Efficient Organization of Large Multidimensional Arrays*

Sunita Sarawagi Michael Stonebraker
Computer Science Division
University of California at Berkeley
sunita@cs.berkeley.edu, mike@cs.berkeley.edu

Abstract

Large multidimensional arrays are widely used in scientific and engineering database applications. In this paper, we present methods of organizing arrays to make their access on secondary and tertiary memory devices fast and efficient. We have developed four techniques for doing this: (1) storing the array in multidimensional "chunks" to minimize the number of blocks fetched, (2) reordering the chunked array to minimize seek distance between accessed blocks, (3) maintaining redundant copies of the array, each organized for a different chunk size and ordering and (4) partitioning the array onto platters of a tertiary memory device so as to minimize the number of platter switches. Our measurements on real data sets obtained from global change scientists demonstrate that accesses on arrays organized using the above techniques are often an order of magnitude faster than on the original unoptimized data.

1 Introduction

Scientific and engineering applications often utilize large multidimensional arrays. Earth scientists routinely process satellite images in the form of large two and three dimensional arrays [DOZ91]. Their simulations of atmosphere and ocean climatic conditions generate large regular arrays of floating point numbers as output [MEC92]. For example, typical runs of the UCLA

*This research was sponsored by the National Science Foundation under grant IRI-9107455, the Defense Advanced Research Projects Agency under grant T63-92-C-0007, and the Army Research Office under grant 91-G-0183. Additional support was provided by the University of California and Digital Equipment Corporation under Research Grant 1243. Other industrial and government partners include the State of California Department of Water Resources, United States Geological Survey, Construction Engineering Research Laboratory (CERL) of the U.S. Army Corps of Engineers, the National Aeronautics and Space Administration (NASA), Epoch Systems, Inc., Hewlett-Packard Corp., Hughes Aircraft Company, MCI, Metrum Corporation, PictureTel Corporation, Research Systems Inc., Science Applications International Corporation, Siemens Inc., and TRW Space and Electronics.

General Circulation Model (GCM) generate five dimensional arrays of size 5 to 50 Gigabytes. Other areas where large arrays are commonly used include image processing [WAD84], computational chemistry, structural dynamics and seismology. Because of the large storage requirements for such arrays, they are usually allocated to tertiary storage devices. Achieving high performance in spite of the non-uniform access times and the high latency of such storage devices requires good allocation strategies [STO91a].

The traditional method of storing a multidimensional array is *linear allocation* whereby the array is laid out linearly by a nested traversal of the axes in some predetermined order. This strategy, which mimics the way FORTRAN stores arrays in main memory, can lead to disastrous results on a secondary or tertiary memory device. Because users typically access large arrays in several different ways, FORTRAN order will optimize for one access pattern while making all others very inefficient. Optimizing the allocation of the array becomes increasingly important as array dimension and size increases. In this paper, we explore methods of structuring arrays to reduce latency and improve speed of data accesses. The strategies we explore are:

- **chunking** : dividing the array into chunks (multidimensional tiles) that are stored and accessed together;
- **reordering** : permuting the dimensions of the chunked array to reduce average seek distance.
- **redundancy** : storing redundant copies of the array which are organized differently to optimize for different patterns of access; and
- **partitioning** : allocating an array to platters of a tertiary memory device to minimize the number of platter switches.

The above techniques can be used, in combination, to tune the array's internal structure to an access pattern obtained from either an end user or from statistical sampling by a data management system.

Chunking in the context of image processing has been used to build tiled virtual memory systems [WAD84] [REU80] [FRA92]. Whereas those systems deal only with two dimensional arrays and assume magnetic disk as the storage device, our interest is in multidimensional arrays with both magnetic disk and tertiary memory as storage devices. A more theoretical approach to organizing multidimensional arrays is presented in [ROS75]. Their approach organizes data without regard to access pattern, whereas our work considers access patterns to optimize layout.

Array organization is related to the general problem of data clustering. Most clustering algorithms [JAI88] work on a collection of records that are not structured in any way. Arrays have a regular structure that facilitates a different approach to storage organization.

The rest of this paper is organized as follows. In Section 2 we present the different schemes we used for organizing arrays, namely chunking, reordering, redundancy and partitioning. In Section 3 we describe our implementation of multidimensional arrays in the next generation DBMS Postgres [STO91b]. Section 4 presents our simulation of several earth science arrays used by global change researchers in the Sequoia 2000 project [STO91c] and shows the results of our array organization schemes on this data. Lastly, we present future work and conclusions in Section 5.

2 Storage of Arrays

We begin this section by presenting the access pattern model that we use for optimization of array layout. Then, we describe each of our four organization strategies and sketch an algorithm that can be used to implement it in a DBMS.

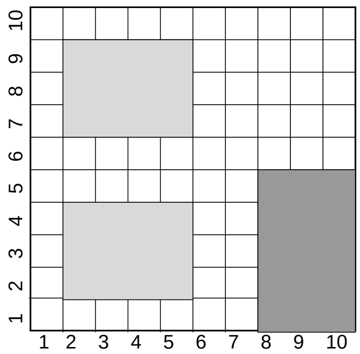

Figure 1: An Example Array

Consider an array of n dimensions. We model each user access request as a n-multidimensional rectangle located somewhere within the array. Furthermore, we group user accesses into collections of classes $L_1, \ldots L_K$ such that each L_i contains all rectangles of a specific size $(A_{i1}, \ldots A_{in})$ located anywhere within the array. Lastly, we assume that an access occurs to some rectangle in the ith class with probability P_i. Therefore, the access pattern for an array can be described by the set:

$$\{(P_i, A_{i1}, A_{i2}, \ldots, A_{in}) : 1 \le i \le K\}$$

Figure 1 illustrates an example on a 10×10 array. The three shaded rectangles (each accessed with probability $\frac{1}{3}$) represent access in two classes corresponding to the following access pattern:

$$\{(\tfrac{2}{3}, 3, 4), (\tfrac{1}{3}, 5, 3)\}$$

The access pattern can either be provided by an end user or can be determined by statistically sampling array accesses in a database management system.

2.1 Chunking

Instead of using FORTRAN style linear allocation, we can decompose the array into multidimensional chunks, each the size of one storage *block* A block is the unit of transfer used by the file system for data movement to and from the storage device. The shape of the chunk is chosen to minimize the average number of block fetches for a given access pattern. To illustrate the significance of chunking we consider the example shown in Figure 2. Figure 2(a) shows a 3-dimensional array of size X_1=100, X_2=2000 and X_3=8000 stored using linear allocation and Figure 2(b) illustrates the same array stored using a chunked representation. Assume the array

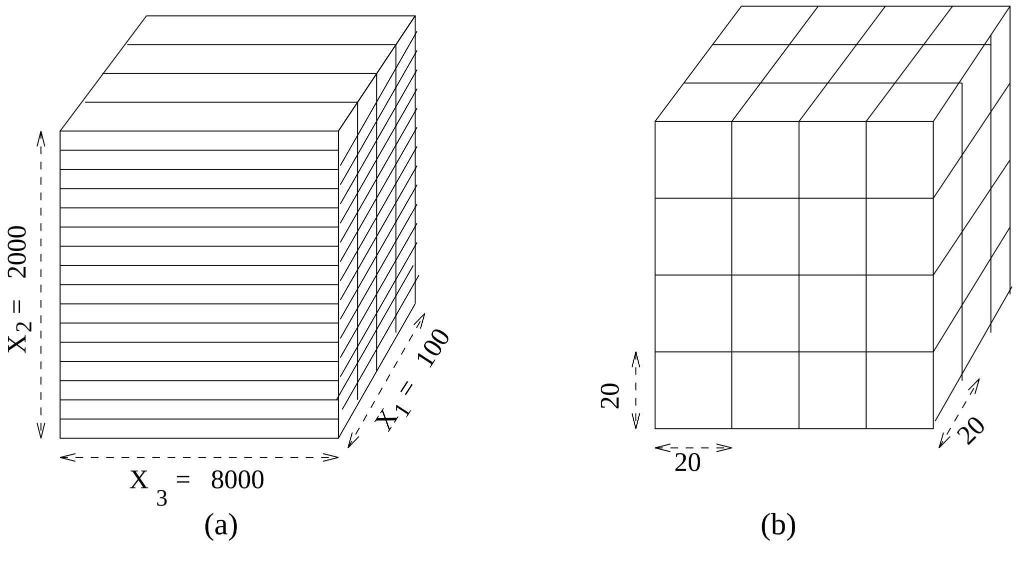

Figure 2: An example of array chunking

is stored on a magnetic disk and data transfer between main memory and disk occurs in 8kB pages (we assume 8kB = 8000 bytes for this example). Let the access pattern for this array be

$$\{(0.5,\ 10,\ 400,\ 10),\ (0.5,\ 20,\ 5,\ 400)\}.$$

If the array is stored linearly with X_3 as the innermost axis followed by X_2 and then X_1, as shown in Figure 2(a), then each disk block will hold just one row of values along X_3. The first access fetches a total of $10 \times 400 \times 1 = 4000$ blocks, and the second access fetches $20 \times 5 \times 1 = 100$ blocks. Hence, this access pattern fetches an average of $4000 \times 0.5 + 100 \times 0.5 = 2050$ blocks. Since, the bytes requested can fit in 5 blocks on the average, the amount of data fetched is 410 times the amount of useful data.

Suppose we divide the array into 8kB chunks. The shape of each chunk is a (20, 20, 20) cube as shown in Figure 2(b). For the same access pattern, the number of blocks fetched is $1 \times 20 \times 1$ for the first access and $1 \times 1 \times 20$ for the second access, assuming that the start of the access rectangle aligns perfectly with the start of a chunk. The average number of blocks fetched is $20 \times 0.5 + 20 \times 0.5 = 20$ as compared to 2050 for the unchunked array. Thus, chunking results in more than a factor of 100 reduction in the number of blocks fetched. In order to realize these improvements, we need a way to optimize the shape of a chunk.

We now present a formal definition of the problem. Given an n-dimensional array $[X_1, X_2 \ldots X_n]$ where X_i is the length of the i-th axis of the array, block size C and an access pattern $\{(P_i, A_{i1}, A_{i2}, \ldots, A_{in}) : 1 \leq i \leq K\}$, the objective is to find the shape of the chunk into which the array should be decomposed such that the average number of blocks fetched is minimized. The shape of the chunk is specified by a tuple $(c_1, c_2, \ldots, c_n)$ where c_i is the length of the ith axis of the multidimensional chunk. The size of the chunk puts the following additional constraints on each c_i:

$$\prod_{i=1}^{n} c_i \leq C$$

The average number of blocks fetched for a specified access pattern and chunk shape is given by:

$$\sum_{i=1}^{K} \left(\prod_{j=1}^{n} \left\lceil \frac{A_{ij}}{c_j} \right\rceil \right) P_i \tag{1}$$

The goal is to choose the chunk shape, satisfying the constraints, that minimizes (1).

The presence of the ceiling function in (1) makes a closed form solution difficult. One can always find the optimal solution by exhaustive search of all possible shapes that satisfy the size constraint. In this case, the number of shapes generated is exponential in the dimensionality of the array. Various techniques can be used to prune the search space. For example:

- Instead of considering all possible shapes, we only generate the ones which are *maximal*. A shape is maximal when increasing the length of any one of the sides of the shapes will violate the size constraint. For example, if $C = 15$ and $n = 2$, then shape (5,3) is maximal whereas (4,3) and (5,2) are not.

- Instead of considering all possible shapes, we first generate an approximate solution by only considering shapes for which the length of each side is a power of 2. This solution is then refined by considering the shapes that are in the "neighborhood" of this shape. The neighborhood consists of sides varying between double and half of the corresponding side in the approximate solution.

2.2 Reordering

Once the array is chunked, we require a good method of laying out the chunks on disk. The natural way is to lay out the chunks by traversing the chunked array in the axis order. Hence, different axes order will result in different chunk layout. The time to fetch the blocks for a requested rectangle can be greatly reduced by choosing the right axis order. We now derive a simple formula for finding a good ordering of the array axes so that the average seek distance to retrieve an average rectangle in the access pattern reduces. We assume that the blocks of the array are laid out contiguously on disk and a cylinder is totally occupied before allocation of data on the next cylinder. The analysis below is relevant only if the disk is used exclusively for retrievals on the array data.

Consider an n dimensional array $[X_1, X_2 \ldots X_n]$ divided into chunks of shape $[c_1, c_2 \ldots c_n]$.

Lemma 1 *The number of tracks to seek on disk for an access request $(y_1, y_2 \ldots y_n)$ is at least*

$$\frac{(z_1 - 1)(d_2 d_3 \ldots d_n) + \ldots + (z_i - 1)(d_{i+1} \ldots d_{n-1} d_n) + \ldots + (z_{n-1} - 1)d_n + z_n}{B} \tag{2}$$

where $z_i = \lceil y_i / c_i \rceil$, $d_i = X_i / c_i$ (assuming c_i divides X_i exactly) and B is the number of blocks per cylinder on the disk.

Proof. Transform all indices to a new coordinate system where chunk $[c_1, c_2, \ldots c_n]$ is the basis element. In the new coordinate system the array dimension is $[X_1/c_1, X_2/c_2, \ldots X_n/c_n]$ which is equal to $[d_1, d_2, \ldots d_n]$ and the access request is $(\lceil y_1/c_1 \rceil, \lceil y_2/c_2 \rceil, \ldots \lceil y_n/c_n \rceil)$ which is equal to $(z_1, z_2, \ldots z_n)$. We now have an array $[d_1, d_2, \ldots d_n]$ with an access request $(z_1, z_2, \ldots z_n)$ on it. If the array is laid out linearly in the axis order $1, 2, \ldots n$, with n as the innermost axis, the number of blocks between the start block and the end block of the access rectangle is given by the numerator of (2). If each disk cylinder holds B blocks, the number of cylinders over which these blocks will span is given by (2). ∎

Lemma 2 *Given an access pattern, the value of expression* (2) *averaged over all elements of the access pattern is minimized for the order* $1, 2, \ldots n$ *(with n as the innermost axis) if*

$$\frac{a_1 - 1}{d_1 - 1} \leq \frac{a_2 - 1}{d_2 - 1} \leq \ldots \frac{a_n - 1}{d_n - 1}, \qquad d_i \neq 1$$

where $a_j = \sum_{i=1}^{K} A'_{ij} P_i$, $A'_{ij} = \lceil A_{ij}/c_j \rceil$ *and* $d_i = X_i/c_i$.

Proof. Substituting a_i for z_i in (2) gives the expression to be minimized. We only need to consider the numerator of this expression. Rewriting it with $a_i - 1$ replaced by x_i, $\forall i$ we get,

$$(((\ldots(x_1 d_2 + x_2)d_3 + \ldots)d_i + x_i)d_{i+1} + \ldots)d_j + x_j)\ldots + x_{n-1})d_n + x_n. \tag{3}$$

Interchanging positions of dimensions d_i and d_j $(i < j)$ gives,

$$((\ldots(x_1 d_2 + x_2)d_3 + \ldots)d_j + x_j)d_{i+1} + \ldots)d_i + x_i)\ldots + x_{n-1})d_n + x_n. \tag{4}$$

If (3) is minimal then (3) $\leq$ (4) which is true iff

$$((x_i d_{i+1} + x_{i+1})\ldots)d_j + x_j \leq ((x_j d_{i+1} + x_{i+1})\ldots)d_i + x_i \tag{5}$$

It can be easily proved by induction that (5) holds if,

$$\frac{x_i}{d_i - 1} \leq \frac{x_{i+1}}{d_{i+1} - 1} \leq \ldots \frac{x_j}{d_j - 1} \tag{6}$$

Extending this for any pair (i, j) such that $i < j$ and substituting $a_i - 1$ for x_i completes the proof for Lemma (2). ∎

To illustrate the advantage of re-ordering the array axes reconsider the example in Figure 2(b). Suppose the number of blocks per cylinder is 60. Then for the access pattern assumed for Figure 2, the average number of tracks to seek (from Lemma 1) is 67 for the array axis order (X_1, X_2, X_3). Using Lemma 2, if we reorder the axis as (X_1, X_3, X_2) the number of tracks to seek is reduced to 17.

2.3 Redundancy

Data layout using one chunk size minimizes *average* access cost, meaning it is efficient for some rectangles but inefficient for others. We propose maintaining redundant copies of the array which are organized differently to optimize for the various classes in the access pattern. Specifically, we divide the classes in the access pattern into as many partitions as there are proposed copies and optimize each copy for its associated partition. Hence, the first step is to find R partitions, where R is the number of copies, such that the cumulative access time for the queries in the classes of the access pattern is minimized. We can do this using one of the following two approaches:

- Use brute force to try all possible partitions and choose the best. In the worst scenario, the number of partitions to be considered is exponential in the number of elements in the access pattern.

- Use vector clustering techniques [LIN80] to group classes into clusters. We have a starting set of K classes and wish to divide them into R clusters. Initially, each class belongs to a different cluster and we progressively merge pairs of clusters with the minimal weighted distance between them until R clusters remain. Algorithms for computing minimal distance are given in [EQU89].

When a read request arrives for a replicated array, the runtime system first finds the replica with the smallest estimated access cost. The estimated cost is a weighted sum of the number of block fetches, seek distance and media switches (in case of tertiary devices). The least cost replica is then used to answer the query.

2.4 Partitioning

Tertiary memory devices are robo-line storage systems consisting of a large number of storage media (tapes or platters) and a few read-write drives. A robot arm switches the media between the shelves and drives in typically ten seconds. To improve performance the number of media switches required to access a requested rectangle should be minimized. The array should be partitioned such that the parts of the array accessed together frequently lie on the same media. We can extend the chunking methodology to deal with media switches by:

- making the size of the chunk a platter instead of a disk block.

- minimizing the number of platter switches instead of number of page fetches.

Partitioning can be used for minimizing the media switches for both disk and tape tertiary devices. However, for tapes the average seek time (45 seconds) is large compared to the switch time. Hence, minimizing media switches is less crucial than minimizing seek time.

3 Implementation in POSTGRES

POSTGRES [STO91b] is an extended relational database system being developed at Berkeley. We have built into POSTGRES a generalized interface for multidimensional arrays. POSTGRES is well-suited for handling massive amounts of data; it supports large objects that allow attributes to span multiple pages and it has a generalized storage structure that supports huge capacity storage devices as tertiary memory [STON93].

In our implementation, arrays are first class objects. Therefore any attribute of a class can be declared to be an array of any base type. The internal representation of arrays is a variable length structure with the following fields:

int	`array_size`
int	`ndim`
int_array	`dim`
int_array	`lbound`
int	`flags`
byte_array	`data`

In this structure, `array_size` is the total size of the array (data and meta-data); `ndim` is the number of dimensions of the array; `dim` and `lbound` are integer arrays of size `ndim` where the array `dim` stores the size of each dimension of the array and the array `lbound` stores the lower index of each dimension; `flags` is a bit mask that stores information about the array type. The contents of the `data` field depend on the kind of array stored and are described later.

Our implementation supports a variety of convenient features:

- Array elements can be stored in one of the following two formats depending on the total array size:
  - Store the array on the same page as the rest of the tuple. POSTGRES tuples cannot span pages, so the entire array must be smaller than the page size (currently 8KB). The `data` field in this case is used to store the array elements contiguously in their respective internal representation

- Store the array as a POSTGRES large object [MOS92] and keep a pointer to the large object in the `data` field of the array structure. The large object interface in POSTGRES provides a file-oriented access to data that span multiple pages. This implies that the only limit on the size of an array is the maximum object size ($\approx$ 17 Tbytes).

  A bit in the `flag` field of the array structure indicates the format used by an existing array.

- Arrays of both variable and fixed length base types are supported. If the array base type is of variable length, the actual data element is preceded by an integer that is the size of the data element. For example, the `data` field for an array of text {"abc", "xy"} will be stored as {3 a b c 2 x y }.

- Any sub-array of an array can be read by specifying the range of indices. For example, an access to a subarray starting at the fifth array element and ending at the ninth is posed as:

```
retrieve (rel1.a[5:9])
```

- The values stored in an array can be updated any time; it is not necessary to fill the entire array at creation time. At array creation time the user can simply specify the dimensions of the array and initialize it as an empty array. At any later time, a replace command can be used to assign values to any part of the array, as shown in the example below:

```
append (rel2.a[4][5] = "{}")
replace (rel2.a[1:2][3:4] = "{1,2,3,4}").
```

  A limitation in the current prototype is that arrays cannot grow in size after creation. As a consequence, updates on variable length base elements are not supported.

- Operators can be defined on arrays, so that arrays of the same base type can be compared for equality. For example,

```
retrieve (x = 1) where rel2.a = rel3.a
```

  returns `1` only if `rel2.a` and `rel3.a` are arrays of the same base type, dimension and have the same values.

The POSTQUEL query language has been extended to provide the necessary array interface.

Chunking

At array creation time, the user can specify whether the array should be chunked. If so, the user can either specify the access pattern or use the default chunking provided. The default chunking chooses the size of each axis of the chunk to be proportional to the length of corresponding array axis. If the access pattern is provided, the method in Section 2 is used for finding the optimal chunk shape. For example, the POSTQUEL query

```
append (a[100][100][50] = "input_array -chunk acc_pattern")
```

creates a 3 dimensional array for which the array elements are obtained from the file `input_array`. The `-chunk` flag specifies that the array should be chunked using the access pattern provided in the file `acc_pattern`. The input array is organized into chunks and the chunked array is stored as another large object. Since the array organization scheme used cannot work in-place, it is necessary to make a separate copy of the chunked file. A bit in the `flag` field is set to indicate that the array is chunked. The `data` field is arranged as a structure with the first field pointing to the newly created large object and the second field storing the chunk shape.

For automatic generation of the access pattern for an array we intend to augment POSTGRES with a user option whereby all read requests to an array will be monitored and access statistics collected. At a later time, the user may invoke the chunking algorithm which will use this collected statistics for the access pattern.

4 Performance

In this section we present the performance improvement provided by our organization techniques. Our experiments were done on a DECstation 5000/200 running Ultrix 4.2. Measurements were made on two different storage devices. The first set of results is for a local 1 GB magnetic disk using the Ultrix file system. The block size, C was set to 8 KB, which is the file system block size. A second set of results was taken from data stored on a write-once optical jukebox [SON89]; the tertiary storage device currently supported by POSTGRES [STON93]. The jukebox consists of 50 double sided platters, each of which has a 3.27 GB capacity per side. At any time a maximum of two platters can be physically mounted, and mounting a platter takes about ten seconds. A custom storage manager transfers data between disk and tertiary memory in units of 256 KB and hence block size is 256 KB.

To make our measurements realistic we considered arrays actually used by global change

benchmark #	array size	dimension	element size	storage media
data set 1	182.25 MB	[025 135 027 100 05]	4 bytes	magnetic disk
data set 2	342.75 MB	[112 180 170 020 05]	4 bytes	magnetic disk
data set 3	4.255 GB	[072 090 038 144 30]	4 bytes	tertiary memory
data set 4	4.255 GB	[114 360 180 024 06]	4 bytes	tertiary memory

Table 1: Benchmarks

scientists in the Sequoia project [STO91c]. The first source of data was ocean model output from the General Circulation Model (GCM) simulations done at UCLA [MEC92] [WEI93]. The arrays consist of three-dimensional snapshots of the ocean (covering the world or a region of it) taken at regular intervals of time with horizontal grid resolution varying from $\frac{1}{3}^\circ$ to 1°. For each point in the three dimensional space there are 5 model variables namely, temperature, salinity and three velocity components along the x, y and z direction in space. Hence the arrays have five dimensions: time, latitude, longitude, depth and the variables. The UCLA scientists currently store the array by a nested traversal of the array axes in the order time, latitude, longitude, depth and variables with time as the outermost axis.

The second data source was atmospheric output from the UCLA GCM. In this model, the entire earth (180° latitude by 360° longitude) is divided into regular grids with resolution varying from 1.25° to 5° for 9 to 57 horizontal layers of the atmosphere. For each point in the three dimensional grid, a collection of 38 variables are recorded at regularly spaced time steps. Thus, the output is another five dimensional array of time, elevation, latitude, longitude and an index of model variables. The UCLA scientists currently store the array by a nested traversal of the array axes in the order time, latitude, variables, longitude and elevation with time as the least rapidly varying dimension.

We selected four benchmark arrays from the two sources described above as summarized in Table 1. The third column indicates the number of values along each of the five array dimensions. Data sets 1, 2 and 4 are chosen from the ocean GCM and 3 from the atmosphere GCM. The first two benchmarks were studied on a local magnetic disk and the next two on a sony jukebox.

For each of the data sets, we obtained a collection of queries (10 to 20 in number) by consulting UCLA scientists. Some sample queries ran include:

- making surface plots of some variables over some portion of the total surface

- finding the mean or variance of a variable over time or elevation
- making cross-section plots of some variable over some region.

To study the performance improvement with the array organization techniques we performed the following measurements for each of the four data sets:

We first determined the optimal chunk shape for the user provided access pattern using the exhaustive search method discussed in Section 2.1. The time to find the optimal chunk size for all the four data sets took less than a minute. We organized the array into chunks and ran the benchmark queries on the chunked array. The total execution time, CPU time and the number of blocks fetched for executing the queries were recorded. Next, we reorganized the chunked array using the axis order specified by Lemma 2 and repeated the measurements using the same query set. Finally, we made two copies of the array as described in Section 2.3 and measured performance by executing each query on the array copy that has the smaller estimated cost.

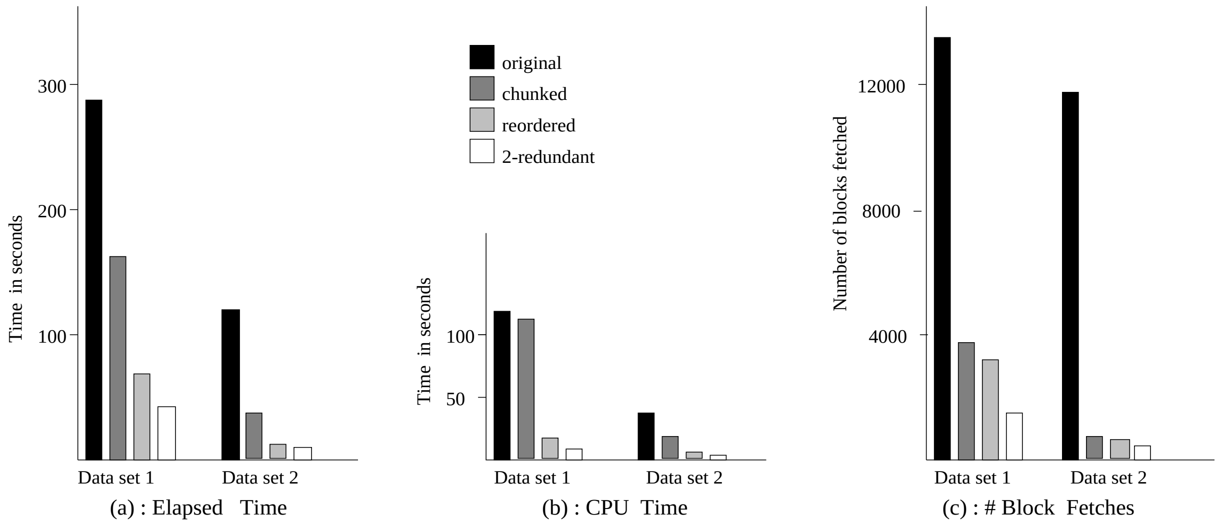

Figure 3: Performance measurements in POSTGRES

Figure 3 summarizes our measurements in POSTGRES for arrays stored on magnetic disk. For data set 1, chunking results in a 40% reduction in elapsed time. Reordering results in a further 60% reduction in elapsed time. Similar improvements are observed for data set 2. The number of blocks fetched by the file system drops even more dramatically with chunking; there is a factor of 4 and 13 reduction for data sets 1 and 2 respectively. Since both chunked and reordered arrays are organized using the same chunk shape, the number of blocks fetched for the two cases should theoretically be the same. In practice, a slight reduction (compare bars 2 and 3 in Figure 3(c)) is observed because of prefetching in the Ultrix file system. Prefetching works better for a reordered array since a greater fraction of accesses become sequential with reordering. 2-level redundancy yields a 27% reduction in elapsed time for data set 1 but for data set 2 redundancy does not provide much benefit.

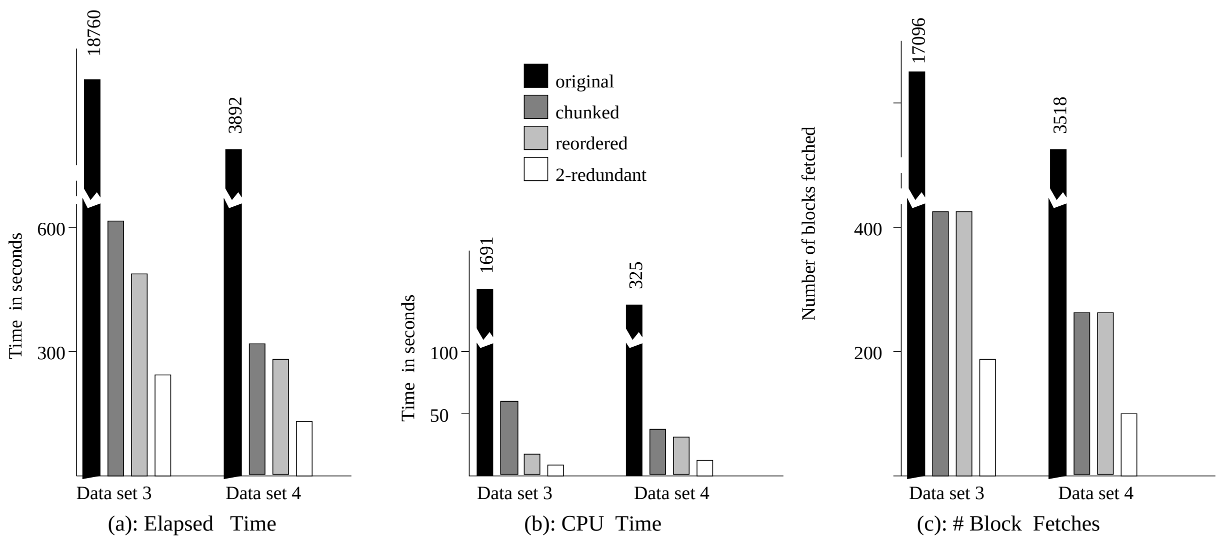

Figure 4: Performance measurements on Tertiary Memory data

Figure 4 shows the results of applying various organization schemes on data sets 3 and 4 stored on the optical jukebox. Comparison of bars 1 and 2 for data set 3 shows that queries on the unorganized data takes 5.2 hours to complete compared to 10.2 minutes on the chunked array. Similarly for data set 4 we observe a factor of 12 reduction in elapsed time. Reordering also works well and a 20% and 12% reduction in access times is achieved for data set 3 and 4 respectively. With 2-level redundancy the number of blocks fetched is lowered by another 60% and the access time by 50% as compared to the best single copy version for both data sets.

Effect of Access Pattern

In all of the optimization strategies discussed, the input access pattern has played a crucial role. To evaluate the role of the access pattern, we measured performance on arrays that are chunked without using any access pattern. Instead, each array is organized using a default chunk, each side of which is chosen to be proportional to the side in the original array. Figure 5 shows the difference in total execution time between an array chunked using a perfect access pattern and

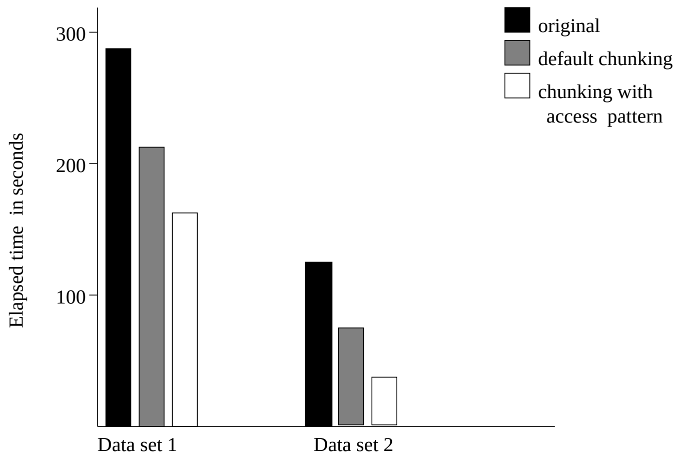

Figure 5: Performance of default chunking

an array chunked using the default strategy. From the figure it is clear that for both data sets there is at least a 40% improvement when perfect knowledge of the access pattern exists. On the other hand, compared to the original array the default chunking shows significant improvement. Hence even when no knowledge of the access pattern is available it is a good idea to do chunking.

5 Conclusion

In this paper, we presented a number of strategies for optimizing layout of large multidimensional arrays on secondary and tertiary memory devices. Based on a suitably captured access pattern, we used chunking of arrays to reduce the number of blocks fetched and reordering of array axes to reduce seek distance between accessed blocks. In cases where it is affordable, we suggested the use of redundancy to organize multiple copies of the same array based on different access patterns. Very often the size of the array is too large to be stored in conventional secondary storage media. In such cases, arrays must be migrated to large capacity tertiary storage devices that are slow and require different methods of optimization. We suggested partitioning as a method to reduce the media switch costs for such devices.

We extended the POSTGRES database system to support multidimensional arrays. Our implementation provides a generalized array interface that allows arrays of arbitrary size and dimension. Moreover, large array can be chunked (on the user's discretion) for fast processing of queries on such arrays.

These optimization techniques were tested for their effectiveness in reducing the enormous access time on large arrays. Towards this end, we collected data from real users of large multidimensional arrays. Our measurements based on their usage patterns showed significant reduction of access times with our optimization strategies.

6 Acknowledgements

Paul Aoki, Ginger Ogle, Mike Olson, Carol Paxson and Mark Sullivan deserve special thanks for going through initial drafts of the paper and providing useful feedback. Professor Tom Anderson suggested the idea of default chunking. Chung Chu Ma, Joseph Spahr and William Weibel helped form the benchmark data sets and provided the access pattern for the benchmark arrays.

References

[DOZ91] Jeff Dozier and H.K. Ramapriyan. Planning for the EOS Data and Information System. In *Global Environment Change*, volume 1. Springer-Verlag, Berlin, 1991.

[EQU89] William H. Equitz. A New Vector Quantization Clustering Algorithm. *IEEE Transactions on Accoustics, Speech and Signal Processing*, 37(10), 1989.

[FRA92] James Franklin. Tiled Virtual Memory for UNIX. In *Proceedings of USENIX, San Antonio,TX*, June 1992.

[JAI88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[LIN80] Yoseph Linde, Andres Buzo, and Robert Gray. An Algorithm for Vector Quantizer Design. *IEEE Transactions on Communications*, 28(1), 1980.

[MEC92] C. Mechoso et al. Parallelization and Distribution of a Coupled Atmosphere-Ocean General Circulation Model, 1992. Submitted to *Monthly Weather Review*, Aug 1992.

[MOS92] Claire Mosher. POSTGRES Reference Manual, version 4.0. Electronics Research Laboratory, University of California, Berkeley, CA-94720, 1992. No. UCB/ERL M92/85.

[REU80] J.L. Reuss, S.K. Chang, and B.H. McCormick. Picture Paging for Efficient Image Processing. In S.K. Chang and K.S. Fu, editors, *Pictorial Information Systems*, Springer-Verlag, 1980.

[ROS75] Arnold L. Rosenberg. Preserving Proximity in Arrays. *SIAM Journal on Computing*, 4, 1975.

[SON89] Sony Corporation, Japan. *Writable Disk Auto Changer WDA-610 Specifications and Operating Instructions*, 1989. 3-751-106-21(1).

[STO91a] Michael Stonebraker and Jeff Dozier. Large Capacity Object Servers to Support Global Change Research. Technical Report 91/1, University of California at Berkeley, 1991.

[STO91b] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next Generation Database Management System. *Communications of the ACM*, 34 (10), 1991.

[STO91c] Michael Stonebraker. An Overview of the Sequoia 2000 Project. Technical Report 91/5, University of California at Berkeley, 1991.

[STON93] Michael Stonebraker and Michael Olson. Large Object Support in POSTGRES. Proc. 9th Intl. Conf. on Data Engineering, April 1993, Vienna, Austria.

[WAD84] B.T Wada. A Virtual Memory System for Picture Processing. *Communications of the ACM*, 27, 1984.

[WEI93] William Weibel, Chung Chu Ma and Joseph Spahr. Personal Communication.