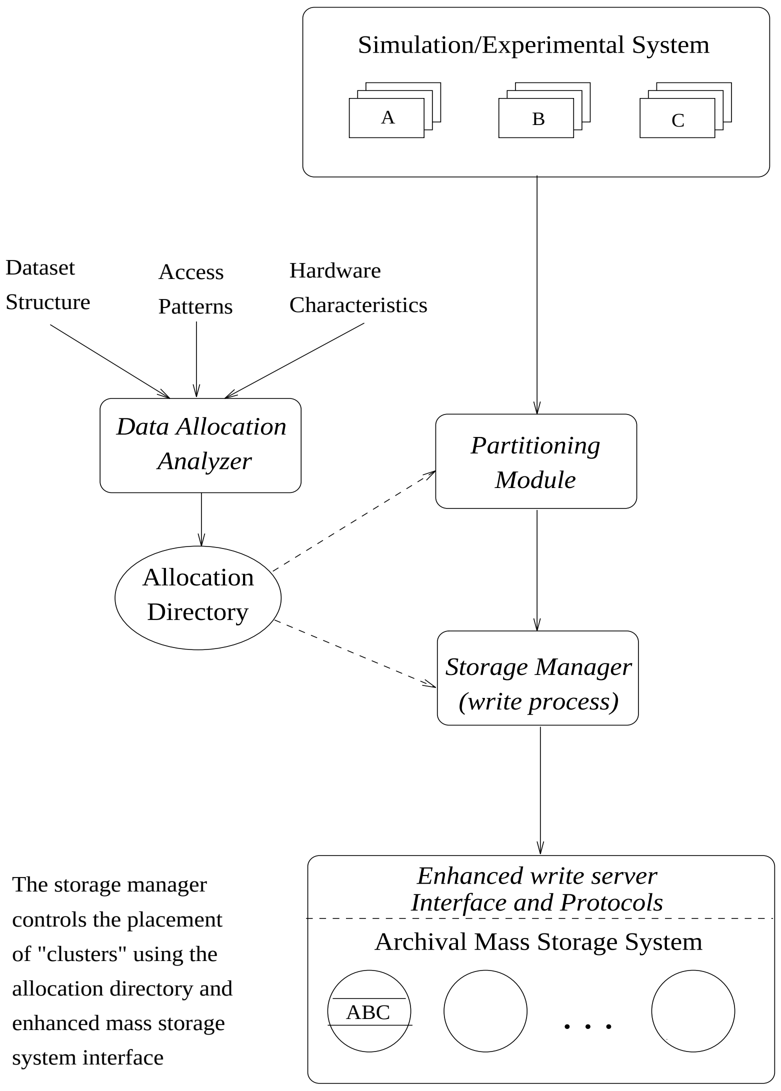
Simulation/Experimental System
A
B
C
Dataset Structure
Access Patterns
Hardware Characteristics
Data Allocation Analyzer
Partitioning Module
Allocation Directory
Storage Manager (write process)
Enhanced write server Interface and Protocols
Archival Mass Storage System
ABC
. . .
The storage manager controls the placement of "clusters" using the allocation directory and enhanced mass storage system interface

Fig. 3: Dataset Reorganization and Partitioning Process

the volume of data cached will become smaller as we tend to retrieve data only if it matches a query. In that way the cache will tend to only hold relevant data which is frequently accessed, thus improving the efficiency of the cache.

3.2. The Subset Assembly Process

Answering a user query generally involves picking parts from various clusters in the mass storage system and then generating the result in the format and order needed by the application. This is the function of the subset assembly module. Note, that we do not consider here the possibility of assembling a single subset from multiple datasets. Such "joining" of results is very much application dependent and is usually done at the application level after multiple subsets are retrieved. For this reason, we consider the study of various ways of joining the results from multiple datasets to be beyond the scope of this project.

The best situation one can hope for is that each query matches exactly a single cluster. But, typically, a query needs to read a small number of clusters, still benefiting from reading only a small portion of the original dataset. It is worth noting that the assembly algorithm has some similarity to the well-known problem of transposing matrices on secondary and tertiary storage

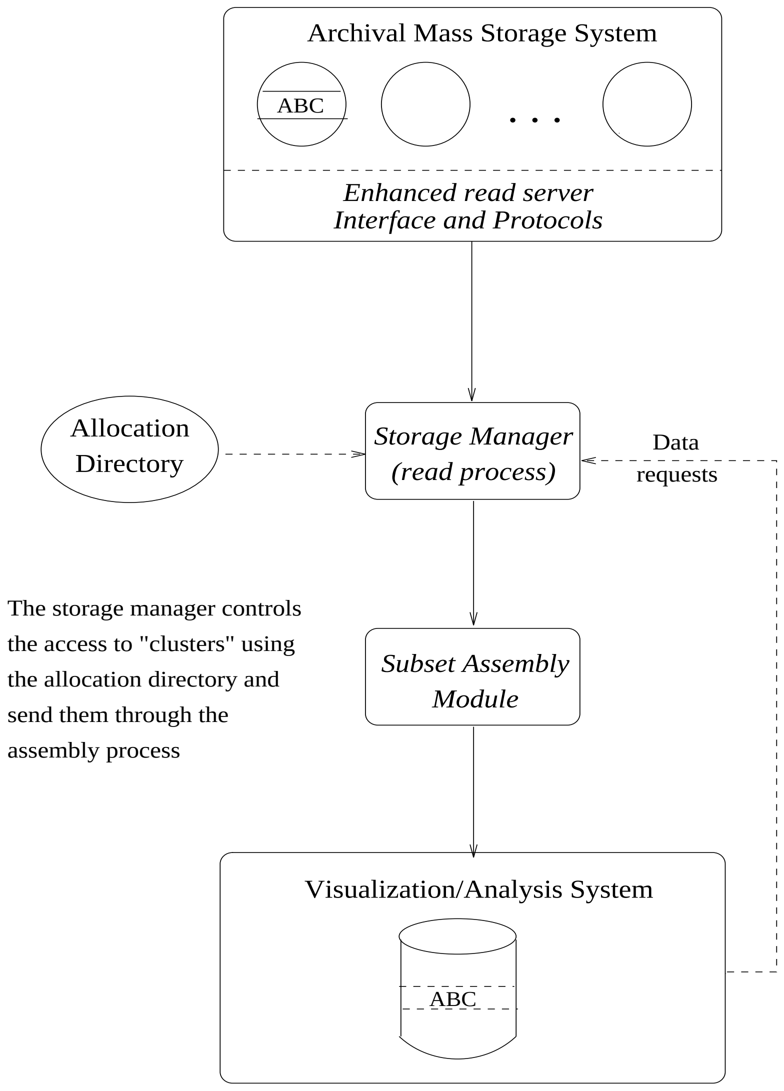
Archival Mass Storage System
ABC
. . .
Enhanced read server
Interface and Protocols
Allocation Directory
Storage Manager (read process)
Data requests
The storage manager controls the access to "clusters" using the allocation directory and send them through the assembly process
Subset Assembly Module
Visualization/Analysis System
ABC

Fig. 4: Subset Assembly Process

systems with limited buffer space [10].

As the assembly task is a crucial step in the process of forming an answer to a query, it is beneficial to use a powerful machine (preferably one with multiple processors) with a large buffer size. In general, given a set of clusters, the running time of the assembly algorithm (as is the case with transposition algorithms) is inversely related to the size of the buffer available for the task. One can also take advantage of the potential for parallelism in the assembly algorithm due to the well known speedups gained by parallel sorting algorithms.

Although it may be desirable to have a dedicated assembly machine as an extension of the mass storage system, we do not initially assume so. Rather, we assume that the assembly will be done on the scientist's workstation. Thus, the assembly software needs to take into account the size of available disk as well as the available buffer space on the assembly machine. It then tries to re-arrange the data according to specifications with minimum number of passes over the data. Since the techniques for the assembly software are well-known [10], we do not discuss these further here. Rather the focus of this paper is on the partitioning algorithms and the interface design to the mass storage system.

3.3. Management of the Allocation Directory and Associated Metadata

The allocation directory generated by the allocation analyzer needs to be managed as well. In addition to the allocation directory, there is relevant metadata information associated with the various datasets, such as: who produced the dataset, when was it generated, variable descriptors, what regions they cover, etc. The metadata as well as the allocation directory form a relatively small (but important) database that needs to be managed and maintained. Such a database can be managed by existing commercial relational database systems. However, the use of such systems requires expertise that may be a burden to application scientists. Scientists need to interact with this database on a regular basis, to insert new entries and to browse the metadata. For this reason, more user friendly, perhaps customized interface systems should be used. We have developed and implemented such a tool, called the "partitioning workbench", for specifying dataset characteristics, query types, and for guiding the user in selecting the most desirable cluster partitioning. Details are presented in Section 6.

4. THE DATASET PARTITIONING ALGORITHM

We describe in this section the method of partitioning datasets into clusters and the algorithms used. Below we give a brief characterization of the problem, the assumptions made, and our solution to the problem.

Our goal is to take a multidimensional array with multiple variables and layout the data on multiple one dimensional storage volumes, such that the retrieval response time is minimized for anticipated queries. Before we can proceed we need to characterize the dataset structure, the anticipated queries, and the hardware properties, and determine a measure for the quality of a solution.

The structure of datasets must satisfy the output of typical climate simulation models and observations. Each dataset 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.

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. In the next subsection, we elaborate on the details of specifying query types.

The next issue we describe is that of defining a measure of "goodness" for a given solution. We assume that the scientists working with the system specify a weight for each query type that they define. A high weight means that the associated query type is "important", in the sense that it should be executed as fast as possible. The algorithms described here make use of these weights. We do not make a distinction between weights specified by a single user or multiple users. We assume that a single person, referred to as the designer, collects all the query types from multiple users, and assigns weights to them. It is possible to develop interfaces that will permit joint design by multiple users, but this was not an important issue for this project.

We found that in practice, the specification of weights is not an intuitive task. Rather, illustrating the effects of the weights in terms of the estimates on actual response time to queries is much more meaningful to users. Consequently, we have developed an interactive "workbench" that lets the designer see the trade-off on the performance of the set of query types for the various partitioning solutions. In presenting the solutions to the designer, we order the solutions using a measure of goodness based on a formula that compares the estimates on actual response time to the best possible time (optimal time) for each query. The designer can then evaluate the trade-offs between the solutions. Once a choice is made, we derive relative weights that reflect this choice. We describe the workbench, the formula for the measure of goodness, and the assignment of weights in Section 6.

Given a weight, W_i, for a query type, Q_i, the weight, w_{ij}, of a single query, q_{ij}, that belongs to this query type is determined by dividing W_i by the number of queries that belong to Q_i. Thus, if the query type in the example above (which represented 480 queries) had a weight of 0.5, then the weight of each query that belongs to this query type is $\frac{0.5}{480}$. This assumes that all queries belonging to a query type are equally likely to be requested. The rationale for doing so is as follows. The interpretation that scientists we worked with gave to the weight of a query type is that this value is assigned to the entire class of queries rather than the individual queries. This interpretation prevents query types that have a large number of queries from dominating query types that have a few queries. As mentioned above, we had to find a more intuitive way of deriving weights for query types, but once they have been derived, the above ratio is used to assign weights to individual queries. The weights of individual queries are used in the algorithms described in Appendices B and C.

Next, we describe in more detail our assumptions on the query types and the hardware model.

4.1. Characterization of Query Types

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. Any: if one value along the dimension is requested for this query type. Note that the value itself is not specified, it is assumed that any one of the values within this dimension is equally likely to occur.
3. One(coordinate): if exactly one point (the coordinate element) of the dimension is requested.
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.

While we believe that the above class of query types is useful for many applications, other query types may be necessary. For example, it is likely that scientists working on earthquake faults will benefit from a query type that specifies the fault line, so that tiles along the fault line will be put into a single cluster.

It is worth mentioning that in a recently published paper [7], a similar approach of partitioning large multidimensional arrays was taken in the context of optical jukeboxes. Although addressing this problem for tertiary storage is a different problem, it is interesting to see what assumptions were made regarding access patterns. In that paper, the assumptions made about the query types are different than ours. Specifically, it was assumed that a query type is expressed by giving a range size for each dimension (such as range size on the X dimension is 100, etc.). The interpretation of specifying a range size is that all possible combinations on each range are equally likely (e.g. 1-101, 2-102, ... etc.). The variables are considered as a additional dimension of the data array. Thus, if the range size on this dimension was 3, for example, then any 3 variables are as likely as any other 3 variable to be accessed together. While these assumptions led to a closed analytical solution, we found them unsuitable for our application domain. For example, certain groupings of variables are much more likely than others in climate modeling applications.

4.2. Characterization of Tertiary Storage Devices

The optimal partitioning depends also on the characteristics of the tertiary storage devices. Because we do not want to limit this work to a particular tertiary storage device, 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. $T_s(x)$ (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: $T_s(x) = 10 + (x/20)$. In cases where it is difficult to determine the constant value, the seek function is simply x divided by the seek speed.
4. T_m (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 $FO = (T1 - T2) \times R$, 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.

4.3. The Partitioning Algorithm

Having described the hardware and query type model, we can now describe our approach to solving the partitioning problem. It is easy to show (see Appendix A) that the problem, in general, is NP-Complete. Thus, the best we can hope for is to find a near optimal solution within

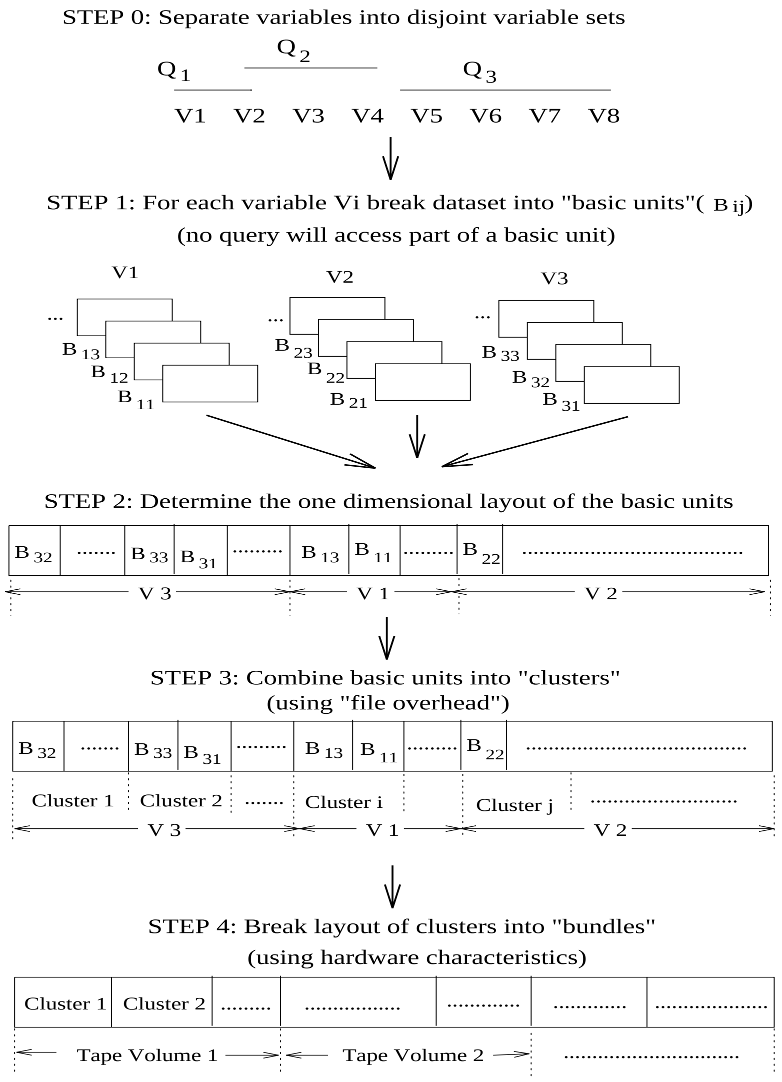
STEP 0: Separate variables into disjoint variable sets
Q1
Q2
Q3
V1 V2 V3 V4 V5 V6 V7 V8
STEP 1: For each variable Vi break dataset into "basic units"(B ij)
(no query will access part of a basic unit)
V1
V2
V3
B 13
B 12
B 11
B 23
B 22
B 21
B 33
B 32
B 31
STEP 2: Determine the one dimensional layout of the basic units
V 3
V 1
V 2
STEP 3: Combine basic units into "clusters"
(using "file overhead")
Cluster 1
Cluster 2
Cluster i
Cluster j
STEP 4: Break layout of clusters into "bundles"
(using hardware characteristics)
Tape Volume 1
Tape Volume 2

Fig. 5: Dataset Partitioning Steps

a reasonable amount of time. We accomplish this by solving the problem in steps, as shown in Figure 5. We outline these steps below, and discuss details of each step in subsequent subsections.

First, we split up the dataset into disjoint groups of variables, such that all variables accessed together by any query type are always in the same group. This preprocessing step is labeled STEP 0 in Figure 5. It illustrates that variables V1, V2, V3, V4 are in the same group because of the overlap between query types Q_1 and Q_2, while V5, V6, V7, V8 are in the same group because they are all accessed by a single query type Q_3. Note that for each group of variables there is a corresponding group of query types, and that all variables in a query type are contained in a single group of variables. Thus, we use the term "group" below as containing a set of query types and the corresponding set of variables. This grouping of query types allows us to split the original problem into several smaller problems, since most query types only involve one or two variables. We can now solve the partitioning problem group by group, since each group has its own unique and disjoint set of variables and query types.

Within each group, we solve the problem in 4 steps. STEPS 1-4 in Figure 5 illustrate this for 3 variables V1, V2, and V3. First, we individually take each variable in the group and sub-partition it into basic units. A basic unit is the largest chunk of data such that, for every possible query of any query type in the group, the query either needs all the data in the basic unit or it needs none of the data in the basic unit. Splitting each variable into basic units allows us to simplify the remaining stages since we now only need to consider how to lay out the basic units, instead of having to lay out the individual bytes. Note that each variable in a group may have a different set of basic units depending on the queries that overlap it.

After breaking all variables into basic units, the second step determines a one dimensional layout of all the basic units, such that the weighted sum of the distance between the first and last basic unit needed by each query of all query types is minimized. The reasoning behind this step, is that tapes are fundamentally one dimensional. By trying to minimize the distance between the first and last basic unit needed by each query, we are designing a layout in which tape seeking and mounting time is minimized when data for the query needs to be retrieved. By using the weight of queries in the weighted sum we favor the queries with the higher weights. This step is the only one of the four steps that is NP-Complete (also see Appendix A). Consequently, we rely on some assumptions and use heuristics to solve this step, which are discussed in Section 4.3.2.

The third step determines the optimal way to combine multiple neighboring basic units into "clusters" such that the one dimensional array of basic units becomes a one dimensional array of clusters. This step is needed because the interface to the storage system we currently use, UniTree, assumes that the unit of retrieval is a file. Therefore, a file can either be completely retrieved for a query, or not retrieved at all, but it cannot be partially retrieved. Because of this, we must "cluster" basic units together into files. Thus, the term cluster is used to represent the set of basic units that is stored in a physical file. The terms "cluster" and "file" will be used interchangeably throughout this paper. Note that in this step we need to use only the "file overhead" parameter. No other hardware characteristics are needed.

We make the assumption that each cluster resides entirely on a single tape. Usually, the size of clusters are small relative to typical tape capacities, since the clusters are made as small as possible in order to minimize the amount of data accessed. However, if a cluster exceeds the tape capacity, we assume that the storage system will handle that as it handles any large file that exceeds the tape capacity.

Finally, the fourth step further combines neighboring clusters into "bundles" that will each fit into a tape volume, and thus determines which clusters should be put on the same tape volume. This last step uses the hardware characteristics, and therefore it is best if it is supported by the mass storage management system. For this project, this "bundling" step was made part of the UniTree system, as discussed in some detail in Section 7.

Each one of these four steps will be discussed in more detail below.

4.3.1. Breaking Data Into Basic Units

We form basic units by individually examining the way each variable is accessed by all the queries belonging to a query type within the group. We first identify which dimensions have "All" specified by all query types that access the variable. We then take the multidimensional subset formed by combining data of these dimensions together, and call this a basic unit. By doing this we are guaranteed that no query will ever ask for a fraction of a basic unit. The rationale for having this step is to minimize the number of "units" that the remaining three steps will need to deal with.

4.3.2. Determining the One Dimensional Layout of Basic Units

The goal of this step is to layout all the basic units in one dimension such that all the units that will be accessed by a query will be as close together as possible. In general, because of conflicting query demands, the basic units for satisfying a query, might be interleaved with basic units of other queries. If one considers all the basic units in between the first and last unit requested by a query, and assume that they all need to be accessed in order to answer the query, then the goal of this step is to find a one dimensional ordering of the basic units such that the weighted sum of the length (in bytes) of data that needs to be retrieved for each query is minimized. Since, as mentioned above, this general problem is NP-Complete, we use a heuristic to solve it suboptimally by considering two independent subproblems:

1. determining a linearization order for the dimensions of each variable
2. determining an ordering of the variables within a group

To solve the first problem, we utilize the fact that most queries either want all the data along a dimension or need only one element of the dimension, and only consider layouts of variables such that each dimension is completely laid out before we layout the next dimension. This is similar to the way a multidimensional array is linearized into a single dimension. Obviously the dimension with the fastest changing index (i.e. the last to be linearized) should be a dimension in which most query types ask for "all", and the dimension with the slowest changing index should be the one in which most query types ask for "any" or "one". We determine the best ordering of dimensions by considering all possible permutations for each variable. This is possible for a small number of dimensions. In our case, each variable has at most seven dimensions and thus there are at most 7! = 5040 possibilities.

Our solution to the second problem, namely, that of ordering the variables within a group, is influenced by a few factors. We need to consider the subset of the variables that are requested together by the same query type. Intuitively, we would like variables which are requested together to be close to each other in the ordering. However conflicts between queries may exist. For example, if we have 3 query types in a group, Q1, Q2, Q3, each accessing a pair of variables [V1,V2], [V2,V3], and [V1,V3], respectively, then any order of the variables will cause one of the query types to have its two variables separated by the third. One factor that can help resolve such conflicts is query type weights. A second factor we need to consider is the volume of data used by each variable, as this volume may affect the tape distances traveled for answering those queries which request several variables together. Fortunately, our analysis of typical access patterns reveals that very few variables (typically less than three) appear in the same group and we can therefore exhaustively evaluate all possible orderings of the variables and use the one which minimizes the weighted distance traveled by all queries in the group. One such ordering is shown in STEP 2 of Figure 5.

4.3.3. Combining Basic Units Into Clusters

At this stage, the basic structure of the data has already been determined. Next we need to determine which basic units should be combined into clusters and how the clusters should be laid out within tape volumes. In this step the basic units are grouped together to form clusters. This step is only needed for a storage system that does not support partial file reads. In such systems, an entire file needs to be read even though only a portion of it is needed. Because of this, there is the tendency to let each basic unit simply be a file, since this guarantees that we will never read any data that is not actually needed by the query. But this would result in an enormous amount of files, and the file overhead associated with each extra file would dominate the response time of queries that retrieve lots of data. On the other hand, we could simply have all the data of a variable be stored in one file. This would minimize the file overhead involved in big queries, but queries that only ask for a small portion of a file will have to retrieve the entire file. In order to strike an optimal balance between these two tradeoffs, we have devised an $O(N^2)$ time algorithm that utilizes dynamic programming to optimally solve this problem. Details of this algorithm are described in Appendix B.

4.3.4. Breaking The 1-D Array of Clusters into Tape Volumes (Bundling Clusters)

Similar to combining neighboring basic units into clusters, we must also combine neighboring clusters into bundles that fit in a tape volume, since the one dimensional layout of clusters will not necessarily fit on one tape. In order to do this, we have also designed an $O(N^2)$ time dynamic programming algorithm that finds the optimal solution for this problem. The algorithm considers both the seek time and the mounting time required for each query, based on all the different possible ways of breaking the linear array of clusters into tape volumes. When the function describing the seek time (based on seeking distance) for the tape system is a linear function, the algorithm can be further simplified to take only $O(N)$ time. The basic $O(N^2)$ time dynamic programming algorithm is given in Appendix C, and various variations on it appear in [8].

	Capacity (MB)	Transfer Rate (MB/s)	Seek Speed (MB/s)	Mounting (Seconds)	File Overhead (MB)
Exabyte	4500	0.265	31.25	315	0.064
Ampex	25000	12.864	503.32	39	141.506

Table 1: Measurements of Hardware Characteristics

4.3.5. Reasons for Minimizing Hardware Dependency in the Algorithm's Steps

The last step of determining bundles is the only step that requires the knowledge of hardware parameters (besides file overhead). Also, the file overhead is more a parameter of UniTree than a parameter of the hardware. Therefore, the first three steps of the partitioning process are completely hardware independent. Consequently, we can incorporate the dynamic programming algorithm of this last step into the storage system and define the interface to it at a level in which we simply supply an ordered one dimensional array of clusters to it. This approach was taken with the NSL-UniTree implementation, as discussed in Section 7. It permits the migration of data from one tape system to another to be completely transparent to the user, since UniTree would take care of the detail of trying to maintain this one dimensional order of clusters as best as it can. Even when UniTree must break this one dimensional array of clusters into multiple tape volumes, it selects the break in between clusters such as to minimize the query response time due to multiple tape mounts.

This notion, of postponing hardware dependency as late into the algorithm as possible, is the reason that we first determined a one dimensional layout of basic units for the entire dataset. The alternative is to first determine how to break the dataset into multiple tape volumes, and then try to determine the optimal layout within each tape. While this alternative method may sometimes generate better optimized results than ours, it makes the crucial assumption that we have full control of the storage hardware, which is impractical with most storage systems. This alternative approach was taken by [7], but applied to optical disks only.

5. SIMULATION AND EXPERIMENTAL RESULTS

5.1. 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 Table 1.

Note that the file overhead for the Ampex system is quite large (11 seconds, which is equivalent to 141 MB) due to lack of control over the behavior of the robotic system [†]. The effects of this fact will be discussed in the following sections.

5.2. Response Time Results

Based on these experimental measurements, we were able to make a comparison of the estimated response time of queries before and after we apply the previously described partitioning method to an actual PCMDI dataset. The response times are expressed in minutes and are shown for a few query types in Table 2, 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.

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 described in Section 4) and 62 query types. These query types

[†]Recent tuning of the UniTree system to match the characteristics of the Ampex reduced the file overhead from 11 seconds to 3.5 seconds

Exabyte Carousel Response Times					
	Query Types for Group 1	Optimal	Original	New	Ratio
1	U,V,W for any month at ground level	6.45	174.97	6.45	27.13
2	U,V,W for any month at all heights	92.85	174.97	92.85	1.88
3	U,V for any day at any height	1.72	30.55	4.08	7.48
4	U for any month at all heights for any Y	7.35	174.97	92.85	1.88
5	V for any year at all heights for range of Y	274.27	2142.60	1118.72	1.92
	Query Types for Group 2				
6	T for any month for all heights for all X,Y	32.01	174.97	40.83	4.28
7	T for range of 3 months at any height	6.45	535.65	6.45	83.05
8	T for any sample any height for range of Y	3.25	30.55	3.25	9.40
	Query Types for Group 3				
9	A cloud variable for any month for all X,Y	8.44	237.30	8.44	28.12

Ampex D2 Tape System Response Times					
	Query Types for Group 1	Optimal	Original	New	Ratio
1	U,V,W for any month at ground level	0.75	9.80	2.73	3.59
2	U,V,W for any month at all heights	2.52	9.80	2.73	3.59
3	U,V for any day at any height	0.65	1.60	2.73	0.59
4	U for any month at all heights for any Y	0.65	9.80	2.73	3.59
5	V for any year at all heights for range of Y	3.47	117.00	29.07	4.03
	Query Types for Group 2				
6	T for any month for all heights for all X,Y	1.27	9.80	10.67	0.92
7	T for range of 3 months at any height	0.75	29.30	1.90	15.42
8	T for any sample any height for range of Y	0.65	1.60	1.90	0.84
	Query Types for Group 3				
9	A cloud variable for any month for all X,Y	0.72	9.80	0.72	13.61

Table 2: Response Time Results for Two Different Tape Systems

were derived after extensive interviews with scientists interested in this dataset. The query types were partitioned into groups as explained in Section 4.3, and each group was analyzed separately. Table 2 only shows 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. For this particular set of query types only two groups have more than one query type. We chose to show these two groups in the tables because they represent conflicting requirements of the query types. We also show one representative group with a single query type, which obviously can be optimized.

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.

In order to interpret the results shown in Table 2, we discuss the results of the partitioning next. For this experiment we used equal weights for the query types, since the scientists involved wanted to see the effects of partitioning if all query types are equally likely. The algorithm determined that the optimal way to store the U,V,W group on the Exabyte Carousel is as follows: Each half a month of U,V,W data for a single height level was assigned to a file, and the 2 files that form a month need to be stored right next to each other. Next, similar files for different height levels, but of the same month, are stored next to each other, and finally, neighboring months are stored adjacent to each other. For the Exabyte, it turned out that 2 months of U,V,W data for all X,Y, and Z could be stored in single tape, with 6 tapes needed to hold the entire 1 year dataset.

Next, we give an explanation of the response time results. The 2 consecutive files that hold one month of data for all X,Y and one Z level, are all that needs to be retrieved for a query belonging to query type 1, and in doing so, no unnecessary data is retrieved. For queries that belong to query type 2, all 19 Z levels must be read to retrieve the one month worth of data. This represents reading half an Exabyte tape and requires 92.85 minutes. Although this may seem long at first, this is inevitable since a large amount of data is requested by each query. No unnecessary data was retrieved by queries of query type 2 under this partitioning scheme. The fact that half a month is stored in one file was determined by the dynamic programming algorithm of Step 3, and represents a compromise between query type 3 and all the other query types that access U,V, and W. This is because all the other query types access at least a month worth of data, whereas query type 3 only accesses data for one day. This compromise would result in roughly 15 days worth of data being read by query type 3 when only one day is needed. This does not turn out to be a serious problem though, since these queries retrieve so little data anyway. A good portion of the response time is due to the mounting time of the tape (100 seconds), and the fact that 15 times more data was retrieved only represented a two fold increase in response time. Query type 4 retrieves the same amount of data as query type 2 even though only one Y value is needed, and query type 5 retrieves the entire 6 tapes even though only a range of Y is needed. The response time of these two query types were sacrificed in order to speed up the first three query types.

As for the layout of the T variable on the Exabyte tape system, the algorithm determined that each file should contain exactly one month of all X,Y data for one Z level. In this way, all files containing the same Z level for different months will be next to each other, followed by the files for the next Z level, and so on. This emphasis on the "Month" dimension as a more important dimension to keep together than the Z dimension, is the result of query type 7 requesting 3 months at a time. This is in contrast to what the partitioning algorithm did for U,V,W, in which packing the Z dimension together was more important than packing the "Month" dimension together. The net result is that query type 7 has a very small response time under this new partition, which can be observed from the fact that its response time improved by a factor of 83 over the original.

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 effects of the file overhead are further discussed in the next subsection). 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, our algorithm can take advantage of partial file reads (no file overhead) to significantly improve performance.

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

Query Type	Original	with F/O	without F/O	Old Ratio	New Ratio
1	9.80	2.73	0.75	3.59	13.07
2	9.80	2.73	2.53	3.59	3.87
3	1.60	2.73	0.75	0.59	13.07
4	9.80	2.73	2.53	3.59	3.87
5	117.00	29.07	24.60	4.02	4.76
6	9.80	10.67	8.19	0.92	1.20
7	29.30	1.90	0.75	15.42	39.07
8	1.60	1.90	0.69	0.84	2.32

Table 3: Effects of File Overhead on Results

5.3. 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, see footnote in Section 5.1). Consequently, the size of each cluster was relatively large (the average size per cluster was about 200 MB). 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.

The reason for the high file overhead is unimportant for our experiments, and will be fixed in the HPSS version (see section 7). The important thing to notice is that even with a large file overhead the overall improvement of the partitioning algorithm is very significant. As was shown in Table 2, 5 of the 8 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.

To understand the effect of the file overhead better, we performed a simulation for the same set of query types, assuming no file overhead at all. This 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 bundle 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.

The simulation results are shown in Table 3, along with the original and measured response times that were shown in Table 2 for comparison purposes.

As can be seen, the new ratio 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. 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.

5.4. Effects of Partitioning on Unanticipated Query Types

Physical database design is a process that often involves a trade-off between competing needs and resources. In cases in which access patterns to the data are contradictory in terms of the physical organization best for each, a compromise needs to be struck. What if the compromise is not satisfactory? What will be the effect on unanticipated queries?

In general, given a particular physical organization, it is always possible to find a query that will perform badly. Thus, considering random queries is not a valuable exercise. However, there are several issues worth observing with respect to the above questions.

1. It is important to identify characteristics of access patterns that apply to nearly all queries of an application. In the application described here, queries typically involve one or two variables. It is rare that three or more variables are accessed in a single query. Taking advantage of this fact alone leads to great improvements to access efficiency. In the case of

climate model datasets, the datasets are typically generated in time intervals. For example, the dataset in our experiments was originally partitioned into 144 files, each containing all variables for all x, y, and z for a period of 5 days over 2 years (in these models all months are represented as having 30 days). This organization is best suited for queries that need all variables for a certain time period, not for queries that require a few variables at a time. Organizing the dataset for queries involving a few variables at a time is likely to help unanticipated queries as well.

Another common characteristic access pattern is locality of access in space and/or time. Even if no other information is available (i.e. no query types are specified) one can take advantage of this information to organize the data for efficient access. Known techniques for organizing data for queries that have locality of reference are discussed in Section 8.1.

2. The smaller the size of the clusters the better the solutions are. As explained above the size of the clusters depend on the file overhead. This was observed in the experiments above, where the Exabyte system had a relatively small overhead, and thus the number of clusters was about 3200 for the 2 year dataset. In contrast, the Ampex system had a relatively large file overhead, and the number of clusters was 465 for the same dataset. This explains the fact that all the queries in the Exabyte experiment performed better than the original time, while in the Ampex there were three query types that did not perform as well as the original time. Note, however, that the overall performance was greatly improved, and that a solution could be chosen to favor a particular query type at the expense of others. In general, the effect of a large number of clusters benefits unanticipated queries as well.

3. In cases in which a compromise solution is not satisfactory, data duplication can be considered. The trade-off between access time and space is a well known principle in physical database design. In the case of multidimensional datasets, a small percentage of duplication of data can alleviate access pattern conflicts. This is particularly true for queries that involve only one or two variables. We have not studied this effect so far. We plan to develop algorithms that take into account a percentage increase in space requirements in order to achieve improved solutions. The designer will then have a way to evaluate the benefits of data duplication.

4. Dealing with unanticipated queries is a difficult problem with all database and file systems. The best way of dealing with this problem is to monitor the access patterns, to collect statistics, and to permit reorganization from time to time, especially for frequently accessed datasets. Here again, analysis tools such as the partitioning workbench, and automatic reorganization software are needed to facilitate this fairly expensive operation.

6. IMPLEMENTATION OF THE PARTITIONING ALGORITHM AND THE WORKBENCH

6.1. Design Considerations of the Partitioning Workbench

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. However, we found out that assigning weights is not a meaningful task for the scientists, and that the assignment of weights was confusing in practice. 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. The result is an interactive “partitioning workbench”. The goal of the workbench is to help the scientists in designing the partitioning of a dataset, to see the trade-offs between possible solutions, and to make a meaningful selection of the most desirable solution.

The difficulty with assigning weights to query types can be illustrated with an example. Suppose that we permit weight values W_i to be between 1 and 10 (10 designating that Q_i is a very important query type). It became clear that such assignment of values is relative. For example, assigning a value W_i=8 while the other query types are assigned values below 2, makes this query type dominant. However, if all the other query types were assigned the weight 10, then this query type is viewed as “less important”. Even if we can normalize the final value assigned, the relative distance between weights carries no real meaning as to their effect on the partitioning solution. Consequently, we decided to present the effects of a solution in terms of estimates on actual times that each query will take under a given solution. In the following, we assume that all queries that belong to the same query type will have the same response time. This approximation is reasonable since each query that belongs to the same query type needs to access the same amount of data.

To make the estimates on actual times meaningful it is necessary to present them relative to the best possible time for each query that belongs to a specific query type Q_i. The best possible time (optimal), O_i, is calculated assuming that all the information to answer the query in Q_i is in a single file, and that the file is positioned at the beginning of a tape. Thus, O_i 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 increasing the weight on that query type. For comparison purposes, we calculate the original time that the query would have taken to run without the reorganization of the datasets (for our example dataset, the original organization consisted of 292 files, where each file contained 5 days for all variables).

We produce the multiple solutions presented to the designer as follows. We start by assigning all queries Q_i the same weight, and run step 2 of the algorithm (described in Section 4.3.2). It produces a solution for each permutation of the dimensions. Each solution consists of the the estimates on actual response times for all query types. Thus, for each query in Q_i and each solution option j, the actual response time is A_{ij}. We then select the solution that minimizes the overall response time relative to best possible times. We use the formula $\sum_i log(\frac{A_{ij}}{O_i})$ to find the value assigned to each solution j, and select the solution with the smallest value. The reason for considering the ratios is that we are interested in evaluating response times relative to the corresponding optimal times of queries, and not the absolute measure of response times. Further, the reason for using the sum of the log of these ratios (which has the same effect as taking the product of the ratios) is that we are interested in the improvement of ratios relative to each other. In other words, we prefer solutions where these ratios are as close as possible to each other. Thus, an improvement of a query from 200 minutes to 100 minutes is considered equivalent to an improvement of another query from 2 to 1 minutes. This is shown in Figure 6, for the first five query types described in Section 5 for the Exabyte system.

As can be seen in Figure 6, the estimates on actual response times for all query types has improved significantly relative to original times. For this solution, Q_1, Q_2, and Q_3 are at optimal or close to optimal. Q_4 is roughly a factor of 12 from optimal, and Q_5 is roughly a factor of 4 from optimal. The usefulness of this approach can be seen by realizing that Q_5 takes over 4 hours at best (it is a post-processing query type), so trying to improve it further would make little difference. On the other hand, improving Q_4 from about 92 minutes to 7 minutes makes a big difference for near-line usage.

Now, suppose that the designer wishes to improve Q_4. Using the workbench, he/she can display other solution options for Q_4 to see if there are more desirable solutions. The designer can then interactively choose other solutions and see the effect on the response time of other queries. We found by experience that 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. This is discussed further in Section 8.

Once a desired solution is selected, the weights for each query type are calculated and used in the remaining steps of clustering and bundling the basic units. This is described in the next subsection as part of the LAYOUT module.

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, and is used to store up to 25 tables

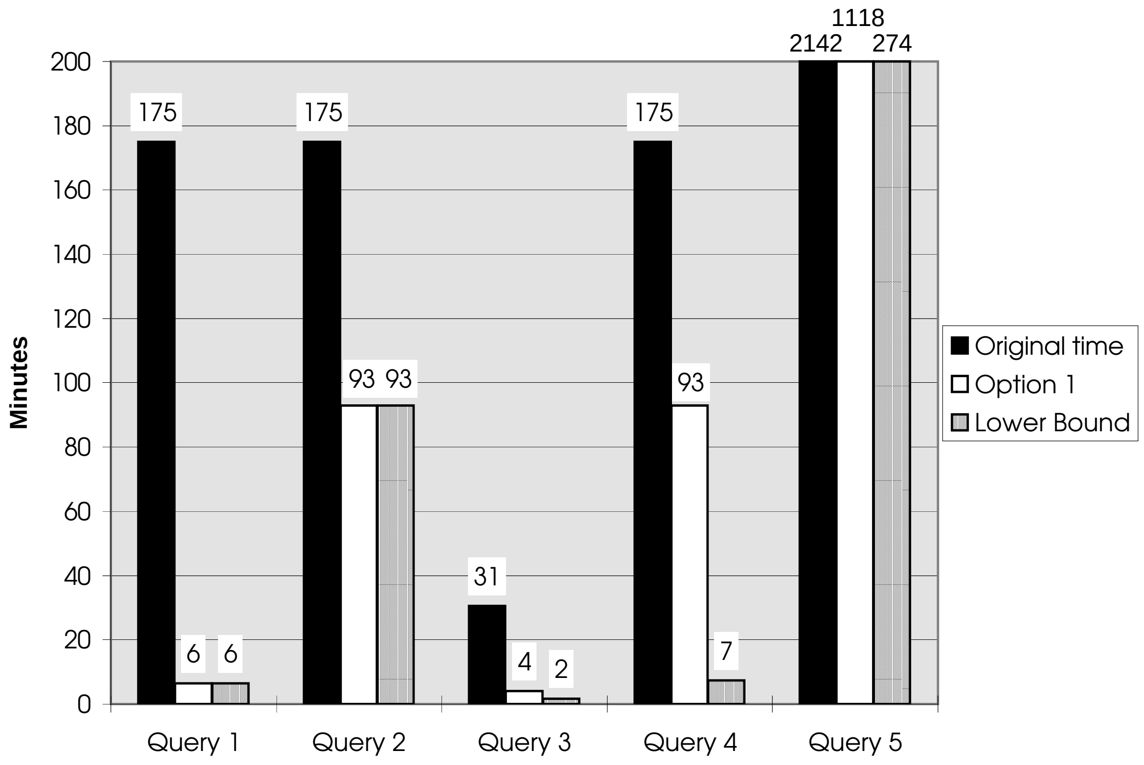

Fig. 6: One Partition Option for the U,V,W Group

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. These programs are discussed next.

6.2. Implementation of Partitioning Algorithms for the Workbench

The algorithms for data partitioning were developed and implemented. The partitioning algorithm was broken into three separate programs, so that the scientist's workbench can allow the user to interact with the partitioning process. The three programs are described below:

GROUP: This program first determines how to separate all the variables of the dataset into groups, such that any query type that accesses multiple variables, will always access variables of the same group. This step is separated out from the other steps so that the user may first examine the grouping of variables, and then individually work on each group. This program also computes the optimal response time O_i for each query type Q_i.

PERMUTE: This module performs the first two steps of the partitioning algorithm in which variables in each group are broken into basic units, and different permutations of the dimensions are tried to determine an optimal one dimensional ordering of the basic units. Equal weights are assigned to each query type, and different permutations of the dimensions are tried as options. As explained above, for each option, $\sum_i log(\frac{A_{ij}}{O_i})$ is computed to determine an ordering of the options, where options that have a smaller value represent a more desirable option. This ordered set of options is returned to the GUI part of the workbench, for the designer to examine in order to select the most desirable permutation of dimensions.

LAYOUT: After the user has selected the most desired permutation of dimensions (i.e., option) from the output of PERMUTE, the result is fed to this module. This module then performs the dynamic programming algorithms of steps 3 and 4, in order to determine what data belongs in each cluster and which clusters should be put together into a bundle on a single tape. The weights for each query type, used in these dynamic programming algorithms are generated by making them inversely proportional to the estimated response time of each query type. For example, suppose that three query types are involved, and that the estimates on actual response times are 2, 5, and 10 minutes for Q_1, Q_2, and Q_3, respectively, then the relative weights are 1/2, 1/5, and 1/10 respectively. The final weights are obtained by normalizing these relative weights, to yield W_1=0.625, W_2=0.25, and W_3=0.125, respectively. The reason for this choice of assigning weights is that in most applications it is not important to optimize queries with long response times, as compared with queries having short response times. Other weight assignment formulas can easily be used with the workbench if desired. The resulting layout is also used by the workbench to estimate the response time for any ad-hoc query, not only queries belonging to the original query types. This can be used to see the effect of reorganization on unanticipated queries.

7. THE STORAGE SYSTEM INTERFACE DESIGN

The implementation of the interface to the mass storage system is performed at the National Storage Laboratory (NSL) at Lawrence Livermore National Laboratory (LLNL). The NSL provides two important functions: a site where experiments can be performed with a variety of storage devices, and the support for the data structures necessary to support storage and access of multiple clusters.

The first software system developed at the NSL features network-attached storage, dynamic storage hierarchies, layered access to storage-system services, and extensive storage-system management capabilities. A commercial version of this system, called NSL-UniTree [†], was announced late in 1992 by IBM Federal Systems Company. Work on a second software system, called the High-Performance Storage System (HPSS), is also under way. A central goal of this new effort is to move large data files, using parallel I/O, between storage devices and massively parallel computers. Another goal is the efficient support of scientific data management applications.

Current versions of mass storage systems strive to present clients with a file system view, usually hierarchical, and compatible with widely used operating systems such as UNIX. This view is not necessarily the most efficient one for clients of mass storage systems that would prefer to deal with scientific data objects (such as multidimensional datasets) instead of files. The current NSL-UniTree implementation is no exception to the file system view: access is available via standard FTP and NFS file transfer protocols, or via a file-oriented client application programming interface. We have developed the specification for a more suitable interface between mass storage systems and application software to allow for better control over data storage organization and placement for the benefit of data management clients such as the data partitioner and subset assembler discussed here.

Enhancements necessary to support this work were made to the NSL-UniTree. In order for the Storage Manager Write Module 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 the pre-allocation of space for "clusters" and the piece-wise writing and reading of the "clusters". Such protocol enhancements were added to the NSL-UniTree.

The Storage Manager Read Module supports the efficient reading of clusters. In the case that only a single cluster is needed to satisfy the subset request, it needs to mount the proper volume, fast forward to the position of the cluster and read this cluster only. In the case that multiple clusters are needed to satisfy a subset request, it needs to order the reading of clusters in such a way as no unnecessary rewinds of the volume take place. This capability is already supported by NSL-UniTree.

[†] A previous version of a hierarchical storage system, called Unitree, was originally developed at LLNL, and later marketed commercially.

7.1. Summarization of Interface Requirements

We have designed and implemented a functional interface between the partitioning and subset assembly engines and the mass storage system which provides the ability to control allocation of space and physical placement of data. The approach that has been taken is to define several "class of service" attributes which are associated with clusters and cluster sets and provided to the storage system via a modified FTP interface. These 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 tells the storage system the desired order of storing the clusters. The frequency of use parameters indicate the desirability of storing a cluster close to the beginning of a tape (or to a dismount station) 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 is used to determine whether two clusters should be split across tape volumes.

A table of these attributes is provided prior to delivery of the individual clusters to the storage system to permit the storage system to bundle the clusters appropriately for the desired tertiary storage device. Thus, the last step of the partitioning process, (i.e. the bundling of clusters, such that each bundle can fit on a tape), is done by the storage system. This point was considered necessary to allow for the possibility that precise storage system parameters may not be known by the partitioning engine, and to provide storage system managers with the ability to override the partitioning engine's decisions in order to prevent storage system overload.

The interface implementation in the context of NSL-UniTree is shown in Figure 7. As can be seen, this mechanism allows the partitioning engine to determine optimal data layout on a given destination, but it also gives final control to the storage system. Data is transferred to the storage system in cluster sets. While there is no strict requirement that all clusters be contained within a cluster set, such a requirement is necessary if the benefits of data association are to be realized. Clusters which are provided to the storage system independently will be stored individually – as normal files would be.

So as not to constrain the partitioning engine unnecessarily, no limits on cluster set size have been established. This however means that any particular cluster set may be larger than the available storage system disk cache. To prevent cache overflow, the cluster sets are "bundled" into pieces which the storage system can easily manage.

Once the cluster set attribute table is available in the storage system, a "bundling" function is called to create bundled data for internal use by the storage system migration server. This server then builds a bundle table and examines that table for the availability of complete bundles. In order to provide the necessary inter-cluster cohesion, bundles are migrated to tertiary storage levels only when complete. Once a complete bundle is available to the migration process, this process provides a list of clusters (files) to be migrated to the destination server.

To ensure that clusters are stored without unnecessary volume breaks (which would result in an additional media mount penalty for the reading process), code has been added to the migration server to allow it to check the destination volume space availability prior to actual bundle migration. If this code finds insufficient available space to store the entire bundle on the destination volume, the bundle migration will be deferred for a finite period which can be determined by a storage system management process. The idea behind this deferral is to allow non-bundle files to be migrated first with the hope that doing so results in the mounting of a new (empty) volume. Code has been provided to permit storage system management to override this deferral in cases where the system cache space becomes exhausted.

A basic implementation of the interface design was developed for NSL-UniTree. The assumed storage objects were clusters and cluster sets. For this initial implementation, we assumed no partial read or write operations, even though they were available through the NFS access broker

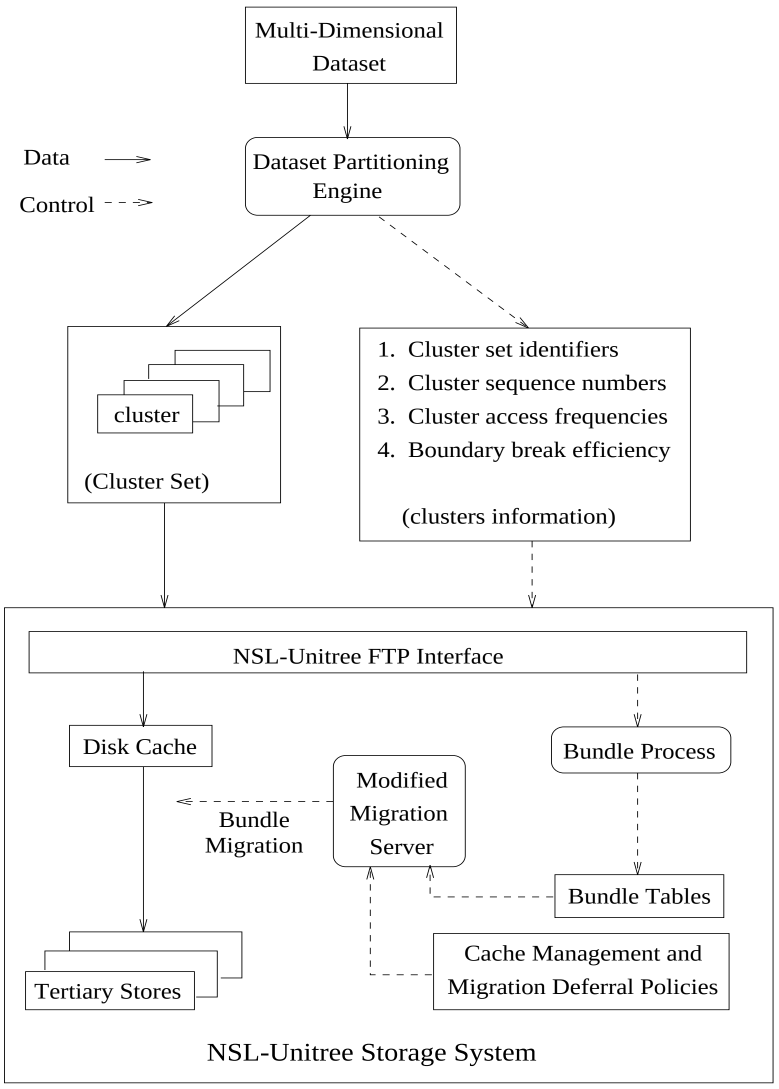
Multi-Dimensional Dataset
Data
Control
Dataset Partitioning Engine
cluster
(Cluster Set)
1. Cluster set identifiers
2. Cluster sequence numbers
3. Cluster access frequencies
4. Boundary break efficiency
(clusters information)
NSL-Unitree FTP Interface
Disk Cache
Modified Migration Server
Bundle Process
Bundle Migration
Bundle Tables
Tertiary Stores
Cache Management and Migration Deferral Policies
NSL-Unitree Storage System

Fig. 7: NSL-Unitree Enhancements Implementation

and client libraries. For this implementation data transfer was restricted to the FTP interface. This restriction will be removed as part of planned future work.

8. FUTURE EXTENSIONS

8.1. Partially Specified Query Types

In general, query types may be specific in some dimensions but non-specific in others. For example, a query type may specify that in the time domain we are interested in one month at a time (All D, Any M), and one height at a time (Any Z), but nothing about X and Y. A reasonable interpretation of that is that the query can request any (non-specified) region in the X-Y dimension. Thus, it is desirable to maintain the spatial locality in the X-Y dimensions. How should clusters be organized in this case?

To illustrate our approach to this problem, consider an extreme situation where no information exists on query types at all. In that case the multidimensional space is partitioned into spatial hyper-cubes. The linear ordering of these hyper-cubes has been the subject of several studies that suggest that a z-ordering (also called Peano ordering) [11] yields the best results for random queries. Another ordering method, called Hilbert ordering [9] was shown to be better for range queries [9]. We assume that if no other information exists, most query types involve range queries, and therefore we select the Hilbert ordering (H-ordering) method.

Our approach is to use a hybrid of the total ordering of dimensions that are specified in the query types and H-ordering for dimensions that are not specified. For the query type example above, a solution can be represented using the following notation: Z,(X-Y),M,D. The interpretation is that the ordering is first by Z, then the X-Y dimensions are H-ordered, then by M (month), then by D (day). Thus, a particular cluster will be for a particular height, for a particular X-Y rectangular region for that height, and will contain all days of a single month.

The above method is planned to be implemented and incorporated into the partitioning software in a future version.

8.2. Other Extensions

Another important extension to the partitioning methods is to permit partial duplication of data. In the case of query type conflicts for queries that are considered essential, it may be necessary to duplicate some of the data. The goal is to provide the scientist with trade-offs between additional stored data and the improvement to the desired query types. Again, it is important to provide the scientist the ability to analyze the trade-offs interactively, using the workbench. The problem is one of finding the best solutions while minimizing the duplication of data.

Duplication of data introduces synchronization problems between the duplicated versions when the data is modified (updated, deleted, or new data inserted). However, the typical scientific application generates the dataset once, and does not modify them.

Another aspect that we plan to study in the future is the effect of hardware devices with different fundamental characteristics than linear tapes. In particular, there is emerging new technology called “serpentine tape” where a single tape has multiple tracks and the data can be laid out on these tracks in a serpentine fashion [16]. This technology does not change the basic value of partitioning the data into clusters, but rather changes the layout of these clusters on the tracks.

Finally, it is necessary to emphasize that our work applies only to data that is fundamentally organized as grid data (or uniform meshes). While this represents a large class of applications (earth science, environmental science, etc.) there are applications with different topography. Some simulations permit adaptive meshes that can vary from region to region, some may use irregular topography for representing turbulence of flow within irregular boundaries, such as rivers. Such application areas are yet to be explored.

It is also worth pointing out that we have considered query types that are basically range queries; that is queries that specify ranges on the different dimensions. Some application may have different dominant style. For example, in the study of beach erosion, the desired proximity of X-Y tiles is along the patterns of the beaches. Thus, it will be desirable to store the tiles physically along such patterns. A similar situation exists in the study of earthquake fault lines. Characterizing such query types and analyzing clustering methods for such applications is yet to be studied.

9. 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. Recent research on data allocation and placement based on application access patterns in the context of parallel disks and RAID technology is reported in [15]. However, the problem is much more acute when dealing with robotic tertiary systems, and the solutions are different. We have chosen to work closely with a specific application area (climate modeling) where this problem is effecting 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 from 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.).

Another challenging area is the development of interfaces that are natural and familiar to users of each scientific discipline. Experience shows that scientists are reluctant to learn new systems, new concepts and new query languages (such as SQL). This reluctance is justified as learning new systems takes away from the effectiveness of the scientist's work. Interfaces need to be tailored to the application and to the discipline. The development of the scientist "workbench" was necessary in order to facilitate the interaction with the scientists who specify the query types and select the most advantageous partitions. Similar interfaces can be designed for scientists to express queries over datasets.

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 [14]). 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.

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] Coyne, R. A., Hulen, H., Watson, R. W., Storage Systems for National Information Assets, Supercomputing '92 Proceedings, Minneapolis, MN, November 1992.

[3] Graf, O., Nguyen, J., WOCE-CME Data System Prototype, Overview of IBM GBSI IR&D and Earth Sciences Programs, July 1991.

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

[5] McDonald, K. R., Calvo, S., Accessing Earth Science Data from the EOS Data and Information System, Proceedings of the IEEE Symposium on Mas Storage Systems, Monterey, April 1993.

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

[7] Sarawagi, S., Stonebraker, M., Efficient Organization of Large Multidimensional Arrays, Tenth International Conference on Data Engineering, February, 1994.

[8] Chen, L. T., and Rotem, D., Optimizing Storage of Objects on Mass Storage Systems with Robotic Devices, EDBT (Extending Database technology) 94, Cambridge, U.K.

[9] Faloutsos, C. and Roseman, S., Fractals for Secondary Retrieval, Eighth ACM Symposium on Principles of Database Systems (PODS). March, 1989. Pages 247-252.

[10] Tsuda, T. and Sato, T., Transposition of Large Tabular Data Structures with Applications to Physical Database Organization. Part I, Acta Informatica 19:13-33 (1983), Part II, Acta Informatica, 19:167-182 (1983).

[11] Samet, H., The design and analysis of Spatial Data Structures, Volume I, page 14, Addison Wesley 1990.

[12] Orenstein, J., Merret, A class of Data Structures for Associative Searching, PODS, 1984, Pages 181-190.

[13] Gary and Johnson, Computers and Intractability, W.H.Freeman and Company, San Francisco 1979.

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

[15] Scheuermann, P., Weikum, G., and Zabback, P. Data Partitioning and Load Balancing in Parallel Disk Systems, Department Informatik, ETH Zurich, January 1994, Technical Report number 209.

[16] Graves, D., Major Capacity and Data Rate Advances in IBM Longitudinal Magnetic Tape Recording Technology, 13th Symposium on Mass Storage Systems, June 1994, Annecy France.

Acknowledgements — We would like to thank the anonymous reviewers and the editors for their insightful and detailed comments. This work is supported by the Office of Scientific Computing of the Office of Energy Research, U.S. Department of Energy. This report was issued as LBL report LBL-35441.

APPENDIX A: COMPLEXITY OF THE ONE DIMENSIONAL LAYOUT PROBLEM

Let us consider the simplest version of the Layout problem with one tape, equal query weights and zero seek time on tapes. We will show that even this simplified version is NP-Complete thus proving that our more complex problem with unequal weights is also NP- complete. The complexity of the Layout problem is closely related to the problem known as Consecutive Ones Property (CO) in binary matrices. A binary matrix has the CO property if its columns can be permuted such that the 1's in each row appear consecutively. Given n basic units and m queries, we can think of our problem as a binary matrix $A(a_{ij})$ where $a_{ij} = 1$ if and only if query i needs to read basic unit j in its answer set. Assume the i'th query requests p_i basic units for its answer. We measure query response time by the distance between the first and last basic units needed for its answer, this may include some irrelevant basic units it has to skip through. For simplicity let us assume each basic unit read represents one response time unit. Then the optimum expected response time for a given Layout problem is clearly $R = \sum_{i=1}^{m} p_i$ which occurs if we can permute our basic units such that for every query, all its requested basic units occur consecutively. We will now show that the problem of arranging our basic units along one tape is NP-Complete using a reduction from the problem known as COMA (Consecutive Ones Matrix Augmentation [13, pg. 229].)

The COMA problem is stated as follows:

Given an $m \times n$ matrix A of 0's and 1's, and a constant K, find a matrix A' obtained from A by changing K or fewer 0's to 1's such that A' has the CO property.

Proposition 1 *The Layout problem is NP-Complete.*

Proof. In the following we denote a sequence of consecutive 1's as a 1-block. In case a row contains a single 1-block the corresponding query has no overhead. Otherwise the number of 0's separating the 1-blocks represent overhead. Given an instance of the COMA problem we translate it to the Layout problem with m queries and n basic units and ask if a solution with response time less than R+K exists. In this case K represents the overhead. A solution with response time less than R+K for the Layout problem exists if we have at most a total of K 0's separating our 1-blocks over all the rows. On the other hand if COMA has a solution with K or fewer changes of 0's to 1's, a solution to the Layout problem with response time less than R+K exists. □

APPENDIX B: OPTIMAL COMBINING OF BASIC UNITS INTO CLUSTERS

In Appendices B and C we describe dynamic-programming solutions to the problems of splitting the basic units into clusters (each treated as a separate file) and bundling the clusters (i.e. files) into volumes. Although there are some similarities between the algorithms, the defining equations turn out to be different. In both Appendices, we assume that each query has an associated weight reflecting its "importance" to the user. The method of derivation of query weights for our specific application is described in Sections 4 and 6.

The problem we consider here is that of breaking a linear stream of N basic units, $B_1, B_2, .., B_N$, into files, such that the extra number of bytes that needs to be read by all queries is minimized. Extra bytes need to be read due to the fact that an entire file needs to be read even though only a portion of it is needed by the query. Extra bytes, also come from the fact that a file overhead equivalent to reading FO bytes, is needed for each file that needs to be retrieved by the query. A break can occur anywhere between a pair of consecutive basic units but not in the middle of a basic unit.

We assume each query needs to access some subsequence of contiguous basic units from the stream $B_1, B_2, .., B_N$. Based on the information from these queries, we can collect the following information for each basic unit B_i:

1. L_i: The length of the basic unit (in bytes).

2. Y_i: The sum of the weights of all queries that read B_i.

3. X_i: The sum of the weights of all queries that read both B_{i-1} and B_i.

Given the above information, we can easily compute in O(N) time:

1. P_i: The starting position (in bytes) of basic unit B_i measured from the beginning of B_1. It can be computed by letting $P_1 = 0$, and $P_i = P_{i-1} + L_{i-1}$.

2. S_i: The sum of weights of all queries that *start* reading from this basic unit. This can be simply computed by $S_i = Y_i - X_i$.

3. E_i: The sum of weights of all queries that *stop* reading after basic unit B_{i-1}. This can be simply computed by $E_i = Y_{i-1} - X_i$.

We now describe an O(N^2) time dynamic programming algorithm to solve this problem. Initially the algorithm assumes that each query simply reads the basic units that it needs and that no file overhead is involved (i.e 0 extra bytes are read). Then, starting from the last basic unit B_N, and working our way down to B_1, we start to consider the cost of the file overhead, and the weighted extra number of bytes that needs to be read due to its existence, i.e., for each affected query we multiply its weight by the extra number of bytes it reads and sum this value over all affected queries. For each B_i along the way, we compute a D_i that is equal to the weighted extra number of bytes that needs to be read, under the following conditions:

1. Basic units B_{i-1} and B_i are split into 2 different files, and the file overhead for splitting here is considered.

2. No extra bytes are read from basic units B_1 through B_{i-1}, i.e., each of these $i-1$ basic units is in a file by itself.

3. Basic units B_i through B_N have been split into files in the optimal way, such that D_i is minimized.

Dynamic programming can be employed efficiently in this case, because each D_i can be computed in O(N) time by only using D_j values where $j > i$, based on the following formula:

$$D_i = X_i \times FO + \min_{i<j\leq N+1}(D_j + U(i,j))$$

where

$$U(i,j) = \sum_{k=i}^{j-1}(S_k \times (P_k - P_i) + E_k \times (P_j - P_k))$$

Initially we set $D_{N+1} = 0$. Assume by induction that D_m is correctly computed for all $m > i$, i.e., D_m represents the additional number of bytes to be read under the optimal way to break basic units B_m through B_N into files. Given that there is a break between B_{i-1} and B_i, let the immediate break on the right of it be between B_{j-1} and B_j for some $j > i$. This implies that the basic units $B_i, B_{i+1}, \ldots, B_{j-2}, B_{j-1}$ will be placed in one file. The term $B_i \times FO$ reflects the extra file overhead needed for all the queries that access both B_{i-1} and B_i. $U(i,j)$ reflects the extra number of bytes that needs to be read due to that fact that basic units B_i through B_{j-1} are now in one file, and all of these basic units need to be read by any query that needs any portion of these basic units. For all queries that start reading from basic unit B_k, $(P_k - P_i)$ extra bytes will have to be read at the beginning of the file. Since S_k is the weighted sum of all queries that start reading from basic unit B_k, $S_k \times (P_k - P_i)$ will be equal to the weighted extra number of bytes read at the beginning of the file, by all these queries. Similarly, $E_k \times (P_j - P_k)$ represents the weighted extra number of bytes read at the end of the file, by all queries that stop reading within this file.

This formula will compute the extra number of bytes to be read under the optimal way to break basic units B_i through B_N into files because all possible j's, in which basic unit B_i through B_{j-1} can belong in one file are considered, and because the D_j's have been correctly computed by the induction hypothesis.

Note that $U(i,j)$ can be accumulated while we are enumerating through the j's for each D_i, this reduces the running time of the algorithm to O(N^2) since each D_i can be computed in O(N) time.

At each step in which a new D_i is computed, for each i, we record the newly found minimal value in D_i and also record the j value j_{min}, that resulted in this minimal value. Let the recorded pointer be R_i (i.e. $R_i = j_{min}$). At the end, when we have finally computed the minimal value for D_1, we can simply follow the pointers starting from R_1, in order to find all the positions in which to break the stream of basic units into files. Since this breaking resulted in D_1 obtaining its minimal value, it is guaranteed to be the optimal way to break the entire stream into multiple files, such that the extra number of bytes to be read by all queries is minimized.

APPENDIX C: OPTIMAL BUNDLING OF CLUSTERS INTO TAPE VOLUMES

The problem we consider here is that of breaking a linear stream of N clusters, $C_1, C_2, .., C_N$, into bundles such that each bundle fits on a tape volume, and the summed weighted query response time in minimized. A break can occur anywhere between a pair of consecutive clusters but not in the middle of a cluster.

As before, we assume each query has a weight associated with it, and needs to access some subsequence of contiguous clusters from the stream $C_1, C_2, .., C_N$. Based on the information from these queries, we can collect the following information for each cluster C_i:

1. L_i: The length of the cluster (in bytes).

2. Y_i: The sum of weights of all queries that read C_i.

3. X_i: The sum of weights of all queries that read both C_{i-1} and C_i.

The parameter X_i can be viewed as the penalty incurred by placing the clusters C_{i-1} and C_i on two different tapes. This is because if a tape break actually occurred between clusters C_{i-1} and

C_i, all the queries counted in X_i would have an increased response time of T_m seconds, since a new tape would need to be mounted to answer the remaining part of the query.

Given the above information, we can easily compute in O(N) time:

1. P_i: The starting position (in bytes) of cluster C_i measured from the beginning of C_1. It can be computed by letting $P_1 = 0$, and $P_i = P_{i-1} + L_{i-1}$.

2. S_i: The sum of weights of all queries that *start* reading from this cluster. This can be simply computed by $S_i = Y_i - X_i$.

An O(N^2) time dynamic programming algorithm to solve this problem is as follows: Initially the algorithm assumes that all the clusters are on one long imaginary tape that can hold everything (i.e., the leftmost break is on the right of cluster C_N). Let the total weighted response time under this one tape configuration be some constant K. Starting from the last cluster C_N, and working our way down to C_1, we compute a cost D_i, which is the minimal extra response time (relative to K) involved in a configuration in which:

1. There is a break between C_{i-1} and C_i.

2. There are no breaks between C_1 through C_{i-1}.

3. The other breaks between C_i through C_N have been chosen optimally, under the constraint that no tape contains more than M bytes.

Note that D_i could be a negative number, since it represents the gain or the loss in response time due to the placement of such breaks. Dynamic programming can be employed efficiently because each D_i can be computed in O(N) time by only using D_j values where $j > i$, based on the following formula:

$$D_i = X_i \times T_m + \min_{i<j\leq N+1}^{P_j - P_i \leq M} (D_j - U(i,j))$$

where

$$U(i,j) = \sum_{k=i}^{j-1} S_k \times (T_s(P_k) - T_s(P_k - P_i))$$

Initially we set $D_{N+1} = 0$. Assume by induction that D_m is correctly computed for all $m > i$, i.e., D_m represents the additional cost of the optimal way to break clusters C_m through C_N into tapes. Given that there is a break between C_{i-1} and C_i, let the immediate break on the right of it be between C_{j-1} and C_j for some $j > i$. This implies that the clusters $C_i, C_{i+1}, \ldots, C_{j-2}, C_{j-1}$ will be placed on one tape. The constraint $P_j - P_i \leq M$ in the above equation guarantees that in computing the minimum, we are only considering values of j such that these clusters do in fact fit on one tape. The term $X_i \times T_m$ reflects the extra response time needed to mount a new tape for all the queries that access both C_{i-1} and C_i. On the other hand, $U(i,j)$ reflects the decrease in response time due to that fact that the seek time for all clusters C_k in the set $C_i, \ldots, C_{j-1}$ will now be reduced from $T_s(P_k)$ to $T_s(P_k - P_i)$ since C_i is now at the beginning of a tape. Note that this reduction in seek time only affects the queries that *start* reading from cluster C_i. This is because, a query only needs to seek once to the first cluster, and all other clusters are read consecutively without any more seeks.

This formula will compute the relative response time of the optimal way to break the clusters from C_i through C_N, because all possible j's, in which cluster C_i through C_{j-1} can fit on one tape are considered, and because the D_j's have been correctly computed by the induction hypothesis.

Note that $U(i,j)$ can be accumulated while we are enumerating through the j's for each D_i. This reduces the running time of the algorithm to O(N^2) since each D_i can be computed in O(N) time.

At each step in which a new D_i is computed, for each i, we record the newly found minimal value in D_i and also record the j value j_{min}, that resulted in this minimal value. Let the recorded pointer be R_i (i.e. $R_i = j_{min}$). At the end, when we have finally computed the minimal value for D_1, we can simply follow the pointers starting from R_1, in order to find all the positions in which to break the stream of clusters. Since this breaking resulted in D_1 obtaining its minimal value, it is guaranteed to be the optimal way to break the entire stream into multiple tapes, such that the weighted response time for all queries is minimized.

The interested reader should consult [8] for a more detailed description of this algorithm, along with a description of how this algorithm can be improved to require only $O(N)$ time when the tape seeking function T_s is a linear function.