A Storage Scheme for Multi-dimensional Databases Using Extendible Array Files

Ekow J. Otoo and **Doron Rotem**

Lawrence Berkeley National Laboratory
1 Cyclotron Road
University of California
Berkeley, CA 94720, USA
ekw,@hpcrd.lbl.gov,d_rotem@lbl.gov

Abstract

Large scale scientific datasets are generally modeled as k-dimensional arrays, since this model is amenable to the form of analyses and visualization of the scientific phenomenon often investigated. In recent years, organizations have adopted the use of on-line analytical processing (OLAP), methods and statistical analyses to make strategic business decisions using enterprise data that are modeled as multi-dimensional arrays as well. In both of these domains, the datasets have the propensity to gradually grow, reaching orders of terabytes. However, the storage schemes used for these arrays correspond to those where the array elements are allocated in a sequence of consecutive locations according to an ordering of array mapping functions that map k-dimensional indices one-to-one onto the linear locations. Such schemes limit the degree of extendibility of the array to one dimension only. We present a method of allocating storage for the elements of a dense multidimensional extendible array such that the bounds on the indices of the respective dimensions can be arbitrarily extended without reorganizing previously allocated elements. We give a mapping function $\mathcal{F}_*()$, and its inverse $\mathcal{F}_*^{-1}()$, for computing the linear address of an array element given its k-dimensional index. The technique adopts the mapping function, for realizing an extendible array with arbitrary extendibility in main memory, to implement such array files. We show how the extendible array file implementation gives an efficient storage scheme for both scientific and OLAP multi-dimensional datasets that are allowed to incrementally grow without incurring the prohibitive costs of reorganizations.

Proceedings of the third Workshop on STDBM
Seoul, Korea, September 11, 2006

1 Introduction

Datasets used in large scale scientific applications, are generally modeled as multidimensional arrays and matrices. Matrices are 2-dimensional rectangular array of elements (or entries) laid out in rows and columns. Although the logical view of a rectangular array of elements need not be the same as the actual physical storage, the elements of the arrays are mapped into storage locations according to some linearization function of the indices. An array file is simply a file of the elements of an array in which the k-dimensional indices of the elements map into linear consecutive record locations in the file. The mapping of k-dimensional indices to linear addresses may either be by a computed access function, an indexing mechanism, or a mixture of both. Operations on these array files involve accessing and manipulating sub-arrays (or array chunks) of one or more of these array files. Such models for manipulating datasets are typical in large scale computing performed by various Scientific and Engineering Simulations, Climate Modeling, High Energy and Nuclear Physics, Astrophysics, Computational Fluid Dynamics, Scientific Visualization, etc.

On-line analytical processing (OLAP) and decision support systems depend highly on efficiently computing statistical summary information from multi-dimensional databases. A view of the dataset as a multi-dimensional array model forms the basis for efficient derivation of summary information. The literature distinguishes OLAP on *relational model* of data called *ROLAP* and that on multi-dimensional model termed *MOLAP*.

Consider a dataset that maintains monitored values of *temperature* at defined locations given by the *latitude* and *longitude* at various time-steps. Figure 1 illustrates a simple multi-dimensional view of a sample data as a 3-dimensional array A[4][3][3] with assigned ordinal coordinates. The entries shown in the cells correspond to the linear addresses of the cells as the relative displacements from cell $A\langle 0,0,0\rangle$.

Each cell stores the temperature value (not shown) as the *measure*. Suppose we wish to maintain information on sales of some enterprise that has stores at different locations. The locations are defined by their spatial coordinates of *latitude* and *longitude* often abbreviated as *Lat/Long*. The sales information recorded for say three consecutive months can be represented by Figure 1 which gives the same view of the data as before but with values for sales as the *measures*. A multi-dimensional array correctly models the dataset in both domains of scientific applications and MOLAP. Further common characteristics associated with multi-dimensional model of data under these domains are that:

1. The data incrementally grow over time by appending new data elements. These may reach orders of terabytes, as in data warehousing.
2. The datasets are mainly read-only. However, they may be subject to expansions in the bounds of the dimensions, e.g., as new monitoring locations are enabled or new stores are opened.
3. The number of dimensions (i.e., the *rank*) of the array may be extended.
4. The array representation can be either dense or sparse.

In the illustration of Figure 1, the array may grow by appending data for new time-steps. When new locations are added, the bounds of the Lat/Long dimensions may be required to be extended. The mapping function depicted in the above figure (represented by the label inside each cell) corresponds to that of conventional array mapping that allows extendibility in one dimension only; namely the dimension in the mapping scheme that is least varying. We desire an array addressing scheme that allows extensions in both the bound and rank of the array file and also efficiently manages sparse arrays. In this paper we address primarily the storage scheme for managing dense extendible array efficiently. We discuss how the technique is tailored for handling sparse arrays as well without detailed experimentation due to space limitation.

Over the years special file formats have been extensively studied and developed for storing multi-dimensional array files that support sub-array accesses in high performance computing. These include NetCDF [10], HDF5 [7] and disk resident array (DRA) [11]. DRA is the persistent counterpart of the distributed main memory array structure called Global Array [12]. Except for HDF5, these array files allow for extendibility only in one dimension. We say an array file is extendible if the index range of any dimension can be extended by appending to the storage of previously allocated elements, so that new elements can be addressed from the newly adjoined index range. The address calculation is done without relocating elements of the previously allocated elements and without modifying the addressing function.

Let $A[\mathbb{N}_0][\mathbb{N}_1]\ldots[\mathbb{N}_{k-1}]$, denote a k-dimensional array where $\mathbb{N}_j, 0 \leq j < k-1$, is the bound on the indices of dimension j. An element, denoted by $A\langle i_0, i_1 \ldots, i_{k-1}\rangle$, is referenced by a k-dimensional index $\langle i_0, i_1 \ldots, i_{k-1}\rangle$. Let $\mathcal{L} = \{\ell_0, \ell_1 \ldots, \ell_{\mathbb{M}-1}\}$, be a sequence of consecutive storage locations where $\mathbb{M} = \prod_{j=0}^{k-1} \mathbb{N}_j$. An allocation (or mapping) function $\mathcal{F}()$, maps the k-dimensional indices one-to-one, onto the sequence of consecutive indices $\{0, 1, \ldots, \mathbb{M}-1\}$, i.e., $\mathcal{F} : \mathbb{N}_0 \times \mathbb{N}_1 \times \cdots \times \mathbb{N}_{k-1} \rightarrow \{0, 1, \ldots, \mathbb{M}-1\}$. Given a location index j, the inverse mapping function $\mathcal{F}^{-1}(j)$, computes the k-dimensional index $\langle i_0, i_1 \ldots, i_{k-1}\rangle$ that corresponds to j. The significance of inverse mapping functions have not been much appreciated in the past but has important use in computing the addresses of the neighbors of an element given its linear storage address and also for managing a general k-dimensional sparse array as opposed to sparse matrices which is 2-dimensional.

Modern programming languages provide native support for multidimensional arrays. The mapping function $\mathcal{F}()$ is normally defined so that elements are allocated either in *row-major* or *column major* order. We refer to arrays whose elements are allocated in this manner as conventional arrays. Conventional arrays limit their growth to only one dimension. We use the term *row-major order* in a general sense, beyond row-and-column matrices, to mean an order in which the leftmost index of a k-dimensional index is the least varying. This is also sometimes referred to as the *lexicographic order*. A *column-major order* refers to an ordering in which the rightmost index varies the slowest.

In conventional k-dimensional arrays, efficient computation of the mapping function is carried out with the aid of a vector that holds k multiplicative coefficients of the respective indices. Consequently, an N-element array in k-dimensions, can be maintained in $O(N)$ storage locations and the computation of the linear address of an element, given its k-dimensional coordinate index, is done in time $O(k)$ since this is done by $k-1$ multiplications and $k-1$ additions. We achieve similar efficiency in the management of extendible arrays by maintaining vectors of multiplying coefficients each time the array expands. The computation of the linear address, corresponding to a k-dimensional index, uses these stored vectors of coefficients. The vectors of multiplicative coefficients maintain records of the history of the expansions and are organized into *Axial-Vectors*. There is one *Axial-Vector* for each dimension and holds entries that we refer to as *expansion records*. The fields of an expansion record are described later.

Consider a k-dimensional array of N elements and after some arbitrary extensions dimension j maintains $\mathcal{E}_j$ records of the history of expansion for dimension j. Our approaches computes the linear address from the k-dimensional index in time $O(k + \sum_{j=0}^{k-1} \mathcal{E}_j)$ using $O(k \sum_{j=0}^{k-1} \mathcal{E}_j)$ additional space.

The technique being presented in this paper can be adopted by compiler developers for generating mapping functions for dense multidimensional extendible arrays in memory as well. The Standard Template Library (STL) in C++ also provides support for resizable vectors. However, the general technique for allowing array extensions without the extensive cost of reallocating storage is what we desire. The approach we propose may be used in implementing special libraries such as the Global Array (GA) library [12] and the disk resident array (DRA) [11] that manage dense multidimensional arrays.

To handle sparse arrays, we adopt the technique of *array chunking* [4, 7, 5], where the linear address of a chunk is computed as for a dense array. The generated address forms the key of an array chunk that is used in an index scheme that stores only non-empty chunks. Partial chunks can be further compressed and grouped into physical buckets to maintain efficient storage utilization. A thorough discussion of our approach is in a follow-up paper.

The main contributions in this paper are:

- A definition of a mapping function for extendible arrays that is applicable for both dense in-core arrays and out-of-core arrays. We call the approach the *The Axial-Vector* method. The general principle is not new. The idea was first proposed with the use of an *auxiliary array*. However, the storage overhead using an auxiliary array could be prohibitive. The new *Axial-Vector* method obviates the storage overhead and also gives a new method for computing the linear addresses.
- When extremely large array files are generated and stored, any dimension can still be expanded by appending new array elements without the need to reorganize the already allocated storage which could be of the order of terabytes.
- The mapping function proposed in this paper incurs very little storage overhead and can be used as a replacement for the access function for array files such as *NetCDF* and the data chunks in the *HDF5* file format.
- We discuss an extension of the technique to handle sparse extendible arrays for both in-core and out-of-core arrays.

The organization of this paper is as follows. In the next section we present some details of the definition of the mapping function for dense multidimensional extendible arrays in main memory since the same mapping function is adopted for accessing elements of extendible array files. In section 3 we describe how an array file is implemented. We describe how sparse multidimensional array files are managed in section 4. We give some experimental comparison of the array mapping functions for extendible and conventional arrays in section 5. We conclude in section 6 and give some directions for our future work.

2 The Mapping Function for an Extendible Array

2.1 Addressing Function for a Conventional Array

First we explore some details of how conventional arrays are mapped onto consecutive storage locations in memory. Consider a k-dimensional array $A[\mathbb{N}_0][\mathbb{N}_1]\dots[\mathbb{N}_{k-1}]$, where $\mathbb{N}_j, 0 \le j < k-1$, denote the bounds of the index ranges of the respective dimensions. Suppose the elements of this array are allocated in the linear consecutive storage locations $\mathcal{L} = \{\ell_0, \ell_1 \dots, \ell_{\mathbb{M}-1}\}$, in row-major order according to the mapping function $\mathcal{F}()$. In the rest of this paper, we will always assume row-major ordering and for simplicity we will assume an array element occupies a unit of storage. A unit of storage could be one word of 4 bytes, a double word of 8 bytes, etc. An element $A\langle i_0, i_1, \dots i_{k-1}\rangle$ is assigned to location ℓ_q, where q is computed by the mapping function defined as

$$q = \mathcal{F}(\langle i_0, i_1, \dots i_{k-1}\rangle) = i_0 * C_0 + i_1 * C_1 + \dots + i_{k-1} * C_{k-1}$$
$$\text{where} \quad C_j = \prod_{r=j+1}^{k-1} \mathbb{N}_r, 0 \le j \le k-1. \tag{1}$$

and $A\langle 0, 0, \dots 0\rangle$ assigned to ℓ_0.

In most programming languages, since the bounds of the arrays are known at compilation time, the coefficients $C_0, C_1, \dots, C_{k-1}$ are computed and stored during code generation. Consequently, given any k-dimensional index, the computation of the corresponding linear address using Equation 1, takes time $O(k)$.

Suppose we know the linear address q of an array element, the k-dimensional index $\langle i_0, i_1, \dots i_{k-1}\rangle$ corresponding to q can be computed by repeated modulus arithmetic with the coefficients $C_{k-2}, C_{k-3}, \dots, C_1$ in turn, i.e., $\mathcal{F}^{-1}(q) \rightarrow \langle i_0, i_1, \dots i_{k-1}\rangle$. When allocating elements of a dense multidimensional array in a file, the same mapping function is used where the linear address q gives the displacement relative to the location of the first element in the file, in units of the size of the array elements. The limitation imposed by $\mathcal{F}$ is that the array can only be extended along dimension 0 since the evaluation of the function does not involve the bound of $\mathbb{N}_0$.

We can still circumvent the constraint imposed by $\mathcal{F}$ by shuffling the order of the indices whenever the array is extended along any dimension. The idea of extending the index range of a dimension is simply to adjoin a block (or a hyperslab) of array elements whose sizes on all dimensions remain the same except for the dimension being extended.

2.2 The Mapping Function for an In-Core Extendible Array

The question of organizing an extendible array in memory, such that the linear addresses can be computed in the manner similar to those of a static array described above, is a long standing one [15]. Some solutions exist for extendible arrays that grow to maintain some predefined shapes [14, 15]. A solution was proposed that uses an auxiliary array [13] to keep track of the information needed to compute the mapping function. In [16], a similar solution was proposed that organized the content of the auxiliary array with a B-Tree. The use of auxiliary arrays can be prohibitively expensive in storage depending on the pattern of expansions of the array. We present a new approach to organizing the content of the auxiliary array with the use of *Axial-Vectors*. The idea of using axial-vectors to replace the auxiliary array was introduced in [20] but only for 2-dimensional arrays. The method introduced maintains the same information as in the auxiliary-array approach and does not generalize easily to k-dimensional arrays. Furthermore since it requires that information be stored for each index of any dimension, the method incurs the same prohibitive cost for certain array shapes. The method introduced in this paper avoids these problems. First we show how an in-core extendible array is organized with the aid of axial-vectors since the same mapping function is used for array files.

2.3 The Axial-Vector Approach for Extendible Arrays

Consider Figure 2a that shows a map of the storage allocation of 3-dimensional extendible array. We denote this in general as $A[\mathbb{N}_0^*][\mathbb{N}_1^*][\mathbb{N}_2^*]$, where $\mathbb{N}_j^*$ represents the fact that the bound has the propensity to grow. In this paper we address only the problem of allowing extendibility in the array bounds but not its rank. The labels shown in the array cells represent the linear addresses of the respective elements, as a displacement from the location of the first element.

Suppose initially the array is allocated as $A[4][3][1]$, where the corresponding axes of Latitude, Longitude and Time have the instantaneous respective bounds of $\mathbb{N}_0^* = 4, \mathbb{N}_1^* = 3$ and $\mathbb{N}_3^* = 1$. The array was extended by one time-step followed by another time-step. The sequence of the two consecutive extensions along the same time dimension, although occurring at two different instances, is considered as an *uninterrupted extension* of the time dimension. Repeated extensions of the same dimension, with no intervening extension of a different dimension, is referred to as an interrupted extension and is handled by only one expansion record entry in the axial-vector.

The labels shown in the array cells represent the linear addresses of the respective elements. For example, in Figure 2a the element $A\langle 2,1,0\rangle$ is assigned to location 7 and element $A\langle 3,1,2\rangle$ is assigned to location 34. The array was subsequently extended along the longitude axis by one index, then along the latitude axis by 2 indices and then along the time axis by one time-step. A *hyperslab* of array elements can be perceived as an array *chunk* (a term used in [17, 7]), where all but one of the dimensions of a *chunk* take the maximum bounds of their respective dimensions.

Consider now that we have a k-dimensional extendible array $A[\mathbb{N}_0^*][\mathbb{N}_1^*]\dots[\mathbb{N}_{k-1}^*]$, for which dimension l is extended by λ_l, so that the index range increases from $\mathbb{N}_l^*$ to $\mathbb{N}_l^* + \lambda_l$. The strategy is to allocate a *hyperslab* of array elements such that addresses within the *hyperslab* are computed as displacements from the location of the first element of the *hyperslab*. Let the first element of a hyperslab of dimension l be denoted by $A\langle 0,0,\dots,\mathbb{N}_l^*,\dots,0\rangle$. Address calculation is computed in row-major order as before, except that now dimension l is the least varying dimension in the allocation scheme but all other dimensions retain their relative order. Denote the location of $A\langle 0,0,\dots,\mathbb{N}_l^*,\dots,0\rangle$ as $\ell_{\mathbb{M}_l^*}$ where $\mathbb{M}_l^* = \prod_{r=0}^{k-1}(\mathbb{N}_r^*)$. Then the desired mapping function $\mathcal{F}_*()$ that computes the address q^* of a new element $A\langle i_0,i_1,\dots i_{k-1}\rangle$ during the allocation is given by:

$$q^* = \mathcal{F}_*(\langle i_0,i_1,\dots i_{k-1}\rangle) = \mathbb{M}_l^* + (i_l - \mathbb{N}_l^*)C_l^* + \sum_{\substack{j=0\\ j\neq l}}^{k-1} i_j C_j^*$$

$$\text{where}\quad C_l^* = \prod_{\substack{j=0\\ j\neq l}}^{k-1}\mathbb{N}_j^* \quad\text{and}\quad C_j^* = \prod_{\substack{r=j+1\\ r\neq l}}^{k-1}\mathbb{N}_r^* \tag{2}$$

We need to retain for dimension l the values of $\mathbb{M}_l^*$ - the location of the first element of the hyperslab, $\mathbb{N}_l^*$ - the fisrt index of the adjoined hyperslab, and $C_r^*, 0 \le r < k$ - the multiplicative coefficients, in some data structure so that these can be easily retrieved for computating an element's address within the adjoined hyperslab. The *axial-vectors* denoted by $\Gamma_j[\mathcal{E}_j], 0 \le j < k$, and shown in Figure 2b, are used to retain the required information. $\mathcal{E}_j$ is the number of stored records for axial-vector Γ_j. Note that the number of elements in each axial-vector is always less than or equal to the number of indices of the corresponding dimension. It is exactly the number of uninterrupted expansions. In the example of Figure 2b, $\mathcal{E}_0 = 2, \mathcal{E}_1 = 2$, and $\mathcal{E}_2 = 3$.

The information of each expansion record of a dimension is a record comprised of four fields. For dimension l, the i^{th} entry denoted by $\Gamma_l\langle i\rangle$ consists of $\Gamma_l\langle i\rangle.\mathbb{N}_l^*$; $\Gamma_l\langle i\rangle.\mathbb{M}_l^*$; $\Gamma_l\langle i\rangle.C[k]$ - the stored multiplying coefficients for computing the displacement values within the hyperslab; and $\Gamma_l\langle i\rangle.S_{i_l}$ - the memory address where the hyperslab is stored. Note however that for computing record addresses of array files, this last field is not required, since new records are always allocated by appending to the existing array file. In main memory an extendible array may be formed as a collection of disjoint hyperslabs since a block of memory acquired for each new hyperslab may not necessarily be contiguous to a previously allocated one. Contiguiety in memory allocation is only guranteed for unintrrupted expansion of a dimension, i.e., when the same dimension is repeatedly expanded.

Given a k-dimensional index $\langle i_0,i_1,\dots,i_{k-1}\rangle$, the main idea in correctly computing the linear address is in determining which of the records $\Gamma_0\langle z_0\rangle,\Gamma_1\langle z_1\rangle\dots\Gamma_{k-1}\langle z_{k-1}\rangle$, has the first maximum starting address of its hyperslab. The index z_j is given by a modified binary search algorithm that always gives the highest index of the axial-vector where the expansion record has a maximum starting address of its hyperslab less than or equal to i_j.

For example, suppose we desire the linear address of the element $A\langle 4,2,2\rangle$, we first note that $z_0 = 1, z_1 = 0$, and

Array Method	Address calculation	Inverse addr. calculation	Storage overhead
Conventional	$O(k)$	$O(k)$	0
Axial Vectors	$O(k+$ $k\log(k+\mathcal{E})$ $-k\log k)$	$O(k+\log\mathcal{E})$	$O(k\mathcal{E})$

Table 1: Summary of features of extendible array realization.

$z_2 = 1$. We then determine that

$$\begin{aligned} \mathbb{M}^*_l &= \max(\Gamma_0\langle 1\rangle.\mathbb{M}^*_0, \Gamma_1\langle 0\rangle.\mathbb{M}^*_1, \Gamma_2\langle 1\rangle.\mathbb{M}^*_2) \\ &= \max(48, -1, 12); \end{aligned} \tag{3}$$

from which we deduce that $\mathbb{M}^*_l = 48, l = 0,$ and $\mathbb{N}^*_l = \mathbb{N}^*_0 = 4$. The computation $\mathcal{F}_*(\langle 4,2,2\rangle) = 48 + 12 \times (4-4) + 3 \times 2 + 1 \times 2 = 48 + 0 + 6 + 2 = 56$. The value 56 is the linear address relative to the starting address of 0. The main characteristics of the extendible array realization is summarized in the following theorem

Theorem 2.1. *Suppose that in a k-dimensional extendible array, dimension j undergoes $\mathcal{E}_j$, uninterrupted expansions. Then if $\mathcal{E} = \sum_{j=0}^{k-1}\mathcal{E}_j$, the complexity of computing the function $\mathcal{F}_*()$ for an extendible array using axial-vectors is $O(k + k\log(k+\mathcal{E}) - k\log k)$, using $O(k\mathcal{E})$ worst case space.*

Proof. The worst case sizes of the axial-vectors occur if each dimension has the same number of uninterrupted expansions; i.e., $\mathcal{E}_j = \mathcal{E}/k$. The evaluation of $\mathcal{F}_*()$ involves $k \log\mathcal{E}_j$ followed by k multiplications k additions and 1 subtraction, giving a total of $O(k + k(\log(1+\mathcal{E}/k))) = O(k + k\log(k+\mathcal{E}) - k\log k)$.

The additional space requirement for the k axial-vectors is $O((k+3)\sum_{j=0}^{k-1}\mathcal{E}_j) = O(k\mathcal{E})$. □

Given a linear address q^*, it is easy to find, an entry in the axial vectors that has the maximum starting address less than or equal to q^* using a modified binary search algorithm as in the address computation. The repeated modulus arithmetic as described in section 2.1 is then used to extract the k-dimensional indices. We state without a formal proof the following.

Theorem 2.2. *Given a linear address q of an element of an extendible array realized with the aid of k axial-vectors, the the k-dimensional index of an element is computable by the function $\mathcal{F}_*^{-1}$ in time $O(k + \log\mathcal{E})$.*

The main results of the extendible array realization are summarized in Table 1.

3 Managing Multidimensional Extendible Array Files

An array file is simply a persistent storage of the corresponding main memory resident array in a file, augmented with some meta-data information, either as a header in the same file or in a separated file. We will consider array files as formed in pairs: the primary file F_p and the meta-data file F_m. For extremely large files, the content in memory at any time is a subset, or a subarray of the entire disk resident array file. Scientific applications that process these arrays consider array chunks as the unit of data access from secondary storage. Most high performance computations are executed as a parallel program either on a cluster of workstations or on massively parallel machines. The model of the data is a large global array from which subarrays are accessed into individual workstations. The subarrays of individual nodes together constitute tiles of the global array. Actual physical access of array elements is carried out in units of array chunks for both dense and sparse arrays. We discuss this in some detail in the next section.

For the subsequent discussions, we will ignore most of the details of the organization of the array files and also the details of the structure of the array elements. For simplicity, we consider the array files as being composed of fixed size elements, where each element is composed of a fixed number of attributes. Each attribute value has a corresponding ordinal number that serves as the index value. Mapping of attribute values to ordinal numbers is easily done for array files. The primary file F_p, contains elements of the multidimensional array that continuously grows by appending new data elements whenever a dimension is extended. For example, a dimension corresponding to time may be extended by the addition of data from new time-steps. A meta-data file F_m stores the records that correspond to the axial vectors. The contents of the meta-data file F_m are used to reconstruct the memory resident axial vectors. Each expansion of a dimension results in an update of an axial vector and consequently the meta-data file as well.

Suppose an application has already constructed the memory resident axial vectors from the meta-data file, then the linear address of an element (i.e., a record), of the array file given its k-dimensional coordinates, is computed using the mapping function $\mathcal{F}_*()$. Essentially, $\mathcal{F}_*()$ serves as a hash function for the elements of the array file. Conversely, if the linear address q^*, of an element is given and one desires the neighbor element that lies some units of coordinate distances along some specified dimensions from the current, such an element can be easily retrieved with the aid of the inverse function $\mathcal{F}_*^{-1}()$. First we compute the k-dimensional coordinate values from $\mathcal{F}_*^{-1}(q)$, adjust the coordinate values along the specified dimensions and compute the address of the desired element by $\mathcal{F}_*()$. The relevant algorithms for these operations are easily derived from the definitions of the functions presented in the preceding sections. One of the main features of our scheme is that when extremely large array files are generated, each dimensions can still be expanded by appending new array elements without the need to reorganize the allocated storage of the files.

Two popular data organization schemes for large scale scientific datasets are *NetCDF* [10] and *HDF5* [7]. The *NetCDF* maintains essentially an array file according to a row-major ordering of the elements of a conventional array. Consequently, the array can only be extended in one dimension. The technique presented in this paper can be easily adopted as the mapping function of the *NetCDF* storage scheme to allow for arbitrary extensions of the dimensions of the NetCDF file structure, without incurring any additional access cost. *HDF5* is a storage scheme that allows array elements to be partitioned in fixed size sub-arrays called *data-chunks*. A chunk is physically allocated on secondary storage and accessed via a B^+-tree index. Chunking allows for extendibility of the array along any dimension and also for the hierarchical organization of the array elements, i.e., elements of a top level array is allowed to be an array of a refined higher resolution and so on. The mapping function introduced can be used as a replacement of the B^+-tree indexing scheme for the *HDF5* array chunks. Other applications of the mapping function introduced here include its use for the *Global-Arrays* [12] data organization and *Disk-Resident-Array* files [11].

4 Managing Sparse Extendible Array Files

Besides the characteristics that multi-dimensional databases incrementally grow over time, they also have the unkind property of being sparse. Techniques for managing large scale storage of multi-dimensional data have either addressed the sparsity problem of the array model [2, 3, 4, 5, 17, 18] or the extendibility problem [16, 19] but not both simultaneously. The sparsity of multidimensional array is managed by array *chunking* technique. An array chunk is defined as a block of data that contains extents of all dimensions from a global multi-dimensional array. Even for dense array, an array chunk constitutes the unit of transfers between main memory and secondary storage.

Figure 3a shows a 3-dimensional array partitioned into chunks using index-intervals of 3. Addressing elements of the array is computed in two levels. The first level address computation gives the chunk address of the element. The second level address is computed as the displacement of the array element within the array chunk. The extendible array addressing method maps the k-dimensional index of each chunk into a *Table Map*. The table map of the chunks contains pointers to the physical addresses of the data blocks that hold chunks. An array chunk forms the unit of data transfer between secondary storage and main memory. A table map pointer is set to *null* if the chunk holds no array elements. As in extendible hashing, the use of table map to address chunks guarantees at most 2 disk accesses to locate a chunk. Array chunks that have less than some defined number of array elements can be compressed further.

The table map is one of the simplest techniques for handling sparse extendible array but suffers from the problem of a potential exponential growth for high dimensional datasets. Instead of a table map for maintaining array chunks, we use a dynamically constructed PATRICIA trie [9] to manage the table map of the chunks. Other methods of handling sparse multi-dimensional arrays have been described in [4, 8]. Some of the techniques for handling sparse matrices [1] can also be adopted.

4.1 Alternative Methods for Addressing Array Chunks

The use of a table map for locating the physical locations of array chunks relaxes the need for directly addressing array elements with an extendible array mapping function. One can further relax this constraint by doing away with the extendible array mapping function entirely. Rather, a method for constructing a unique address $I_{\langle i_0,\dots i_{k_0}\rangle}$ of an array chunk from the k-dimensional index $\langle i_0,\dots i_{k_0}\rangle$ is all that is required. One such method is given by concatenating the binary representation of the coordinate indices of an array chunk. The unique address generated is then used in an index scheme such as a B^+-tree, to locate the physical chunk where an array element is stored. This approach is actually implemented in the HDF5 storage format. There are two problems with this approach;

1. Either the k-dimensional index or the generated identifier for the chunk must be stored with the chunk. For the latter case, an inverse function for computing the k-dimensional chunk index from the chunk identifier is needed but is less space consuming.
2. It does not handle both memory resident array and disk resident arrays uniformly.

In general the table map can be replaced with any indexing scheme that maintains $O(N)$ chunk address for exactly N non-empty chunks. The use of a PATRICIA trie guarantees that. Using extendible array mapping function for computing the linear address of a chunk has the advantage of:

1. giving us a uniform manner of managing extendible arrays resident both in-core and out-of-core.
2. allowing the replacement of the global k-dimensional address of an array element by one which only defines its linear location within an array chunk and yet enables us to compute the global k-dimensional index from the linear address.

4.2 Operations on Multidimensional Array Files

Besides the creation of the multi-dimensional array files, and operations for reading, writing (i.e., accessing) and appending new element, the application domain dictates the type of operations the array files are subjected to. While multi-dimensional OLAP applications see the efficient computation of the *Cube* operator [6, 21] as a significant operation, applications in other scientific domains require efficient extractions of sub-arrays for analysis and subsequent visualization. In both domains, efficiently accessing elements of a sub-array is vital to all computations carried out.

The mapping function provides direct addressing for array chunks and subsequently to the array elements. A naive approach to performing sub-array extraction operation would be to iterate over the k-dimensional coordinates of the elements to be selected and retrieve each array elements independently. Suppose the cardinality of the response set of the first selection class is $\Re$. The naive approach performs $\Re$ independent disk accesses. But one can do better than this worst case number of disk accesses. An efficient method for processing any of the above queries is to compute the chunk identifies of the element; and for each chunk retrieved, extract all elements the chunk that satisfies the request. Due to space limitation, detailed discussions on the structures, algorithms and experiments on extendible sparse multidimensional arrays is left out in this paper.

5 Performance of Extendible Array Files

The theoretical analysis of the mapping function for extendible arrays indicates that it is nearly of the same order of computational complexity as that of conventional arrays. The main difference being the additonal time requied by the mapping function for an extendible array to perform binary searches in the axial-vectors. We experimentally tested this by computing the average access times of both the conventional array and extendible array for an array size of approximately 10^8 elements of double data types. We varied the rank of the array from 2 to 8 while keeping the size of the array about the same. We plotted the average time over 10000 random element access for static arrays. These experiments were run on a 1.5GHz AMD Athlon processor running Centos-4 Linux with 2GByte memory. Figure 4a show the graphs of the access times averaged over 10000 element access.

The graphs indicate that for static arrays, the cost of computing the extendible array access function varies more significantly with the rank of the array than for the conventional array access function. Even though the complexity of computing the access functions are both $O(k)$, the extendible array access function shows strong dependence on the array's rank k due to the fact that k binary searches are done in the axial-vectors.

We also considered the impact on the average access cost when the arrays undergo interleaved expansions. The experiment considered arrays of ranks 2 and 3 where the initial array size grew from about 10000 elements to about 10^6 elements. For each sequence of about 10000 accesses the array is allowed to undergo up to 16 expansions. A dimension selected at random, is extended by a random integer amount of between 1 and 10. Each time the conventional array is extended, the storage allocation is reorganized. The average cost of accessing array elements with interleaved expansions is shown in Figure 4b. Under this model, we find that the average cost of accessing array elements for extendible arrays is significantly less than for conventional arrays that incur the additional cost of reorganization.

Similar experiments were conducted, for accessing elements of array files instead of memory resident arrays. Figure 5a compares the average time to access elements for 2, 3 and 4 dimensional static files. There is very little variation in the times of the conventional and extendible array functions. However, when these times are computed with interleaved expansions, the extendible array clearly outperforms the conventional array methods by several orders of magnitude. Figure 5b shows the graphs for $2,3$ and 4-dimensional extendible array files only. The extra time and storage required to reorganize the conventional array files with interleaved expansions become prohibitive as the array becomes large. One can infer from these results that, in handling large scale dense multi-dimensional dataset that incrementally grow by appending elements, the extendible array mapping function should be the choice for addressing storage.

We should mention that a competitive model for comparisons of multi-dimensional array file implementations should be with the HDF5 data schemes. Our implementation currently does not include array caching of data pages which the HDF5 implementation uses. Work is still ongoing to add main memory buffer pools for data caching at which time a fair comparison would be made.

6 Conclusion and Future Work

We have shown how a k-dimensional extendible array file can be used to implement multi-dimensional databases. The technique applies to extendible arrays in-core just as much as for out-of-core extendible arrays that can be either dense or sparse. The method relies on a mapping function that uses information retained in axial-vectors to compute the linear storage addresses from the k-dimensional indices. Given the characteristics of multi-dimensional databases that they incrementally grow into terabytes of data, developing a mapping function that does not require reorganization of the array file as the file grows is a desirable future.

The method proposed is highly appropriate for most scientific datasets where the model of the data is perceived typical as large global array files. The mapping function developed can be used to enhance current implementations of array files such as *NetCDF*, *HDF5* and *Global Arrays*. Work is still on-going to incorporate our proposed solution to multidimensional array libraries and to extend the technique for multi-dimensional datasets whose dimensions or ranks are allowed to expand. We are also conducting comparative studies on the different techniques for managing sparse multi-dimensional extendible arrays.

Acknowledgment

This work is supported by the Director, Office of Laboratory Policy and Infrastructure Management of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098. This research used resources of the National Energy Research Scientific Computing (NERSC), which is supported by the Office of Science of the U.S. Department of Energy.

References

[1] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Sparse Matrix Storage Formats, in Templates for the solution of algebraic eigenvalue problems: A practical guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[2] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *Proc. ACM-SIGMOD, 1998*, pages 259–270, 1998.

[3] P. Furtado and P. Baumann. Storage of multidimensional arrays based on arbitrary tiling. In *Proc. of 15th Int'l. Conf. on Data Eng. (ICDE'99)*, page 480, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[4] S. Goil and A. N. Choudhary. Sparse data storage schemes for multidimensional data for olap and data mining. Technical Report CPDC-TR-9801-005, Center for Parallel and Dist. Comput, Northwestern Univ., Evanston, IL-60208, 1997.

[5] S. Goil and A. N. Choudhary. High performance multidimensional analysis of large datasets. In *Int'l. Wkshp on Data Warehousing and OLAP*, pages 34–39, 1998.

[6] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[7] Hierachical Data Format (HDF) group. *HDF5 User's Guide*. National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana-Champaign, Illinois, Urbana-Champaign, release 1.6.3. edition, Nov. 2004.

[8] Nikos Karayannidis and Timos Sellis. Sisyphus: The implementation of a chunk-based storage manager for olap data cubes. *Data snf Knowl. Eng.*, 45(2):155–180, 2003.

[9] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Mass., 1997.

[10] NetCDF (Network Common Data Form) Home Page. http://my.unidata.ucar.edu/content/software/netcdf/index.html.

[11] J. Nieplocha and I. Foster. Disk resident arrays: An array-oriented I/O library for out-of-core computations. In *Proc. IEEE Conf. Frontiers of Massively Parallel Computing Frontiers'96*, pages 196 – 204, 1996.

[12] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169 – 189, 1996.

[13] E. J. Otoo and T. H. Merrett. A storage scheme for extendible arrays. *Computing*, 31:1–9, 1983.

[14] M. Ouksel and P. Scheuermann. Storage mappings for multidimensional linear dynamic hashing. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 90 – 105, Atlanta, March 1983.

[15] A. L. Rosenberg. Allocating storage for extendible arrays. *J. ACM*, 21(4):652–670, Oct 1974.

[16] D. Rotem and J. L. Zhao. Extendible arrays for statistical databases and OLAP applications. In *8th Int'l. Conf. on Sc. and Stat. Database Management (SSDBM '96)*, pages 108–117, Stockholm, Sweden, 1996.

[17] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimenional arrays. In *Proc. 10th Int'l. Conf. Data Eng.*, pages 328 – 336, Feb 1994.

[18] Kent E. Seamons and Marianne Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proc. 7th Int'l. Conf. on Scientific and Statistical Database Management*, pages 218–227, Washington, DC, USA, 1994. IEEE Computer Society.

[19] T. Tsuji, A. Isshiki, T. Hochin, and K. Higuchi. An implementation scheme of multidimensional arrays for molap. In *DEXA, Workshop*, pages 773 – 778, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[20] T. Tsuji, H. Kawahara, T. Hochin, and K. Higuchi. Sharing extendible arrays in a distributed environment. In *IICS '01: Proc. of the Int'l. Workshop on Innovative Internet Comput. Syst.*, pages 41–52, London, UK, 2001. Springer-Verlag.

[21] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. ACM-SIGMOD Conf.*, pages 159–170, 1997.

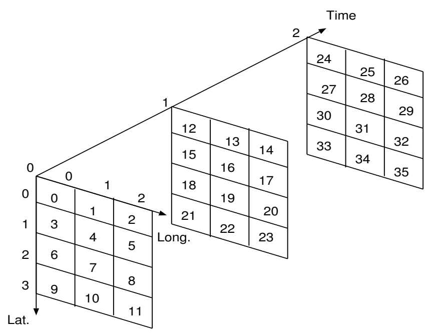
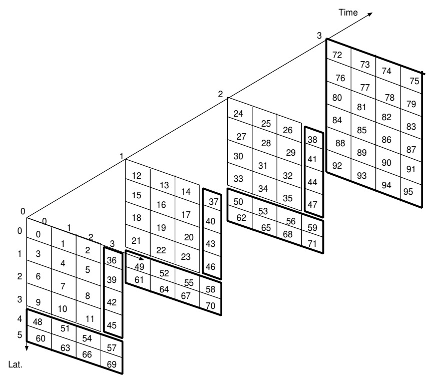
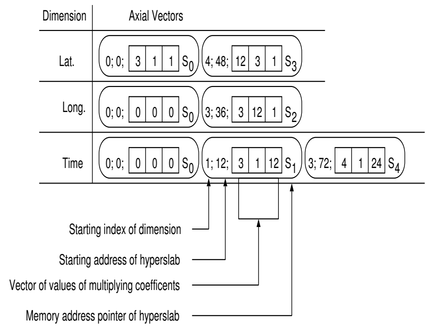


Figure 1: A Lat., Long. and Time 3-D model of the data

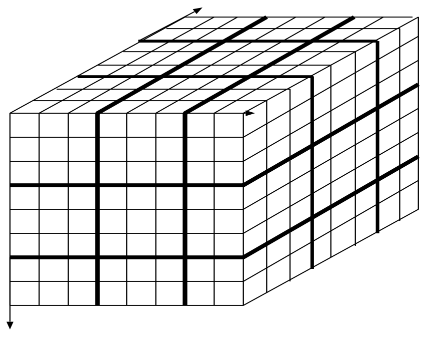


(a) A storage allocation a 3-dimensional array varying in both space and time

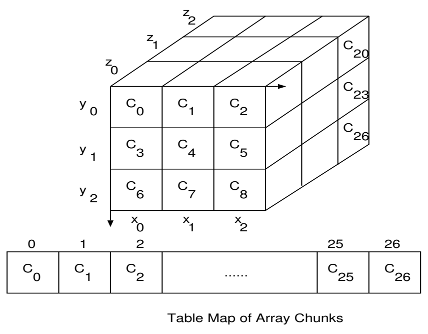


(b) The corresponding 3 distinct Axial-Vectors

Figure 2: An example of the storage allocation scheme of a 3-D extendible array

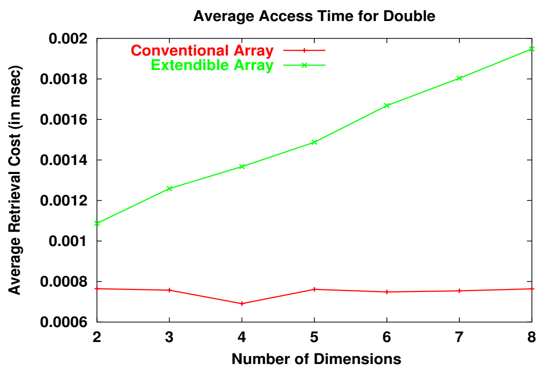


(a) A 3-dimensional array partitioned into chunks

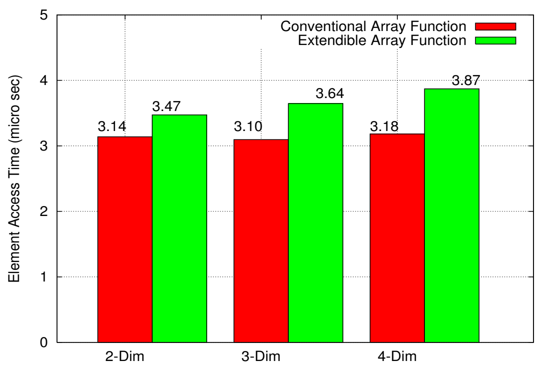


(b) Chunked Array with its Table Map

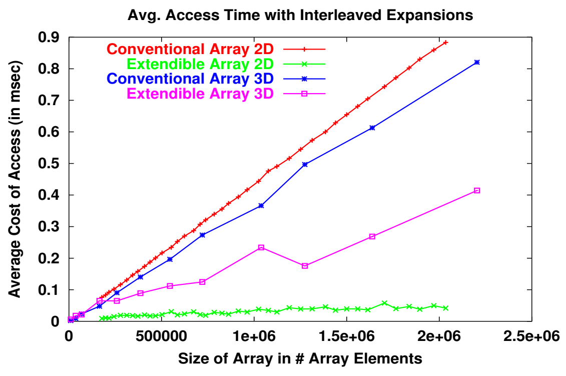
Figure 3: An allocation scheme of a 3-D sparse extendible array



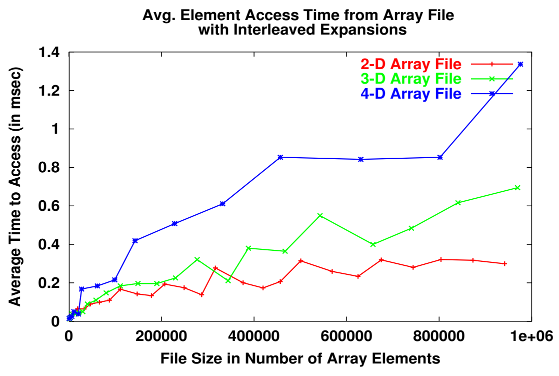
(a) Element access costs for static array



(a) Access cost from a static array file



(b) Element access costs with interleaved extensions



(b) Access cost from a file with interleaved extensions

Figure 4: Average time to access an element in an array of type double

Figure 5: Average time to access an element from an extendible array file of type double