Sparse Data Storage of Multi-Dimensional Data for OLAP and Data Mining *

Sanjay Goil Alok Choudhary
Department of Electrical & Computer Engineering and
Center for Parallel and Distributed Computing,
Northwestern University,
Technological Institute, 2145 Sheridan Road, Evanston, IL-60208
{sgoil,choudhar}@ece.nwu.edu

Abstract

Decision support systems use On-Line Analytical Processing and data mining to find interesting information from large databases. Queries posed on such systems are quite complex and require different views of data. Analytical models need to capture the multi-dimensionality of the underlying data, a task for which multi-dimensional databases are well suited. Traditional multi-dimensional databases store data in multi-dimensional arrays on which analytical operations are performed. Multi-Dimensional arrays are good to store dense data. Since most datasets are sparse, sparse storage schemes are required for efficient storage of data. However, there is a trade-off between storage and access of data. Complex operations, such as required for knowledge discovery and data mining can be very expensive in terms of data access time if efficient data structures are not used.

In this paper we introduce the bit-encoded sparse structure(BESS) for storing multi-dimensional sparse data. A comparison with other sparse data structures is presented in terms of storage efficiencies and data access times for the typical accesses required for OLAP and data mining workloads. The gain from reduced storage requirements is usually offset by overhead required in data accesses. We present an analysis of some typical OLAP operations using sparse storage schemes for higher dimensional data. This helps in highlighting the benefits of BESS. Performance results are presented which show good performance for OLAP queries using BESS over other storage schemes. We evaluate BESS with a space $\times$ time metric and find that its performance is better than the other structures. Furthermore, the analysis and performance results presented in this paper can be used by a data modeler to compare and contrast the storage versus data access performance trade-offs in choosing a suitable multi-dimensional structure for a given application.

1 Introduction

Multi-Dimensional databases have been increasingly used for On-Line Analytical processing (OLAP) and data mining since these require complex operations on data. In contrast to traditional OLTP applications, where the need is to support a large number of small transactions, mainly of the type of storing and retrieving information, the need now is to analyze large amounts of data from data repositories, called *data warehouses*. Analysis of data and discovery of patterns and trends from it can greatly help management in making better business decisions.

*This work was supported in part by NSF Young Investigator Award CCR-9357840 and NSF CCR-9509143.

Decision support systems use OLAP and data mining to find interesting information from large databases. Queries posed on such systems are quite complex and require different views of data. Analytical models need to capture the multi-dimensionality of the underlying data, a task for which multi-dimensional databases are well suited. Traditional multi-dimensional databases store data in multi-dimensional arrays on which analytical operations are performed. Multi-Dimensional arrays are good to store dense data. Since most datasets are sparse, sparse storage schemes are required for efficient storage of data. Less storage requirements will reduce the I/O time. However, there is a trade-off between storage and efficient access of data. Each cell in a multi-dimensional array can be accessed by an index in each of its dimension. Hence, finding an element in it is a trivial operation. However, if data is sparse most of the cells will remain unused resulting in wastage of memory space. On the other hand, sparse data structures can provide efficient storage mechanisms, but locating an element in the sparse data structure would require some additional work. Sparse data structures have been used in scientific computing to represent sparse matrices and perform operations on them. However, these techniques have been limited to 2 and 3 dimensions. Sparsity in higher dimensions has not received much attention in the past in that community.

Since most data in practice is sparse, multi-dimensional databases can benefit from sparse structures to efficiently store this data provided they are efficient in data retrieval. It is important to weigh the trade-offs involved in reduction in storage space offered versus the increase in access time for each sparse data structure when compared with multi-dimensional arrays. These trade-offs are dependent on many parameters some of which are the number of dimensions, sizes of dimensions and the degree of sparsity of the data. Complex operations such as required for data-driven data mining can be very expensive in terms of data access time if efficient data structures are not used.

Some indexing schemes have been reported in the literature for support of multi-dimensional sparse data. Array storage issues for sparse data have been addressed in [17]. Data compression and storage of chunk-offsets have been used within chunks of data. In Essbase [2], dimensions are split into a sparse set and a dense set. The sparse dimensions are stored in an index structure that has pointers to the corresponding multi-dimensional array for the remaining dimensions. Extensions of R-trees have been considered in [13]. The issue of sparse and dense clusters for storage in multiple dimensions is raised there. The tree searches for data on a R-tree may become cumbersome for some OLAP queries if the search is not very restrictive, leaving some of the dimensions unspecified.

In this paper we introduce a Bit-encoded sparse storage method for storing sparse data in chunks which stores the index in all dimensions and the value at that location. Retrieval of dimensional index for OLAP operation along specified dimensions can be achieved by bit operations which are comparatively inexpensive. We also present a comparative analysis with various other sparse data structures that can be used in multi-dimensional databases for handling OLAP and data mining operations. This would allow designers to evaluate the various trade-offs in terms of storage requirements and retrieval performance in selecting a suitable structure. Performance results show that BESS provides the best performance among the sparse structures and has better retrieval performance than the multi-dimensional array for densities of 2-3%, which are commonly found in real data sets. We evaluate space $\times$ time performance of BESS with multi-dimensional arrays and the Offset-Value storage, the two structures which either provide good storage or provide good data access performance. BESS provides the best performance.

Furthermore, we present analysis and performance results in this paper in terms of design parameters : number of dimensions, dimension sizes, sparsity of data, cell size, which can serve to compare and contrast the storage versus data access performance trade-offs in choosing a suitable multi-dimensional structure for a given application.

The rest of the paper is organized as follows. Section 2 describes multi-dimensional databases and arrays. Section 3 introduces the bit-encoded sparse structure. Other sparse data structures and their storage analysis is presented in Section 4. Section 5 characterizes the operations required for OLAP and data mining and motivates the query requirements for such workloads. Section 6 analyzes the performance of the various sparse data structures on some of these operations. Section 7 presents performance results and how they compare with our analysis. Conclusions are presented in Section 8.

2 Multi-Dimensional databases

The information industry has undergone a steady evolution in database power and flexibility, starting from flat files, hierarchical to relational and distributed relational technologies, data structures have evolved to match more closely the way humans visualize and work with data. A recent development in the trend has been the recognition of the usefulness of multi-dimensional databases for data analysis applications. Multi-Dimensional database technology facilitates flexible, high performance access and analysis of large volumes of complex and interrelated data [11]. It is more natural and intuitive for humans to model a multi-dimensional structure. Table 1 shows the rows of a sales database table with Model, Color and Sales as the attributes. An alternative way to represent this table-based data is shown in Figure 1. This data matrix is known as a "cross tab" view. The matrix is an example of a two dimensional array which is the fundamental structure of a multi-dimensional database. In an array each axis is called a *dimension* and each element within it is called a *position*. In Figure 1, the dimensions are MODEL and COLOR, and the positions for them are {Mini Van, Coupe, Sedan} and {Red, Blue, White, Green} respectively. The sales volume figures, called *measures* are located at the intersections of the dimension positions. The intersections are called *cells* and are populated with measures.

Table 1: Relations representing the sales volumes

Model	Color	Sales
Mini Van	Red	6
Coupe	Blue	5
Sedan	Red	4
Mini Van	Blue	5
Mini Van	White	4
Coupe	Red	3
Coupe	White	5
Sedan	Green	3
Sedan	Blue	3
Mini Van	Green	4
Coupe	Green	7
Sedan	White	2

There are several reasons for an array to be an efficient and effective means of presenting data over a relational table. Firstly, a great deal of information is gleaned immediately upon direct inspection of an array. Secondly, the array conveniently groups like information in columns and rows. This shows that a

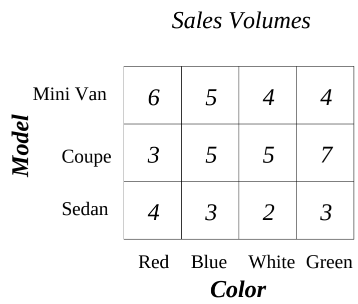


Figure 1: A two dimensional array representing sales volumes

multi-dimensional array structure represents a higher level of organization than the relational table. The structure contains valuable information regarding the relationships between the data elements because our "perspectives" are embedded directly in the structure as dimensions as opposed to being placed as fields. All possible values of each dimension are presented as positions along the dimension. This structure apart from presenting information, also has quantifiable performance advantages. As an example, if sales volume for all blue sedans is needed from the example in Table 1 then we must scroll through each individual record to identify all possible matching records. On the other hand, the multi-dimensional array structure presents the data for blue sedans in a pre-organized way. In the above discussion we have considered a 2 dimensional example for the ease of presentation. This discussion will naturally extrapolate to higher dimensions.

Since multi-dimensional arrays are the basic building blocks of a multi-dimensional database, we will discuss them next.

Table 2 describes the parameters used in the analysis presented in the following sections. These parameters control the amount of memory and the layout of data in memory for the data storage scheme under consideration. k and n_i together control the number of cells in the data. The valid data among these is controlled by the density D. c, the cell size is the amount of data stored with each combination of dimensions.

Table 2: Parameters for analysis

Parameter	Description
k	Number of dimensions
n_i	Size of dimension, $0 \leq i < k$
d_i	Dimension numbers, $0 \leq i < k$
D	Density
c	Cell size

2.1 Multi-Dimensional arrays

Multi-Dimensional arrays have been traditionally used to store multi-dimensional data. A cell position in a k-dimensional array is given by an index containing an offset in each of the k dimensions. Access to an element is done by using these index values which makes the calculation of the position of the data element very fast. A multi-dimensional array can be stored in memory in multiple ways. Row major or column major organizations are commonly used. Figure 2 illustrates row major and column major storage in memory. These organizations store the elements of a dimension contiguously, striding the other dimensions appropriately. Consider a k-dimensional array, with dimensions $d_0, d_1 \ldots d_{k-1}$ and sizes $n_0, n_1 \ldots n_{k-1}$. Let there be an ordering of the dimensions with d_0 being the inner most dimension and d_{k-1} be the outermost. For such an organization of data in memory, d_0 is stored contiguously. Adjacent elements in the multi-dimensional array will be stored with a stride of size n_0 for dimension d_1. Similarly, the other dimensions will be strided with increasing stride lengths. If the multi-dimensional array is stored on disk, data accesses to access consecutive elements in a higher dimension will require more disk reads. This will also result in poor cache performance if the data is already in memory. This effect will worsen as we access data for the outer dimensions, increasing from d_1 to d_{k-1}, as stride lengths increase. This is only true in the case where one dimension is given preference in contiguous storage over another. However, it is important to note that the stride lengths of the dimensions are known from the dimension sizes. On the other hand, chunking, where blocks of multi-dimensional data is stored together in chunks, does not favor any particular dimension for the storage of a dimension.

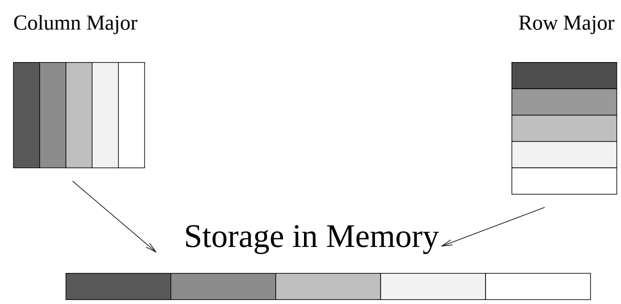
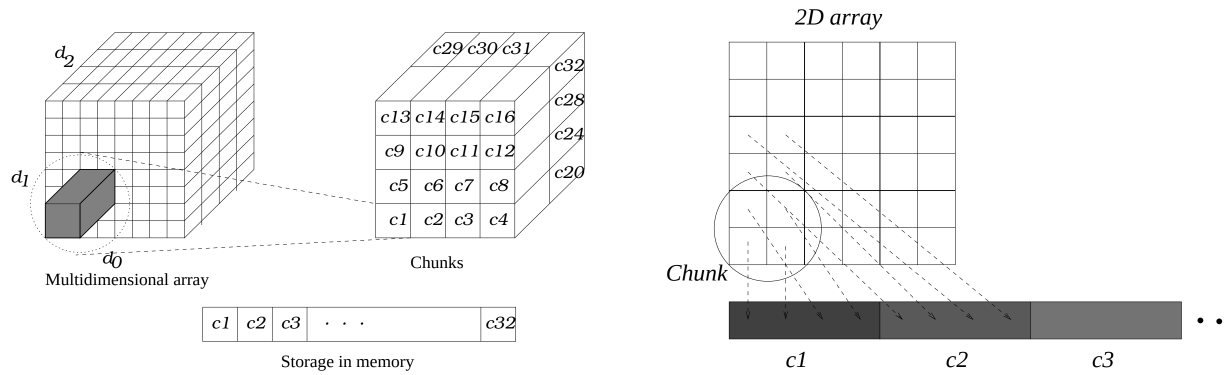


Figure 2: Row major and column major storage

A *chunk* is defined as a block of data from the multi-dimensional array which contains data in all dimensions. A collection of chunks would then define the entire array. Figure 3(a) shows chunking of a three dimensional array. A chunk is stored contiguously in memory and data in each dimension is strided with the dimension sizes of the chunk. Chunk storage for a two dimensional array is shown in Figure 3(b). Accesses for a particular dimension, say d_{k-1} in the row-major storage, do not require strided accesses of stride size $\prod_{i=0}^{k-2} d_i$. Instead, contiguous elements in that dimension will be present in each chunk strided by small amounts since the chunk stores the elements in some dimension order.

2.2 Compressed Multi-Dimensional Array

When multi-dimensional arrays are stored as chunks, each chunk can be either stored as a dense or a sparse structure. Sparse storage by chunk compression has been used in [17]. Chunks are compressed if the data density is below a certain threshold otherwise they are stored using a dense array representation. LZW compression and chunk-offset compression (an instance of the Offset-Value pair storage scheme discussed later) have been used. The saving in space by compression of multi-dimensional arrays is dependent on several factors. Firstly, it is data dependent, different data sets might yield differing compression ratios. Secondly, access of a compressed chunk is done by first decompressing it and then accessing it as a multi-dimensional array. Hence, as far as operations on the data are concerned it behaves exactly as a multi-dimensional array. Aggregation operations in OLAP and data mining will also see the invalid cells in a sparse array and will either have to aggregate appropriate NULL values or somehow bypass them. Thirdly, better compression ratios can be obtained when larger sparse chunks are compressed, but since these chunks are decompressed



(a) Chunking for a 3D array

(b) Storage in memory for a 2D chunked array

Figure 3: Storage of data in chunks

for accessing, they must fit in the available memory when they are decompressed. When operations are done on large amounts of data proper scheduling of memory usage can reduce I/O time. The storage requirements for a compressed multi-dimensional array is $\alpha c \prod_{i=0}^{k-1} n_i, 0 < \alpha < 1$. α is the compression ratio factor. α can model the average density of the chunks if chunk based compression is used. The access of data is similar to that for a multi-dimensional array.

3 Sparse Data Storage

Multi-Dimensional arrays are good structures for dense data, in which most cells are occupied by valid data elements. To store sparse data, space is wasted since many cells will remain unused. In the scientific computing community, considerable work has been done to represent sparse matrices in 2 and 3 dimensions using suitable sparse data structures. To the best of our knowledge, higher dimension sparse data structures have not received much attention in the scientific domain.

Most sparse data may not be uniformly sparse. Dense clusters of data can be found and stored as multi-dimensional arrays. Sparse data structures are needed to store the sparse portions of data. Data in a multi-dimensional database can be divided into chunks for storage. These chunks can then either be stored as dense arrays or stored using an appropriate sparse data structure as illustrated in Figure 4. Chunks also act as an index structure which helps in extracting data for queries and OLAP operations.

In this section we introduce a sparse data structure for multi-dimensional data useful for OLAP workloads in terms of storage requirements and data access performance. Typically, sparse structures have been used for advantages it provides in terms of storage, but operations on data have been performed on a multi-dimensional array which is constructed from it. However, this is not always possible when either the dimension sizes are large or the number of dimensions is large. Since we are dealing with multi-dimensional structures for a large number of dimensions, we will be interested in performing operations on the sparse structure itself. This is desirable to reduce I/O costs by having more data in memory to work on. This is one of the primary motivations for BESS.

In what follows, N_{total} is the total number of valid cells in the data. This can be controlled by adjusting

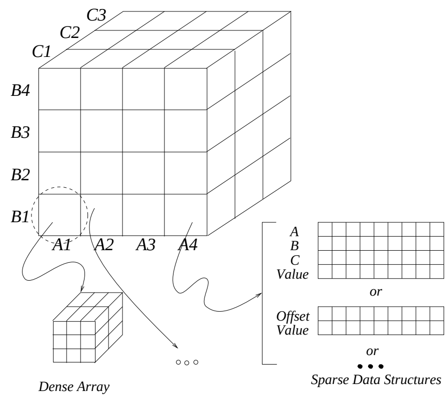


Figure 4: Dense and sparse chunk storage for the datacube

the density factor D. N_{chunks} is the average number of valid cells contained in a chunk.

3.1 Bit-Encoded Sparse Storage (BESS)

A sparse data structure that treats dimensions symmetrically will be really helpful for OLAP queries which are ad-hoc and not any dimension specific. Typical operations need fast access to index an element in each dimension to perform summary operations similar to what is implicit in a multi-dimensional array.

Bit encoded sparse storage maintains indices for a mapping of each dimension in bit encodings. Unique values in each dimension d_i can be represented by an integer value between 0 and n_i. This will require $\lceil \log n_i \rceil$ bits to represent dimension d_i. Since the dimension sizes are known, we can assign the appropriate number of bits to represent each dimension. The total storage required in this representation is $(c + \lceil \frac{\sum_{i=0}^{k-1} \lceil \log n_i \rceil}{wordsize} \rceil) N_{total}$. Figure 5 shows a storage of the indices for 3 dimensions, with $|A| = 100, |B| = 1000, |C| = 10000$, requiring 7, 10 and 13 bits respectively. These can be stored in a 4-byte integer and represents a 3-dimensional cube having its product of dimensions as 10^9. If each cell contains an integer the cube will require 3.72 GB.

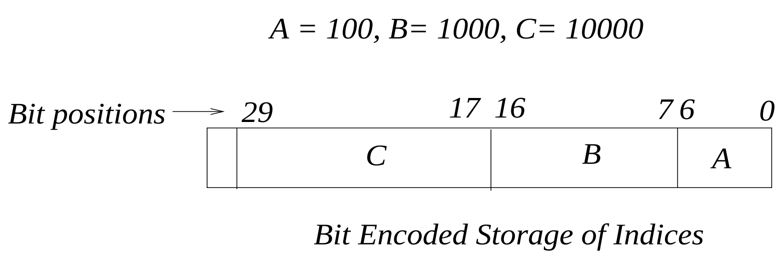


Figure 5: Bit Encoded Sparse Storage

The storage requirement for this method can be improved if chunking is used. Sizes of chunks in the various dimensions are much smaller than the total dimensions sizes, and hence can be represented in fewer number of bits. If h_i, $0 \le i < k$, is the size of a chunk in each dimension, the number of chunks is $\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil$. The storage required is $(\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil) * ((c + \lceil \frac{\sum_{i=0}^{k-1} \lceil \log h_i \rceil}{wordsize} \rceil) N_{chunk})$

Figure 6 illustrates an example for storing a two dimensional sparse chunk of size 3×2. Values $v1$ and $v2$ in chunk 0 are stored using a bit-encoded structure. Dimension x can be encoded in 2 bits since there are only 3 distinct values. Dimension y needs 1 bit to differentiate between 0 and 1. Hence 3 bits are needed for an index. Higher dimensional structures will use more bits but the minimum storage is 32 bits since data is word aligned. Thus, a large number of dimensions can be stored if 2 integers are used.

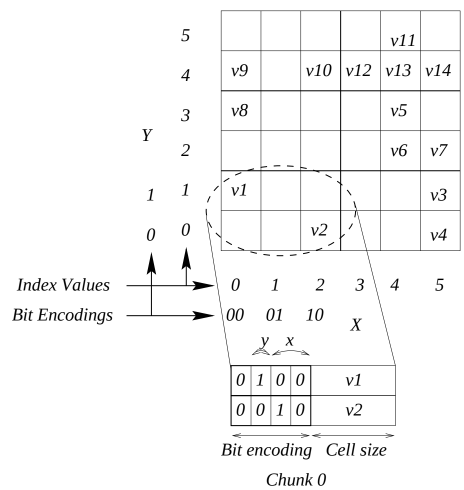


Figure 6: An example chunk storage with Bit Encoded Sparse Storage

A chunk index structure stores the chunk offsets for each chunk. Chunks are stored in memory in some order of dimensions. Without loss of generality let us assume that the order of storage is in the order X followed by Y in Figure 7. To reference an element in the chunk, first the chunk offset is dereferenced to get the dimension index values. Since the dimension extents are known for the chunk this can be done easily. The second step is to add the index value in the chunk to this by retrieving it from the bit encoding. This can be done by shifting bits in the bit-encoded structure by the number of bits and using a bit-mask to extract the bits that are used to encode each dimension.

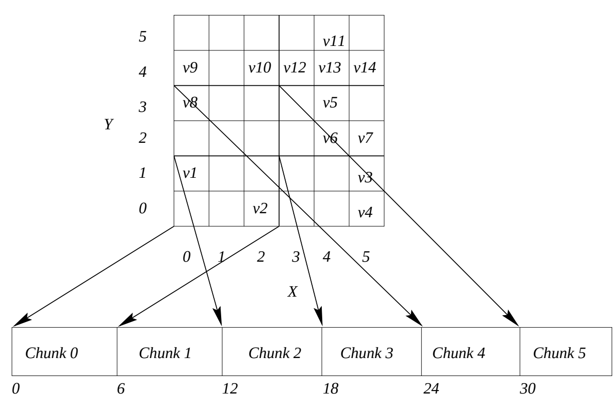


Figure 7: Storage of chunk offsets for index retrieval

Operations on BESS are described in a later section comparing them with other sparse storage schemes.

4 Other Sparse Storage Schemes

Next, we look at sparse data structures reported in the literature for multi-dimensional data and compare them in terms of storage requirements and data access performance with the bit-encoded sparse storage structure introduced in the previous section. The analysis parameters are the same as described in Table 2.

4.1 Index-Value pairs

The simplest scheme to store sparse data is the index of a position in a multi-dimensional array together with data. Figure 8(a) shows this form of sparse storage for a 2-dimensional array. Each entry stores a value of each of the dimensional index of the array along with the value. Without imposing any restriction on the values for the sizes of the dimensions an integer value is needed for each dimension to store its index. Hence $(c+k)$ storage is required for each cell. The total storage is hence $(c+k)N_{total}$.

The performance of this storage scheme depends on the density D, and the relation between c and k. For a low D, and if $c \geq k$, no more than twice the space required for the cells is needed. As D increases and with $k > c$, the effects of storing k integer values for the dimension indices will result in poor storage.

Storage by chunks can be exploited for this method, by storing the sparse chunks as Index-Value pairs. This would require $(c+k)N_{chunk}$ storage for each sparse chunk.

4.2 Offset-Value pairs

Offset-Value pair stores calculates an offset value from the start of memory location where the multi-dimensional array is stored. This offset value is stored along with each data cell (Figure 8(b)). Hence only one 32-bit integer value is required to store the offset. The data storage requirement is $(c+1)N_{total}$.

The Offset-Value pair storage scheme linearizes the multi-dimensional representation by calculating the offset. Accesses to data values usually require the values of the various dimensional index in multi-dimensional operations. These can be obtained from the offset value by performing suitable operations. These would typically involve a division, a multiply and a subtract operation using the dimension sizes.

Chunking can be used to store sparse data using this scheme similar to the one described for BESS. This would require $(c+1)N_{chunk}$ storage per chunk.

This method has the lower overhead in terms of storage costs. Since it stores an offset from the start of the chunk as an index, computation needs to be done on the offset to retrieve each of the dimension index values. This is often required in queries which perform operations along a dimension or a combination of dimensions. An operation involving a divide, multiply and a subtract is done to retrieve each dimension.

4.3 Compressed Sparse Dimensions

Compressed Sparse Row (CSR) storage has been extensively used for operations on sparse matrices in scientific computing. This is an extension of CSR for higher dimensions. Each data element is stored contiguously in some order of its dimensions. See Figure 8(c). Suppose the order of dimensions is from $d_0 \ldots d_{k-1}$ from innermost to outermost. The valid cells of the array are stored contiguously in this dimension order. k levels of index values are maintained to dereference these values. All dimension indices are stored for d_0. Indices into this array are stored for d_1, pointing to the offsets for the values in this dimension. An example is illustrated in Figure 8. In this scheme the order of the dimensions affect the storage requirements for these dimension indices. The storage required for the indexes is calculated as follows. The storage for d_0 is N_{total} elements. For d_1, $\prod_{i=1}^{k-1} n_i$ entries are needed. For each subsequent d_j, $\prod_{i=j}^{k-1} n_i$ entries are stored. Data size is cN_{total}. Hence, the total storage for this scheme is $(c+1)N_{total} + \sum_{j=1}^{k-1} \prod_{i=j}^{k-1} n_i$.

Consider the dimensions to be sorted in non-decreasing order of their sizes, $n_0 \leq n_1 \leq \ldots \leq n_{k-1}$. Now consider two extreme cases, one with dimension ordering from $d_0, d_1 \ldots d_{k-1}$ and another in the order $d_{k-1}, d_{k-2} \ldots d_0$. The storage requirements for the index when d_0 is the innermost dimension and d_{k-1} is the outermost is $\sum_{j=1}^{k-1} \prod_{i=0}^{k-j-1} n_i$. On the other hand, the storage required for a dimension ordering with d_{k-1} as the innermost dimension and d_0 as the outermost is $\sum_{j=1}^{k-1} \prod_{i=j}^{k-1} n_i$. Clearly, $\sum_{j=1}^{k-1} \prod_{i=0}^{k-j-1} n_i \leq \sum_{j=1}^{k-1} \prod_{i=j}^{k-1} n_i$ for $n_0 \leq n_1 \leq \ldots \leq n_{k-1}$. Consider an example with $k = 4, n_0 = 10000, n_1 = 1000, n_2 = 100, n_3 = 10$. For the dimension order d_{k-1} (innermost) to d_0 (outermost) the storage for the indexes is $n_0 \times n_1 \times n_2 + n_0 \times n_1 + n_0$. Thus, $10^9 + 10^7 + 10^4$ entries are needed for this. We are not considering the storage for the innermost dimension because an index is kept for each data element which is the same for both dimension orderings. For a dimension ordering of d_0 (innermost) and d_{k-1} (outermost) the number of entries required are $n_1 \times n_2 \times n_3 + n_2 \times n_3 + n_3$, which is $10^6 + 10^3 + 10$. Keeping the largest dimension as the innermost dimension and progressively smaller ones as outer dimensions is the better choice for this scheme.

Data access performance for this structure is presented in a later section.

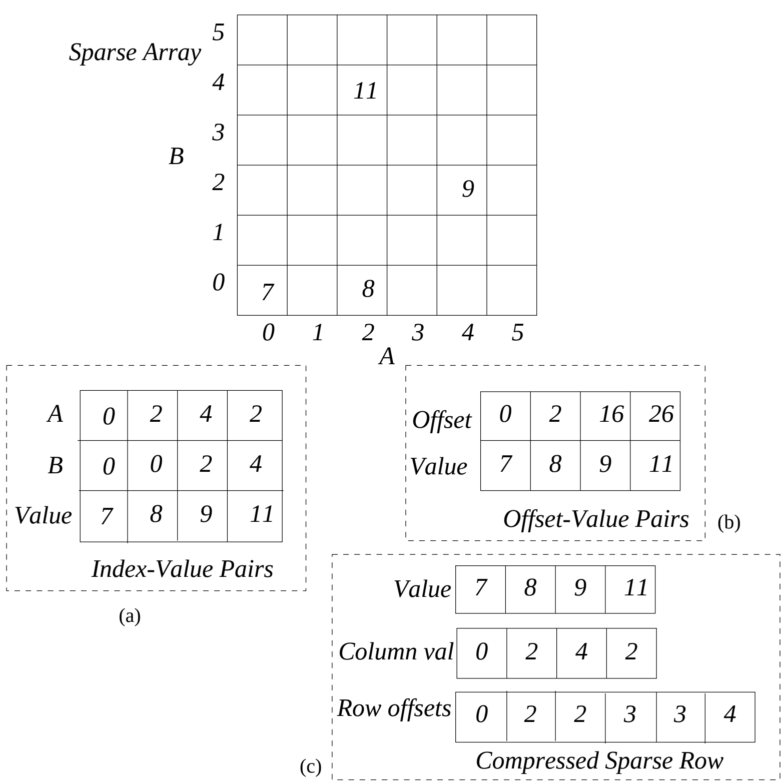


Figure 8: Sparse data structures for a sample sparse array: Index-Value pairs, Offset-Value pair and Compressed Sparse Dimensions format

4.4 Sparse-Dense split storage

This method has been used in *Essbase*, a multi-dimensional OLAP product from ArborSoft [2] [15]. Dimensions are split into dense and sparse dimensions. The dense dimensions are stored as a multi-dimensional array blocks. The sparse dimensions act as an index into the dense blocks. Among the many possible ways to store the sparse dimensions two methods have been elaborated as illustrated in Figure 9. Either each combination of the sparse dimensions is stored, each valid combination having a pointer to its data in the dense array or a binary tree like index structure is maintained, each node of which has pointers to the corresponding dense block. We will consider the first scheme to store sparse dimensions which stores all possible combinations. Without loss of generality, consider the first j dimensions $d_i, 0 \leq i < j$ to be dense. The storage required for this is $c\prod_{i=0}^{j-1} n_i$. The remaining $k-j$ dimensions are sparse and are stored in an index structure.

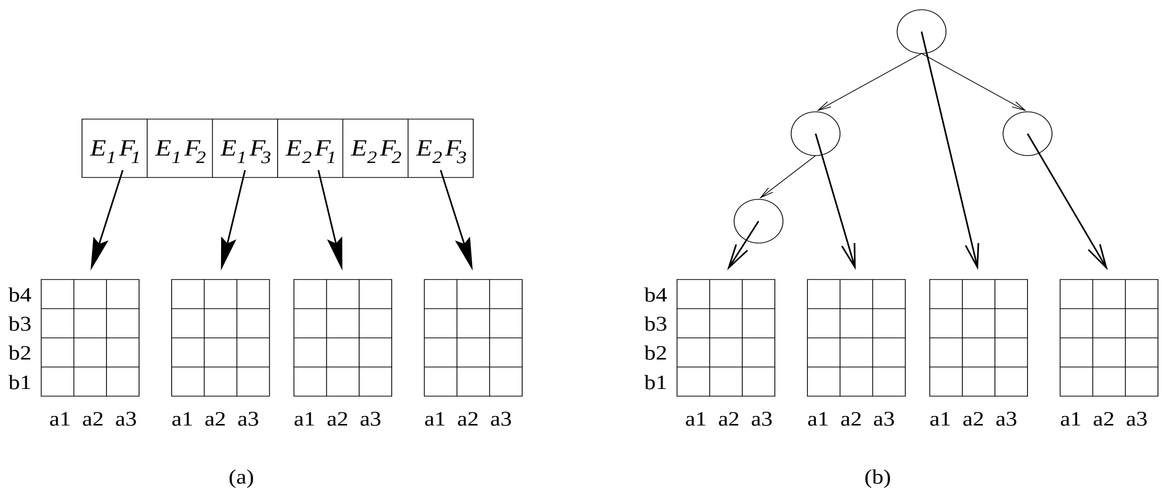


Figure 9: Splitting sparse and dense dimensions (a) block pointer array structure (b) binary tree index structure

One method stores pointers from all the possible combinations of the sparse dimensions to data blocks of the dense dimensions. Storage for a pointer is required for all combinations, but only those combinations that point to data are actually useful. Storage for the sparse dimensions when all combinations are stored is $\prod_{i=j}^{k-1} n_i$. Let D^* is the density of the chosen sparse dimensions. Then $D^* \prod_{i=j}^{k-1} n_i$ have pointers to dense data. The storage requirement for this structure is $(D^* \prod_{i=j}^{k-1} n_i) * c\prod_{i=0}^{j-1} n_i = (cD^* \prod_{i=0}^{k-1} n_i)$.

The second method stores the sparse dimensions in a binary tree. Each node of the tree points to a dense data block. This has the advantage of having entries for those combinations that have data associated with them. However, now a pointer for the section number, left child, right child and the data block have to be stored. Additionally, a search over the binary tree has to be made to access the right data block. Periodic balancing of the binary tree might be required during insertion of data. The storage requirements for it are $(D^* \prod_{i=j}^{k-1} n_i) * (4 + c\prod_{i=0}^{j-1} n_i)$.

We will consider method 1 in our analysis of this structure. It is not always easy to split dimensions into sparse and dense dimensions. Some statistical techniques may have to be employed to get a good split. Also, this structure treats the dimensions asymmetrically. It is easier to find elements if the sparse dimensions are specified and the dense dimensions are left unspecified. This is true because an index structure is present for the sparse dimensions. However, a search made with a few dense dimensions specified and the sparse ones left unspecified, will be expensive.

Table 3 summarizes the storage requirements for each sparse data structure. A comparison of the storage requirements shows that the Offset-Value pair and BESS are the best in terms of storage requirements for higher values of k. The growth rate of storage for other structures is much higher for an increasing k.

Table 3: Storage required for the various sparse storage formats

Method	*Storage*
Multi-Dimensional Array	$c \prod_{i=0}^{k-1} n_i$
Compressed Array	$\alpha Dc \prod_{i=0}^{k-1} n_i, 0 < \alpha < 1$
Bit-Encoded Sparse Storage	$(\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil) * (c + \lceil \frac{\sum_{i=0}^{k-1} \lceil \log h_i \rceil}{wordsize} \rceil) N_{chunk}$
Index-Value pair	$(\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil) * (c + k) N_{chunk}$
Offset-Value pair	$(\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil) * (c + 1) N_{chunk}$
Compressed Sparse Dimensions (extension of CSR)	$(c+1) N_{total} + \sum_{j=1}^{k-1} \prod_{i=j}^{k-1} n_i$
Sparse-Dense Split Storage	$(D^* c \prod_{i=0}^{k-1} n_i)$

5 Data Access Operations in OLAP and Data Mining

OLAP is used to summarize, consolidate, view, apply formulae to, and synthesize data according to multiple dimensions. Queries posed on such systems are quite complex and require different views of data. Traditionally, a relational approach (relational OLAP) has been taken to build such systems. Relational databases are used to build and query these systems. A complex analytical query is cumbersome to express in SQL and it might not be efficient to execute. More recently, multi-dimensional database techniques (multi-dimensional OLAP) have been applied to decision-support applications. Data is stored in multi-dimensional arrays which is a more natural way to express the multi-dimensionality of the enterprise data and is more suited for analysis. A "cell" in multi-dimensional space represents a tuple, with the attributes of the tuple identifying the location of the tuple in the multi-dimensional space and the measure values represent the content of the cell.

Data mining can be viewed as an automated application of algorithms to detect patterns and extract knowledge from data [4]. An algorithm that enumerates patterns from, or fits models to, data is a data mining algorithm. Data mining is a step in the overall concept of knowledge discovery in databases (KDD). Large data sets are analyzed for searching patterns and discovering rules. Automated techniques of data mining can make OLAP more useful and easier to apply in the overall scheme of decision support systems. Data mining techniques like *Associations, Classification, Clustering and Trend analysis* [4] can be used together with OLAP to discover knowledge from data.

5.1 Data Cubes

Data cube computes aggregates along all possible combinations of dimensions [3]. This operation is useful for answering OLAP queries which use aggregation on different combinations of attributes. Data can be organized into a data cube by calculating all possible combinations of GROUP-BYs. For a data set with k attributes this would lead to 2^k GROUP-BY calculations. A data cube treats each of the k aggregation attributes as a dimension in k-space. An aggregate of a particular set of attribute values is a point in this space. The set of points form a k-dimensional cube.

Data Cube operators generalize the histogram, cross-tabulation, roll-up, drill-down and sub-total constructs required by financial databases. The following operations can be defined on the data cube.

- **Pivoting:** This is also called *rotating* and involves rotating the cube to change the dimensional orientation of a report or page on display. It may consist of swapping the two dimensions (row and column in a 2D-cube) or introducing another dimension instead of some dimension already present in the cube.
- **Slicing-dicing :** This operation involves selecting some subset of the cube. For a fixed attribute value in a given dimension, it reports all the values for all the other dimensions. It can be visualized as *slice* of the data in a 3D-cube.
- **Roll-up :** Some dimensions have a hierarchy defined on them. Aggregations can be done at different levels of hierarchy. Going up the hierarchy to higher levels of generalization is known as roll-up. For example, aggregating the dimension up the hierarchy ($day \rightarrow month \rightarrow quarter..$) is a roll-up operation.

- **Drill-down:** This operation traverses the hierarchy from lower to higher levels of detail. Drill-down displays detail information for each aggregated point.

- Trend analysis over sequential time periods

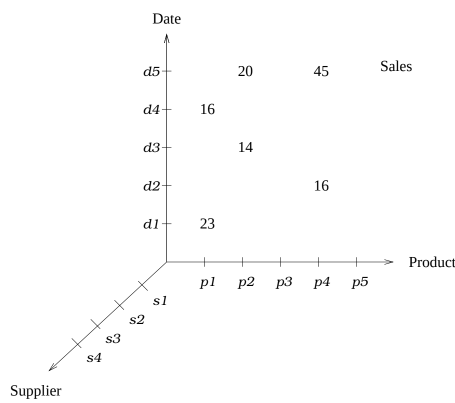


Figure 10: Example database of sales data

Figure 10 shows a multi-dimensional database with *product*, *date*, *supplier* as *dimensions* and *sales* as a *measure*. Partitioning a data set into dimensions and measures is a design choice. Dimensions usually have a hierarchy associated with them, which specify aggregation levels and the granularity of viewing data. For example $day \rightarrow month \rightarrow quarter \rightarrow year$ is a hierarchy on date.

Data mining techniques are used to discover patterns and relationships from data which can enhance our understanding of the underlying domain. Discovery of quantitative rules is associated with quantitative information from the database. The data cube represents quantitative summary information for a subset of the attributes. Attribute-oriented approaches [5] [9] [10] to data mining are data-driven and can use this summary information to discover association rules.

5.2 Basic Operations

The operations needed for OLAP and data mining require access to data along a particular dimension or a combination of dimensions. Usual operations are of the type of aggregation along a particular attribute, which is represented by a dimension in the multi-dimensional database. The set of operations required for OLAP and data mining should be efficiently supported by the sparse data structure for good performance. The operations fall under the following categories.

1. **Retrieval of a random cell element**

   A single cell is retrieved by specifying the values of all dimensions. For a k-dimensional cube, the dimensions are specified where d_1 takes value val_1, d_2 takes value val_2 and so on, represented by $(val_1, val_2, \ldots val_k)$ dimension vector.

2. **Retrieval along a dimension or a combination of dimensions**

   If a dimension or a combination of dimensions is unspecified, all values are returned for that dimension for the combination of the specified values. As an example, consider the case when d_1 and d_2 are unspecified and the rest of the dimensions are assigned values, represented by $RETRIEVE(*, *, val_3, \ldots val_k)$. A 2 dimensional plane, with all values in dimensions d_1 and d_2, is retrieved for the specific values of the other dimensions. Any m of the k dimensions may remain unspecified and this will return values for a m dimensional hypercube. Figure 11 is an illustration of retrieval along dimensions d_1 and d_2 done by fixing a value for dimension d_0 to 5, specified as $RETRIEVE(5, *, *)$.

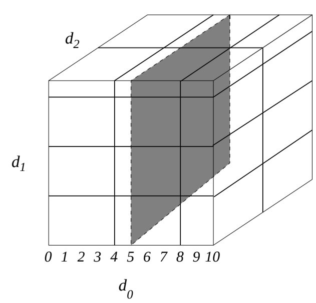


Figure 11: Retrieval along d_1 and d_2 by restricting dimension $d_0 = 5$

3. **Retrieval for values of dimensions within a range (Range Queries)**

   Often values are needed between a range in several dimensions. If d_1 is constrained between val_{11} and val_{12} and the rest of the dimensions are specified at a certain value, represented by $(val_{11} : val_{12}, val_2, \ldots, val_k)$. Any m of the k dimensions can have a range specified and values will be retrieved within that range. This is a m-dimensional range query. Figure 12 illustrates a range query in 3 dimensions, specified as $RETRIEVE(3 : 6, 2 : 10, 0 : 6)$.

4. **Aggregation operations on dimensions**

   Summarization of data along a particular dimension or combination of dimensions is often required. The aggregation can be for a single dimension or a combination of dimensions. Typically, in the construction of a data cube, a single dimension is aggregated from one level to another. To maintain a full data cube, aggregation of a single dimension is done for each dimension for the cubes present at a level. For any

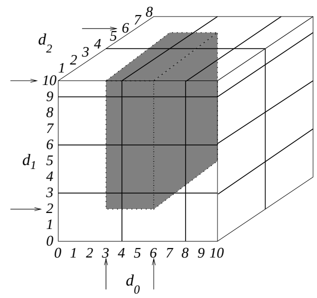


Figure 12: Range query in 3 dimensions, defining a range in all dimensions

specified dimension j, values of cells along dimension j are aggregated using an aggregation operation. Each cell along dimension j is aggregated resulting in a $k-1$ dimensional cube. This access pattern is different from the one in which retrieval is done along dimension j, because here all values are accessed to apply the aggregation operation whereas in the previous operation only the particular values for dimension j are accessed in the remaining $k-1$ dimensions. Figure 13 illustrates aggregation for a 3 dimensional cube along dimension d_0, represented as $AGGREGATE(*, d_1, d_2)$.

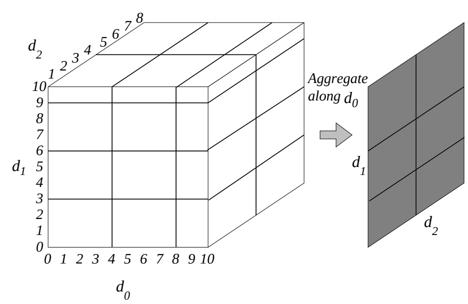


Figure 13: Aggregate along a single dimension (d_0) for a 3 dimensional cube

5. **Multi-dimensional Aggregation (Generalization/Consolidation: lower to higher level in hierarchy)**

   Another type of aggregation is one that aggregates values in all dimensions. This is required for ascending a generalization hierarchy in which a k dimensional section is aggregated to get a single value for a cell at a higher level in the hierarchy. Alternatively, a subset of dimensions can be aggregated to reduce dimensions and to answer specific queries. This is a generalized case of the previous operations, where the aggregation was on a single dimension. The operational complexity can perhaps be reduced if multiple dimensions are aggregated together. This aggregation can be represented as $AGGREGATE(d_0 : level, d1 : level, \ldots, d_n : level)$, where $level$, denotes the level of aggregation for the dimension. Figure 14 illustrates an aggregation in 3 dimensions, for a single cell, denoted by $AGGREGATE(d_0 : 1, d_1 : 1, d_2 : 1)$.

In the next section we will study the implications of a sparse data structure on each of these operations and provide an analysis for each operation.

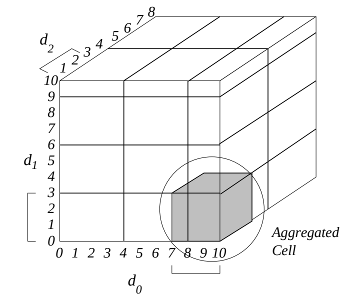


Figure 14: Aggregate along all dimensions

6 Data Access Performance

Values of indices in the various dimensions are implicit in a multi-dimensional array but are not so in a sparse representation of data. Sparse data schemes need to be evaluated for their efficiency to perform these data access operations. The various cost parameters required for such an analysis are summarized in Table 4. The cost of the operations are from a 132 MHz, 64MB main memory, HP B132L and a 120 MHz, 96MB main memory, IBM RS6000/E30 workstation. We used the native C compiler on each machine without any loop optimizations to run a million iterations of each operation and factored out the loop cost. This gives us an idea on the relative performance of the operations and where we can look for optimizations in terms of operations.

Table 4: Typical costs of operations on a workstation

Cost	Operation	HP B132L	RS6000/E30
t_{mem}	Memory access	101 ns	123 ns
t_{add}	Addition (integer)	45 ns	8 ns
t_{comp}	Comparison	23 ns	8 ns
t_{mul}	Multiply (integer)	129 ns	16 ns
t_{div}	Division (integer)	648 ns	166 ns
t_{mod}	Modulo	701 ns	111 ns
t_{shift}	Shift	30 ns	16 ns
t_{and}	And	45 ns	16 ns
t_{or}	Or	45 ns	16 ns
t_{op}	Aggregate Operation	–	–

6.1 Operations

In the following sections each basic operation is analyzed with respect to BESS and sparse data structures discussed previously.

6.1.1 Retrieval of a random cell element

A random element can be retrieved if we have its index. For k-dimensional data, this is a specification of the type $(d_0 = val_0, d_1 = val_1, \ldots, d_{k-1} = val_{k-1})$. Table 5 presents the costs of retrieving an element for each type of structure.

For a multi-dimensional array, an offset is created from the start of the array by using each val_i and n_i, $0 \leq i < k$ to get the correct offset in each dimension. $k-1$ multiply and add operations are needed.

Table 5: Costs for retrieval of a random element

Structure	Cost
Multi-dimensional array	$(k-1)(t_{mul} + t_{add})$
BESS	$\sum_{i=0}^{k-1} O(\log\lceil \frac{n_i}{h_i} \rceil) + \frac{1}{2}[t_{mem} + k(t_{shift} + t_{and} + t_{comp})]N_{chunk}$
Index-Value	$\sum_{i=0}^{k-1} O(\log\lceil \frac{n_i}{h_i} \rceil) + \frac{1}{2}[k(t_{mem} + t_{comp})]N_{chunk}$
Offset-Value	$\sum_{i=0}^{k-1} O(\log\lceil \frac{n_i}{h_i} \rceil) + \frac{1}{2}[t_{mem} + (k-1)(t_{div} + t_{mul} + t_{sub} + t_{comp})]N_{chunk}$
Compressed Sparse Dimension	$k(t_{mul} + t_{mem}) + t_{comp} \log n_0$
Sparse-Dense Split storage	$(k-1)(t_{mul} + t_{add})$

For BESS, the appropriate chunk has to be found in each dimension. This can be obtained by k binary searches, one for each dimension. For n sorted elements, a binary search takes $O(\log n)$ time. In the desired chunk, the dimension index values are decoded and compared with the needed val_is. This requires k bit and operations, k shift operations and comparisons for each element. On an average half the elements in the sparse chunk are searched.

For the Index-Value pair, the cost for searching the chunk is the same as BESS since both use the chunk index structure. However, within the sparse chunk, dimension index values are available and can be directly compared with val_i. Hence, k fetch and comparison operations are needed for each element.

The Offset-Value pair stores an offset in the chunk. To get the values of each dimension index $k-1$ division, multiply and subtract operations are required for each element. As can be seen from Table 4, this can be an expensive operation and has to be done for each entry in the chunk.

The index values in each dimension are already known in the compressed sparse dimension. Starting at the index of the outermost dimension, the indices are followed to the innermost dimension. Here a search for the index in the innermost dimension has to be made. The split storage scheme behaves like the multi-dimensional array for this access. The correct pointer to the dense block in the sparse dimensions index can be found by add and multiply operations on the dimension values.

Comparing costs in Table 5, retrieval of a random element is the fastest for multi-dimensional array and the split storage schemes. This is because elements are stored at regular strides and it is easy to index into the desired element. Compressed sparse dimension also has pointers set for all dimensions and following the appropriate k level pointers a random element can be retrieved. For the other three schemes, the decoding of the dimension index values is not needed for the Index-Value pairs, BESS provides with fast decoding of the dimension index values and Offset-Value pair is the slowest.

6.1.2 Retrieval along a dimension

Let $j, 0 \leq j < k$ be the dimension specified. Retrieval of values will restrict dimension j with a value val_j. This will return a $k-1$ dimensional hypercube if the operation is performed on a k dimensional hypercube. Table 6 shows the costs for this operation. For a multi-dimensional array, a binary search over the sorted list of the unique values in the dimension j is needed. This takes $O(\log n_j)$ time. Using this base address, the elements are retrieved using a multiply and an add over the remaining $k-1$ dimensions.

When using BESS, the chunk containing val_j is found by searching the sorted chunk list for dimension j using binary search. A chunk's base offset is converted into a dimensional index for each dimension. This is added to the dimension index within the chunk which is retrieved from BESS by using k bit and operations and k bit shift operations. This is done for each chunk in the $k-1$ dimensional plane. A similar analysis holds for the Index-Value pair, except that the dimension index values are already available. For the Offset-Value pair, the dimension index have to be calculated from the offset. This takes $k-1$ division, multiply and subtract operations for each element. These are computationally expensive operations when compared to the bit shift used by BESS.

The compressed sparse dimension stores pointers in a dimension ordering from the outermost to the innermost. These can be used to retrieve the data and requires traversing k levels of pointers. There are two cases. First, when j is not the innermost dimension. In this case a value for an intermediate dimension is known and only pointers from that value need to be followed. n_{avg} is the average number of elements found in the innermost dimension. In the second case, j is the innermost dimension. Each element found in the innermost dimension has to be compared with the value specified in this case.

The split storage structure can either have j belong to the sparse dimensions or in dense dimensions. In the former case, it is easy to calculate which index elements will belong to the set of possible block pointers that need to be followed to the dense blocks. When j belongs to the dense dimensions, each index element in the dense dimension is followed and the restriction is made for each dense block. Addition and multiply are the operations to do the pointer math. Again, D^* is the density in the sparse dimensions.

Table 6: Costs for retrieval along a dimension $j, 0 \leq j < k$

Structure	Cost
Multi-dimensional array	$(t_{mem} + t_{comp}) \log n_j + (t_{mem} + t_{mul} + t_{add}) \prod_{i=0,i\neq j}^{k-1} n_i$
BESS	$(t_{mem} + t_{comp}) \log \lceil \frac{n_j}{h_j} \rceil +$ $(\prod_{i=0,i\neq j}^{k-1} \lceil \frac{n_i}{h_i} \rceil)[k(t_{div} + t_{mul} + t_{sub}) + (t_{mem} + t_{shift} + t_{and} + t_{comp}))N_{chunk}]$
Index-Value	$(t_{mem} + t_{comp}) \log \lceil \frac{n_j}{h_j} \rceil +$ $(\prod_{i=0,i\neq j}^{k-1} \lceil \frac{n_i}{h_i} \rceil)[(t_{mem} + t_{comp})N_{chunk}]$
Offset-Value	$(t_{mem} + t_{comp}) \log \lceil \frac{n_j}{h_j} \rceil +$ $(\prod_{i=0,i\neq j}^{k-1} \lceil \frac{n_i}{h_i} \rceil)[(t_{mem} + t_{comp} + 2(t_{div} + t_{mul} + t_{sub}))N_{chunk}]$
Compressed Sparse Dimension (j is not innermost)	$t_{mem} n_{avg} \prod_{i=1,i\neq j}^{k-1} n_i$
(j is innermost)	$(t_{mem} + t_{comp}) n_{avg} \prod_{i=1}^{k-1} n_i$
Sparse-Dense Split storage ($j \in s$)	$(D^* \prod_{i=0,i\neq j}^{s-1} n_i) * (t_{mem} + t_{add} + t_{mul}) \prod_{i=s}^{k-1} n_i$
($j \notin s$)	$(D^* \prod_{i=0}^{s-1} n_i)(t_{mem} + t_{add} + t_{mul}) \prod_{i=s,i\neq j}^{k-1} n_i$

From the cost expressions in Table 6, multi-dimensional array has the best performance. However, the cost for accesses will increase as the strides of accesses increase, that is as the dimension order to be retrieved increases from innermost to outermost. Compressed sparse dimension is efficient when j is not innermost and so is the split storage scheme. For the other three schemes which operate on chunks, Index-Value has the least overhead of processing within a chunk, followed by BESS and then Offset-Value pair. For low values of N_{chunk}, the operation count is much less for the sparse data structures and hence might provide better retrieval performance than the multi-dimensional array.

6.1.3 Retrieval for values of dimensions within a range (Range Queries)

Range queries on multi-dimensional arrays require retrieving values between the upper limit u_i and the lower limit l_i of the dimensions in which the range is specified. Let r include the dimensions in which the range is specified. Clearly, $|r| \leq k$. In the other structures, the operation is not so straightforward. In BESS, the chunk containing u_i and the chunk containing l_i in all specified range dimensions are identified from the chunk index structure. Values are then retrieved from the chunks within this range and for all chunks in the unspecified dimensions. Checks for ranges are needed for elements in the boundary chunks only. A similar analysis hold true for the Index-Value and the Offset-Value structures. The compressed sparse dimensions will use the pointers within the range specified for the dimensions. For the split storage structure, if the range is specified only in the sparse dimensions then the index structure is restricted to follow pointers only in the range selected and elements from the dense block are retrieved for all the valid pointers in the range. Otherwise, if the range is specified in the dense dimension then all the valid pointers to the dense blocks are used and the values are retrieved within the range from the dense blocks. s is the set of sparse dimensions and the set $k - s$ contains the dense dimensions.

Table 7: Costs for range query between l_i and u_i, $i \in r$.

Structure	Cost
Multi-dimensional array	$(t_{mem} + t_{add}) \prod_{i \in r} (u_i - l_i) * \prod_{i \in k-r} n_i$
BESS	$2(t_{mem} + t_{comp}) \sum_{i \in r} \log \lceil \frac{n_i}{h_i} \rceil +$ $\prod_{i \in r} (\lceil \frac{u_i - l_i}{h_i} \rceil \prod_{i \notin r} \lceil \frac{n_i}{h_i} \rceil * [k(t_{div} + t_{mul} + t_{sub}) + (2t_{mem} + k(t_{shift} + t_{and} + t_{comp}))N_{chunk}]$
Index-Value	$2(t_{mem} + t_{comp}) \sum_{i \in r} \log \lceil \frac{n_i}{h_i} \rceil +$ $\prod_{i \in r} \lceil \frac{u_i - l_i}{h_i} \rceil \prod_{i \notin r} \lceil \frac{n_i}{h_i} \rceil * [k(t_{div} + t_{mul} + t_{sub}) + k(t_{mem} + t_{comp})N_{chunk}]$
Offset-Value	$2(t_{mem} + t_{comp}) \sum_{i \in r} \log \lceil \frac{n_i}{h_i} \rceil +$ $\prod_{i \in r} \lceil \frac{u_i - l_i}{h_i} \rceil \prod_{i \notin r} \lceil \frac{n_i}{h_i} \rceil * [k(t_{div} + t_{mul} + t_{sub}))(N_{chunk} + 1)]$
Compressed Sparse Dimensions	$\prod_{j=1, j \in r}^{k-1} (u_i - l_i) \prod_{j=1, j \notin r}^{k-1} n_i * (t_{mem} + t_{comp}) n_{avg}$
Sparse-Dense Split storage (range only in sparse dimensions)	$D^* \prod_{i \in s} (u_i - l_i)[t_{comp} + \prod_{i \in k-s} n_i * (t_{mem} + t_{add})]$
(range in dense dimensions)	$D^* \prod_{i \in s} n_i [t_{comp} + \prod_{i \in k-s} (u_i - l_i) * (t_{mem} + t_{add})]$

Comparison of cost expression from Table 7 show us that the range query is the most efficient on the multi-dimensional array. Range search for split-storage searches for the valid pointers in the sparse structure and then restricts the dense blocks on the ranges specified in the dense dimension. For operations within chunks, the Index-Value pair is better than BESS which is better than the Offset-Value pair storage. However, for low densities, the sparse structure can perform better than the multi-dimensional array.

6.1.4 Aggregation of a dimension

Each element is accessed and aggregated to the resultant structure with the same dimension value in the appropriate dimension. For a multi-dimensional array, the resultant structure is an array without the dimension which is to be aggregated. The result is initialized with zeros and each element is added to the result in the appropriate dimension. For a k dimensional cube aggregation of a single dimension results in a $k - 1$ dimensional cube. Table 8 compares the cost of aggregating a dimension.

For the BESS, operations are performed for each chunk. First the chunk dimension index values are calculated from its offset. For each element in the chunk, the appropriate dimension is retrieved from the encoded structure by bit shifting. This is aggregated to the result chunk which is also stored in BESS format. The dimension which is being aggregated is removed from the bit representation by appropriately shifting the remaining bits. The resultant bit representation has $k-1$ dimensions and values prior to the aggregation. The list of values in the chunk is then sorted on this bit representation. This gets the values that need to be aggregated adjacent to each other. A traversal of the list is then made, aggregating adjacent values which have the same bit representation. This takes $O((t_{shift}+t_{and}+t_{op}+\log N_{chunk})N_{chunk})$. Index-Value pair has the dimension index readily available which is directly used. For the Offset-Value pairs, k mod and subtract operations gets the correct index. A strategy similar to the one described for BESS is then used for these.

For storage in compressed sparse dimensions, the index structure is traversed from the outermost to innermost dimension. For the dimension to be aggregated, its index is traversed in a strided manner and values retrieved from following the pointers are aggregated if they match on the innermost dimension values.

In sparse-dense split storage if a sparse dimension is to be aggregated, then the aggregate operator is applied to the corresponding dense block pointers along the dimension to be aggregated. For aggregating a dense dimension, the aggregation operator is applied to the dense blocks and the sparse index is unaffected.

Table 8: Costs of aggregating a dimension j

Structure	Cost
Multi-dimensional array	$(t_{mem}+t_{add}+t_{op})\prod_{i=0}^{k-1} n_i$
BESS	$\prod_{i=0}^{k-1}\lceil\frac{n_i}{h_i}\rceil * [k(t_{div}+t_{mul}+t_{sub})+(t_{mem}+t_{shift}+t_{and}+t_{op}+\log N_{chunk})N_{chunk}]$
Index-Value	$\prod_{i=0}^{k-1}\lceil\frac{n_i}{h_i}\rceil * [k(t_{div}+t_{mul}+t_{sub})+(kt_{mem}+t_{op}+\log N_{chunk})N_{chunk}]$
Offset-Value	$\prod_{i=0}^{k-1}\lceil\frac{n_i}{h_i}\rceil * [k(t_{mem}+t_{div}+t_{mul}+t_{sub})+t_{op}+\log N_{chunk}]N_{chunk}$
Compressed Sparse Dimensions	$\sum_{i=0}^{k-1}\prod_{l=i}^{k-1} n_l * (t_{mem}+t_{comp}+t_{op})n_{avg}$
Sparse-Dense Split storage	$(D^*\prod_{i\in s} n_i)[t_{comp}+\prod_{i\in k-s} n_i * (t_{mem}+t_{add}+t_{op})]$

6.2 Choosing a sparse structure

From the storage perspective, the multi-dimensional array is the most inefficient for sparse data. Index-Value pair and Compressed sparse dimension structures are next, since they store index values for each dimension. For large data sets and high number of dimensions these structures are not efficient. However, these structures provide good access performance as can be seen from the analysis in the previous section.

BESS, Offset-value and Split storage structures give good storage performance for sparse data. The restriction with the split storage, however is that dimensions should be partitioned into groups of sparse and dense dimensions which might not always be possible or be inefficient for a large number of dimensions. Among these, performing dimension oriented operations discussed in earlier sections are the most expensive on Offset-Value structure. BESS provides good performance for such operations. The split storage scheme is asymmetric and provides good performance only when the operations restrict the upper level sparse structure. When comparing BESS and Offset-Value pair, BESS stores the bit encoded dimension indices in $\sum_{i=0}^{k-1}\lceil\log n_i\rceil$ bits. Offset-Value pair stores the offset from the start of the chunk in memory in $\log\prod_{i=0}^{k-1} n_i$ bits. The difference is at-most k bits in the representation and is much lower on an average. For operations requiring dimension index values, BESS uses k bit shift operations and k bit and operations for k dimensions and Offset-Value uses $k-1$ division, multiply and subtract operations. The cost of a bit shifting operation is an order of magnitude less than the cost of a division operation as can be seen from Table 4.

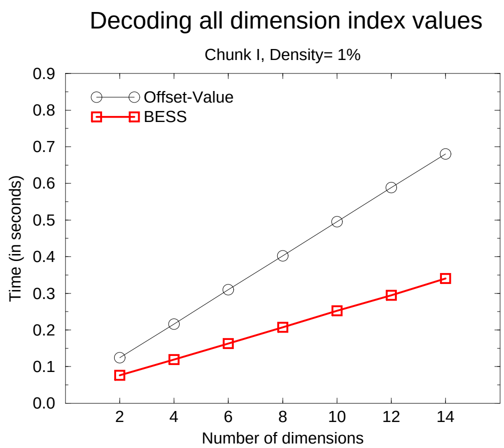
As has been seen in the previous sections, OLAP workloads need a sparse storage mechanism with the following criteria. First, dimensions should be treated symmetrically. Secondly, it should optimize on storage requirements and finally it should give good accesses for data retrieved along dimensions for multi-dimensional analysis. The last two requirements are conflicting and a good trade-off between the two is desired. We believe, the bit-encoded sparse structure (BESS) provides a good balance between the two. In the next section, we provide an experimental comparison of BESS with multi-dimensional arrays and the Offset-Value pair storage.

7 Performance Results

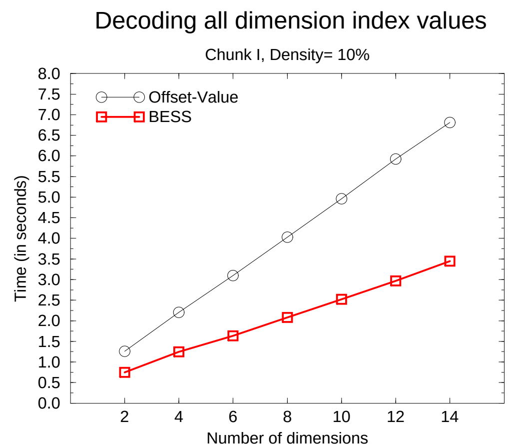
We consider chunked data storage and perform data operations on a chunk. Data in a chunk can either be stored a multi-dimensional array, Offset-Value pairs or using BESS. Performing operations on a data set will involve performing operations on a collection of chunks and will have the same comparative performance as that for a chunk. Results presented in this section are on uniformly distributed random data. The number of dimensions are varied while keeping the chunk size constant. The first data set, has the product of all the chunk dimensions as 10 million. Each cell in a multi-dimensional array contains a 4 byte integer, resulting in about a 38 MB dense chunk (Chunk I). Similarly, another data set fixes the product of all dimensions as 20 million to make a 76MB dense chunk (Chunk II). Queries are run by using varying densities of a chunk. For BESS and Offset-Value storage chunk densities of 1%, 5% and 10% are used. Offset-Value pair stores 8 bytes, a 4 byte measure value and a 4 byte offset, in the data sets chosen. BESS also contains 8 bytes, a 4 byte measure value and a 4 byte bit encoding. This makes the actual chunk sizes to be 0.76MB, 3.8MB and 7.6MB for the first chunk and 1.52MB, 7.6MB and 15.2MB for the second chunk. The number of dimensions, k, is varied from 2 onwards for each query. The dimension sizes, i.e the number of unique values in each dimension, vary accordingly. Again, k is the number of dimensions and D is the density of the chunk. In the following sections performance results are presented for the various queries.

7.1 Decoding dimension index values

Figure 15 presents results for decoding all k dimensions from the value stored as a cell index for the first chunk on the IBM machine. k is varied from 2 to 14 in steps of 2. For example, when $k = 8$, a 8-dimensional cube is considered for dimension decoding. Offset-Value decodes an offset into the individual dimension index values, whereas BESS decodes the bits stored as the index. Figure 16 shows the performance for the second chunk on the IBM machine. Here, k is varied between 2 and 20 in steps of 2. BESS performs much better than Offset-Value storage for all dimensions and the difference gets more pronounced as we increase the number of dimensions. Similar results are observed on the HP machine. Figure 17 shows the performance using the first chunk on the HP machine.

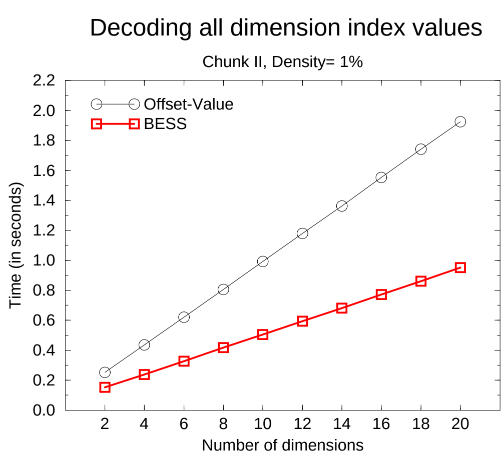


(a)

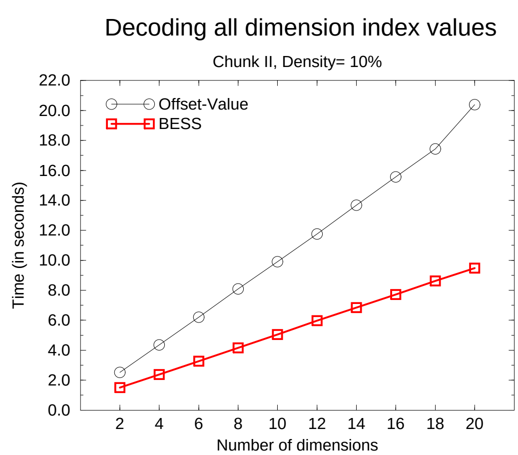


(b)

Figure 15: Decoding all the dimension index values for chunk I (run on IBM) (a) Chunk density = 1% (Data size = 0.76MB)(b) Chunk density = 10% (Data size = 7.6MB)



(a)



(b)

Figure 16: Decoding all the dimension index values for chunk II (run on IBM) (a) Chunk density = 1% (Data size = 1.52MB)(b) Chunk density = 10%(Data size = 15.2MB)

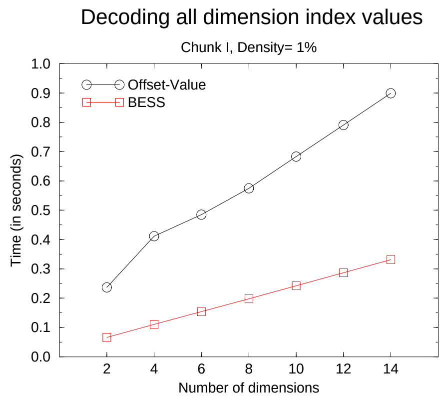


Figure 17: Decoding all the dimension index values for chunk I with density 1% (Data size = 0.76MB) (run on HP)

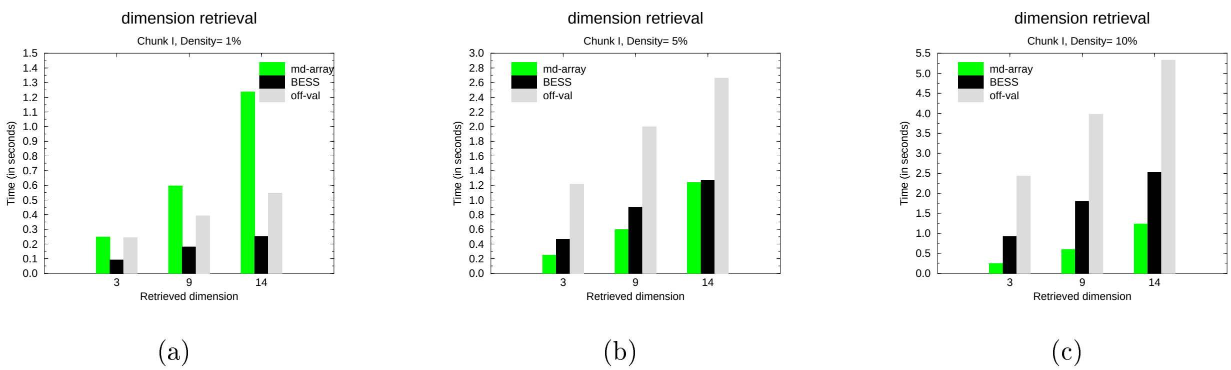
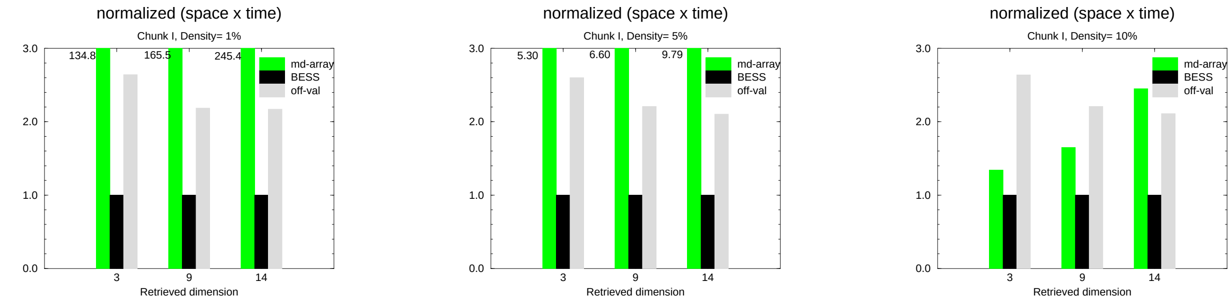


Figure 18: Time for dimension retrieval for a chunk I with 14 dimensions (run on HP) (a) density = 1% (Data size = 0.76MB) (b) density = 5% (Data size = 3.8MB) (c) density = 10% (Data size = 7.6MB)

7.2 Retrieval along a dimension and Range Queries

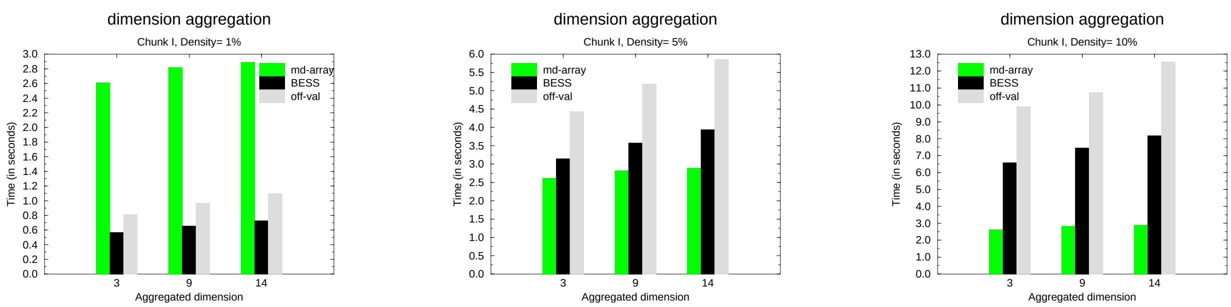
Figure 18 shows the performance of retrieving values along dimensions 3, 9 and 14 in chunk I with 14 dimensions. As can be observed from the figures, the retrieval of the outer dimensions is more costly than the inner dimensions. In multi-dimensional array this is due to the increase in strides of access and the resulting affect on the cache usage. For BESS and Offset-Value pairs, the processing performed to retrieve the dimension index increases as the dimension to be retrieved increases. This can also be observed from the analysis in the previous section, as k increases. The densities considered are 1%, 5% and 10%. The performance of the multi-dimensional array is not affected by the density as expected. The costs of BESS and Offset-Value pair storage increase as density increases since the number of elements to operate on within the chunk increase. It can be observed that BESS provides the best performance for density of 1% for retrieval of any dimension. For 5% density, the retrieval of the outermost dimension performs as well as the multi-dimensional array. Otherwise, for 5% and 10% densities, multi-dimensional array gives the best performance. However, BESS performs much better than Offset-Value storage.

Next, we compare the costs of the three structures in terms of both time and space. A *time-space* metric is calculated as the product of time taken for the operation and the space requirements of the structure. Figure



(a) (b) (c)

Figure 19: Normalized (space $\times$ time) for dimension retrieval with respect to BESS for chunk I with 14 dimensions (run on HP) (a) density = 1% (Data size = 0.76MB) (b) density = 5% (Data size = 3.8MB) (c) density = 10% (Data size = 7.6MB)



(a) (b) (c)

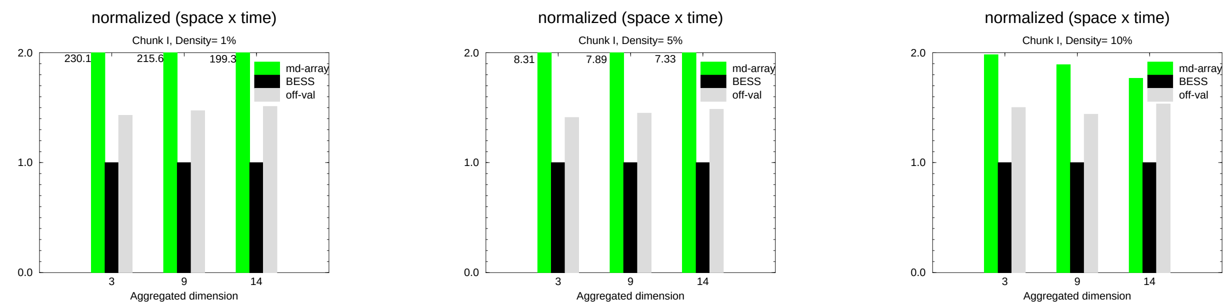
Figure 20: Time for dimension aggregation for chunk I with 14 dimensions (run on HP) (a) density = 1% (b) density = 5% (c) density = 10 %

19 shows this metric for dimension retrieval on a 14 dimensional chunk of size 40MB. This is normalized with respect to BESS. Again, chunk density of 1%, 5% and 10% are considered. It can be observed that BESS performs much better for the time-space metric than the other two structures in all dimensions.

Range queries retrieve data by restricting one or more dimensions. Range queries can be thought of as an extension of retrieval along one dimension. The operations performed for retrieval along a dimension are used for range queries for all three structures. Dimension restrictions can be applied easily to multi-dimensional arrays. For BESS and Offset-Value storage, the dimensions along which the range is specified are retrieved from the encoding and checked if they fall within the range. Performance of dimension decoding and dimension retrieval have been seen in this section. We expect similar results for these structures for range queries.

7.3 Aggregation along a combination of dimensions

Figure 20 shows the performance for aggregation along dimensions 3, 9 and 14 for chunk I with 14 dimensions. Densities of 1%, 5% and 10% are considered. It can be observed from the figures that BESS performs much better than multi-dimensional array and Offset-Value storage for 1% density. For 5% and 10% density the



(a) (b) (c)

Figure 21: Normalized (space × time) for dimension aggregation with respect to BESS for chunk I (run on HP)(a) density = 1% (Data size = 0.76MB) (b) density = 5% (Data size = 3.8MB) (c) density = 10% (Data size = 7.6MB)

multi-dimensional array has a better performance. The number of elements increase with these densities and the aggregation operation contains a $O(n \log n)$ sort operation for both BESS and Offset-Value storage. BESS is better than Offset-Value for this aggregation operation.

However, if we compare the space-time metric in Figure 21, it can be observed that BESS performs much better than the other two structures.

Multi-dimensional aggregation queries can be thought of as an extension of aggregation along one dimension. The operations used for aggregation for all three structures are the same between the two. Aggregation can be done on one or more dimensions. We expect similar results for multi-dimensional aggregation as seen for aggregation done along a dimension.

8 Conclusions

In this paper we have introduced the bit-encoded sparse structure(BESS) for storing multi-dimensional sparse data. A comparison with other sparse data structures is presented in terms of storage efficiencies and data access times for the typical accesses required for OLAP and data mining workloads. The gain from reduced storage requirements is usually offset by overhead required in data accesses. We have presented an analysis of some typical OLAP operations using sparse storage schemes for higher dimensional data. This helped in highlighting the benefits of BESS. Performance results show better performance for OLAP queries using BESS over other storage schemes. We evaluate BESS with a space × time metric and find that its performance is the best overall.

Furthermore, analysis and performance results presented in this paper are in terms of design parameters : number of dimensions, dimension sizes, sparsity of data, cell size, which can serve to compare and contrast the storage versus data access performance trade-offs in choosing a suitable multi-dimensional structure for a given application.

References

[1] Codd E. F., Providing OLAP to user-analysts : An IT mandate, *White Paper*, E.F. Codd and Associates, 1993.

[2] Colliat G., OLAP, Relational, and Multi-Dimensional Database Systems, *SIGMOD Record*, Vol 25, No. 3, September 1996.

[3] Gray J., Bosworth A., Layman A and Pirahesh H., Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, *Proc. 12th International Conference on Data Engineering*, 1996, pp. 152-159.

[4] Fayyad U.M, Piatesky-Shapiro G., Smyth P. and Uthurusamy R., From data mining to knowledge discovery: An overview, *Advances in data mining and knowledge discovery*, pp. 1-34, Cambridge, MA, MIT Press.

[5] Bhandari I., Attribute Focusing: Data mining for the layman, *Research Report RC 20136*, IBM T.J Watson Research Center.

[6] Goil S. and Choudhary A., Parallel Data Cube Construction for High Performance On-Line Analytical Processing, To appear in *Proc. IEEE Conference on High Performance Computing*, December 1997.

[7] Goil S. and Choudhary A., High Performance OLAP and Data Mining on Parallel Computers, *Journal of Data Mining and Knowledge Discovery 1, 1997, pp. 37-63.*

[8] Guting A., An Introduction to Spatial Databases, *VLDB Journal 3(4):357-399, 1994.*

[9] Han J., Cai Y. and Cercone N., Data-Driven Discovery of Quantitative Rules in Relational Databases, *IEEE Trans. on Knowledge and Data Engineering, Vol 5, No. 1, February 1993.*

[10] Han J. and Fu Y., Discovery of Multiple-Level Association Rules from Large Databases, *Proc. of the* 21^{st} *VLDB Conference, Zurich, 1995.*

[11] Kenan Software, An Introduction to Multi-Dimensional Database Technology, available from *http://www.kenan.com/acumate/mddb.htm*

[12] Sarawagi S. and Stonebraker M., Efficient Organization of Large Multi-Dimensional Arrays, *Proc. of the Eleventh International Conference on Data Engineering*, Houston, February 1994.

[13] Sarawagi S., Indexing OLAP Data, *Data Engineering Bulletin, Vol 20, No. 1, March 1997*

[14] OLAP Council Benchmark available from http://www.olapcouncil.org

[15] Earle, Robert J., Arbor Software Corporation, U.S. Patent 5359724., October 1994.

[16] Zhao Y., Tufte K. and Naughton J., On the Performance of an Array-Based ADT for OLAP Workloads, *Technical Report*, University of Wisconsin, Madison, May 1996.

[17] Zhao Y., Deshpande, P. and Naughton J., An Array-Based Algorithm for Simultaneous Multi-Dimensional Aggregates, *Proc. ACM-SIGMOD International Conference on Management of Data*, pp 159-170.