Chunking of Large Multidimensional Arrays

Doron Rotem and **Ekow Otoo**
LBNL, University of California
1 Cyclotron Road
Berkeley, CA 94720

Sridhar Seshadri
Leonard N. Stern School of Business
New York University
44 W. 4th St., 7-60, New York, 10012-1126
sseshadr@stern.nyu.edu

Abstract

Very large multidimensional arrays are commonly used in data intensive scientific computations as well as on-line analytical processing applications referred to as MOLAP. The storage organization of such arrays on disks is done by partitioning the large global array into fixed size sub-arrays called **chunks** or **tiles** that form the units of data transfer between disk and memory. Typical queries involve the retrieval of sub-arrays in a manner that accesses all chunks that overlap the query results. An important metric of the storage efficiency is the expected number of chunks retrieved over all such queries. The question that immediately arises is "what shapes of array chunks give the minimum expected number of chunks over a query workload?" The problem of optimal chunking was first introduced by Sarawagi and Stonebraker [14] who gave an approximate solution. In this paper we develop exact mathematical models of the problem and provide exact solutions using steepest descent and geometric programming methods. Experimental results, using synthetic and real life workloads, show that our solutions are consistently less than 2.0% of the true number of chunks retrieved for any number of dimensions. In contrast, the approximate solution of [14] can deviate considerably from the true result with increasing number of dimensions.

Categories and Subject Descriptors

H.2[Information Systems, Database Management]; H.2.2 [Physical Design]; H.2.8[Database Applications, Scientific databases,Statistical databases]

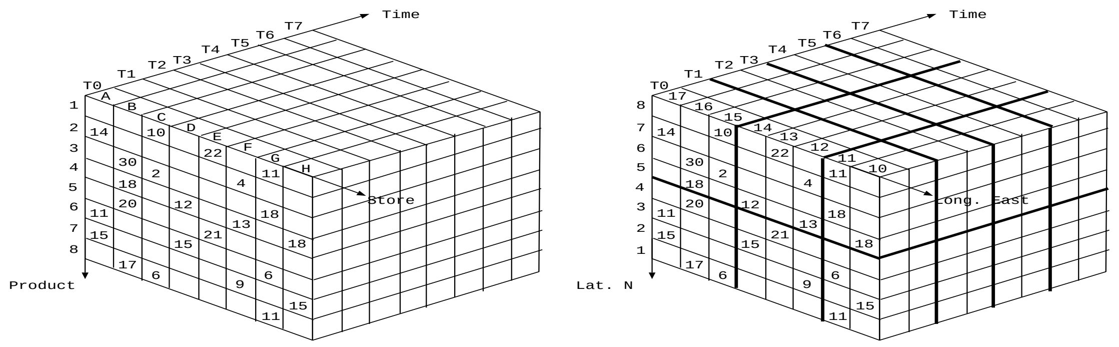
General Terms

Multidimensional Arrays, Algorithms, Array Chunking.

1 Introduction

The computations, analysis and visualization of large scale scientific data involves manipulation of data abstracted as multi-dimensional arrays. The multi-dimensional rectangular arrays, both dense and sparse depending on the context, form the fundamental abstract data structure used in scientific computing. Consequently scientific applications generally center around manipulation of large arrays and array files. Numerous applications in scientific domains such as Physics, Astronomy, Geology, Earth Sciences, Statistics, etc., map their problems space onto matrices and multi-dimensional arrays on which mathematical tools such as linear, non-linear equations solvers and differential equation solvers can be applied. Starting with numeric data arrays from observations, instruments and simulation experiments, these arrays are required to be persistent on disks and subsequently accessed efficiently for scientific analysis.

Another area where multidimensional arrays are commonly used is data warehousing and on-line analytical processing (OLAP) which often require extraction of statistical information for decision support. One gets a better intuitive meaning of the statistical summaries of the data if the data is abstracted as a multi-dimensional dataset. More specifically, usage of optimized multi-dimensional array storage is prevalent in MOLAP (Multidimensional Online Analytical Processing) and HOLAP (Hybrid Online Analytical Process) type products such as Essbase (now officially called Hyperion System 9 BI+ Analytic Services) and Microsoft Analysis Services. A canonical example of a multidimensional array is that of sales data on *products, stores, time* [6, 18], this can be represented as a relation *R(Product, Store, Time, Sales)* on 4 attributes: *products, stores, time* and *Sales*. This information can also be perceived as a 3-dimensional array with 3 independent axes: *Product, Store, Time*, with the values of *Sales*, also termed the *measure*, as the entries in the array. In general a MOLAP model of $k+1$-dimensional attribute relation, $\mathcal{R} \subseteq D_1 \times D_2 \times \ldots \times D_k, \mathcal{Z}$, consists of k-dimensional array, with axes $D_1, D_2, \ldots, D_k$ whose entries are drawn from values of a measure $\mathcal{Z}$, and a representative *null value* ϕ.



(a) A 3-dimensional MOLAP *R(Product, Store, Time, Sales)*

(b) A 3-Dimensional Array of temperature readings over *lat, long, time*, bold grid lines represent chunk boundaries

Figure 1: Multi-dimensional Models of Scientific and MOLAP Datasets

Figure 1a is a simple illustrative 3-dimensional MOLAP view of $\mathcal{R}$. Figure 1b is another simple illustrative view of a 3-dimensional climate data depicting the temperature values of locations indexed by *latitude (lat), longitude (long)* and *time*. Except for the semantic interpretation of the axes, and the entries in the arrays of the two figures, the structural representation are equivalent. Shoshani [16], first showed the similarities and differences between OLAP and statistical databases. The differences however are minor and were primarily attributed to the issues of concern by implementors of statistical and OLAP databases at that time. In the broader sense of comparing the requirements of scientific database management and MOLAP systems today, they are the same in nearly every aspect of storage and access requirements. The problem is that there is currently no adequately defined data model that can be used for their efficient implementations. In general, both scientific and MOLAP datasets can be considered as a collection of multi-dimensional arrays that reside on secondary storage and queries on an array involve an orderly access of either the entire array or a hyper-rectangular sub-array.

To store array elements on disk, one can naively utilize the mapping of multi-dimensional array indices onto linear storage. Two such conventional mapping are the row-major (or C-Language) order, and the column-major (or Fortran Language) order. A layout of the elements in say row-major order only guarantees good performance if the elements are subsequently accessed in the same order. Accessing the elements in a different order, e.g. column-major order, gives very poor performance [15]. Secondly, such a layout is only worth considering if the array is generally *dense*, i.e., almost every array entry exists. Thirdly, such an array layout on secondary storage is not extensible without storage reorganization. Some major characteristics for consideration in the storage and access of these arrays onto disk then are that:

- the array can be extremely large, requiring gigabytes of disk storage and sometimes tertiary storage.
- the arrays are sparse in that there are fewer valid entries than indexed locations.
- in both scientific data storage and MOLAP storage, the data incrementally grows over time and as such the array storage mapping must be extensible.

Persistent storage organization of multi-dimensional arrays is typically done by partitioning them into coarse-grained hyper-rectangular blocks called *chunks* or *tiles* which form the units of array transfers between disk and memory [14, 15, 5, 9]. A chunk is defined by the index range of values along each dimension. A query over the dataset for analysis retrieves either the entire array or a sub-array in which case all the array chunks that overlap the query result are retrieved. Even though the elements contained in each chunk, are stored either in row-major order, or column major order, the layout of the chunks on disk can be done using some other linear mapping function such as the Morton sequence, Hilbert scan, or Peano scan order [8]. Chunking alleviates some of the concerns in multidimensional array storage since:

- array chunks with all zero entries are not stored and chunks with fewer entries below a specified threshold can be compressed. This results in an improved storage utilization.
- Allocating chunks through an index scheme, e.g., B^+-tree, allows for arbitrary array expansions without storage reorganization.

A question that arises in the use of chunking is that of specifying an optimal chunk shape and chunk size. A chunk is characterized by two parameters: the *chunk size* and the *chunk shape*. The size is defined as the number of elements that can be contained in a chunk. Suppose a k-dimensional array $\mathcal{M}[N_1, N_2, \ldots, N_k]$ is partitioned such that dimension N_j is split into m_j intervals, for $1 \leq j \leq k$. The *chunk shape* is given by $\langle c_1, c_2, \ldots, c_k \rangle$, where $c_j = \lceil N_j/m_j \rceil$, is the number of indices of dimension j addressable in a chunk. A chunk shape implicitly defines a chunk size $C = \prod_{j=1}^{k} c_j$. Note that large chunk sizes may cause unnecessary data to be read for queries with small result set. On the other hand, small chunk sizes, may require more disk accesses to retrieve all chunks required to answer a query. More importantly, the chunk shape influences the number of chunks retrieved in answering a query.

An important metric of the storage efficiency is the expected number of chunks retrieved by queries under the access workload. The problem of optimal chunking was first introduced by Sarawagi and Stonebraker [14], who gave an approximate solution to this problem. We show that the optimal shape derivation given by Sarawagi and Stonebraker is only approximate and under certain circumstances can deviate significantly from the true answer. We propose two different models of the problem and show how the chunking parameters should be determined based on the probabilistic access patterns of sub-array queries. The main contributions of this paper are:

- The development of two accurate mathematical models of the chunking problem;
- Derivation of exact solutions, one using steepest descent and another using geometrical programming method;
- Experimental comparison of the estimation errors induced by the models using synthetic workloads on real life datasets.

In the rest of the paper, Section 2 presents some related work on array chunking where we also discuss how chunks are organized and accessed for both dense and sparse multi-dimensional arrays. Section 3 presents the two mathematical models for defining an optimal chunking shape. The derivations of the optimal chunk shapes and sizes, under both models given some probabilistic access patterns of sub-array queries are presented in Section 4. In section 5, we present the results of our experimental comparisons for a synthetic workload. We conclude with Section 6, giving some direction for future work.

2 Related Work

In nearly all applications that use disk resident large scale multi-dimensional arrays, the physical organization of the array is by chunking. The global array is tessellated into sub-arrays or tiles of size C and shape $\langle c_1, c_2, \ldots, c_k \rangle$. Rather than mapping the elements of the array directly onto consecutive linear storage, the chunks are mapped onto storage and, within each chunk, the array elements are laid out using a conventional row-major or column-major ordering.

The rational for chunking large arrays, whether dense or sparse, is justified in general when efficient I/O performance is desired in applications that access data with a high degree of locality [17]. In [17], Vitter elaborated on the fact that an algorithm that does not exploit locality can be reasonably efficient when the data sets fit entirely in internal memory, but performs miserably when deployed naively on an *External Memory (EM)*, setting and virtual memory is used to handle page management. The linear mapping function for allocating chunks onto disk storage can be done by the row-major or column-major ordering, any one of the mapping functions for space filling curves [8] or done with the use of $B^{+}-tree$ indexing as in HDF5 [7]. We discuss some methods for chunk addressing in subsection 2.1. The problem of chunk addressing is orthogonal to optimizing the chunk shape that requires taking into account the information on sub-array access patterns. This is the problem first raised by Sarawagi and Stonebraker [14].

In [15], the problem of the storage of multidimensional arrays on disks for subsequent efficient access in computational fluid dynamic applications that run on parallel processors is addressed. The approach taken is to distribute all the array chunks among processors during program executions. A similar approach is described in the design and implementation of disk resident array storage (DRA) [11], used in Computational Chemistry applications. DRA extends the use of a memory resident distributed array library, called Global Array (GA), to external memory by a controlled I/O of the sub-arrays onto disks. As in [15], the method employs chunking of persistent dense arrays on disk.

The other domain where array chunking has been predominantly used is in multidimensional on-line analytical processing algorithm (MOLAP) [18, 5, 9, 13, 12]. In [18], the method of computing the CUBE over a multi-dimensional data model was introduced. The authors gave a detailed analysis for the associated on-line analytical processing algorithms. The MOLAP model proposed storing the data as a sparse arrays where the elements of the array are the measures. The encoding of the attribute values, along each dimension, defined the position of the value in the multi-dimensional space. The array is split into chunks of size the same as the block size of the disk storage system. Chunk compression is further used to improve storage utilization.

Goil and Choudhary [5] presented a storage scheme for MOLAP similar to that in [18] but applied a bit-encoded scheme for the position index of the occurring array elements. The method introduced was referred to as the bit-encoded sparse structure(BESS). Not only is BESS applicable to MOLAP data sets, but can be applied to scientific multi-dimensional sparse array data. Variations of the chunking concepts for the storage schemes MOLAP data sets are also proposed in the SISYPHUS storage manager [9].

In all of the above related works, non of the chunking schemes are driven by the query access pattern. Further, given the fact that multi-dimensional databases for data warehousing have the propensity to grow, very little is discussed on how extensibility is managed in these schemes. The problem on handling extensibility in chunked arrays is the research focus reported in [13, 12].

2.1 Addressing Array Chunks

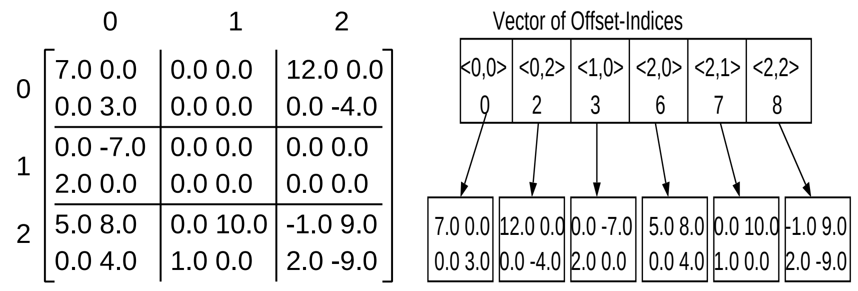
The idea of chunking multi-dimensional arrays has its origins from techniques used in scientific computing for managing memory resident sparse matrices [2] and large sparse and dense matrices in paged and parallel environments [10, 11]. We illustrate an addressing method for array chunks with a technique for sparse multi-dimensional arrays. Consider first an example of a 6×6 array $M = (m_{ij})$, of doubles shown below. The Block Compressed Row storage BCRS [4], for spares matrices forms the basis of a typical chunk addressing method.

$$M = \begin{bmatrix} 7.0 & 0.0 & 0.0 & 0.0 & 12.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 0.0 & 0.0 & -4.0 \\ 0.0 & -7.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 5.0 & 8.0 & 0.0 & 10.0 & -1.0 & 9.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 2.0 & -9.0 \end{bmatrix}$$

2.1.1 Block Compressed Row Storage

The block compressed row-storage partitions the matrix, along each dimension into intervals, to form small blocks. Each blocks is then treated as a dense matrix. Figure 2 illustrates the storage scheme.

The block addressing is done in two levels. The first level concerns locating the block that an element falls in and the second level concerns the location of the element within the block. The first level block organization is treated as a sparse array in which all blocks containing only zero elements are discarded. Each block has a coordinate index $\langle i, j \rangle$. The offset-values computed from a linear mapping function are organized in an offset-value vector. Only the offset-values of block with non-zero entries are retained. One only needs to define a linear mapping function



(a) Blocked Matrix (b) Block Addressing

Figure 2: Block Compressed Row Storage

$I_{ij} = g(\langle i,j\rangle)$ that takes the coordinate index $\langle i,j\rangle$ and returns an offset value I_{ij}, relative to the position of $I_{0,0}$ and also the inverse function $\langle i,j\rangle = g^{-1}(I_{ij})$ that takes an offset value and returns the coordinate index. Figure 2b illustrates the offset-indices of the blocks when the mapping function is the row-major linearizing function. Given the coordinates $\langle i,j\rangle$, one determines if the element a_{ij} exists by first computing the offset index of the block I_{ij} using the mapping function $g()$ and then determining whether I_{ij} occurs in the offset index vector or not. Searching the offset index vector can be done with an interpolation search or alternatively, organizing the pairs of offset index and block pointers as a balanced binary search tree.

The BCRS method generalizes naturally to multi-dimensional arrays. The chunking process is equivalent to the block partitioning method used for matrices. For extremely large multi-dimensional arrays the first level chunk organization can be done with $B^+{-}tree$. In HDF5 [7], a popular multi-dimensional array file format used extensively in scientific computing, the chunk accessing is done through a $B^+{-}tree$ storage scheme.

3 Access Models of Arrays

A k dimensional array, $M[N_1, N_2, \ldots, N_k]$, consists of $\prod_{i=1}^{k} N_i$ elements. Each of its elements, $m\langle i_1, i_2, \ldots, i_k\rangle$, is indexed by k indices where $0 \leq i_j < N_j$ is its index with respect to the j^{th} dimension. We wish to store M on disk subject to the constraint that each disk block can hold at most C elements of M. This is done by partitioning M into equal shape rectangular chunks such that each chunk fits on a disk block, i.e., if each chunk has dimensions $\langle c_1, c_2, ..., c_k\rangle$ then $\prod_{i=1}^{k} c_i \leq C$.

The system supports queries that retrieve rectangular sub-arrays of M.
A query $q = \langle [l_1 : u_1), [l_2 : u_2), \ldots, [l_k : u_k)\rangle$ specifies a lower bound l_i and an upper bound u_i on each of the k dimensions. The query retrieves all elements $m\langle i_1, i_2, ..., i_k\rangle$ of M such that $l_j \leq i_j < u_j$ for $1 \leq j \leq k$. The cost of answering this query is directly related to the number of chunks (disk blocks) that overlap the sub-array defined by the query. In [14], it was shown that knowledge of the predicted query access patterns can be efficiently used to select chunk dimensions that result in a significant reduction in the cost of answering queries. Prediction of query access patterns is usually based on query statistics that are collected using query history logs, sampling, or other statistical methods. Next we present two models commonly used for query access pattern prediction: *The Independent Attribute Range* model and *The Query Shape* model.

3.1 Independent Attribute Range (IAR)

For a query $q = \langle [l_1 : u_1), [l_2 : u_2), \ldots, [l_k : u_k) \rangle$, we define its shape to be $\langle A_1, A_2, \ldots, A_k \rangle$ where A_i is the size of its range on the i^{th} dimension, i.e., assuming the ranges are closed on the left and open on the right, $A_i = u_i - l_i$. In both models a query shape can fall randomly anywhere within the array. In the IAR model, a probabilistic distribution of the possible range values is calculated separately for each of the k dimensions. It is assumed that the specifications of ranges of attributes in queries are independent of each other [1]. This assumption means that the estimated probability of a query shape is calculated as a product of the estimated probabilities of its components, i.e., the probability of a shape $\langle a_1, a_2, ..., a_k \rangle$ is estimated as $\prod_{i=1}^{k} p(a_i)$ where $p(a_i)$ is the estimated probability that the value of the range for the i^{th} dimension is a_i. More detailed treatment of this case is provided in Section 4.2.

3.2 Query Shape (QS)

This model is attributed to Sarawagi and Stonebraker [14], As in the IAR model, each query is associated with a shape $\langle A_1, A_2, \ldots, A_k \rangle$. The difference is that in the QS model the query access pattern is estimated in terms of probability distribution of complete query shapes rather than distributions of ranges of individual dimensions.

The QS model, groups the queries into a collection of classes $L_1, L_2, L_3, \ldots, L_q$ such that the class L_i contains all queries of shape $\langle A_{i1}, A_{i2}, \ldots A_{ik} \rangle$. Each class L_i is associated with a probability P_i, such that $\sum_i P_i = 1.0$. The access pattern is defined then by the set of pairs $\{\langle A_i 1, A_i 2, \ldots A_i k \rangle : P_i\}, 1 \leq i \leq q$. Our exact analysis of this model is presented in Section 4.3.

3.3 Illustrative Example

Under both models (IAR) and (QS), the actual location of a query shape relative to the array is assumed to be uniformly distributed. The following small example illustrates the difference between the two models.

Table 1: Queries

Query number	**Query**	**shape**
1	$< 1:3, 2:5 >$	$< 2, 3 >$
2	$< 4:7, 6:10 >$	$< 3, 4 >$
3	$< 5:9; 3:6 >$	$< 4, 3 >$
4	$< 6:8, 4:7 >$	$< 2, 3 >$

Example: For a 2-dimensional array, we assume access pattern estimation is based on a sample of 4 queries given in Table 1. Range distribution for each dimension is given in Table 2 and shape distributions according to the two models are shown in Table 3. Note that under model (IAR), some shapes that were not observed in the sample are assumed to have non-zero probability whereas under model (QS) only observed shapes have non- zero probability.

4 Optimizing Array Chunk Shapes

4.1 Analysis of Expected Chunk Overlaps

We will first estimate the number of chunks overlapping a fixed shape and then compute the expected number of chunks under the probabilistic assumptions for each of the two models.

Given a shape $A = \langle A_1, A_2, \ldots, A_k \rangle$, assuming the array M is split into chunks of dimensions $c = \langle c_1, c_2, ..., c_k \rangle$, we will denote by $E(A, c)$ the expected number of chunks overlapping the shape A assuming it can be located randomly anywhere in the array M.

Lemma 4.1. $E(A, c) = \prod_{i=1}^{k} (\frac{A_i - 1}{c_i} + 1)$

Proof. It is easy to see that

$$E(A, c) = \prod_{i=1}^{k} E(A_i, c_i)$$

where $A_i = \langle A_i \rangle$ is a one dimensional shape and $c_i = \langle c_i \rangle$ is a one dimensional chunk. It is therefore sufficient to calculate the number of chunks overlapping a shape on a one dimensional array M. For simplicity we will omit the angular brackets whenever it is clear from the context whether we are discussing a shape vector or its dimensions.

We will now proceed to show that given a one dimensional shape of size A_i and assuming a one dimensional array M is partitioned into chunks of size c, the expected number of chunks overlapping this shape is

$$E(A_i, c) = \frac{A_i - 1}{c} + 1.$$

Let us number the points within each chunk from 0 to $c - 1$ (see Figure 3). We can express A_i as $A_i = mc + r$ where m (quotient) and r (remainder) are integers with $0 \leq m$ and $0 \leq r < c$.

Under the assumption that a shape can fall uniformly anywhere in the array, the left endpoint of a shape can fall on any of the c points within a chunk with equal probability $1/c$. This assumes A_i is relatively small compared to the total size of M and ignores small "edge" effects due to the constraint that the right endpoint of a range must also fit in the array. Let us denote by R_1 and R_2 the sub-intervals within each chunk consisting of the leftmost $c - (r - 1)$ and rightmost $r - 1$ points respectively. In the event that the left endpoint of the shape falls in R_1 it will overlap $m + 1$ chunks and if it falls in region R_2 it will overlap $m + 2$ chunks. For example, in Figure 3 this is illustrated for the case $A_i = 8$ and $c = 5$, i.e., $m = 1$ and $r = 3$. The possible positions within a chunk where the query shape may fall are labeled as r_i. We see that the positions r_0,r_1 and r_2 overlap $m + 1 = 2$ chunks whereas the positions r_3 and r_4 overlap $m + 2 = 3$ chunks. In this case the expected number of chunks is $3/5 \times 2 + 2/5 \times 3 = 12/5$. In general, the expected number of chunks that are overlapped by the shape is therefore

$$E(A, c) = \frac{c - (r - 1)}{c}(m + 1) + \frac{r - 1}{c}(m + 2).$$

By using $m = (A_i - r)/c$ and rearranging, we get the required result of

$$E(A_i, c) = \frac{A_i - 1}{c} + 1 \tag{4.1}$$

Table 2: Individual range probabilities under model (IAR)

Dimension #	**Range value**	**Appears in query#**	**Range probability**
1	2	1,4	1/2
1	3	2	1/4
1	4	3	1/4
2	3	1,3,4	3/4
2	4	2	1/4

Table 3: Shape probabilities under the two models

Query Shape	Prob. Model (IAR)	In Query #	Prob. (QS)
$<2,3>$	$1/2 \times 3/4 = 3/8$	1,4	1/2
$<2,4>$	$1/2 \times 1/4 = 1/8$	-	0
$<3,3>$	$1/4 \times 3/4 = 3/16$	-	0
$<3,4>$	$1/4 \times 1/4 = 1/16$	2	1/4
$<4,3>$	$1/4 \times 3/4 = 3/16$	3	1/4
$<4,4>$	$1/4 \times 1/4 = 1/16$	-	0

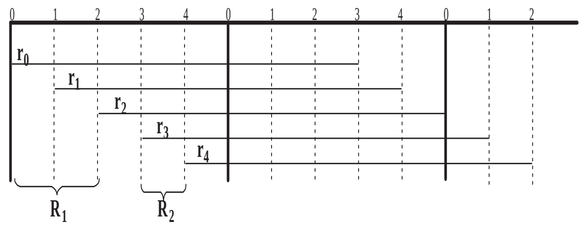


Figure 3: Example of range position effects on number of chunks retrieved.

from which the the lemma follows. ☐

As we can see, the expression in Lemma 4.1 involves subtracting 1 from each range size. For convenience we will refer to these reduced ranges as "adjusted range sizes".

4.2 Analysis of the Independent Attribute Range Model

We recall that this model assumes that the probability of a random query shape is calculated from the individual probability distributions for range values in each dimension. More specifically, we assume there are m_i possible range values A_{ij} for the i^{th} dimension where each such value appears in a random query shape with probability p_{ij} and $\sum_{j=1}^{m_i} p_{ij} = 1$ for $1 \le i \le k$.

Lemma 4.2. *The expected number of chunks that overlap a random query shape is*

$$\prod_{i=1}^{k} \left(\frac{\bar{A}_i}{c_i} + 1 \right)$$

where $\bar{A}_i$ is the expected value of the adjusted range size for the i^{th} dimension, i.e., $\bar{A}_i = \sum_{j=1}^{m_i} p_{ij}(A_{ij} - 1)$

Proof. (Outline): Using Lemma 4.1 and the fact that the probability of a query shape is equal to the product of the probabilities of its components, we can show that the expected overlap is

$$\prod_{i=1}^{k} \left[\sum_{j=1}^{m_i} p_{ij} \left(\frac{A_{ij} - 1}{c_i} + 1 \right) \right].$$

This by definition is equal to

$$\prod_{i=1}^{k} \left(\frac{\bar{A}_i}{c_i} + 1 \right).$$

□

The chunk overlap minimization problem we wish to solve can be stated as follows:

$$\min \prod_{i=1}^{k} \left(\frac{\bar{A}_i}{c_i} + 1 \right) \tag{4.2}$$

Subject to

$$\prod_{i=1}^{k} c_i \leq C \tag{4.3}$$

where the c_i's are integers.

We will first show how to solve this problem by relaxing this latter integrality constraint and then discuss the integral solution. Optimization problems where the objective function and/or constraints contain products rather than sums are known as geometrical programming(see [3]). Our case is more involved than typical geometrical programming problems as the solution represents chunk sizes which must be integers.

Theorem 4.1. *The solution of the system represented by Equations* (4.2),(4.3) *for real* c_i*'s is*

$$c_i = \bar{A}_i \sqrt[k]{\frac{C}{\prod_{i=1}^{k} \bar{A}_i}}$$

Proof. We first represent the above optimization problem using base 2 logarithms. We define new variables y_i where $y_i = \log_2 c_i$ and $C' = \log_2 C$ and solve the system for the new variables y_i. Equation (4.2) can be rewritten as

$$\prod_{i=1}^{k} \left(\frac{\bar{A}_i}{2^{y_i}} + 1 \right) \tag{4.4}$$

and (4.3) becomes

$$\sum_{i=1}^{k} y_i = C' \tag{4.5}$$

Let $E = \prod_{i=1}^{k} \left(\frac{\bar{A}_i}{2^{y_i}} + 1 \right) - \lambda(\sum_{i=1}^{k} y_i - C')$. Using Lagrange multipliers, E is minimized when

$$\frac{\partial E}{\partial y_i} = 0 \text{ for all } 1 \leq i \leq k$$

This is

$$-\bar{A}_i (\ln 2) 2^{-y_i} \prod_{j \neq i}^{k} \left(\frac{\bar{A}_j}{2^{y_j}} + 1 \right) - \lambda = 0 \quad \text{for } 1 \leq i \leq k. \tag{4.6}$$

or

$$\bar{A}_i 2^{-y_i} \prod_{j \neq i}^{k} \left(\frac{\bar{A}_j}{2^{y_j}} + 1 \right) = \frac{\lambda}{\ln 2} \text{ for } 1 \leq i \leq k \tag{4.7}$$

From (4.7) we get for any pair of variables y_r, y_s

$$\bar{A}_r 2^{-y_r} \prod_{j \neq r}^{k} \left(\frac{\bar{A}_j}{2^{y_j}} + 1 \right) = \bar{A}_s 2^{-y_s} \prod_{j \neq s}^{k} \left(\frac{\bar{A}_j}{2^{y_j}} + 1 \right) \text{ for } 1 \leq r, s \leq k \tag{4.8}$$

Re-arranging

$$\frac{\bar{A}_r}{\bar{A}_s} \frac{2^{-y_r}}{2^{-y_s}} \left(\frac{\bar{A}_s}{2^{y_s}} + 1 \right) = \left(\frac{\bar{A}_r}{2^{y_r}} + 1 \right) \tag{4.9}$$

using $c_i = 2^{y_i}$ for $1 \leq i \leq k$

$$\bar{A}_r c_s \left(\frac{\bar{A}_s}{c_s} + 1 \right) = \bar{A}_s c_r \left(\frac{\bar{A}_r}{c_r} + 1 \right) \tag{4.10}$$

$$\bar{A}_r \bar{A}_s + \bar{A}_r c_s = \bar{A}_s \bar{A}_r + \bar{A}_s c_r$$

and finally we get

$$\frac{c_r}{c_s} = \frac{\bar{A}_r}{\bar{A}_s}. \tag{4.11}$$

This means that the ratio between each pair of c_i's is equal to the ratio between their respective $\bar{A}_i$'s or in other words $\frac{\bar{A}_i}{c_i}$ is a constant. The result of the theorem follows directly from Equation (4.11). □

4.2.1 Integral solution

In most practical cases, the disk block size C is an integral power of 2. In that case, $C' = \log_2 C$ and the y_i's in the above equations must be all integers. In this section we will show how to solve the above problem optimally for this case by rounding up or down the non-integral solutions obtained in Theorem 4.1. Our approach uses some techniques developed in Aho and Ullman [1] for solving a different problem related to bucketing multidimensional data for partial match retrieval.

Let $\langle y_1, y_2, ..., y_k \rangle$ be the non-integral solution obtained above and $\hat{Y} = \langle \hat{y}_1, \hat{y}_2, ..., \hat{y}_k \rangle$ be an integral solution which is as good as any other integral solution. As mentioned above, equation 4.11 of Theorem 4.1 shows that $\frac{\bar{A}_i}{c_i} = \bar{A}_i 2^{-y_i}$ is a constant. We will denote it by e. Let $e_i = \bar{A}_i 2^{-\hat{y}_i}$, then by the optimality of the solution $\hat{Y}$, subtracting 1 from $\hat{y}_i$ and adding 1 to $\hat{y}_j$ cannot improve the value of the objective function in Equation 4.2. . Let $\hat{O}$ be the value of the objective function (see 4.2) resulting from using the solution $\hat{Y}$ and let $\hat{O}_{ij}$ be the value of the objective function obtained by a solution where we transfer 1 from $\hat{y}_i$ to $\hat{y}_j$ leaving all other terms unchanged. The two terms that are different between these two objective functions (see 4.2) are $(\bar{A}_i 2^{-\hat{y}_i} + 1)$ and $(\bar{A}_j 2^{-\hat{y}_j} + 1)$ in $\hat{O}$ which are changed to $(\bar{A}_i 2^{-(\hat{y}_i - 1)} + 1)$ and $(\bar{A}_j 2^{-(\hat{y}_j + 1)} + 1)$ in $\hat{O}_{ij}$ respectively. Using the notation above and noting that due to the optimality of the solution $\hat{Y}$ the ratio between the two objective functions, $\frac{\hat{O}_{ij}}{\hat{O}} \geq 1$, we get

$$\frac{1 + 2\hat{e}_i}{1 + \hat{e}_i} \frac{1 + \hat{e}_j/2}{1 + \hat{e}_j} \geq 1$$

¿From it follows that $2\hat{e}_i \geq \hat{e}_j$. Dividing both sides of this inequality by the constant e we finally get

$$(y_i - \hat{y}_i) \leq 1 - (\hat{y}_j - y_j) \tag{4.12}$$

Using similar arguments to the ones in [1] it follows that the optimal integral solution is obtained by rounding each y_i either up or down. We also note that in the case that y_i is rounded up its fractional part is $1 - (\hat{y}_i - y_i)$ and if it is rounded down its fractional part is $y_i - \hat{y}_i$. As equation (4.12) must hold for every pair of indices i and j, it follows that in an optimal solution, the fractional parts of each of the y_i's that are rounded up must be equal or larger than the fractional parts of the ones that are rounded down. We can therefore obtain an integral solution to our problem following the arguments in [1] as follows: Let the sum of the fracional parts of the non-integral solution be M (clearly M must be an integer). We can obtain an integral solution from it by rounding up the M components, y_i , with the largest fractional parts and rounding down the rest. The complete process of obtaining a solution is illustrated in the example below.

Row #		**Dim 1**	**Dim 2**	**Dim 3**	**Dim 4**	**Dim 5**
1	Input: Expected adjusted range size in queries	5.7	9.4	12.5	24.9	30.2
2	c_i= Non integral solution for chunk size (product is 8192)	2.501088	4.124602	5.484843	10.92581	13.25138
3	Solution in logarithms $y_i=\log_2 c_i$ (sum is 13)	1.322556	2.044255	2.45545	3.449668	3.728071
4	Fractional part of y_i (sum of fractional parts is 2)	0.322556	0.044255	0.45545	0.449668	0.728071
5	Integral solution (in logarithms) after rounding up y_3 and y_5 and rounding down the rest (sum is 13)	1	2	3	3	4
6	Final integral chunk size	2	4	8	8	16

Table 4: Example of analysis of the IAR model

Example: The table 4 shows an example of a 5-dimensional optimization problem with block size $C=2^{13}=8192$. In row #1, we show the input to the problem in terms of expected range sizes on each dimension. Row #2 shows the non-integral optimal solution obtained from Theorem 4.1, the product is 8192 as required by block size constraint. Row #3 shows the solution in terms of base 2 logarithms, note that their sum is 13 as required by block size constraint. The last two rows illustrate the conversion of the non-integral solution to an integral one. In row #4, we show the corresponding fractional parts of the solution. The fractional parts add up to 2. This means that to obtain an optimal integral solution we need to pick the largest M=2 fractional parts round up their corresponding y_i's (y_3 and y_5 in this case) and round down the rest. In row #5 we show the result of performing this rounding. The chunk size obtained as a final solution is shown in row #6.

4.3 Analysis of the Shape Model

Using the results of lemma 4.1 and the discussion of the QS model in 3.2 we can foormulate the optimization problem for the query shape model as:

$$\min \sum_{j=1}^{q} P_j \prod_{i=1}^{k} \left(\frac{A_{ij}}{2^{y_i}} + 1 \right); \quad s.t. \sum_{i=1}^{k} y_i \le C'; \quad \mathbf{y} \in S \tag{4.13}$$

where, S is the set of k-tuple strictly positive integers, and $\mathbf{y}$ denotes the k-tuple (y_1, y_2, ..., y_k), $y_i = \log_2 c_i$, the "adjusted" shape of the j-th query is $\langle A_{1j}, A_{2j}, \cdots, A_{kj} \rangle$ (for simplicity of notation we assume here that the A_{ij}'s are obtained by subtracting 1 from each original range as explained previously), and $C' = \log_2 C$. The probability of the j-th shape is $P_j,\ j = 1, 2, \cdots, q$.
.

This problem is more complex than the optimization of the IAR model as no closed form solution is known. However, we will show that a greedy algorithm can be used to solve this problem.

We shall denote,

$$\phi(\mathbf{y}) = \sum_{j=1}^{q} -P_j \prod_{i=1}^{k} \left(\frac{A_{ij}}{2^{y_i}} + 1 \right)$$

Let $\mathbf{e}_i$ denote the k-dimensional unit vector with unity in the i-th place. We shall use the following lemma:

Lemma 4.3. *If $\lambda \ge 0$ and $\mathbf{y}^* \in S$ maximize the Lagrangian $\phi(\mathbf{y}) - \lambda g(\mathbf{y})$ over all $\mathbf{y} \in S$, then $\mathbf{y}^*$ maximizes $\phi(\mathbf{y})$ over all $\mathbf{y} \in S$ and $g(\mathbf{y}) \le g(\mathbf{y}^*)$.*

Proof. The proof is by contradiction. Assume that there is a $\mathbf{y} \in S$ such that $g(\mathbf{y}) < g(\mathbf{y}^*)$ and for which the value of $\phi(\mathbf{y}) > \phi(\mathbf{y}^*)$. Then $\mathbf{y}^*$ cannot maximize the Lagrangian $\phi(\mathbf{y}) - \lambda g(\mathbf{y})$ over all $\mathbf{y} \in S$. ☐

Consider the following algorithm which we cal QS_Algorithm:

```
1. $\mathbf{y}^0 = (0,0,\ldots,1)^T$ – initialize all $y_i$'s to 0.
2. n = 1.
3. Do while $\sum_i y_i \le C'$
4. $\mathbf{y}^n = \mathbf{y}^{n-1} + \mathbf{e}_i$, where $\mathbf{e}_i$
is the i-th unit vector and i is any index for which $[\phi(\mathbf{y} + e_i) - \phi(\mathbf{y})]$ is maximum.
5. Set $n = n + 1$
End.
```

As the number of iterations in the algorithm is $C' = \log_2 C$ and in each iteration we have to compute k difference computations the total running time of the algorithm is $O(k \log_2 C)$.

Definition 4.1. *Allocation $\mathbf{y}$ belongs to $T(\lambda)$
if $[\phi(\mathbf{y} + e_i) - \phi(\mathbf{y})] \le \lambda$ and $[\phi(\mathbf{y} - \delta e_i) - \phi(\mathbf{y} - (\delta + 1)e_i)] > \lambda,$ for $y_i > 1$ and $\delta \le y_i - 1$.*

It follows that if $\mathbf{y}$ belongs to $T(\lambda)$ then it is a global maximum of $\phi(\mathbf{y}) - \lambda g(\mathbf{y})$ over S, where $g(\mathbf{y}) = y_1 + y_2 + \ldots + y_k$. To see this, say in any alternate $\mathbf{y'}$ the value of $\phi(\mathbf{y}') - \lambda g(\mathbf{y}')$is larger. If for any index i y'_i is larger than y_i then the gain in the objective function is less than or equal to zero due to concavity (the gain in $\phi(\mathbf{y})$ is offset by the loss due to $\lambda g(\mathbf{y})$. If y'_i is smaller then again the gain is less than zero using a similar reasoning (the loss in $\phi(\mathbf{y})$ is greater than the gain in $\lambda g(\mathbf{y})$).

Theorem 4.2. *The allocations generated by the algorithm belong to $T(\lambda^n)$ where λ^n equals the largest difference found in step 3 at iteration n.*

Proof. Notice that the theorem holds for n=1. We shall use induction on n. Also,

$$\frac{\partial \phi}{\partial y_i} = \ln 2 \sum_{j=1}^{q} P_j \frac{A_{ij}}{2^{y_i}} \prod_{l=1, l\neq i}^{k} \left(\frac{A_{lj}}{2^{y_l}} + 1 \right) \quad .$$

Thus, the differences found in step 3 are decreasing in n. Therefore, $0 < \lambda^n < \lambda^{n-1}$.

By construction (that is choice of i in step 3), for any index l

$$\left[\phi(\mathbf{y}^{n-1} + e_l) - \phi(\mathbf{y}^{n-1})\right] \leq \lambda^n.$$

Assume that z was the index of the y that was increased at step $(n-1)$. Then for any index l

$$\phi(\mathbf{y}^{n-1} - \delta e_l) - \phi(\mathbf{y}^{n-1} - (\delta+1)e_l) \geq \lambda^{n-1}; \quad 0 \leq \delta \leq y_l^{n-1} - 1. \tag{4.14}$$

$$\phi(\mathbf{y}^{n-1} + e_z) - \phi(\mathbf{y}^{n-1}) = \lambda^{n-1}. \tag{4.15}$$

Because the differences are decreasing we need to prove that ((4.14)) holds for $\delta = 0$ and n.

Due to global optimality (from induction and Lemma 4.3)

$$\phi(\mathbf{y}^{n-1}) - \phi(\mathbf{y}^{n-1} - e_l + e_z) \geq 0. \tag{4.16}$$

Therefore, using the fact that $\phi(\mathbf{y^n}) = \phi(\mathbf{y^{n-1}} + e_z)$ and ((4.16))

$$\begin{aligned} \phi(\mathbf{y}^n) - \phi(\mathbf{y}^n - e_l) \geq {} & \phi(\mathbf{y}^{n-1} + e_z) - \phi(\mathbf{y}^{n-1}) \\ = {} & \lambda^{n-1} > \lambda^n. \end{aligned} \tag{4.17}$$

Thus, ((4.14)) holds for n. ☐

Theorem 4.3. *The algorithm terminates with the optimal solution to the query shape problem* .

Proof. The proof follows from Theorem 4.2 and Lemma 4.3. ☐

4.3.1 An Example of Applying Theorem 4.2

	shape 1	shape 2	shape 3	shape 4
probability	.4	.2	.3	.1
Dim 1	100	75	80	165
Dim 2	17	14	10	26
Dim 3	23	12	14	9
Dim 4	35	60	45	70
Dim 5	40	30	21	34

Table 5: Example of input shapes to the QS_Algorithm

Example: the table 5 shows an example 5-dimensional input problem with 4 shapes. The block size constraint is 2^{16}. In table 6 we show the first 4 iterations and the final two iterations of the algorithm before it finds the optimal solution.

Itr.	y1	y2	y3	y4	y5	Obj. funct.
0	0	0	0	0	0	46,560,641.59
1	1	0	0	0	0	23,503,316.20
2	2	0	0	0	0	11,974,653.50
3	2	0	0	1	0	6,122,765.65
4	2	0	0	1	1	3,147,627.72
..	...	...	...	..	...	
14	4	2	2	3	3	6233.26
15	5	2	2	3	3	3537.00
16	5	2	2	4	3	2041.87

Table 6: Iterations of the QS_Algorithm

5 Experimental Results

The first observation in our mathematical models is the difference in the expressions, between our model and that of Sarawagi and Stonebraker, used in calculating averaged number of blocks fetched for a specified access pattern. The two principal expressions are:

$$\sum_{j=1}^{q} P_j \prod_{i=1}^{k} \left(\frac{A_{ij} - 1}{2^{y_i}} + 1 \right); \tag{5.1}$$

and from [14]

$$\sum_{i=1}^{k} \left(\prod_{j=1}^{q} \left\lceil \frac{A_{ij}}{c_i} \right\rceil \right) p_i \tag{5.2}$$

Under an assumption of equal probability of shapes, the principal terms in the two expressions for the number of chunks fetched for a given shape become $\prod_{i=1}^{k}((A_i - 1)/c_i + 1)$ and $\prod_{i=1}^{k}\lceil (A_i)/c_i \rceil$. For the same $c_i' s$ and random values of $A_i' s$ we tested the accuracy of these expressions relative to the actual number of chunks that overlap the query region. We conducted such a simulation in a Linux environment, using randomly generated queries. The results show a considerable discrepancy in how close the values are to the actual count of chunks retrieved. In our experiments, we formulated random queries on 2, 3, 4 and 5 dimensions, each dimension having equal probability of access. Figure 4 shows the result of the errors relative to the actual true count of the overlapping chunks retrieved. In the legend SS_Error is that obtained with the expression 5.2.

5.1 Real Data

5.1.1 Query Workloads and Data Distribution

The next set of experiments is based on a large real data set from the Sloan Digital Sky Survey (SDSS), Data Release 1. SDSS is an astronomical survey project that maps one quarter of the entire sky in order to determine the positions and absolute brightnesses of more than 100 million celestial objects. The survey also measures the distances to more than a million galaxies and quasars.

The data set of Data Release 1 consists of 168 million records and some 500 attributes. We selected a representative subset of 4 attributes for which query workload was available for

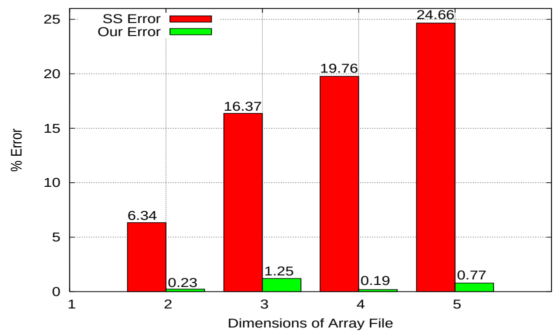


Figure 4: Bar Charts of Percentage Errors for Number of Chunks Retrieved

studying the query performance of our optimized chunking algorithms. For this purpose we did an extensive study of the real query workloads from astronomers of the SDSS collaboration over a few weeks. We extracted 5,000 queries and identified four attributes that were by far the most commonly used ones in all observed queries. Namely the variables *ra*, *dec*, *petromag_z*, etc. The variables *ra* and *dec* describe the position of celestial objects in the sky in terms of *right ascension* and *declination*, and *petromag_z* defines the *Petrosian flux*.

From the query workload we computed probability distribution of range sizes for attributes *dec, ra* shown respectively in the graphs of figures 6 and 7 respectively. We also computed the average range sizes of queries on four attributes *dec, ra, u* and *z* as decribed above. In table 7, we show the performance of our optimal chunking method for different block sizes. The results are compared with symmetric chunking, i.e., chuck shapes in which all dimensions have equal sizes. The results are also shown in the graph 5 Analysis of the SDSS cost for different block sizes is included in table 7

attrs.	**dec**	**ra**	**u**	**z**	**Opt. chunk cost**	**Sym. chunk. cost**
adj. avg. range	22.7	54.79	146.04	71.5		
blk. 2048	2	8	16	8	9755.44	15862.39
blk. 4096	4	8	16	8	5272.677	5763.278
blk. 8192	4	8	32	8	2896.653	3846.639
blk. 16384	4	8	32	16	1594.07	1961.929

Table 7: Chunk sizes for different block sizes for SDSS data

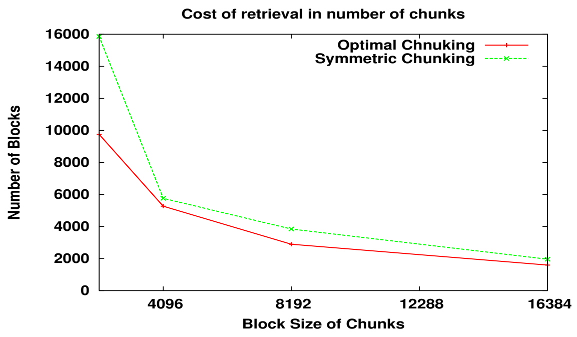


Figure 5: Comparison of Optimal and Symmetric chunking

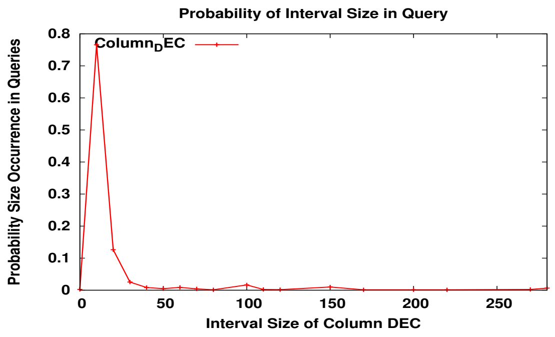


Figure 6: Probability distribution of ranges in queries for attribute dec

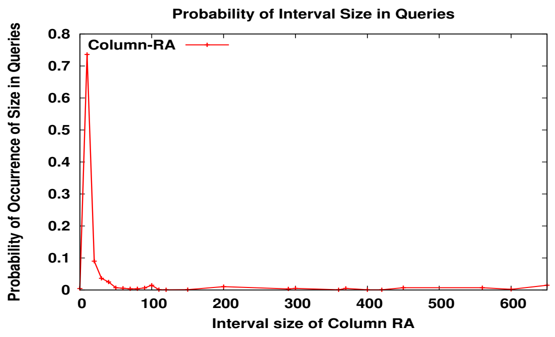


Figure 7: Probability distribution of ranges in queries for attribute ra

6 Conclusion

In high performance data intensive scientific computing and also in on-line analytical processing, multi-dimensional arrays form the principal fundamental data structure for managing the data. Array chunking constitutes the prevalent method for performing I/O between primary and secondary storage and is embodied in the prevalent file formats for array data such as HDF5. The specification of an array chunk shape that optimizes subsequent query processing is a problem that have not been adequately addressed. We have presented exact mathematical models of the problem and have presented solutions to both models with two different approaches; one using geometrical programming and the other using steepest descent optimization method. The analysis in this paper provides accurate estimations of the number of chunks that overlap hyper-rectangular query regions. There is currently lack of query workloads for driving our optimizing.

However, a synthetic workload on real data validates our analysis. Future work will include addressing the problem with factors such as array sparseness, compression and possible variable chunk sizes.

References

[1] A. V. Aho and J. D. Ullman. Optimal partial-match retrieval when fields are independently specified. *ACM Trans. on Database Syst*, 4(2):168 – 179, Jun. 1979.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 2nd edition, 1994.

[3] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, 2004.

[4] 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.

[5] 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.

[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] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *SIGMOD '90: Proc. Int'l. Conf. on Management of Data*, pages 332–342, New York, NY, USA, 1990. ACM Press.

[9] 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.

[10] A. C. McKeller and E. G. Coffman. Organizaing matrices and matrix operations for paged virtual memory. *Comm. ACM*, 12(3):153 – 165, 1969.

[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] Ekow J. Otoo and Doron Rotem. Efficient storage allocation of large-scale extendible multi-dimensional scientific datasets. In *Proc. 18th Int'l. Conf. Scientific and Statistical Database Management (SSDBM'06)*, Vienna, Austria, Jul. 3 - 5 2006. LBNL Tech Report No LBNL-61119.

[13] 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.

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

[15] 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.

[16] A. Shoshani. Olap and statistical databases: Similarities and differences. In *Proc. ACM-PODS Conf.*, pages 185–196, 1997.

[17] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209 – 271, Jun. 2001.

[18] 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.