Design and Implementation of a Scalable Parallel System for Multidimensional Analysis and OLAP [*]

Sanjay Goil Alok Choudhary
Department of Electrical & Computer Engineering,
Northwestern University,
Technological Institute,
2145 Sheridan Road, Evanston, IL-60208.
Email: {sgoil,choudhar}@ece.nwu.edu

Abstract

Multidimensional Analysis and On-Line Analytical Processing (OLAP) uses summary information that requires aggregate operations along one or more dimensions of numerical data values. Query processing for these applications require different views of data for decision support. The Data Cube *operator provides multi-dimensional aggregates, used to calculate and store summary information on a number of dimensions.*

The multi-dimensionality of the underlying problem can be represented both in relational and multi-dimensional databases, the latter being a better fit when query performance is the criteria for judgment. Relational databases are scalable in size and efforts are on to make their performance acceptable. On the other hand multi-dimensional databases perform well for such queries, although they are not very scalable. Parallel computing is necessary to address the scalability and performance issues for these data sets.

In this paper we present a parallel and scalable infrastructure for OLAP and multidimensional analysis. We use chunking *to store data either as a dense block using multi-dimensional arrays (md-arrays) or a sparse set using a Bit encoded sparse structure (BESS). Chunks provide a multi-dimensional index structure for efficient dimension oriented data accesses much the same as md-arrays do. Operations within chunks and between chunks are a combination of relational and multi-dimensional operations depending on whether the chunk is sparse or dense. We present performance results for data sets with 3, 5 and 10 dimensions for our implementation on the IBM SP-2 which show good speedup and scalability.*

1 Introduction

On-Line Analytical processing (OLAP) and multi-dimensional analysis is used for decision support systems to find interesting information from large databases. Multi-dimensional databases are suitable for OLAP and data mining since these applications require dimension oriented operations on data. Traditional multi-dimensional databases store data in md-arrays on which analytical operations are performed. These are good to store dense data, but since most datasets are sparse in practice, other efficient sparse storage schemes are required.

It is important to weigh the trade-offs involved in reducing the storage space versus the increase in access time for each sparse data structure, in comparison to md-arrays. These trade-offs are dependent on many parameters some of which are (1) number of dimensions, (2) sizes of dimensions and (3) degree of sparsity of the data. Complex operations such as required for OLAP can be very expensive in terms of data access time if efficient data structures are not used.

Sparse data structures such as the R-tree and its variants have been used for OLAP [6]. Range queries with a lot of unspecified dimensions are expensive because many paths have to be traversed in the tree to calculate aggregates. Chunking has been used in [7] with dense and sparse chunks. Sparse chunks store an Offset-Value pair for the data present. Dimensional operations on these require materializing the sparse chunk into a md-array and performing array operations on it. For a high number of dimensions this might not be possible since the materialized chunk may not fit in memory. A split storage is described in [1], where the dimensions are split into sparse and dense subsets. The sparse dimensions use a sparse index structure to index into the dense blocks of data stored as md-arrays. Further, none of these address parallelism and scalability to large data sets in a high number of dimensions.

We compare the storage and operational efficiency in OLAP and multi-dimensional analysis of various sparse data storage schemes in [3]. A novel data structure using bit encodings for dimension indices called Bit-Encoded Sparse Structure (BESS) is used to store sparse data in chunks, which supports fast OLAP query operations on sparse data using bit operations without the need for exploding the

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

sparse data into a md-array. This allows for high dimensionality and large dimension sizes.

In this paper we present a parallel and scalable OLAP framework for large data sets. Parallel data cube construction for large data sets and a large number of dimensions using both dense and sparse storage structures is presented. Sparsity is handled by using compressed *chunks* using a bit encoded sparse structure (BESS). Data is read from a relational data warehouse which provides a set of tuples in the desired number of dimensions. Results of our disk-based implementations are presented on the IBM SP2, for large datasets with 3, 5 and 10 dimensions which show good performance and speedups.

The rest of the paper is organized as follows. Section 2 describes OLAP using the data cube operator. Section 3 presents multi-dimensional storage using chunks and BESS for sparse data. Section 4 presents steps in the computation of the data cube on a parallel machine and the overall design. Section 5 describes the algorithms, techniques and optimizations in the parallel building of the simultaneous multi-dimensional aggregates and the factors affecting performance. Section 6 presents implementation results and performance analysis on the IBM SP2. Section 7 concludes the paper.

2 Data Cubes and OLAP

Multidimensional systems store data in multi-dimensional structures which is a 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 can be organized into a data cube by calculating all possible combinations of GROUP-BYs [5]. This operation is useful for answering OLAP queries which use aggregation on different combinations of attributes. For a data set with n attributes this leads to 2^n GROUP-BY calculations. A data cube treats each of the $k, 0 \leq k < n$ aggregation attributes as a dimension in k-space.

Data Cube operators generalize the histogram, cross-tabulation, roll-up, drill-down and sub-total constructs. Figure 1 shows a lattice structure for the data cube with 5 dimensions. At a level $i, 0 \leq i \leq n$ of the lattice, there are $C(n,i)$ sub-cubes (aggregates) with exactly i dimensions, where the function C gives the all combinations having i distinct dimensions from n dimensions. A total of $\sum_{i=0}^{n} C(n,i) = 2^n$ sub-cubes are present in the data cube including the base cube. Optimizations of calculating the aggregates in the sub-cubes can be performed using the lattice structure augmented by the various computations and communication costs to generate a DAG of cube orderings which minimize the cost. This is discussed in a later section.

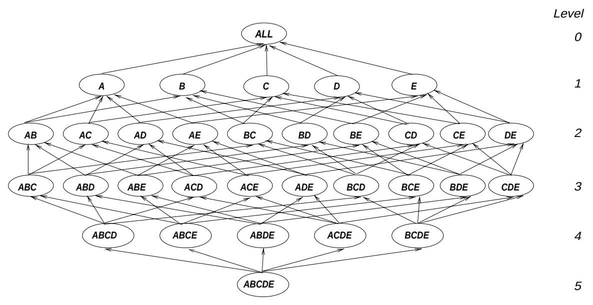
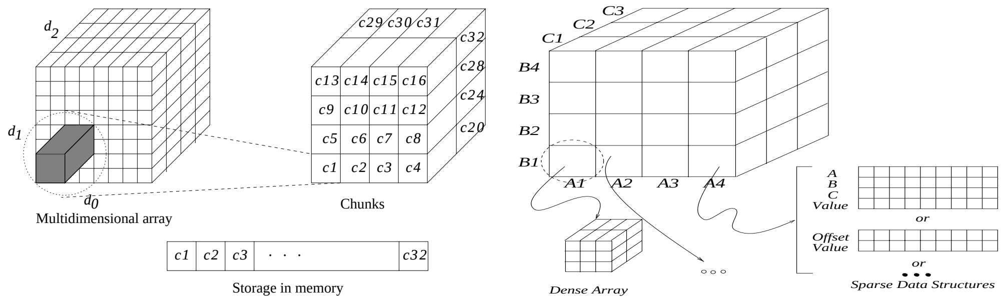


Figure 1. Lattice for cube operator

3 Data Storage: Chunks and BESS

Multidimensional database technology facilitates flexible, high performance access and analysis of large volumes of complex and interrelated data [2]. It is more natural and intuitive for humans to model a multidimensional structure. A *chunk* is defined as a block of data from the md-array which contains data in all dimensions. A collection of chunks defines the entire array. Figure 2(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. Most sparse data may not be uniformly sparse. Dense clusters of data can be stored as md-arrays. Sparse data structures are needed to store the sparse portions of data. These chunks can then either be stored as dense arrays or stored using an appropriate sparse data structure as illustrated in Figure 2(b). Chunks also act as an index structure which helps in extracting data for queries and OLAP operations.

Typically, sparse structures have been used for advantages they provide in terms of storage, but operations on data are performed on a md-array which is populated from the sparse data. 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 multidimensional structures for a large number of dimensions, we are 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 our Bit-encoded sparse storage (BESS). For each cell present in a chunk a dimension index is encoded in $\lceil \log |d_i| \rceil$ bits for each dimension d_i of size $|d_i|$. A 8-byte encoding is used to store the BESS index along with the value at that location. A larger encoding can be used if more bits are required. A dimension index can then be extracted by a bit mask operation. Aggregation along a dimension d_i can be done by masking its dimension encoding in BESS and using a sort operation to get the duplicate resultant BESS values together. This is followed by a scan of the BESS index, aggregating values for each duplicate BESS index. For dimensional analysis, aggregation needs to be done for appropriate chunks along a dimensional plane.



(a) Chunking for a 3D array (b) Chunked storage for the cube

Figure 2. Storage of data in chunks

4 Overall Design

In this section we describe our design for a parallel and scalable data cube on coarse grained parallel machines (e.g IBM SP-2) or a network of workstations, characterized by powerful general purpose processors (few to a few hundred) and a fast interconnection between them. The programming paradigm used is a high level programming language (e.g. C/C++) embedded with calls to a portable communication library (e.g. Message Passing Interface).

In what follows, we address issues of data partitioning, parallelism, schedule construction, data cube building, chunk storage and memory usage on this machine architecture. Moreover, a partial cube can be constructed if the number of dimensions is large or a specific level of the cube is needed. For example, in 2-way attribute-oriented data mining of associations, all cubes at level 2 are materialized by using the base cube and the minimum materializations of sub-cubes at the intermediate levels between 3 and $n-1$ [4].

Data is partitioned on processors to distribute work equitably. In addition, a partitioning scheme for multidimensional has to be *dimension-aware* and for dimension-oriented operations have some regularity in the distribution. A dimension, or a combination of dimensions can be distributed. In order to achieve sufficient parallelism, it would be required that the product of cardinalities of the distributed dimensions be much larger than the number of processors. For example, for 5 dimensional data ($ABCDE$), a 1D distribution will partition A and a 2D distribution will partition AB. Partitioning determines the communication requirements for data movement in the intermediate aggregate calculations in the data cube. We support both 1D and 2D partition in our implementations. Since 2^n cubes are being constructed, we keep them distributed as well. The distribution of these cubes depends on the cardinalities of their largest 1 or 2 dimensions. The same criteria is used here as the one used for the base cube. However, redistribution of dimensions and chunks may be required if a dimension is partitioned anew or is re-partitioned.

Table 1 shows the various distributions for aggregate calculations supported in our framework. The underlined dimensions are partitioned. Calculations are either *Local* or *Non Local*. Local calculations maintain the data distribution on each processor and the aggregation calculation does not involve any inter-processor communication. Non local calculations distribute a undistributed dimension such as in $\underline{AB}C \rightarrow \underline{AC}$, where dimension B is aggregated and C, which was previously undistributed, is distributed. Another calculation is $\underline{AB}C \rightarrow \underline{BC}$, where A is aggregated and B, the second distributed dimension becomes the first distributed dimension, and C gets distributed as the second dimension. These can be categorized as either dimension 1 or dimension 2 being involved in the (re)distribution. The sub-cubes can be stored as *chunked* or as *md-arrays* which are distributed or on a single processor (UNI) with these distributions. The md-arrays are however restricted to a 1D distribution since their sizes are small and 2D distribution will not provide sufficient parallelism. The data cube build scheduler does not evaluate the various possible distributions currently, instead calculating the costs based on the estimated sizes of the source and the target sub-cubes and uses a partitioning based on the dimension cardinalities.

Several optimizations can be done over the naive method of calculating each aggregate separately from the initial data [5]. **Smallest Parent**, computes a group-by by selecting the smallest of the previously computed group-bys from which it is possible to compute the group-by. The next optimization is to compute the group-bys in an order in which the next group-by calculation can benefit from the cached results of the previous calculation. An im-

Table 1. Partitioning of sub-cubes following aggregation calculations

Distribution	*Local*	*Non Local*	
		Dimension 1	*Dimension 2*
$2D \rightarrow 2D$	$\underline{AB}C \rightarrow \underline{AB}$	$\underline{AB}C \rightarrow \underline{BC}$	$\underline{AB}C \rightarrow \underline{AC}$
$2D \rightarrow 1D$		$\underline{AB}C \rightarrow \underline{B}C$	$\underline{AB}C \rightarrow \underline{A}C$
$1D \rightarrow 1D$	$\underline{A}BC \rightarrow \underline{A}B, \underline{A}C$	$\underline{A}BC \rightarrow \underline{B}C$	
$2D \rightarrow UNI$		$\underline{AB}C \rightarrow BC$	$\underline{AB}C \rightarrow AC$
$1D \rightarrow UNI$		$\underline{A}BC \rightarrow BC$	$\underline{A}BC \rightarrow AC$
$UNI \rightarrow UNI$	$ABC \rightarrow AB, AC$		

portant multi-processor optimization is to **minimize interprocessor communication**. In a cube lattice, each node represents an aggregate and an arrow represents a possible aggregate calculation which is also used to represent the cost of the calculation.

4.1 Data Structure Management

For large data sets the sizes of the cubes and the number of cubes will not fit in main memory of the processors. A scalable parallel implementation will require disk space to store results of computations, often many of them intermediate results. This is similar to a *paging* based system which can either rely on virtual memory system of the computer or perform the paging of data structures to the needs of the application. We follow the latter approach.

A global cube topology is maintained for each subcube by distributing the dimension equally on each processor. A dimension of size $d_i, 0 \leq i < n$ gets distributed on p processors, a processor i gets $\lceil \frac{d_i}{p} \rceil$ portion of d_i, if $i < d_i \bmod p$, else it gets $\lfloor \frac{d_i}{p} \rfloor$. Each processor thus can calculate what portion belongs to which processor. Further, a constant chunk size is used in each dimension across subcubes. This allows for a simple calculation to find the target chunk which a chunk maps to after aggregating a dimension. However, the first distribution of the dimensions in the base cube is done using a sample based partitioning scheme which may result in a inexact partition and they are kept the same till any of the distributed dimension gets redistributed.

A *cube directory* structure is always maintained in memory for each cube at the highest level. For each cube this contains a pointer to a *data cube* structure which stores information about the cube and its chunks. It also contains a file offset to indicate the file address if the data cube structure is paged out. A status parameter indicates whether the data cube structure is in memory (INMEM) or on disk (ONDISK).

A data cube structure maintains the cube topology parameters, the number of unique values in each dimension, whether the chunk structure for the cube is in memory (*status*), a pointer to the chunk structure if it is in memory and a file offset if it is on disk. The total number of chunks for the chunk structure of the cube is in *totalchunks*. Additionally, for each chunk of the chunk structure, a chunk status *cstatus* is maintained to keep track of chunk structure paging. The chunk address is a pointer to the chunk structure in memory which stores information for each chunk. This is when cstatus is set to INMEM. Otherwise, cstatus can either be UNALLOCATED or ONDISK. In the latter case the chunk address will be a file offset value. For a md-array, the size of the array and the dimension factor in each dimension are stored to lookup for the calculations involving aggregations instead of calculating them on the fly every time.

A chunk structure for a sub-cube can either be in its entirety or parts of it can be allocated as they are referred to. The cstatus field of the data cube will keep track of allocations. Chunk structure keep track of the number of BESS + value pairs *ntuples* in the chunk, which are stored in *minichunks*. Whether a chunk is dense or sparse is tracked by *type*. A dense chunk has a memory pointer to a dense array whereas a sparse chunk has a memory pointer to a minichunk. Chunk index for each dimension in the cube topology is encoded in a 8 byte value *cidx*. Further, dimensions of the chunk are encoded in another 8 byte value *cdim*. This allows for quick access to these values instead of calculating them on the fly.

Minichunks can either be unallocated (UNALLOCATED), in memory (INMEM), on disk (ONDISK) or both in memory and on disk (INMEM_ONDISK). Initially, a minichunk for a chunk is allocated memory when a value maps to the chunk (UNALLOCATED $\rightarrow$ INMEM). When the minichunk is filled it is written to disk and its memory reused for the next minichunk (INMEM $\rightarrow$ INMEM_ONDISK). Finally, when a minichunk is purged to disk it is deallocated (INMEM_ONDISK $\rightarrow$ ONDISK). A chunk can thus have multiple minichunks. Hence, choosing the minichunk size is an important parameter to control the number of disk I/O operations for aggregation calculations.

5 Parallel Aggregation Calculations

Since chunks can either be sparse or dense, we need methods to aggregate sparse chunks with sparse chunks, sparse with dense chunks and dense with dense chunks. The case of dense chunks to sparse chunk does not arise since a dense chunk does not get converted to a sparse chunk ever. Figure 3 illustrates a local and non local aggregation calculation. Computation and communication costs for aggregation are used by the scheduler by analyzing the chunk aggregation operations and the resulting partitioning. The cost analysis of the various aggregations between 1D, 2D and uniprocessor aggregations is not included in this paper.

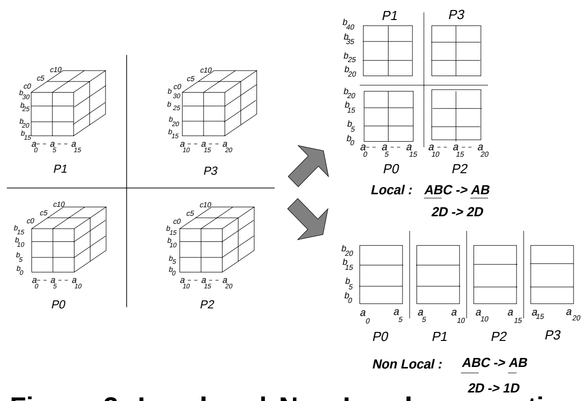


Figure 3. Local and Non Local aggregation calculations

The extents of a chunk of the aggregating cube can be contained in the extents of a chunk of the aggregated cube. In this case the BESS+value pairs are directly mapped to the target chunk, locally or non-locally. However, the BESS index values need to be modified to encode the offsets of the new chunk. If the chunk is overlapping over a target chunk boundary, then each BESS value has to be extracted to determine its target chunk. This is computationally more expensive than the direct case. It is to be noted that a 2 dimensional distribution may result in more overlapped chunks than a 1 dimensional distribution, because the former has more processor boundary area than the latter.

Sparse chunks store BESS+value pairs in minichunks. Sparse to sparse aggregations involve accessing these minichunks. The BESS values are kept sorted in the minichunks to facilitate the aggregation calculations by using sort and scan operations used in relational processing. To aggregate dimension B in a cube ABC, The bit encoding for B is masked from the BESS values for both chunks. A integer sort is done on the remaining encoding of A and C on both the chunks. This gets the values which map to the same A and C contiguous to each other. This is followed by a merge of sorted values aggregating where the values of A and C are the same.

Buffer management for this aggregation depends on the order in which chunks to be aggregated are accessed. There are two alternative ways to access chunks, *dimension oriented* in which the chunks are accessed along the dimension to be aggregated. The other method, *chunk numbering*, accesses chunks in the order in which they are laid out in memory. In the dimension oriented access, a buffer can be filled with the source chunks that map to a single target chunk and the sort and scan operations are done on the buffer in memory. For the chunk order access, the consecutive chunks accessed are mapped to different chunks for all dimensions except the innermost dimension. This results in sort of each individual chunk and merge with the sorted values calculated earlier mapping to the same chunk. This involves a disk read to get the previous result from the minichunk of the target chunk, which was written to disk if it was full.

The sparse-dense chunk aggregation and chunked - md-array aggregation are done by converting a BESS index into an array offset. Due to lack of space we do not elaborate those cases here.

6 Performance Results

In this section we present performance results for our system on a 16-node IBM SP-2 distributed memory parallel computer available to us at Northwestern University. Assume N tuples and p processors. Initially, each processor reads $\frac{N}{p}$ tuples from a shared disk, assuming that the number of unique values is known for each attribute. These are partitioned using a sample based partitioning algorithm (*Partitioning phase*) so that the attribute (dimension) values are ordered on processors and distributed almost equally. To load the base cube, tuples are sorted (*Sorting phase*) on the combined key of all the attributes so that the access to chunks is conformant to its layout in memory/disk. (Sorting in the order $A_0 \rightarrow A_1 \rightarrow A_2 \ldots A_{n-1}$, is conformant to the layout of chunks where A_0 is the outer most dimension and A_{n-1} is the inner most, for loading a sorted run of values.) The base cube is loaded on each processor from these tuples locally on each processor. The sub-cubes of the data cube are calculated from here (*Building phase*).

Table 2. Description of datasets and attributes, (N) Numeric (S) String

	ndim	$\lvert d_i \rvert$	$\prod_i d_i$	Tuples
I	3	1024(S),256(N), 512(N)	2^{27}	1.34 M
II	5	1024(S),16(N),32(N),16(N),256(S)	2^{31}	2.14 M
III	10	1024(S),16(S),4(S),16,4,4,16,4,4,32(N)	2^{37}	1.37 M

We choose 3 data sets, one each of dimensionality 3, 5 and 10 to illustrate performance. Random data with a uniform distribution used for the performance figures. Table 2 illustrates the data sets for our experiments. The chunk sizes in each dimension for the 3 dimensional data set is chosen as 32, 8 and 16. The number of chunks is 2^{15} for the base cube, but this gets distributed across processors. The 5 dimensional data set has chunk sizes in each dimension of 32, 4, 8, 8, 16 and for the 10 dimensional data set the chunk dimension sizes are 64, 4, 2, 4, 2, 2, 4, 2, 2. The number of sub cubes in the datacube for Dataset I is $2^3 = 8$, for Dataset II is $2^5 = 32$ and for Dataset III is $2^{10} = 1024$. We report the results of complete data cube construction here.

Figure 4 shows the time taken by the various phases of the data cube construction algorithm. For 2 dimensional partitioning, 5 and 10 dimensional sets provide good speedups. However, in the 3 dimensional set we observe a good speedup from 4 to 8 processor but 16 processor case results in more communication intensive computations which involve a 2 dimensional distribution to aggregate to uniprocessor sub-cubes. The performance of 1 dimensional distribution is similar to the 2 dimensional distribution for the 10 dimension case.

Figure 5 shows the time taken by the two different methods of accessing chunks for aggregation calculations. Dimension oriented method is better than the access that follows chunk numbering. In all cases we observe that the dimension ordering method works better than the access pattern that follows chunk numbering.

Figure 6 shows the time taken when initial partitioning of the base cube is either 2 dimensional or 1 dimensional. We observe that 1D partitioning works better for lower dimension data sets with 3 and 5 dimensions. This is due to the fact that the lower levels of the lattice (with less number of dimensions) are either distributed on a single dimension

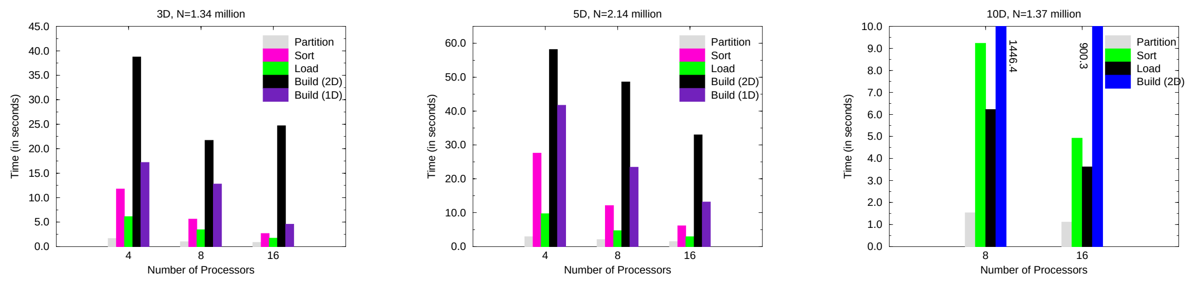


Figure 4. Time taken by various phases of the data cube construction algorithm for datasets with dimensions 3 (8 sub cubes), 5 (32 sub cubes) and 10 (1024 sub cubes)

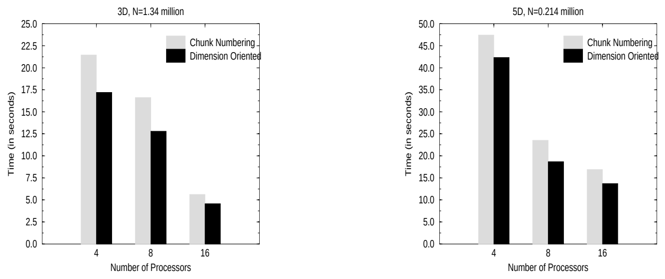


Figure 5. Comparison of the two chunk access methods, chunk numbering and dimension oriented for a dataset with 3 and 5 dimensions

or are on a single processor, hence communication costs are low. For a 2 dimensional partitioning, redistribution is required quite early in the lattice structure which involves larger cubes. Also, the number of overlapping chunks are higher in a 2 dimensional partitioning when compared to a 1 dimensional distribution, in which the individual BESS index values have to be checked for their target mapping.

However, for the 10 dimension case, the 2-dimensional distribution is slightly better. A large number of cubes are calculated in this case and more cubes can benefit from a 2 dimensional partitioning.

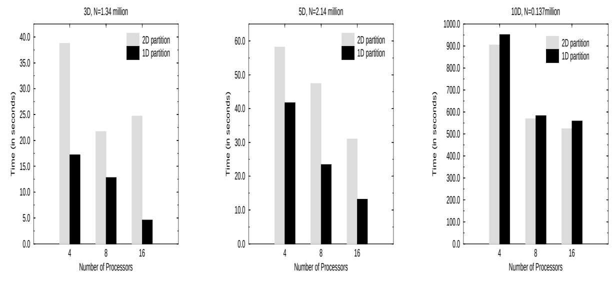


Figure 6. Comparison of partitioning schemes for 3, 5 and 10 dimensional data sets when the initial distribution of the base cube is either 2D or 1D

7 Conclusions

In this paper we have presented the design and implementation of a scalable parallel system for multidimensional analysis and OLAP. Multidimensional data is stored in *chunks* which are either sparse or dense. Sparse chunks are represented by a BESS+value pair and aggregation operations between them use sort and scan operations using the dimension indices encoded in BESS. For maximum efficiency of operations some cubes are stored as md-arrays if the cardinalities of the dimensions involved are not large and the cube size is below a specific threshold. Operations for chunked and md-array cubes are supported. Results on data sets with 3, 5 and 10 dimensions show good performance.

References

[1] G. Colliat. OLAP, Relational, and Multi-dimensional Database Systems. In *SIGMOD Record*, volume 25(3), September 1996.

[2] K. enan Software. An introduction to multi-dimensional database technology. In *http://www.kenan.com/acumate/mddb.htm*, 1997.

[3] S. Goil and A. Choudhary. Sparse data storage schemes for multi-dimensional data for OLAP and data mining. Technical Report CPDC-9801-005, Northwestern University, December 1997.

[4] S. Goil and A. Choudhary. High performance data mining using data cubes on parallel computers. In *Proc. International Parallel Processing Symposium*, March 1998.

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

[6] S. Sarawagi. Indexing OLAP Data. In *Data Engineering Bulletin*, volume 20(1), March 1997.

[7] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multi-dimensional aggregates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 159–170, 1997.