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
sgoil@ece.nwu.edu

Abstract

Multidimensional Analysis and On-Line Analytical Processing (OLAP) operations require summary information on multi-dimensional data sets. Most common are aggregate operations along one or more dimensions of numerical data values. Simultaneous calculation of multi-dimensional aggregates are provided by the *Data Cube* operator, used to calculate and store summary information on a number of dimensions. This is computed only partially if the number of dimensions is large. Query processing for these applications require different views of data to gain insight and for effective decision support. These may either be answered from a materialized cube in the data cube or calculated on the fly.

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 for OLAP and multidimensional analysis and efforts are on to make their performance acceptable. On the other hand multi-dimensional databases have proven to provide good performance for such queries, although they are not very scalable. In this paper we would like to address scalability in multi-dimensional systems for OLAP and multi-dimensional analysis.

Multidimensional databases store data in multidimensional structure on which analytical operations are performed. A challenge for these systems is how to handle large and sparse data sets with large dimensionality. Parallel computing is necessary to address the 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 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 multi-dimensional 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.

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

1 Introduction

On-Line Analytical processing (OLAP) and multi-dimensional analysis is used for decision support systems and statistical inferencing to find interesting information from large databases. Multidimensional databases are suitable for OLAP and data mining since these applications require dimension oriented operations on data. Traditional multidimensional databases store data in multidimensional arrays on which analytical operations are performed. Multidimensional arrays are good to store dense data, but most datasets are sparse in practice for which other efficient 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 multidimensional 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 [Sar97]. 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 [ZDN97] 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 multi-dimensional 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 [Col96], 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 multi-dimensional 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 [GC97b]. 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 sparse data into a multidimensional 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. [GC98b] presents data partitioning, and *sort-based* loading of chunks in the *base* cube (which is a n-dimensional structure at level n of the data cube) from which data cube is computed. Our techniques for multidimensional analysis can also be applied to scientific and statistical databases [LS98].

Finally, 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

OLAP is used to summarize, consolidate, view, apply formulae to, and synthesize data according to multiple dimensions. 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. Alternatively, multi-dimensional database techniques (multi-dimensional OLAP) have been applied to decision-support applications. Data is stored in multi-dimensional structures 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.

We have evaluated the nature of query operations in OLAP and SSDB, which are dimension oriented and classified as:

- **Retrieval of a random cell element**
- **Retrieval along a dimension or a combination of dimensions**
- **Retrieval for values of dimensions within a range (Range Queries)**
- **Aggregation operations on dimensions**
- **Multi-dimensional Aggregation (Generalization/Consolidation: lower to higher level in hierarchy)**

In [GC97b], an analysis is provided for each of these query operations using various sparse data structures and it is shown that BESS is superior to others.

Data can be organized into a data cube by calculating all possible combinations of GROUP-BYs [GBLP96]. 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.

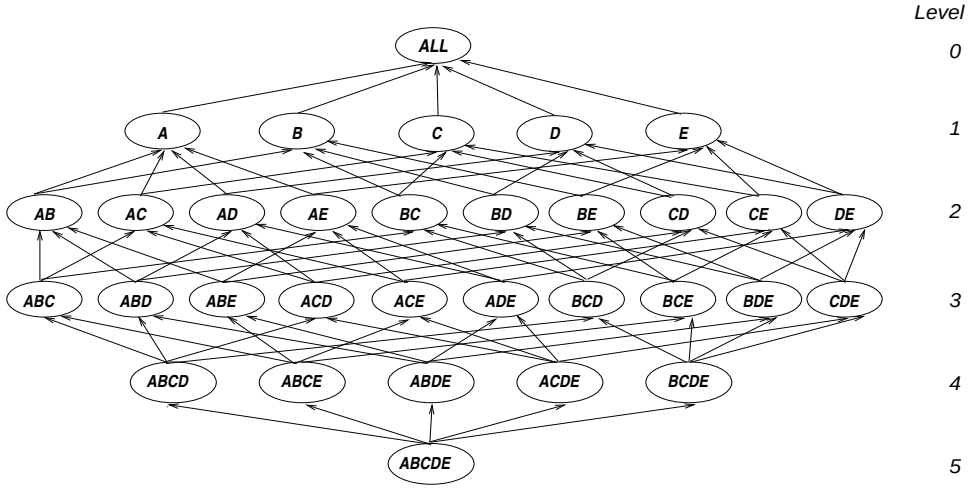
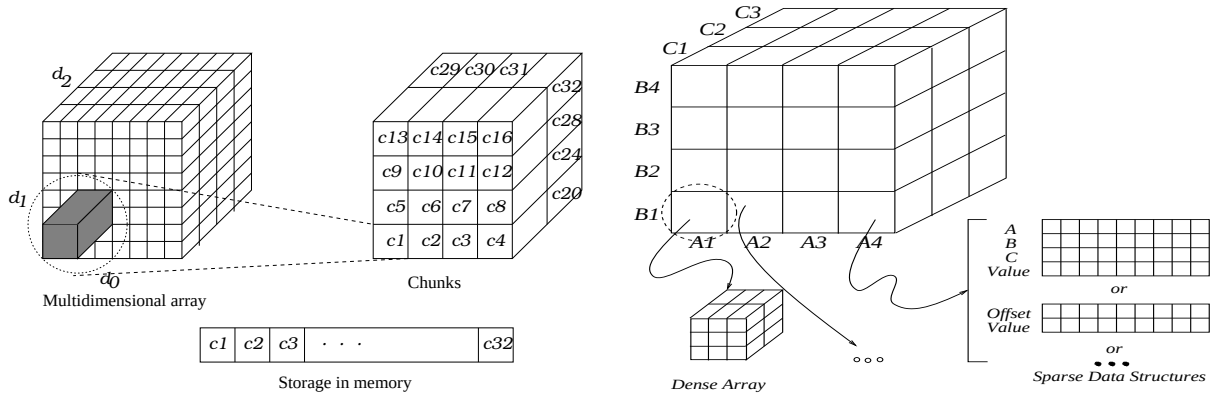


Figure 1: Lattice for cube operator

3 Data Storage: Chunks and BESS

Multidimensional database technology facilitates flexible, high performance access and analysis of large volumes of complex and interrelated data [Sof97]. It is more natural and intuitive for humans to model a multidimensional structure. A *chunk* is defined as a block of data from the multidimensional 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 multidimensional 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 multidimensional 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



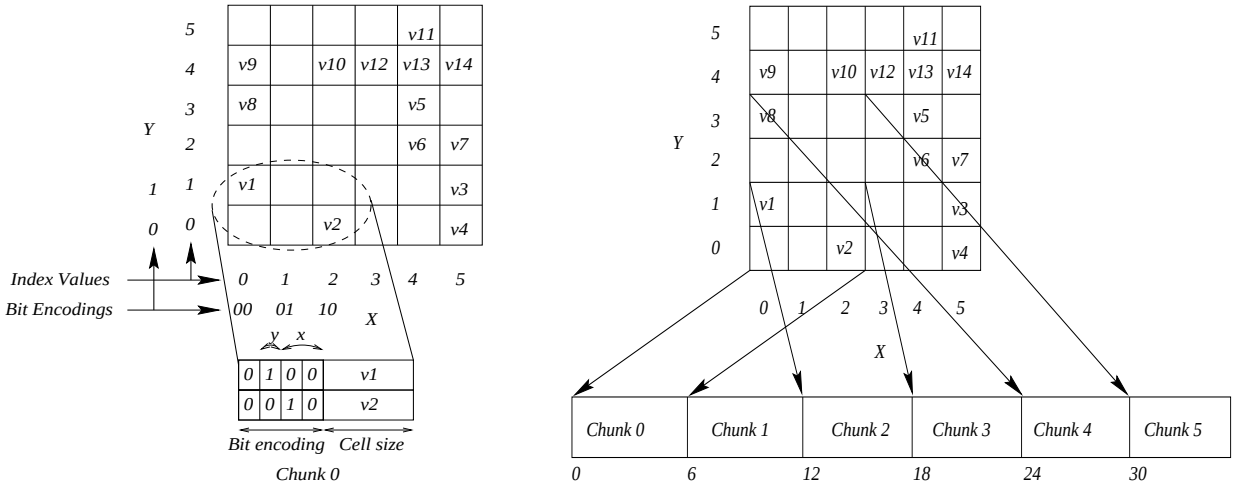
(a) Chunking for a 3D array (b) Chunked storage for the cube

Figure 2: Storage of data in chunks

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 the number of dimensions are larger than 20. 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. This will be elaborated further in a later section.

Figure 3(a) 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.

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 3(b). 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 and using a bit-mask to extract the bits that are used to encode each dimension.



(a) Chunk storage using BESS

(b) Storing chunk offsets for index retrieval

Figure 3: BESS used in Chunk Storage

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. Figure 4 summarizes the various options available for these, especially in terms of storage of cubes, parallelism at different levels, aggregation calculation orderings and the chunk access for aggregations. 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 $n-1$ [GC98a].

4.1 Data Partitioning and Parallelism

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

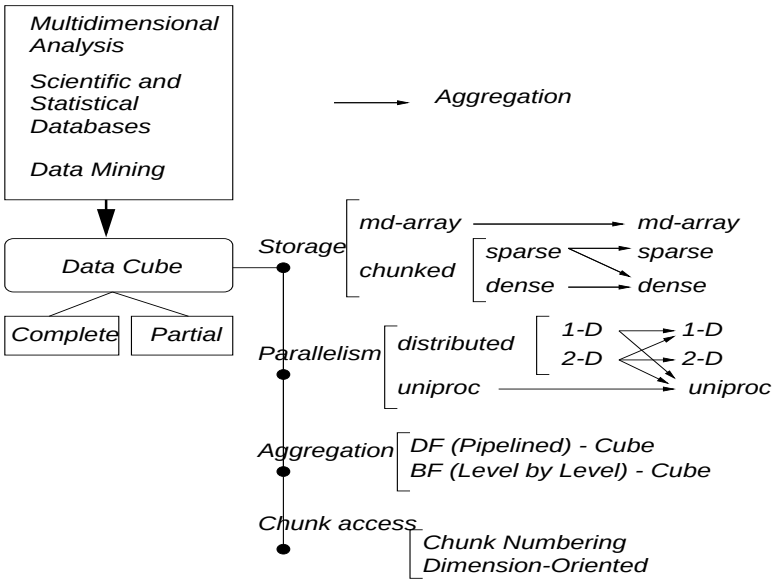


Figure 4: Data Cube Architecture

AB. We assume, that dimensions are available that have cardinalities much greater than the number of processors in both cases. That is, either $|A_i| \gg p$ for some i, or $|A_i||A_j| \gg p$ for some i, j, $0 \leq i, j \leq n-1$, n is the number of dimensions. 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. Let K (1 or 2) be the number of dimensions used for the partitioning. For $K = 2$, a 2 dimensional partitioning, p processors are divided into two groups $(k, \frac{p}{k})$, where k is the number of processors in the group to be created by the first distributed dimension, chosen to be a divisor of p. For each tuple, A_i is used to choose among the k partitions, and then A_j is further used to find the correct processor among the $\frac{p}{k}$ processors in that group. Figure 5 shows the base cube loading for a 3 dimensional cube (ABC) distributed on 4 processors in a 2D distribution, where dimensions A and B are distributed in a (2, 2) grid. Clearly, when $K = 1$, a 1D partitioning is done with $k = p$.

Since 2^n cubes are being constructed, and 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

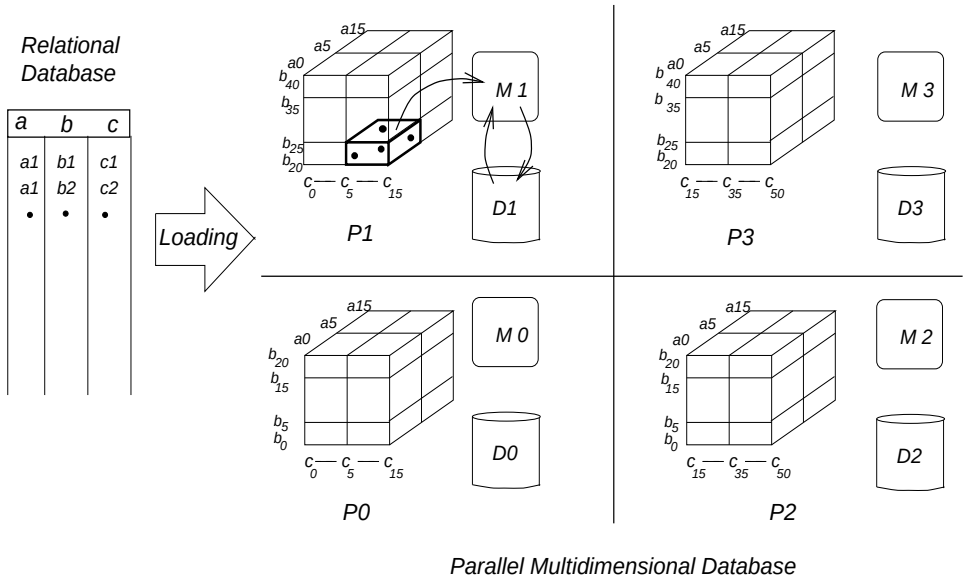


Figure 5: Base cube loading for a 3 dimensional cube on 4 processors

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. Also, as illustrated in Figure 4, the sub-cubes can be stored as *chunked* or as *multi-dimensional arrays* which are distributed or on a single processor with these distributions. The multi-dimensional 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.

Table 1: Partitioning of sub-cubes following aggregation calculations

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

4.2 Schedule Generation for Data Cube

Several optimizations can be done over the naive method of calculating each aggregate separately from the initial data [GBLP96]. **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. Consider a four attribute cube ($ABCD$). Group-by AB can be calculated from $ABCD$, ABD and ABC. Clearly sizes of ABC and ABD are smaller than that of $ABCD$ and are better candidates. 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. This can be extended to disk based data cubes by reducing disk I/O and caching in main memory. For example, after computing ABC from $ABCD$ we compute AB followed by A. An important multi-processor optimization is to **minimize inter-processor communication**. The order of computation should minimize the communication among the processors because inter-processor communication costs are typically higher than computation costs. For example, for a 1D partition, $BC \rightarrow C$ will have a higher communication cost to first aggregate along B and then divide C among the processors in comparison to $CD \rightarrow C$ where a local aggregation on each processor along D will be sufficient.

A lattice framework to represent the hierarchy of the group-bys was introduced in [HRU96]. This is an elegant model for representing the dependencies in the calculations and also to model costs of the aggregate calculations. A scheduling algorithm can be applied to this framework substituting the appropriate costs of computation and communication. A lattice for the group-by calculations for a five-dimensional cube ($ABCDE$) is shown in Figure 1. Each node represents an aggregate and an arrow represents a possible aggregate calculation which is also used to represent the cost of the calculation.

Calculation of the order in which the GROUP-BYs are created depends on the cost of deriving a lower order (one with a lower number of attributes) group-by from a higher order (also called the *parent*) group-by. For example, between ABD $\rightarrow$ BD and BCD $\rightarrow$ BD one needs to select the one with the lower cost. Cost estimation of the aggregation operations can be done by establishing a cost model. Some calculations do not involve communication and are *local*, others involving communication are labeled as *non-local*. Details of these techniques for a parallel implementation using multidimensional arrays can be found in [GC97a]. However, with chunking and presence of sparse chunks the cube size cannot be taken for calculating computation and communication costs. Size estimation is required for sparse cubes to estimate computation and communication costs when dimension aggregation operations are performed. We use a simple analytical algorithm for size estimation in presence of hierarchies presented in [SDNR96]. This is shown to perform well for uniformly distributed random data and also works well for some amount of skew. Since we need reasonable estimates to select the materialization of a sub-cube from a sub-cube at a higher level, this works well.

4.3 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. Figure 6 shows the data structures for our design and the ones which are paged in and out from disk into main memory on each processor.

A global cube topology is maintained for each sub-cube 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 sub-cubes. 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 multidimensional 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)

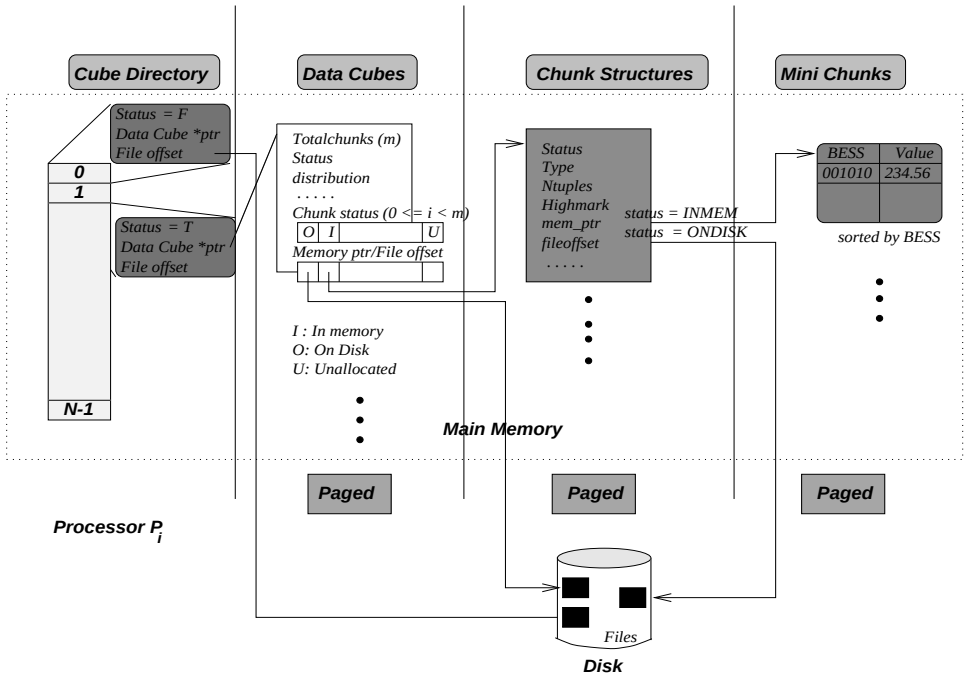


Figure 6: Data Cube Architecture

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.

On each processor, there is one file for the chunk structure of each cube, one file for the minichunks of all the sparse chunks in the cube, one file for the multi-dimensional chunks of the cube (or if the cube is a multi- dimensional array). One file is used for the datacube structure. For 10 (20) dimensions there are 1024 (1048576) chunk files and 1024 (1048576) chunk structure files. With a limit of 2000 open files, file management is required to use the open file descriptors efficiently.

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. Also, a chunked organization may be converted into a multi-dimensional array as illustrated in Figure 4.

5.1 Local - Non Local Aggregation

Figure 7 illustrates the local and non-local aggregation calculations for chunks. The same partitioning dimensions in the source and target sub-cubes result in a local aggregation calculation. For example, $\underline{AB}C \rightarrow \underline{AB}$ has both A and B partitioned in both sub-cubes and this results in a local aggregation. On the other hand, $\underline{AB}C \rightarrow \underline{A}B$, has only A partitioned in the result sub-cube and B goes from being distributed to being undistributed. This results in communication and is a non local aggregation. Other cases are illustrated in Table 1.

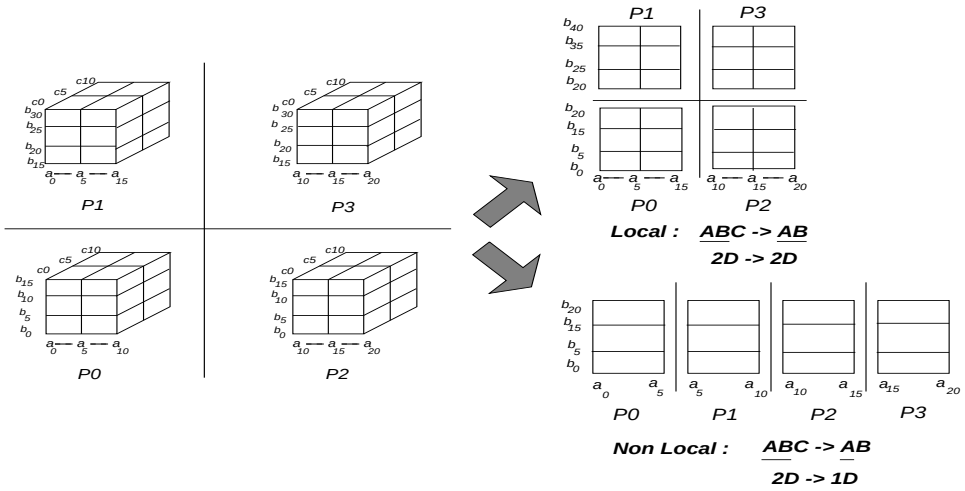


Figure 7: Local and Non Local aggregation calculations

The distribution of the sub-cubes is ascertained by the cardinalities of its two largest dimensions. A sub-cube can be 2D, 1D distributed or be on a single processor. To generate the schedule of data cube calculations, the lattice structure is used, exploiting the various optimization discussed earlier in the section on schedule generation. 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.

The extents of a chunk of the aggregat*ing* cube can be contained in the extents of a chunk of the aggregat*ed* 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.

5.2 Sparse - Sparse Chunk Aggregation

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. Figure 8 shows an aggregation in which B is aggregated in ABC to give AC sub-cube. Here the bit encoding for B is masked from the BESS values for both chunks, which means we are aggregating along dimension B. 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.

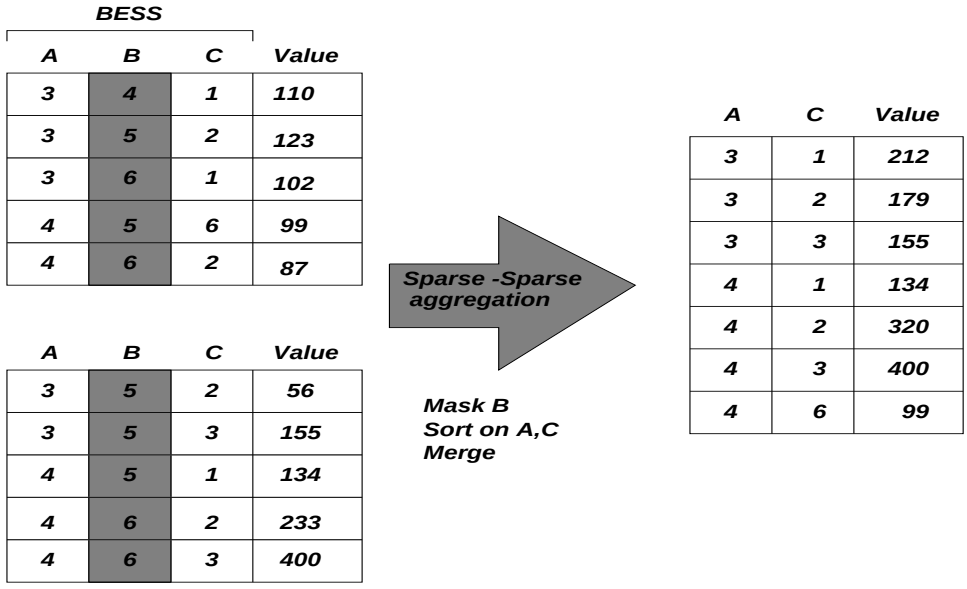


Figure 8: Sparse - sparse chunk aggregation

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. Figure 9 illustrates these two choices. 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.

It is seen that the dimension oriented access performs better than chunk ordering since the sort can be done more efficiently than the sort-merge operation which involves disk I/O.

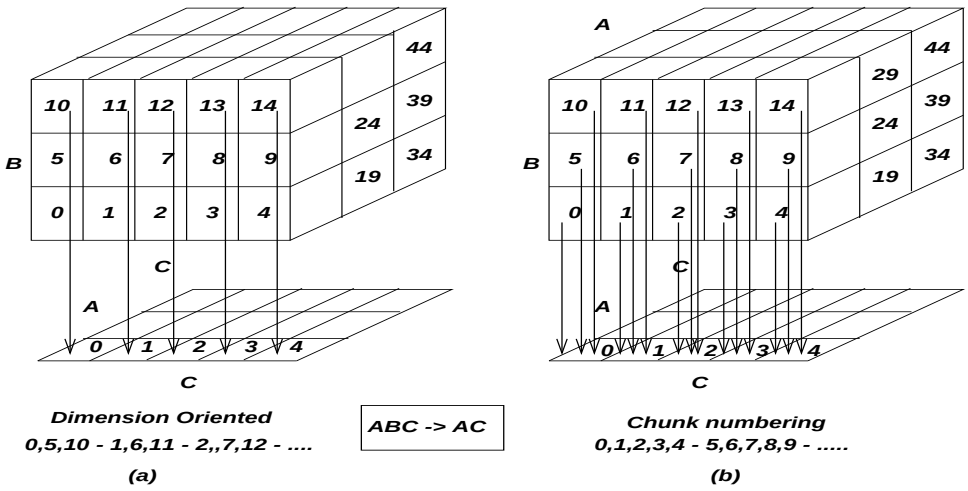


Figure 9: Dimension oriented vs. Chunk numbering access of chunks

5.3 Sparse - Dense Chunk Aggregation

A sparse chunk can be aggregated to a dense chunk by converting the BESS dimension encodings to a chunk offset value. The chunk structure has the values for the index of the chunk in each dimension in the variable *cidx*. This is used to calculate the offset of the chunk in the resulting cube topology by using the number of chunks in each dimension of the result sub-cube.

The BESS index values are extracted and each dimension value is multiplied with the appropriate dimension factor for the chunk stored in the chunk structure. This converts the BESS values to an offset within the chunk which is added to the chunk offset after adjusting the source chunk offset with the resulting chunks offset. The value is then aggregated to this location in the resultant dense chunk.

5.4 Chunked - Multidimensional array Aggregation

A chunk is aggregated to a dense multi-dimensional array in a similar way as a sparse - dense chunk described above. The chunk offset is calculated from *cidx* and the BESS value is used to calculate the offset within the chunk. This is added to the chunk offset which is the correct offset in the resulting multi-dimensional array. This could either be local or non-local. Local values are aggregated to the correct offset. Non local offsets are copied to a buffer for each processor along with the value at that location. Once the buffer is full it is sent to the appropriate processor, which receives it and does the aggregation.

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 (Figure 5) 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

Data set	No. of dimensions	Cardinalities (d_i)	Product ($\prod_i d_i$)	Tuples
Dataset I	3	1024(S),256(N), 512(N)	2^{27}	1.34 million
Dataset II	5	1024(S),16(N),32(N),16(N),256(S)	2^{31}	2.14 million
Dataset III	10	1024(S),16(S),4(S),16,4,4,16,4,4,32(N)	2^{37}	1.37 million

We choose 3 data sets, one each of dimensionality 3, 5 and 10 to illustrate performance. Random data with a uniform distribution is currently used for the performance figures. We are evaluating other types of data (e.g skewed with Zipfian distribution) including real OLAP data sets (including the OLAP benchmark [Cou97], which we used in our earlier study), for a better characterization of performance under different workloads.

Table 2 illustrates the data sets for our experiments. The chunk sizes in each dimension for the 3 dimensional data set are 32, 8 and 16, for a total chunk dimension product of 2^{12}. The number of chunks is 2^{15} for the base cube, but this gets distributed across processors. The number of chunks for each sub-cube is calculated by keeping the chunk dimension sizes fixed and using the dimensions included in each sub-cube. The 5 dimensional data set has chunk sizes in each dimension of 32, 4, 8, 8, 16 for a total chunk dimension product of 2^{17}. The number of chunks is 2^{14} for the base cube and are calculated for each sub-cube depending on the dimensions it has, again getting distributed across processors. Finally, for the 10 dimensional data set the chunk dimension sizes are 64, 4, 2, 4, 2, 2, 4, 2, 2, 4 for a total chunk dimension product of 2^{19} and total number of chunks distributed across processors as 2^{18} for the base cube.

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.

Currently the parallel tuple partitioning is a main memory implementation so we have to restrict the number of tuples to a few million. We are currently implementing a disk based partitioner which will allow us to experiment with larger number of tuples and hence operate at higher densities. We also want to use a larger number of processors but since we are currently using the local system with 16 processors we report those results.

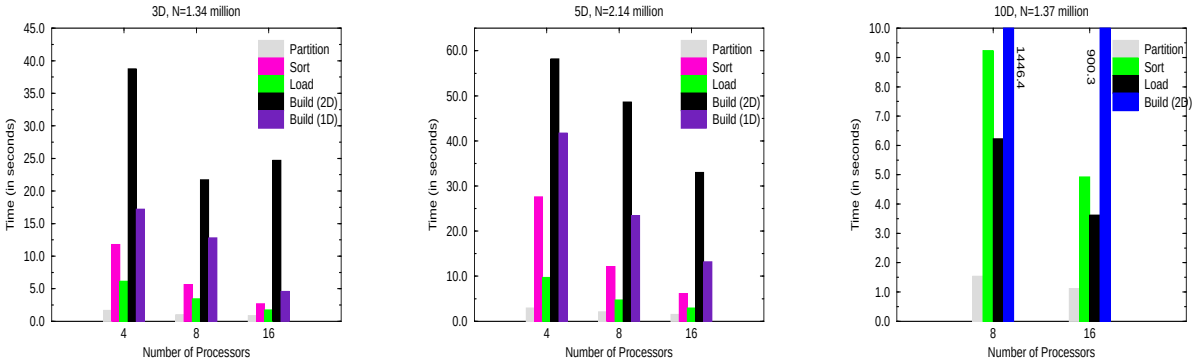


Figure 10: Time taken by various phases of the data cube construction algorithm for datasets with dimensions 3 (8 sub cubes), 5 (32 sub cubes) and 10 (1024 sub cubes)

Figure 10 shows the time taken by the various phases of the data cube construction algorithm. Each phase of the cube construction process shows good speedup for all the data sets when a single dimension is partitioned in the base cube. 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 a few 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.

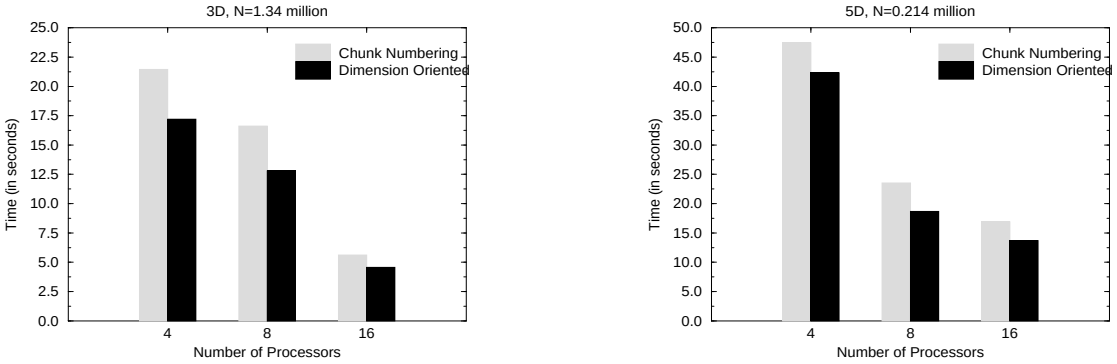


Figure 11: Comparison of the two chunk access methods, chunk numbering and dimension oriented for a dataset with 3 and 5 dimensions

Figure 11 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. This effect is more pronounced on a lower number of processor and lower dimensional data sets because the effect of amortizing sort costs among chunks mapping to the target chunks is more. This is due to the fact that there are more chunks along a given dimension to be aggregated in these cases. This effect will be more pronounced with higher data densities.

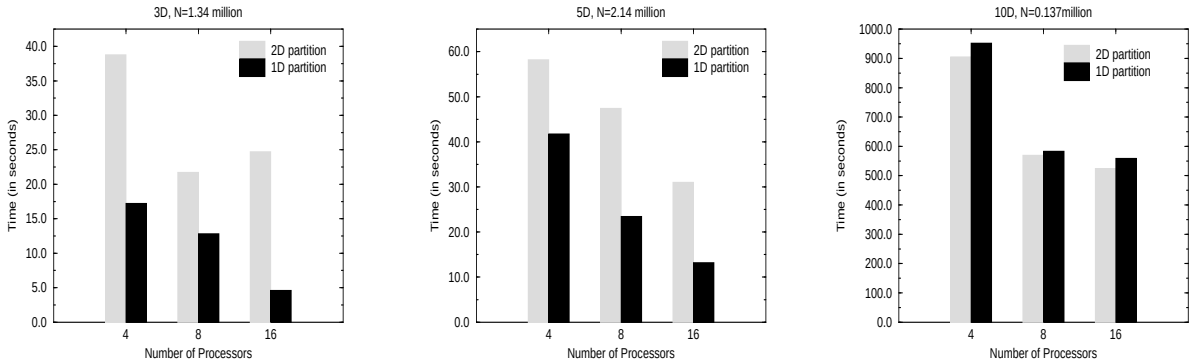


Figure 12: Comparison of partitioning schemes for 3, 5 and 10 dimensional data sets when the initial distribution of the base cube is either 2D or 1D

Figure 12 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 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 checked for their target mapping.

However, for the 10 dimension case, the 2-dimensional distribution is slightly better. This is because the 2 dimensional distribution propagates to higher levels of the cubes and 2 dimensional distribution provides better parallelism in the earlier stages. A large number of cubes are calculated in this case and more cubes can benefit from a 2 dimensional partitioning.

7 Conclusions and Future Work

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 can either be 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 can also be stored as multidimensional arrays if the cardinalities of the dimensions involved are not large and the cube size is below a specific threshold.

Operations between chunked and multi-dimensional array cubes are supported.

The data structures to track the different cubes in a data cube, the chunk structures of each cube and the chunks themselves (using minichunks) use paging to support a large number of cubes, a large number of chunks per cube and a large chunk size. A combination of these results in supporting a large number of dimensions and large data sizes.

For coarse grained parallelism, the cubes can either be partitioned using one largest dimension or a combination of the two largest dimensions. The choice depends on the cardinality of the two largest dimensions and the number of processors. Further, a cube can be allocated on a single processor if its size is small and the overhead of parallelization is high. Operations between these distributions are supported in our framework.

Preliminary results on data sets with 3, 5 and 10 dimensions show good performance. We are currently trying to use skewed data sets, real life data sets, data sets with larger number of dimensions and a larger number of processors to further instrument the performance and study the scalability of the system. We are also implementing a disk based partitioner to allow us to use a larger number of tuples to use higher density for the cubes. This will allow us to investigate the various memory related parameters which affect performance, some of which are the minichunk size, memory for chunk structures and the threshold where sparse chunks are converted to dense chunks. This will allow the number of disk I/O operations in paging the data structures and the minichunks to be reduced.

Query operations on the data cube for OLAP and data mining over the data cube structure are being studied as a part of this research and that is our next step.

References

[Col96] G. Colliat. OLAP, Relational, and Multi-dimensional Database Systems. In *SIGMOD Record*, volume 25(3), September 1996.

[Cou97] OLAP Council. Olap council benchmark. In *http://www.olapcouncil.com*, 1997.

[GBLP96] 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.

[GC97a] S. Goil and A. Choudhary. High performance olap and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, 1(4), 1997.

[GC97b] 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.

[GC98a] S. Goil and A. Choudhary. High performance data mining using data cubes on parallel computers. In *Proc. International Parallel Processing Symposium*, March 1998.

[GC98b] S. Goil and A. Choudhary. High performance multidimensional analysis and data mining (to appear). In *Proc. SC98: High Performance Networking and Computing Conference*, November 1998.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proc. SIGMOD International Conference on Management of Data*, 1996.

[LS98] W. Lehner and W. Sporer. On the design and implementation of the multidimensional cubestore storage manager. In *15th IEEE Symposium on Mass Storage Systems (MSS'98)*, March 1998.

[Sar97] S. Sarawagi. Indexing olap data. In *Data Engineering Bulletin*, volume 20(1), March 1997.

[SDNR96] A. Shukla, P.M. Deshpande, J. Naughton, and K. Ramaswamy. Storage estimation for multidimensional aggregates in the resence of hierarchies. In *Proc. of the 22nd International VLDB Conference*, May 1996.

[Sof97] Kenan Software. An introduction to multi-dimensional database technology. In *http://www.kenan.com/acumate/mddb.htm*, 1997.

[ZDN97] 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.