An Infrastructure for Scalable Parallel Multidimensional Analysis

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

Abstract

Multidimensional Analysis in On-Line Analytical Processing (OLAP), and Scientific and statistical databases (SSDB) use operations requiring summary information on multi-dimensional data sets. Most common are aggregate operations along one or more dimensions of numerical data values and/or on hierarchies defined on them. 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 since a few dimensions are typical for analysis over summary information. Queries may either be answered from a materialized 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 judgement. 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 address scalability in multi-dimensional systems for analysis in SSDB and OLAP applications. We describe our system PARSIMONY - Parallel and Scalable Infrastructure for Multidimensional Online analytical processing. Sparsity of data sets is handled by using chunks *to store data as a sparse set using a Bit encoded sparse structure. 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.*

Performance results for high dimensional data sets on a distributed memory parallel machine (IBM SP-2) show good speedup and scalability.

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 [9]. 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 [12] 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 [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 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 [2]. 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 and data mining 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. [4] 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. OLAP queries can now be answered from the precomputations available in the data cube. We have also used the summary information available in data cubes for calculating support and confidence measures in data mining of association rules. Further, decision-tree based classification is performed by using the multidimensional model to calculate split points efficiently. We do not discuss data mining in this paper. Details can be found in [3]. We believe, our parallel framework for multidimensional analysis and OLAP can also be used in scientific and statistical databases [7, 10] to deliver scalability and performance in execution of analysis tasks.

The rest of the paper is organized as follows. Section 2 describes OLAP using the data cube operator and associated queries. 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 performance results for cube building and and analysis for communication and I/O for it. 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. OLAP technology can lower information system costs and help end-users work more independently, saving time and costly resources. OLAP has been used in applications such as financial modeling (budgeting, planning), sales forecasting, customer and product profitability exception reporting, resource allocation and capacity planning, variance analysis, promotion planning, and market share analysis [8].

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.

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.

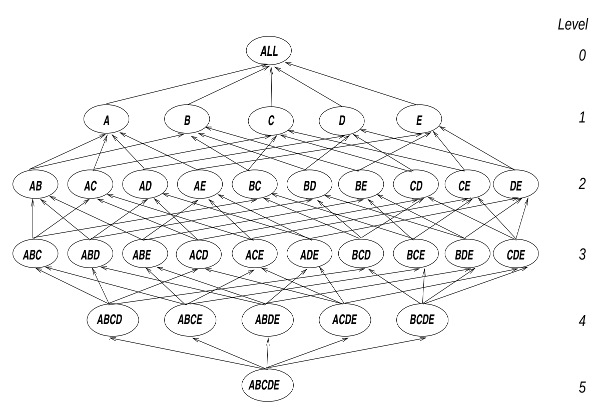


Figure 1. Lattice for cube operator

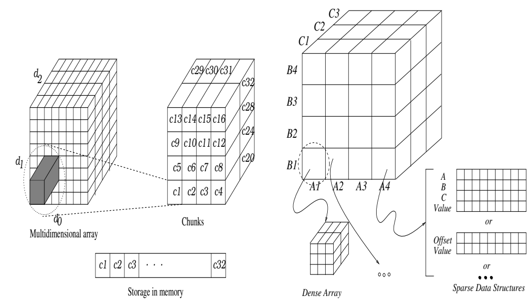
OLAP queries can in many cases be answered by the aggregates in the data cube. Most operations in a data analysis scenario require a multidimensional view of data. **Pivoting** involves rotating the cube to change the dimensional orientation, **Slicing-dicing** involves selecting some subset of the cube, **Roll-up** is an aggregation that can be done at different levels of hierarchy, and **Drill-down** traverses the hierarchy from lower to higher levels of detail. Summarizing these operations, we observe that OLAP requires access to data along a particular dimension or a combination of dimensions. The set of operations required for OLAP and data mining should be efficiently supported by the sparse data structure for good performance. The basic operations include retrieving a random cell element, retrieval of values along a dimension or a combination of dimensions, restricting a range for dimensions in the retrieval, aggregation or some statistical operation for values along a dimension, and multi-dimensional aggregation/consolidation for a hierarchy on dimensions.

Most data dimensions have hierarchies defined on them. Typical OLAP queries probe summaries of data at different levels of the hierarchy. Consolidation is a widely used method to provide roll-up and drill-down functions in OLAP systems. Each dimension in a cube can potentially have a hierarchy defined on it. This hierarchy can be applied to all the sub-cubes of the data cube. We do not go into further detail of these operations as they are well described in the literature.

3 Data Storage: Chunks and BESS

Multidimensional database technology facilitates flexible, high performance access and analysis of large volumes of complex and interrelated data. It is more natural and intuitive for humans to model data. 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 which are 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 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 stor-



(a) Chunking for an array (b) Chunked cube

Figure 2. Storage of data in chunks

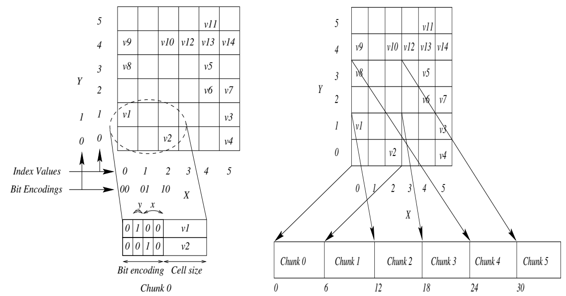
age (BESS). For each cell present in a chunk a dimension index is encoded in $\lceil \log |d_i| \rceil$ bits for each dimension d_i of size $|d_i|$. A 8-byte encoding is used to store the BESS index along with the value at that location. A larger encoding can be used if the number of dimensions are larger 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 logical chunk coordinates in each dimension 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.

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)



(a) Storage using BESS (b) Offsets for index retrieval

Figure 3. BESS used in Chunk Storage

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.

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 one-dimensional (1D) distribution will partition A and a two-dimensional (2D) distribution will partition 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 $|A_i| \times |A_j| \gg p$ 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, dimension A_i is used to choose among the k partitions, and then dimension A_j is further used to find the correct processor among the $\frac{p}{k}$ processors in that group to get the 2D partitioning. Figure 4 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 $k = p$.

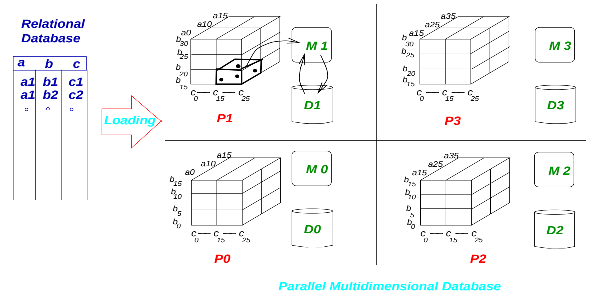


Figure 4. Base cube loading for a 3 dimensional cube on 4 processors

Since 2^n cubes are being constructed in the data cube, 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 in an aggregation calculation, which aggregates a dimension, 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 *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.

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 [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. Consider a four attribute cube ($ABCD$). Group-by AB can be calculated

Table 1. Partitioning of sub-cubes following aggregation calculations

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

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 [6]. 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 [3]. 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 [11]. 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 5 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).

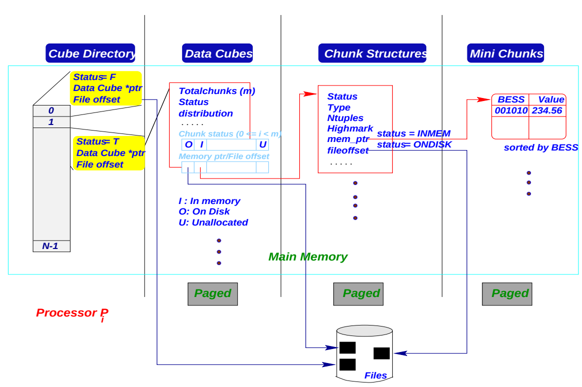


Figure 5. Data Structures on a processor P_i

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) 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 Algorithms and Analysis

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 multidimensional array. In this section we discuss the algorithms for cube aggregations and chunk mappings.

5.1 Chunk mapping to processors

Each chunk in the source cube is processed to map its values to the target chunk. The chunk structure carries information about the chunk's dimensional offsets in *cidx*. This along with *cdim*, the chunk extents, is used to calculate the local value in each dimension. For distributed dimensions we need to add the start of the processor range to calculate the global value. This is then used to calculate the target start and stop values. This is used to determine the destination target processor and the target chunk. The source can map to the same target chunk on the same processor, same target chunk on another processor, split among chunks on the same processor or split among chunks on different processors. These cases are illustrated in Figure 6 for a two dimensional source to target aggregation of chunks. It describes the chunk mapping process and the distinction between *split* and *non-split* chunks, *local* mapping and *non-local* mappings.

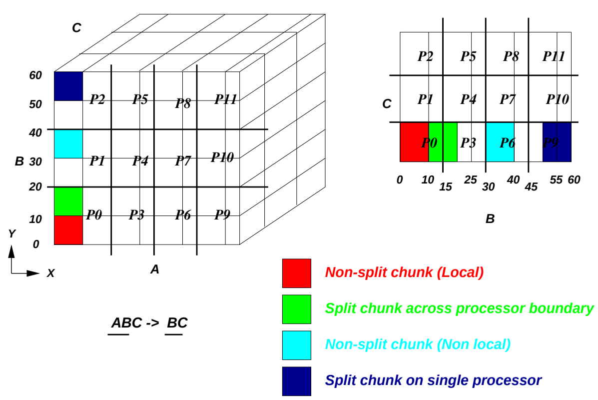


Figure 6. Chunk mapping after aggregation for a 2D cube distribution

Each mapping to a target chunk on the local processor is copied to a local aggregation buffer. If this buffer is full or the current value maps to a different target chunk than the ones in the buffer, it is aggregated with the target. For a non-local calculation a send buffer is kept for each remote processor. A split chunk needs to evaluate each of the index values by decoding the BESS values and adding it to the chunk index values. A target processor needs to be evaluated for the distributed dimensions since this can potentially be different. For a split source chunk, a corrected BESS+value and target chunk id. is sent, otherwise just the BESS+value is sent. Asynchronous send is used to overlap computation and communication. Hence, before the send buffer to a processor is reused, a receive of the previous send must be completed. Asynchronous receive operations are posted from all processors and periodically checked to complete the appropriate sends. A processor receives the BESS values and the target chunk id. and does the aggregation operation.

For a conversion of a chunked source cube to a multidimensional target array, offsets are calculated. Dense chunks are similarly treated. This is explained in a later section.

5.2 Local - Non Local Aggregation

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.

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.

5.3 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, merge 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.

Algorithm 1 *Sparse - sparse chunk aggregation algorithm*

```
for i ← 1 to N_p
        visited[i] = FALSE;
done = FALSE
Initialize prev_target_chunk_id
for i ← 1 to N_p
        nextchunk = i
        if(! visited[nextchunk])
                do {
                /* Check if this is the chunk mapping to the same target */
                        for each BESS value in nextchunk
                                if(Buffer is not full and target_chunk_id =
                                prev_target_chunk_id)
                                        for j ← 1 to ndim_p
                                           Mask dim_agg_p from BESS index and
                                        copy to memory buffer
                                else
                                        Sort memory buffer on masked BESS indices
                                        Merge sorted buffer with sorted BESS values
                                        of target_chunk_id
                                        Empty the buffer for reuse
                /* If dimension oriented chunk acccess, get the next chunk along
                dimension dim_agg_p */
                if(chunk_access = DIM_ORIENTED)
                        if((nextchunk ← Next_chunk()) = FALSE)
                                done = TRUE
                        else
                                done = TRUE
                visited[nextchunk] = TRUE
                } while (! done)
end
```

Figure 7. Chunk aggregation algorithm

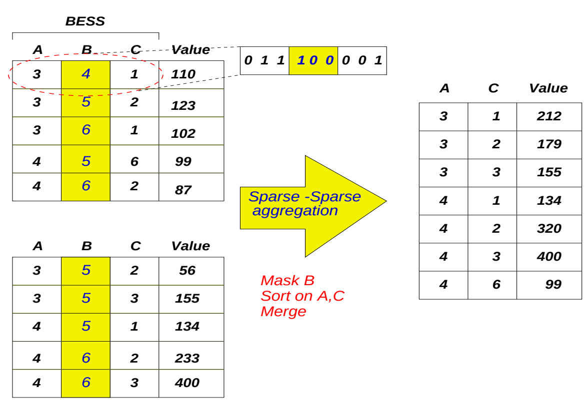


Figure 8. Sparse - sparse chunk aggregation

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 ori-*

ented 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.

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.

5.4 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.

5.5 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.

5.6 Interprocessor communication

The processor with the least load is assigned to a uniprocessor sub-cube for load balancing. To perform a parallel aggregation calculation every processor cooperates since each contains a part of the source and possibly target cube. Since there are dependencies between any two levels of the lattice, i.e between the source and the target, the latter cannot be used for any further calculation unless it is calculated completely. This is especially important when the calculation is in parallel since other processors contribute to the aggregated values. Each processor goes through its chunks assigned to it independently and communicates with other processors by sending and receiving messages. Asynchronous sends and receives are used to overlap computation with communication and reduce the synchronization that results from waiting for messages. For example, When a processor needs to send a message in buffer B, it posts a asynchronous send and goes on to do its processing. But before modifying B again it needs to check that the message has been sent. Hence, each processor also posts a asynchronous receive to get values from other processors. Once data is received from a particular processor, another asynchronous receive is posted for that processor. Processors exchange sentinel values to signal the end of the current communication phase. Figure 9 shows the communication phase on each processor.

Algorithm 2 *Communication in source $\rightarrow$ target aggregation*

for $i \leftarrow 0$ to P, $i \neq myid$
  Post **Asynchronous RECEIVE** from processor i
  Initialize $send\_request[i]$ to **FALSE**
for $j \leftarrow 1$ to N_p (for each chunk)
  ...Computation...
  Find target processor for chunk as $proc$
  While ($send\_request[proc] = TRUE$)
    Wait for b to be received at $proc$ before modifying b
  if($target_{proc} \neq myid$)
    Post **Asynchronous SEND** to processor $proc$
    Set $send\_request[proc]$ to $TRUE$
  for $i \leftarrow 0$ to P, $i \neq myid$
    If($Test\_for\_receives[i] = TRUE$)
      Receive and aggregate buffer
      [This will set $send\_request[myid]$ to FALSE on processor i]
      Post **Asynchronous RECEIVE** from processor i
end

Figure 9. Communication in source $\rightarrow$ target aggregation

5.7 Queries

Queries are supported by executing them from the appropriate aggregated cube. This is determined by the attributes of the query and the dimensions present in the cube. A query is described by giving a range in the dimensions desired, which is translated into chunk numbers to be retrieved. Each processor has a global topology for each cube and determines the inclusion of its local domain in the query. Hierarchies are defined on dimensions which are used for roll-up operations. An attribute with an order defined on its elements keeps it as a dimension with the order. An attribute, with no order is organized as a collection of values ordered by the hierarchy. A hierarchy is defined for each dimension, which is correspondingly used for each cube materialized. Related queries take advantages of the chunks already fetched for another query. Only the remaining chunks need to be read from disk.

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 4) from these tuples locally on each processor. The sub-cubes of the data cube are calculated from here (*Building phase*). This is followed by the analysis and data mining phase on the computed aggregates.

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

Data	Dim	Cardinalities (d_i)	($\prod_i d_i$)	Tuples
I	3	1024(S),256(N), 512(N)	2^{27}	10 million
II	5	1024(S),16(N),32(N),16(N),256(S)	2^{31}	1 and 10 million
III	10	1024(S),16(S),4(S),16,4,4,16,4,4,32(N)	2^{37}	5 and 10 million
IV	20	16(S),16(S),8(S),2,2,2,2,4,4, 4,4,4,8,2,8,8,8,2,4,1024 (N)	2^{51}	1 million

We choose four data sets, one each of dimensionality 3, 5, 10 and 20 to illustrate performance. Random data with a uniform distribution is currently used for the performance figures. Random data with skewed distributions result in similar performance characteristics. OLAP council benchmark [8] which models a realistic business scenario has also been used for performance evaluation. A aubset of results is presented here due to limited space.

Table 2 illustrates the data sets for our experiments. The 3 dimensional data set has chunk sizes in each dimension as 32, 16, 8 for a chunk dimension product as 2^12 and the number of chunks as 2^13 distribute across processors. The other data sets are in 5, 10 and 20 dimensions. The number of sub cubes in the datacube for Dataset I is $2^3 = 8$, dataset II is $2^5 = 32$ and dataset III is $2^{10} = 1024$. We report the results of complete data cube construction for these sets. For Dataset IV we report the results of partial data cube construction in which 1350 sub-cubes are calculated.

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. A two dimensional partitioning performs better than a one dimensional partitioning because there are more chunks in the partitioned dimension that allows for better sparse-sparse aggregation performance.

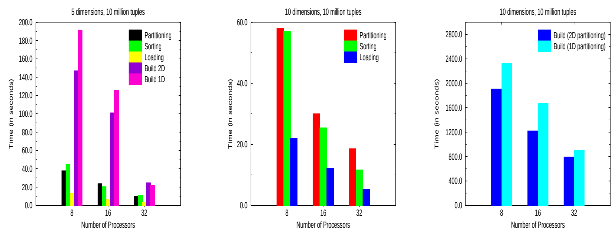


Figure 10. Time taken by phases of the data cube construction algorithm for 5 (32 sub cubes) and 10 (1024 sub cubes) dimensions

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 because of better buffer use in the sparse aggregation calculations. 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.

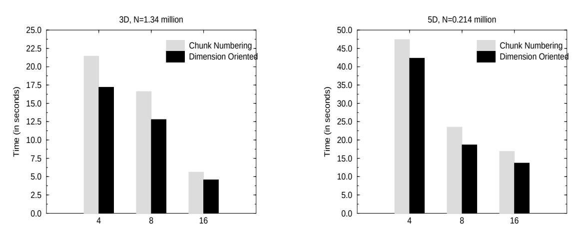
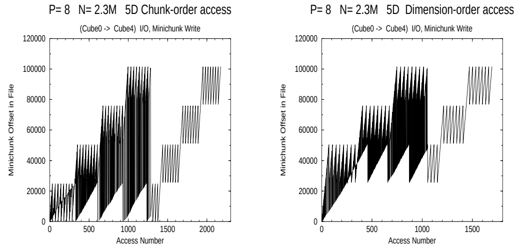


Figure 11. Comparison of the two chunk access methods, chunk numbering and dimension oriented for a dataset with 3 and 5 dimensions

Figure 12 compares the minichunk write performance for the two chunk access orders.

A minichunk write occurs to flush the aggregation buffer while performing the sparse aggregation. This can happen if the buffer is full or when the target mapping of the source chunks change. We observe that there are lesser writes for dimension order access and the file offsets that are written have better locality than the chunk-order access mechanism.



(a) (b)

Figure 12. Minichunk writes to Cube 4 on 8 processors 5D, 2.37M (a) Chunked-order access (b) Dimension-order access

A larger chunksize means lesser number of chunks and hence lesser chunk traversal and aggregation overhead. Hence we observe lower numbers for it in our experiments. Choice of a chunk size depends on the number of dimensions, size of the dimensions and density of the data set. A chunksize should be maximized given these variables. Also observed is the time of aggregation increases as the outermost dimension is aggregated. Time taken for $ABC \rightarrow AB$ is lesser than time taken for $ABC \rightarrow AC$ since the former reads minichunks which are stored almost contiguously. The latter has to index to the correct file location to get to the minichunk offset. This is because the storage order of chunks is C (innermost), B and A (outermost).

Figure 13 shows preliminary results of executing 100 random queries on the base level of data on cubes chosen at the highest two levels (basecube and the next level). Ranges are chosen randomly from a set of these cubes and tuples are extracted from minichunks of chunks in the range. We are currently evaluating query performance on different data sets and will include them in the final version of the paper.

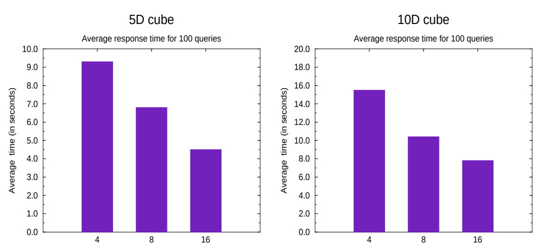


Figure 13. Average time for 100 random range queries for 5D and 10D datasets

7 Conclusions

In this paper we have presented the design and implementation of a scalable parallel system for multidimensional analysis. Using the multidimensional data model, data is stored in *chunks* which can either be sparse or dense. Sparse chunks are represented by a bit encoding which can be used for efficient aggregation operations on compressed data. For maximum efficiency of operations dense regions 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. Use of parallelism further enhances scalability. Parallelism has been used to support a large number of dimensions and large data sets for effective data analysis and decision making. We believe it can serve as a suitable platform for high performance SSDB applications.

References

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

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

[3] S. Goil and A. Choudhary. High Performance OLAP and Data Mining on Parallel Computers. *Journal of Data Mining and Knowledge Discovery*, 1(4), 1997.

[4] S. Goil and A. Choudhary. High performance multidimensional analysis and data mining. In *Proc. SC98: High Performance Networking and Computing Conference*, November 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] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proc. SIGMOD International Conference on Management of Data*, 1996.

[7] 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.

[8] OLAP Council, http://www.olapcouncil.com. *OLAP Council Benchmark*, 1997.

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

[10] A. Shoshani. OLAP and Statistical Databases: Similarities and Differences. In *Proc. Principles of Database Systems*, 1997.

[11] A. Shukla, P. 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.

[12] 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.