A Parallel Scalable Infrastructure for OLAP and Data Mining

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

Decision support systems are important in leveraging information present in data warehouses in businesses like banking, insurance, retail and health-care among many others. The multi-dimensional aspects of a business can be naturally expressed using a multi-dimensional data model. Data analysis and data mining on these warehouses pose new challenges for traditional database systems. OLAP and data mining operations require summary information on these multi-dimensional data sets. Query processing for these applications require different views of data for analysis and effective decision making. Data mining techniques can be applied in conjunction with OLAP for an integrated business solution. As data warehouses grow, parallel processing techniques have been applied to enable the use of larger data sets and reduce the time for analysis, thereby enabling evaluation of many more options for decision making.

In this paper we address (1) scalability in multi-dimensional systems for OLAP and multi-dimensional analysis, (2) integration of data mining with the OLAP framework, and (3) high performance by using parallel processing for OLAP and data mining. We describe our system PARSIMONY - Parallel and Scalable Infrastructure for Multidimensional Online analytical processing. This platform is used both for OLAP and data mining. Sparsity of data sets is handled by using sparse chunks *using a bit-encoded sparse structure for compression, which enables aggregate operations on compressed data. Techniques for effectively using summary information available in data cubes for data mining are presented for mining* Association rules *and decision-tree based* Classification. *These take advantage of the data organization provided by the multidimensional data model.*

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

Keywords: OLAP, Data Cube, Data Mining, High Performance, Parallelism, Multi-dimensional analysis, Chunks, Bit-encoding

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.

We compare the storage and operational efficiency in OLAP and multi-dimensional analysis of various sparse data storage schemes in [6]. 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. Chunks provide a multi-dimensional index structure for efficient dimension oriented data accesses much the same as multi-dimensional arrays do.

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. Precomputed values are used in the probability calculations for association rule mining. Also, classification trees can be built by using the aggregates of class id. counts to calculate the splitting criterion for each dimension.

The rest of the paper is organized as follows. Section 2 describes OLAP and multidimensional analysis 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 describes data mining algorithms like association rule mining and classification on the multidimensional structure and its parallelization. Section 7 presents performance results for cube building and and analysis for communication and I/O for it. Section 8 concludes the paper.

2 OLAP and Multidimensional Analysis

OLAP is used to summarize, consolidate, and synthesize data according to multiple dimensions. It 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 [3]. Multi-dimensional database techniques (MOLAP) have been applied to decision-support applications. 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 [9]. 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. 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.

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.

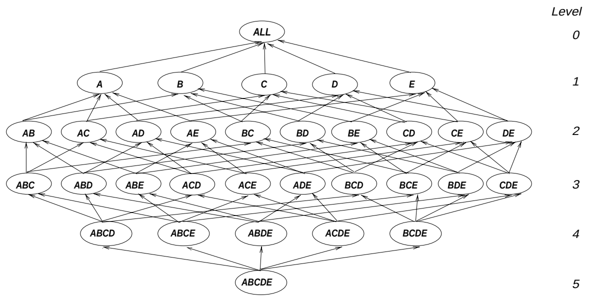
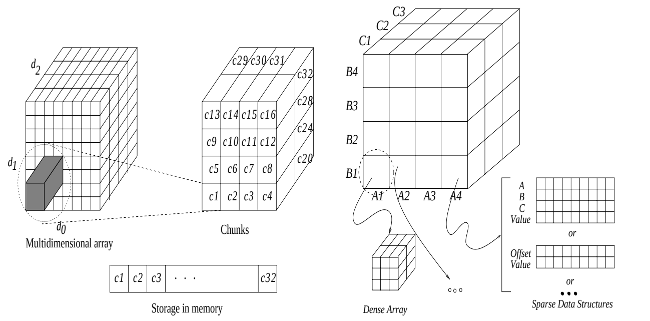


Figure 1. Lattice for cube operator

3 Data Storage: Chunks and BESS

Multidimensional database technology facilitates flexible, high performance access and analysis of large volumes of data [18]. It is more natural for humans to visualize a multi-dimensional structure. Multi-dimensional arrays are the most intuitive of these structures. Chunking has been used to store arrays for better access performance as a collection of dense data blocks [15]. 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.



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

Figure 2. Storage of data in chunks

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 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.

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 3 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 $n-1$ [8].

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 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.

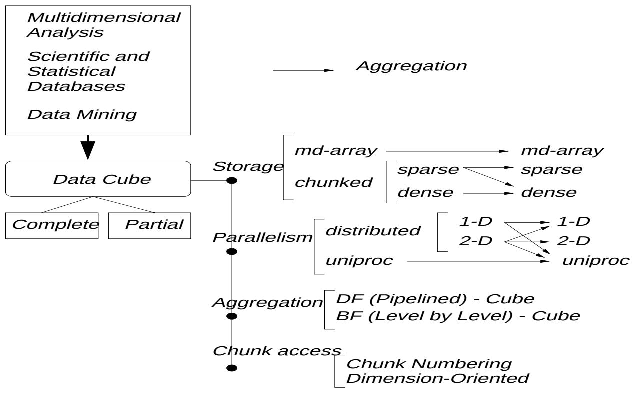


Figure 3. Data Cube Architecture

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 repartitioned.

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. Also, as illustrated in Figure 3, 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 multidimensional 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*
$2D \rightarrow 2D$	$\underline{AB}C \rightarrow \underline{AB}$	$\underline{AB}C \rightarrow \underline{BC}$	$\underline{AB}C \rightarrow \underline{AC}$
$2D \rightarrow 1D$		$\underline{AB}C \rightarrow \underline{B}C$	$\underline{AB}C \rightarrow \underline{A}C$
$1D \rightarrow 1D$	$\underline{A}BC \rightarrow \underline{A}B, \underline{A}C$	$\underline{A}BC \rightarrow \underline{B}C$	
$2D \rightarrow UNI$		$\underline{AB}C \rightarrow BC$	$\underline{AB}C \rightarrow AC$
$1D \rightarrow UNI$		$\underline{A}BC \rightarrow BC$	$\underline{A}BC \rightarrow AC$
$UNI \rightarrow UNI$	$ABC \rightarrow AB, AC$		

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 [9]. **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 [12]. 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 [5]. 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 [17]. 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 4 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,

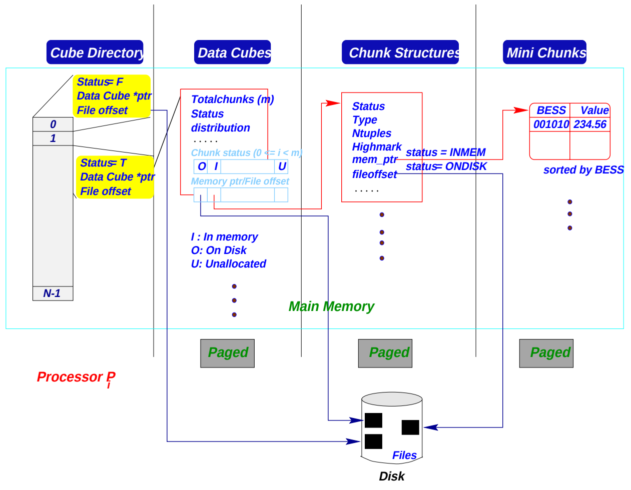


Figure 4. Data Structures on a processor P_i

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.

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. The various options are illustrated in Figure 3. 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 5 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. For a detailed algorithm refer to [7].

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.

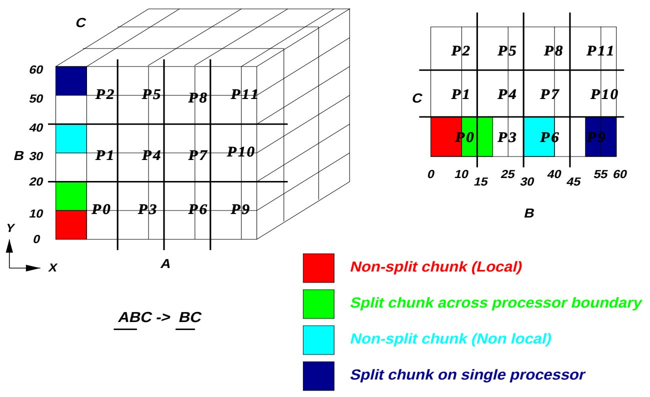

C
P2 P5 P8 P11
P1 P4 P7 P10
P0 P3 P6 P9
B
Y
X
A
ABC -> BC
Non-split chunk (Local)
Split chunk across processor boundary
Non-split chunk (Non local)
Split chunk on single processor

Figure 5. Chunk mapping after aggregation for a 2D cube distribution

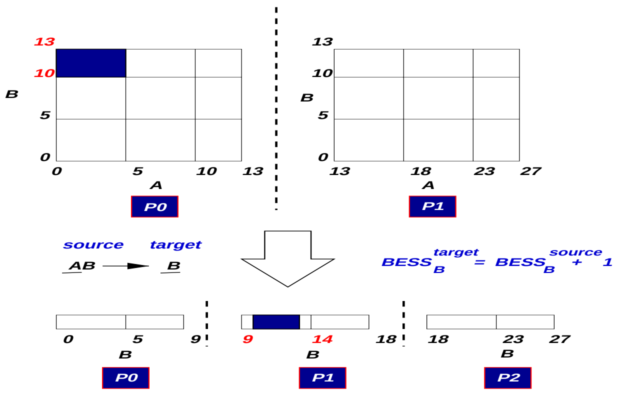


Figure 6. Modification of BESS indices while mapping a non-split chunk

5.2 Aggregation Computations

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 extents of a chunk of the aggregating cube can be contained in the extents of a chunk of the aggregated cube. In this case the BESS+value pairs are directly mapped to the target chunk, locally or non-locally. However, the BESS index values need to be modified to encode the offsets of the new chunk. Figure 6 illustrates a case for $\underline{A}B \rightarrow \underline{A}$ where the chunks with A extents of $10 - 13$ on processor P0 map to the chunk with extents $9 - 13$ on processor P1. The BESS encoding of A needs to be incremented by 1 to correctly reflect the target BESS encoding. If the chunk is overlapping over a target chunk boundary, then each BESS value has to be extracted to determine its target chunk. This is computationally more expensive than the direct case. It is to be noted that a 2 dimensional distribution may result in more overlapped chunks than a 1 dimensional distribution, because the former has more processor boundary area than the latter.

Sparse chunks store BESS+value pairs in minichunks. Sparse to sparse aggregations involve accessing these minichunks. The BESS values are kept sorted in the minichunks to facilitate the aggregation calculations by using sort, merge and scan operations used in relational processing. A sparse chunk can be aggregated to a dense chunk by converting the BESS dimension encodings to a chunk offset value. The detailed algorithms are described in [7].

6 Data Mining

Data mining can be viewed as an automated application of algorithms to detect patterns and extract knowledge from data [4]. We briefly describe an integration of association rule mining and classification with the parallel OLAP and multidimensional analysis infrastructure presented in this paper.

6.1 Association Rule Mining

An association rule is an expression $A \Rightarrow B$, where A and B are sets of items contained in a set of transactions. This means that a transaction in the database that contains the items in A tend to contain the items in B, with a certain probability. These are captured in the metric *support* and *confidence*. Association rule mining has applications in cross-marketing, attached mailing, add-on sales, store layout and customer segmentation based on buying patterns to name a few.

Discovery of quantitative rules is associated with quantitative information from the database. The data cube represents quantitative summary information for a subset of the attributes. Attribute-oriented approaches [1] [10] [11] to data mining are data-driven and can use this summary information to discover association rules. *Support* of a pattern A in a set S is the ratio of the number of transactions containing A and the total number of transactions in S. *Confidence* of a rule $A \rightarrow B$ is the probability that pattern B occurs in S when pattern A occurs in S and can be defined as the ratio of the support of AB and support of A. The rule is then described as $A \rightarrow B$ *[support, confidence]* and a *strong* association rule has a support greater than a pre-determined minimum support and a confidence greater than a pre-determined minimum confidence. This can also be taken as the measure of *"interestingness"* of the rule. Calculation of support and confidence for the rule $A \rightarrow B$ involve the aggregates from the cube AB, A, B and ALL. Support is calculated as Prob($A \cup B$) = $\frac{Number\_of\_transactions(A \cup B)}{Total\_transactions}$, which involve values from cube A, B and ALL. Confidence is Prob($B|A$) = $\frac{Number\_of\_transactions(A \cap B)}{Number\_transactions(A)}$, which involves cubes AB and A. Since the data cube has these summaries for all possible combinations of A and B, computation of support and confidence can be done for all possible pairs. Similarly, association between more attributes can be generated by looking at cubes with more attributes.

Additionally, dimension hierarchies can be utilized to provide multiple level data mining by progressive generalization (roll-up) and deepening (drill-down). This is useful for data mining at multiple concept levels and interesting information can potentially be obtained at different levels. An approach to data mining called Attribute Focusing targets the end-user by using algorithms that lead the user through the analysis of data. Attribute Focusing has been successfully applied in discovering interesting patterns in the NBA [1] and other applications. Since data cubes have aggregation values on combinations of attributes already calculated, the computations of attribute focusing are greatly facilitated by data cubes. We present a parallel algorithm to calculate the interestingness function used in attribute focusing on the data cube.

6.2 Decision-tree based Classification

Classification is used for predictive data mining. Applications that look at the known answers in the past and leverage from it in the future can take advantage of this technique. A set of sample records called the *training data set* is given consisting of several attributes. Attributes can either be *continuous*, if they come from an ordered domain, or *categorical*, if they are from an unordered domain. One of the attributes is the *classifying* attribute that indicates the *class* to which the record belongs. The objective of classification is to build a model of the classifying attribute based upon the other attributes of the record. We use the decision tree model because they are relatively inexpensive to construct, easy to interpret and easy to integrate with data base systems.

Recent work has focused on using the entire data set, in classifiers like SLIQ [14] and SPRINT [16]. A parallel classifier in the same spirit has been developed in ScalParC [13]. The approach of SPRINT, SLIQ is to sort the continuous attribute once in the beginning and maintain the sorted order in the subsequent splitting steps. Separate lists are kept for each attribute which maintains a record identifier for each sorted value. In the splitting phase the same records need to be assigned to a node, which may be in a different order in the different attribute lists. A hash table is used to provide a mapping between record identifiers and the node to which it belongs after the split. This mapping is then probed to split the attribute lists in a consistent manner.

Table 2 is an example training set with three attributes, Age, Car color and Gender, and a class attribute. Figure 7(a) shows the classification tree for it. At each node the attribute to split is chosen that best divides the training set. Several splitting criteria have been used in the past to evaluate the goodness of a split. Calculating the *gini* index is commonly used [2]. This involves computing the frequency of records of each class in each of the partitions. If a parent node having n records and c possible classes is split into p partitions, the *gini* index of the i^{th} partition is $gini_i = 1 - \sum_{j=1}^{c} (n_{ij}/n_i)^2$. n_i is the total number of records in partition i, of which n_{ij} records belong to class j. The *gini* index of the total split is given by $gini_{split} = \sum_{i=1}^{p} (n_i/n) gini_i$. The attribute with the least value of $gini_{split}$ is chosen to split the records at that node. The matrix n_{ij} is called the *count matrix*. The count matrix needs to be calculated for each evaluated split point for a continuous attribute. Categorical attributes have only one count matrix associated with them, hence computation of the gini index is straightforward. For the continuous attributes an appropriate splitting value has to be determined by calculating the $gini_{split}$ and choosing the one with the minimum value. If the attribute is sorted then a linear search can be made for the optimal split point by evaluating the gini index at each attribute value. The count matrix is calculated at each possible split point to evaluate the $gini_{split}$ value. Splitting the split-attribute is straightforward by adjusting pointer values. The challenge is to split the non-split attributes efficiently. Existing implementations such as SPRINT and ScalParC maintain a mapping of the row-id and class-id with the values assigned to each node. The values are split physically among nodes, such that the continuous attribute maintain their sorted order in each node to facilitate the sequential scan for the next split determination phase. A hash list maintains the mapping of record ids to nodes. The record ids in the lists for non-splitting attributes are searched to get the node information, and perform the correct split.

Table 2. Training Set

Row-id	Age	Car-Color	Gender	Class-id
0	10	Green	F	0
1	50	Blue	M	1
2	40	Yellow	F	0
3	30	Green	F	0
4	20	Red	M	1
5	40	Blue	M	0
6	20	Yellow	M	1

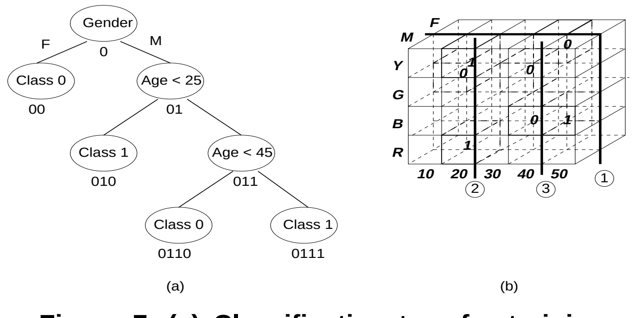


Figure 7. (a) Classification tree for training set (b) Classification tree embedded on the base cube

6.3 Classification on the cube structure

We propose that classification trees can be built using structure imposed on data using the multidimensional data model. Gini index calculation relies on the count matrix which can be efficiently calculated using the dimensional model. Each populated cell represents a record in the array. For the base cube (which is a multidimensional representation of the records without any aggregation) the class value of the record is stored in each cell. The gini index calculation uses the count matrix which has information about the number of records in each partition belonging to each possible class.

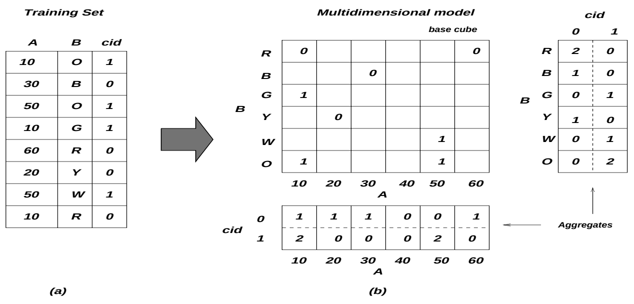


Figure 8. (a) Training set records (b) corresponding multidimensional model

To evaluate split points for a continuous attribute the $gini_{split}$ needs to be evaluated for each possible split point in a continuous attribute and once for a categorical attribute. This means the aggregate calculations present in each of the 1 dimensional aggregates can be used if they have number of records belonging to each class. Therefore for each aggregate we store the number of records in each class. Figure 8(a) gives an example training set with two dimensions, A, a continuous dimension and B a categorical dimension and two class values 0 and 1. Figure 8(b) is the corresponding multidimensional model. The continuous dimensions A is stored in the sorted order. The aggregates store the number of records mapping to that cell for both classes 0 and 1. To calculate the $gini_{split}$ for the continuous attribute attribute A it is now easy to look at the A aggregate and sum the values belonging to both classes 0 and 1 on both sides of the split point under consideration to get the count matrix. Gini index calculation is done on an attribute list which in the case of a multidimensional model is a dimension. Count matrix is repeatedly calculated on the sorted attribute list which is readily available in the cube structure as a higher level one dimensional aggregate. Each dimension is sorted in the dimensional structure as shown in Figure 8(b). Further details are present in [7].

7 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.)

Table 3. Description of datasets and attributes, (N) Numeric (S) String

Data	dim	(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

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

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. We have evaluated other types of data (e.g skewed with Zipfian distribution) including real OLAP data sets (including the OLAP benchmark [3], which we used in our earlier study [5]), for a better characterization of performance under different workloads. 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.

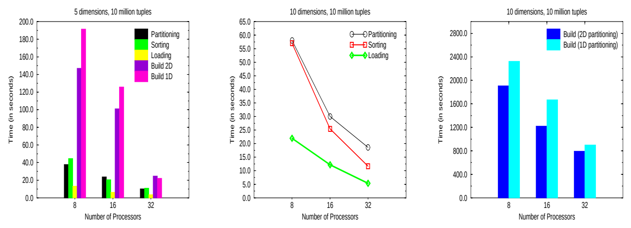
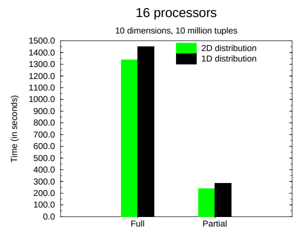


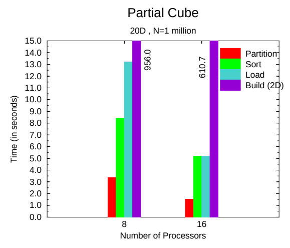
Figure 9. Time taken by various phases of the data cube construction algorithm for 5 (32 sub cubes) and 10 (1024 sub cubes) dimensions

Figure 9 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 shows the time for full and partial cube build for dataset III and for partial cube building for dataset IV. Partial build times are much smaller than the full cube building time.



(a)



(b)

Figure 10. Full and partial cubes (a) 10D, 10 million tuples on 16 processors, (b) partial cubes for 20D on 8 & 16 processors

Other results which are used to evaluate our parallel infrastructure include time to execute OLAP queries, notably range queries and queries over dimension hierarchies. Association rule mining and classification tree building on the multidimensional infrastructure are also evaluated with appropriate data sets. We do not include them here due to lack of space.

8 Conclusions

In this paper we have presented the design and implementation of a scalable parallel system for multidimensional analysis, OLAP and data mining. Using the multidimensional data model, data is stored in *chunks*. 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. This framework has been demonstrated for use in OLAP queries which are ad-hoc in nature and require fast computation times by pre-aggregating calculations. Data mining uses some of the precomputed aggregated calculations to compute the probabilities needed for calculating support and confidence measures for association rules and the split point evaluation in building classification trees. Parallelism has been used to support a large number of dimensions and large data sets for effective data analysis and decision making.

References

[1] I. Bhandari. Attribute Focusing: Data mining for the layman. Technical Report RC 20136, IBM T.J. Watson Research Center, 1995.

[2] L. Breiman, J.H Friedman, R.A Olshen, and C.J Stone. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.

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

[4] U.M. Fayyad, G. Piatesky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in data mining and knowledge discovery*. MIT Press, 1994.

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

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

[7] S. Goil and A. Choudhary. Design and Implementation of a Scalable System for Multidimensional Analysis and OLAP. Technical Report CPDC-9810-020, Northwestern University, October 1998.

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

[9] 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.

[10] J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Trans. on Knowledge and Data Engineering*, 5(1), February 1993.

[11] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. of the* 21^{st} *VLDB Conference, Zurich*, 1995.

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

[13] M. Joshi, G. Karypis, and V. Kumar. Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proc. International Parallel Processing Symposium*, March 1998.

[14] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A Fast Scalable Classifier for Data Mining. In *Proc. of the Fifth Int'l Conference on Extending Database Technology, Avignon*, March 1996.

[15] S. Sarawagi and M. Stonebraker. Efficient organization of large multi-dimensional arrays. In *Proc. of the Eleventh International Conference on Data Engineering*, February 1994.

[16] J.C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A Scalable Parallel Classifier for Data Mining. In *Proc. 22th Int'l Conference on Very Large Databases, Mumbai, India*, September 1996.

[17] 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.

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