PARSIMONY : An Infrastructure for Parallel Multidimensional Analysis and Data Mining*

Sanjay Goil
Performance Technology Group
Sun Microsystems Inc.
Sunnyvale, CA
sgoil@eng.sun.com

Alok Choudhary
ECE Department
Northwestern University
Evanston, IL
choudhar@ece.nwu.edu

Abstract

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

The multi-dimensionality of the underlying problem can be represented both in relational and multi-dimensional databases, the latter being a better fit when query performance is the criteria for judgment. Relational databases are scalable in size for OLAP and multidimensional analysis and efforts are on to make their performance acceptable. On the other hand multi-dimensional databases have proven to provide good performance for such queries, although they are not very scalable. In this article we address (1) scalability in multi-dimensional systems for OLAP and multi-dimensional analysis and (2) Integration of data mining with the OLAP framework. We describe our system PARSIMONY - Parallel and Scalable Infrastructure for Multidimensional Online analytical processing, used for both OLAP and data mining. Sparsity of data sets is handled by using *chunks* to store data either as a dense block using multi-dimensional arrays or a sparse representation 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. Further, we develop parallel algorithms for data mining on the multi-dimensional cube structure for attribute-oriented 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.

*This work was performed while Sanjay Goil was a PhD student in the ECE Department at Northwestern University.

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 [28]. 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 [33] 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 [6], 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. This treats the dimensions non-uniformly, requiring the user to partition the dimensions. 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 [9]. 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 article 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. [12] 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. 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.

Figure 1 motivates the overall design of our parallel MDDB engine for OLAP and data mining. At the lowest level is a parallel machine which can either be a distributed memory message passing machine or a shared memory machine. This machine has a layer of system software that allows programming and provides communication mechanisms. We build the parallel MDDB engine using this layer since the parallel functionality is optimized by the machine vendor. Data distribution, sparse representation and load balancing

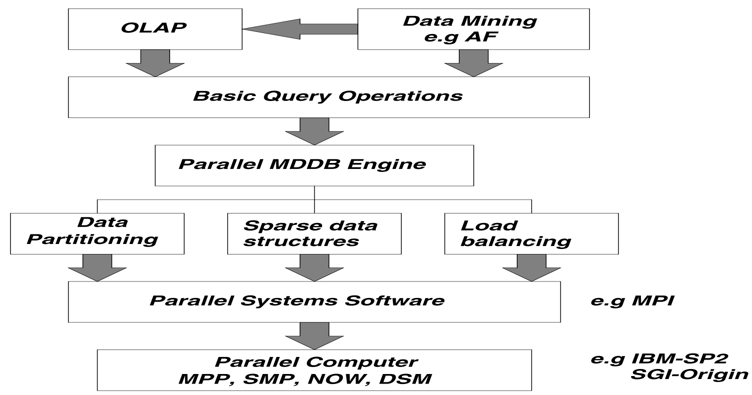


Figure 1: Overall schematic

are issues that are addressed in the parallel design of the framework. Query processing layer for point queries, range queries and hierarchical consolidation queries provide support for OLAP and data mining activities. This layered approach is modular and optimizations are done at each level to tune performance on the underlying architecture. Further details of the this infrastructure can be obtained from [15].

We do not address the question of selective cube materialization here. Our focus in this work is on the scalable maintenance of the multidimensional data cube structure in parallel. Clearly, the selection of sub-cubes for materialization based on their benefit to query response times and cost-saving-ratio, is an important problem and has received much attention recently [20] [7] [29]. We believe these can be easily incorporated in our framework. Our techniques for multidimensional analysis can also be applied to scientific and statistical databases [23] [31].

The rest of the article is organized as follows. Section 2 describes OLAP using the data cube operator and queries on them. 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.Section 7 presents performance results for cube building and and analysis for communication and I/O for it. Section 8 concludes the article.

2 OLAP and Data Cubes

OLAP is used to summarize, consolidate, view, apply formulae to, and synthesize data according to multiple dimensions. Traditionally, a relational approach, using relational databases (relational OLAP) has been taken to build and query such 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), with multidimensional APIs 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 [17]. 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 2 shows an example of a data cube constructed from a data set with three attributes. In relational systems, a *fact* table is constructed, using a *star schema* from the original table. The fact table has keys of the *dimensional* tables and contains the various aggregates, the aggregated attribute represented by a special keyword ALL. Data access for queries is through appropriate relational operations like **join** and **projection**. In SQL, the $CUBE$ operator represents the data cube calculation as illustrated in the Figure. On the other hand, a multidimensional representation uses a multi-cube architecture to store data organized with dimensional information in a suitable multi-dimensional structure. This is a more natural representation of dimensional data and leads to efficient data access operations.

```
SELECT Model, Year, Color, SUM(sales) AS Sales
FROM Sales
WHERE Model IN { 'Ford', 'Chevy'} AND
      Year BETWEEN 1990 AND 1992
GROUP BY CUBE (Model, Year, Color);
```

Model	Year	Color	Sales
Chevy	1990	Red	5
Chevy	1990	Blue	87
Ford	1990	Green	64
Ford	1990	Blue	99
Ford	1991	Red	8
Ford	1991	Blue	7

$\Rightarrow$

Model	Year	Color	Sales
Chevy	1990	Blue	87
Chevy	1990	Red	5
Chevy	1990	ALL	92
Chevy	ALL	Blue	87
Chevy	ALL	Red	5
Chevy	ALL	ALL	92
Ford	1990	Blue	99
Ford	1990	Green	64
Ford	1990	ALL	163
Ford	1991	Blue	7
Ford	1991	Red	8
Ford	1991	ALL	15
Ford	ALL	Blue	106
Ford	ALL	Green	64
Ford	ALL	Red	8
ALL	1990	Blue	186
ALL	1990	Green	64
ALL	1991	Blue	7
ALL	1991	Red	8
Ford	ALL	ALL	178
ALL	1990	ALL	255
ALL	1991	ALL	15
ALL	ALL	Blue	193
ALL	ALL	Green	64
ALL	ALL	Red	13
ALL	ALL	ALL	270

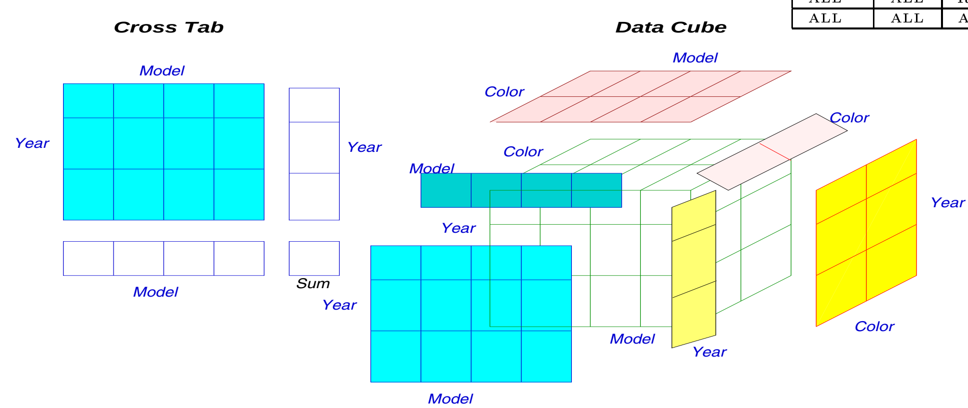


Figure 2: Data Cube Example

Data Cube operators generalize the histogram, cross-tabulation, roll-up, drill-down and sub-total constructs. A lattice structure is used to represent the various aggregate calculations. Figure 3 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 all the 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 computation and communication costs to generate a DAG of cube orderings. The arrows are labeled by the costs of constructing an aggregate and an order of construction is chosen which minimizes the cost [1]. This is described in detail later in the section which presents details on the cost model.

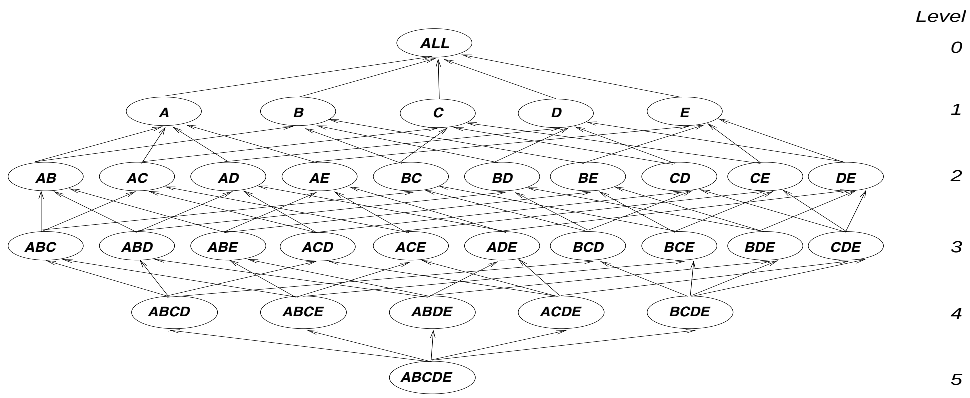


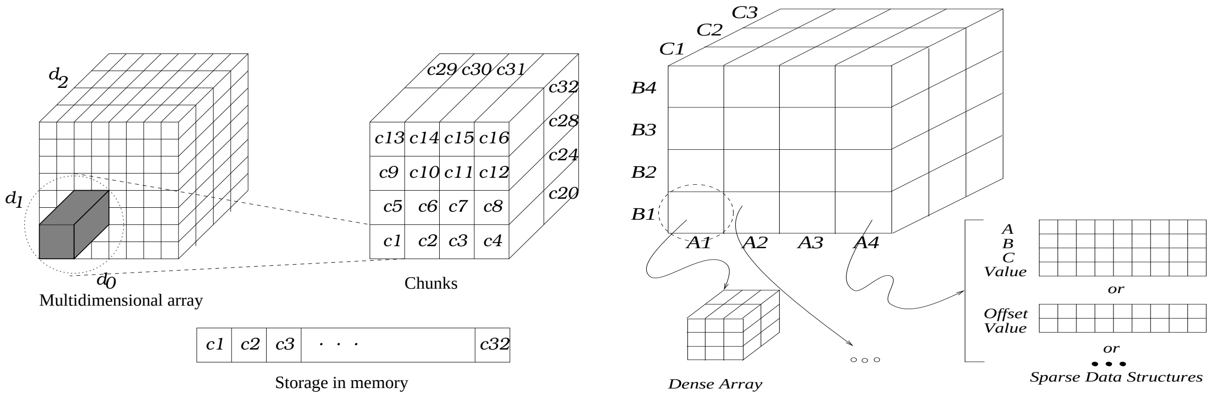
Figure 3: Lattice for cube operator

2.1 OLAP Queries

OLAP queries can be answered by the aggregates in the data cube. The following operations are defined on the cube:

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

The operations needed for OLAP and data mining require access to data along a particular dimension or a combination of dimensions. Usual operations are of the type of aggregation along a particular attribute, which is represented by a dimension in the multi-dimensional database. The set of operations required for



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

Figure 4: Storage of data in chunks

OLAP and data mining should be efficiently supported by the sparse data structure for good performance. The operations fall under the following categories.

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

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 discuss query processing any further, instead concentrate on the design of the storage and computation infrastructure for parallel data cubes.

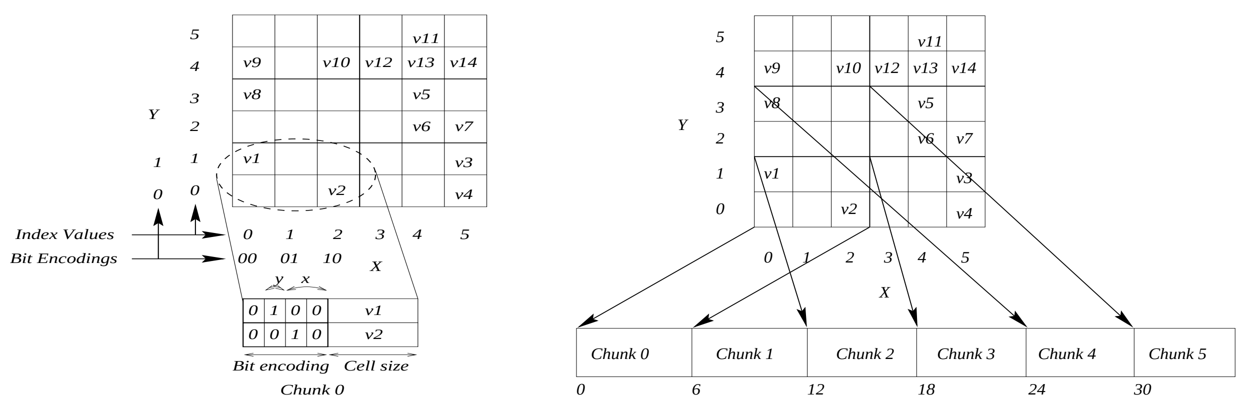
3 Data Storage: Chunks and BESS

Multidimensional database technology facilitates flexible, high performance access and analysis of large volumes of complex and interrelated data [22]. It is more natural and intuitive for humans to model a multidimensional structure. A *chunk* is defined as a block of data from the multidimensional array which contains data in all dimensions. A collection of chunks defines the entire array. Figure 4(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 4(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 storage (BESS). For each cell present in a chunk a dimension index is encoded in $\lceil \log |d_i| \rceil$ bits for each dimension d_i of size $|d_i|$. A 8-byte encoding is used to store the BESS index along with the value at that location. A larger encoding can be used if the number of dimensions are larger than 20. A dimension index can then be extracted by a bit mask operation. Aggregation along a dimension d_i can be done by masking its dimension encoding in BESS and using a sort operation to get the duplicate resultant BESS values together. This is followed by a scan of the BESS index, aggregating values for each duplicate BESS index. For dimensional analysis, aggregation needs to be done for appropriate chunks along a dimensional plane. This will be elaborated further in a later section.

Figure 5(a) illustrates an example for storing a two dimensional sparse chunk of size 3×2. Values $v1$ and $v2$ in chunk 0 are stored using a bit-encoded structure. Dimension x can be encoded in 2 bits since there are only 3 distinct values. Dimension y needs 1 bit to differentiate between 0 and 1. Hence 3 bits are needed for an index. Higher dimensional structures will use more bits but the minimum storage is 32 bits since data is word aligned. Thus, a large number of dimensions can be stored if 2 integers are used.

A chunk index structure stores the chunk offsets for each chunk. Chunks are stored in memory in some order of dimensions. Without loss of generality let us assume that the order of storage is in the order X followed by Y in Figure 5(b). To reference an element in the chunk, first the chunk offset is dereferenced to get the dimension index values. Since the dimension extents are known for the chunk this can be done easily. The second step is to add the index value in the chunk to this by retrieving it from the bit encoding. This can be done by shifting bits in the bit-encoded structure and using a bit-mask to extract the bits that are used to encode each dimension.



(a) Chunk storage using BESS

(b) Storing chunk offsets for index retrieval

Figure 5: BESS used in Chunk Storage

4 Overall Design

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

In what follows, we address issues of data partitioning, parallelism, schedule construction, data cube building, chunk storage and memory usage on this machine architecture. Figure 6 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 $n-1$ [11].

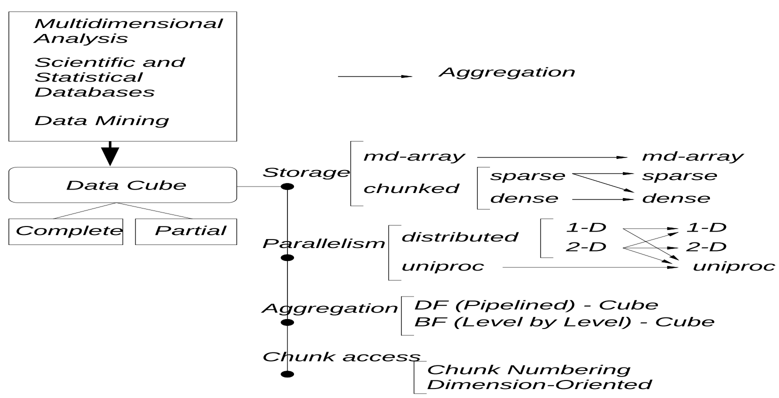


Figure 6: Scheduling for Data Cube

4.1 Data Partitioning and Parallelism

Data is partitioned on processors to distribute work equally. In addition, a partitioning scheme for multi-dimensional 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. Let K (1 or 2) be the number of dimensions used for the partitioning. For $K = 2$, a 2 dimensional partitioning, p processors are divided into two groups $(k, \frac{p}{k})$, where k is the number of processors in the group to be created by the first distributed dimension, chosen to be a divisor of p. For each tuple, A_i is used to choose among the k partitions, and then A_j is further used to find the correct processor among the $\frac{p}{k}$ processors in that group. k can be appropriately chosen to reflect the rations of the dimension sizes used in the partitioning. Figure 7 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 $k = p$.

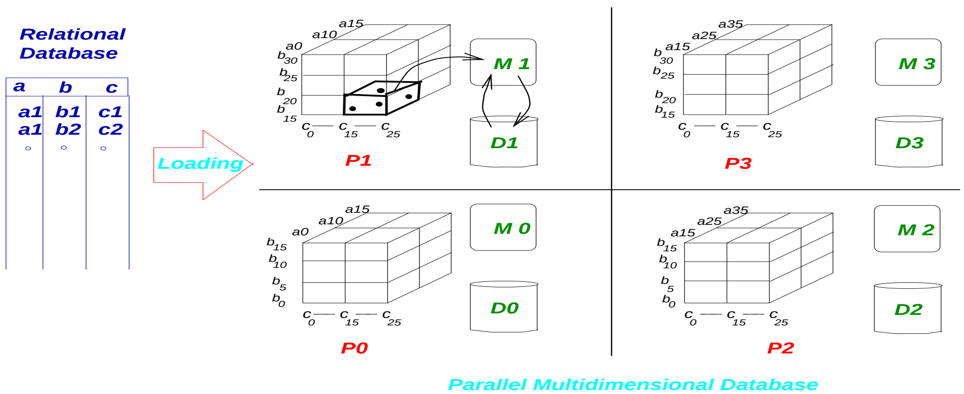


Figure 7: Base cube loading for a 3 dimensional cube on 4 processors

Since 2^n cubes are being constructed, we keep them distributed as well. The distribution of these cubes depends on the cardinalities of their largest 1 or 2 dimensions. The same criteria is used here as the one used for the base cube. However, redistribution of dimensions and chunks may be required if a dimension is partitioned anew or is re-partitioned.

Table 1 shows the various distributions for aggregate calculations supported in our framework. The underlined dimensions are partitioned. Calculations are either *Local* or *Non Local*. Local calculations maintain the data distribution on each processor and the aggregation calculation does not involve any inter-processor communication. Non local calculations distribute a undistributed dimension such as in $\underline{AB}C \rightarrow \underline{AC}$, where dimension B is aggregated and C, which was previously undistributed, is distributed. Another calculation is $\underline{AB}C \rightarrow \underline{BC}$, where A is aggregated and B, the second distributed dimension becomes the first distributed dimension, and C gets distributed as the second dimension. These can be categorized as either dimension 1 or dimension 2 being involved in the (re)distribution. Also, as illustrated in Figure 6, 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 [17]. **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.

Table 1: Partitioning of sub-cubes following aggregation calculations

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

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 [20]. 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 3. 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 [10]. 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 [32]. 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 Cost Model

Size estimation is important to label the edges of the cube lattice with costs of various aggregation calculations. Apart from sizes, computation and communication costs have to be modeled for the various cubes and their distributions. Cubes are partitioned on processors and communication is generated for the various combinations of distributions shown in Table 1. Let $\mathcal{V}(\prod_{i=0}^{n-1} D_i)$ be the estimated value of the number of tuples in the region defined by the dimensions $|D_i|$. Let P, be the number of processors, divided into a $P_x \times P_y$ grid of processors for a 2 dimensional partitioning. The communication cost for a 2D to 2D calculation where the first dimension, D_0 is aggregated, dimension D_1 is re-distributed from P_y to P_x, and dimensions D_2 is distributed on P_y is given by $\mathcal{V}(\frac{|D_0|}{P_x}\frac{|D_1|}{P_y}\prod_{i=2}^{n-1} D_i)T_{calc} + \frac{\mathcal{V}(\frac{|D_0|}{P_x}\frac{|D_1|}{P_y}\prod_{i=2}^{n-1} D_i)}{S_m}T_{I/O} + (T_s + T_w\mathcal{V}(\frac{|D_1|P_x}{P_y}\frac{|D_2|}{P_y}\prod_{i=3}^{n-1} D_i)P)$. Dimension D_0, distributed on P_x processors is aggregated locally, dimension D_1, distributed on P_y is re-distributed to P_x processors, and dimension D_2, which is local is now distributed on P_y processors. The computation cost is the cost of aggregating the tuples in all the chunks on the processor, which are stored as a collection of minichunks for sparse chunks. S_m is the size of a minichunk, and S_b is the size of the aggregate buffer. The I/O cost is in terms of the number of minichunks read and the number of minichunks written after the aggregation. The number of minichunk writes for dimension order traversal is estimated as a ratio of the total number of tuples in the target cube and the minichunk size S_m. We estimate all chunks as sparse chunks for schedule calculations. However, chunks above the user defined threshold density are converted to dense chunks in the implementation. Table 2 summarizes the cost estimations for the various distributions for the aggregation of the first distributed dimension is aggregated. Table 3 summarizes the cost estimation when dimension 2 is aggregated under the various distributions.

Table 2: Cost Model for parallel aggregation calculations for aggregating dimension 1

Distribution	*Cost*		
	T_{calc}	T_{comm}	$T_{I/O}$
$2D \rightarrow 2D$	$\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)$	$P_x(T_s + T_w)\frac{\mathcal{V}(\frac{\lvert D_1\rvert P_x}{P_y}\frac{\lvert D_2\rvert}{P_y}\prod_{i=3}^{n-1} D_i)}{S_{sendbuf}}$	$\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)}{S_{minichunk}}$
$2D \rightarrow 1D$	$\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)$	$P(T_s + T_w)\frac{\mathcal{V}(\frac{\lvert D_1\rvert P}{P_y}\prod_{i=2}^{n-1} D_i)}{S_{sendbuf}}$	$\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)}{S_{minichunk}}$
$2D \rightarrow UNIPRO$	$\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)$	$(T_s + T_w)\frac{\mathcal{V}(\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)}{S_{sendbuf}}$	$\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)}{S_{minichunk}}$
$1D \rightarrow 1D$	$\mathcal{V}(\frac{\lvert D_0\rvert}{P}\prod_{i=1}^{n-1} D_i)$	$P(T_s + T_w)\frac{\mathcal{V}(\frac{\lvert D_1\rvert}{P}\prod_{i=2}^{n-1} D_i)}{S_{sendbuf}}$	$\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P}\prod_{i=1}^{n-1} D_i)}{S_{minichunk}}$
$1D \rightarrow UNIPRO$	$\mathcal{V}(\frac{\lvert D_0\rvert}{P}\prod_{i=1}^{n-1} D_i)$	$(T_s + T_w)\frac{\mathcal{V}(\prod_{i=1}^{n-1} D_i)}{S_{sendbuf}}$	$\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P}\prod_{i=1}^{n-1} D_i)}{S_{minichunk}}$

Table 3: Cost Model for parallel aggregation calculations for aggregating dimension 2

Distribution	*Cost*		
	T_{calc}	T_{comm}	$T_{I/O}$
$2D \rightarrow 2D$	$\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)$	$P_x(T_s + T_w)\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_2\rvert}{P_y}\prod_{i=3}^{n-1} D_i)}{S_{sendbuf}}$	$\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)}{S_{minichunk}}$
$2D \rightarrow 1D$	$\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)$	$P(T_s + T_w)\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\prod_{i=2}^{n-1} D_i)}{S_{sendbuf}}$	$\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)}{S_{minichunk}}$
$2D \rightarrow UNIPROC$	$\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)$	$(T_s + T_w)\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\prod_{i=2}^{n-1} D_i)}{S_{sendbuf}}$	$\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P_x}\frac{\lvert D_1\rvert}{P_y}\prod_{i=2}^{n-1} D_i)}{S_{minichunk}}$
$1D \rightarrow UNIPROC$	$\mathcal{V}(\frac{\lvert D_0\rvert}{P}\prod_{i=1}^{n-1} D_i)$	$(T_s + T_w)\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P}\prod_{i=2}^{n-1} D_i)}{S_{sendbuf}}$	$\frac{\mathcal{V}(\frac{\lvert D_0\rvert}{P}\prod_{i=1}^{n-1} D_i)}{S_{minichunk}}$

[33] uses a minimum memory spanning tree (MMST) for a cube in an optimal dimension order to reduce the memory requirements to overlap the computation of the MMST subtrees. The optimal order is shown to be the increasing order of cardinalities of the dimensions. This is due to the fact that chunks are always accessed in storage order and main memory is allocated for all the target chunks that map in the region

defined by the chunks that are being read. Since the operations are on dense chunks the sizes of the chunks are considered as the product of their dimension cardinalities. A true sparse representation designed for both storage and operations, like BESS, however needs to estimate the sizes of the sparse chunks after group bys.

4.4 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 8 shows the data structures for our design and the ones which are paged in and out from disk into main memory on each processor.

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

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

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

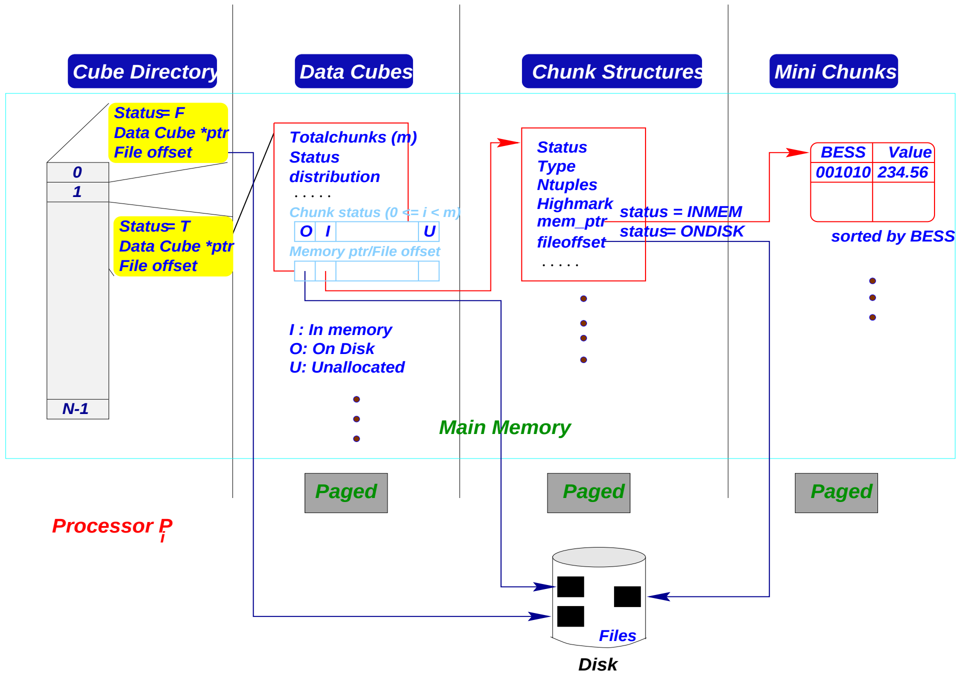


Figure 8: Data Structure Management

a value maps to the chunk (UNALLOCATED → INMEM). When the minichunk is filled it is written to disk and its memory reused for the next minichunk (INMEM → INMEM_ONDISK). Finally, when a minichunk is purged to disk it is deallocated (INMEM_ONDISK → 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 ever get converted to a sparse chunk. Also, a chunked organization may be converted into a multi-dimensional array. The various options are illustrated in Figure 6. In this section we discuss the algorithms for cube aggregations and chunk mappings. Table 4 describes the notation used in the algorithm description.

Table 4: Notation for Algorithms

p	Parent (source) cube
c	Child (target) cube
$\mathcal{D}_k$	Set of dimensions for cube k
$\mathcal{D}_k^d$	Set of *distributed* dimensions for cube k
$\mathcal{D}_k^{nd}$	Set of *non-distributed* dimensions for cube k
$\mathcal{E}_j^k$	Processor extents for cube k in the j^{th} dimension
$\mathcal{F}_j^k$	Dimension factor of chunks for cube k for the j^{th} dimension
$< variable >_A^j$	Value for cube A (parent or child) for the j^{th} dimension
$\mathcal{P}_k$	Processor set partitioned into $\mathcal{P}_k^{d_i}$, $0 \leq i < \lvert\mathcal{D}_k^d\rvert$ for cube k
$\mathcal{B}_{proc}$	Buffer for processor $proc$, communication buffer if $proc \neq myid$ else it is a local buffer
$\mathcal{R}_{proc_i}$	Receive buffer for values from processor $proc_i$,for $0 \leq i < \mathcal{P}_k, i \neq myid$
dim_agg_k	Dimension being aggregated in cube k
N_k	Number of chunks for cube k
$chunks_k[i]$	Chunk structure for cube k, $0 \leq i < N_k$
$cidx_j$	chunk index in dimension j
$cdim_j$	chunk extent in dimension j

5.1 Chunk mapping to processors

Figure 9 describes the algorithm for chunked aggregation of a parent cube with sparse chunks. It describes the chunk mapping process and the distinction between *split* and *non-split* chunks, *local* mapping and *non-local* mappings.

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 10 for a two dimensional source to target aggregation of chunks.

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.

Algorithm 1 *Chunked cube mapping*

```
for i ← 1 to N_p
    for j ← 1 to ndim, j ≠ dim_agg_p
        local_start_index_j^p = chunks_p[i].cidx_j * chunk_extent_j
        local_stop_index_j^p = local_start_index_j^p + chunks_p[i].cdim_j
        /* Calculate the global index in dimension */
        if(j ∈ D_p^d)
            global_start_index_j^p = local_start_index_j^p + E_j^p[myid]
            global_stop_index_j^p = local_stop_index_j^p + E_j^p[myid]
        if(j ∈ D_c^d)
            j' = Rank(j, D_c^d)
            proc_b = BinarySearch(global_start_index_j^p, E_j^p[], P_c^{j'})
            local_start_index_j^c = global_start_index_j^p − E_j^c[proc_b]
            proc_e = proc_b /* (e)end and (b)beginning processors */
            while(global_stop_index_j^p > E_j^c[proc_e])
                proc_e ← proc_e + 1
            if(proc_b ≠ proc_e)
                local_stop_index_j^c − = E_j^c[proc_e]
        else /* For non-distributed dimension j in target */
            local_start_index_j^c = global_start_index_j^p
        chunk_start_num_j^c = local_start_index_j^c / chunk_extent_j
        chunk_stop_num_j^c = local_stop_index_j^c / chunk_extent_j
    /* Check if source chunk maps to the same processor/same chunk or same processor/different chunks or different processors */
    if(proc_b = proc_e and chunk_start_num_j^c = chunk_stop_num_j^c, j ∈ D_c)
        target_chunk_id = 0
        for j ← 1 to ndim, j ≠ dim_agg_p
            target_chunk_id += chunk_start_num_j^c * F_j^c
            /* Calculate the BESS correction values for the target chunk */
            bess_correction_j = |local_start_index_j^p − local_start_index_j^c|
        if(proc_b ≠ myid) /* Off-processor value */
            Correct and copy BESS+value pairs into communication buffer B_{proc_b}
            (If B_{proc_b} is full, do an Asynchronous SEND to proc_b and reuse buffer)
        else /* Local processor value */
            Correct and copy BESS+value pairs into local buffer B_myid
            (If B_myid is full, aggregate buffer and reuse)
    else
    if(proc_b = proc_e) /* Same processor different chunks */
        for each BESS value in chunk
            target_chunk_id = 0, Initialize prev_target_chunk_id
            for j ← 1 to ndim, j ≠ dim_agg_p
                target_chunk_id += [(local_start_index_j^p + BESS_j)/chunks_extent_j] * F_j^c
            if(proc_b ≠ myid) /* Off-processor value */
                Copy (BESS+value, target_chunk_id) pairs into communication buffer B_{proc_b}
                (If B_{proc_b} is full, do an Asynchronous SEND to proc_b and reuse buffer)
            else /* Local processor value */
                if(target_chunk_id ≠ prev_target_chunk_id)
                    Aggregate local buffer B_myid with target_chunk_id of target
                else
                    Copy BESS value to local buffer B_myid
                prev_target_chunk_id = target_chunk_id
    else
    /* Different processors and hence different chunks */
        for each BESS value in chunk
            target_chunk_id = 0, Initialize prev_target_chunk_id
            for j ← 1 to ndim, j ≠ dim_agg_p
                global_index_j = global_start_index_j^p + BESS_j
                if(j ∈ D_c^d)
                    proc = BinarySearch(global_index_j, E_j^p[], P_c^{j'})
                    local_index_j = global_index_j − E_j^p[proc]
                else
                    local_index_j = global_index_j
                target_chunk_id += (local_index_j / chunk_extent_j) * F_j^c
            if(proc = myid) /* Local processor value */
                if(target_chunk_id ≠ prev_target_chunk_id)
                    Aggregate local buffer B_myid with target_chunk_id of target
                else
                    Copy BESS value to local buffer B_myid
                prev_target_chunk_id = target_chunk_id
            else /* Off-processor value */
                Copy (BESS+value, target_chunk_id) pairs into communication buffer B_proc
                (If B_proc is full, do an Asynchronous SEND to proc and reuse buffer)
    Asynchronous RECEIVE from other processors in R_{proc_i} for 0 ≤ i < |P_p|, i ≠ myid, and aggregate to local chunks.
end
```

Figure 9: Algorithm for sparse-sparse chunked cube mapping

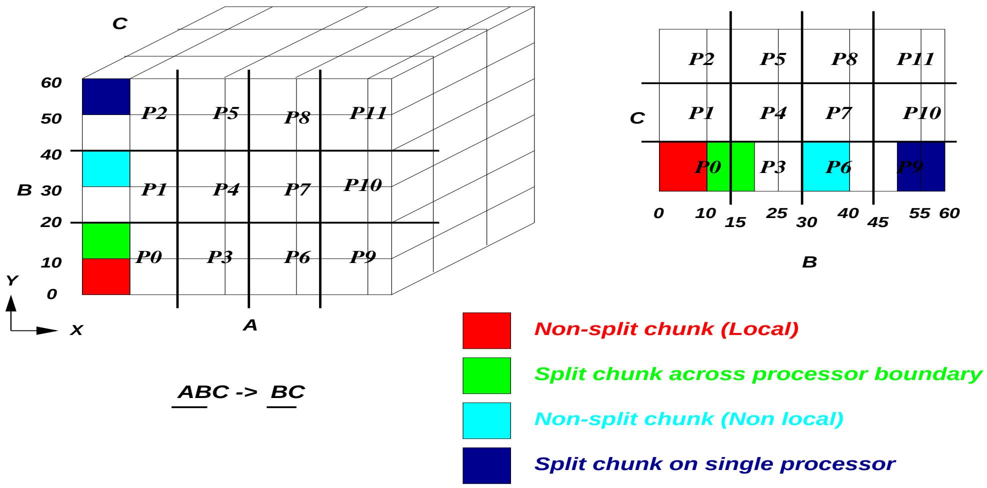


Figure 10: Chunk mapping after aggregation for a 2D cube distribution

5.2 Local - Non Local Aggregation

Figure 11 illustrates the local and non-local aggregation calculations for chunks. The same partitioning dimensions in the source and target sub-cubes result in a local aggregation calculation. For example, $\underline{AB}C \rightarrow \underline{AB}$ has both A and B partitioned in both sub-cubes and this results in a local aggregation. On the other hand, $\underline{AB}C \rightarrow \underline{A}B$, has only A partitioned in the result sub-cube and B goes from being distributed to being undistributed. This results in communication and is a non local aggregation. Other cases are illustrated in Table 1.

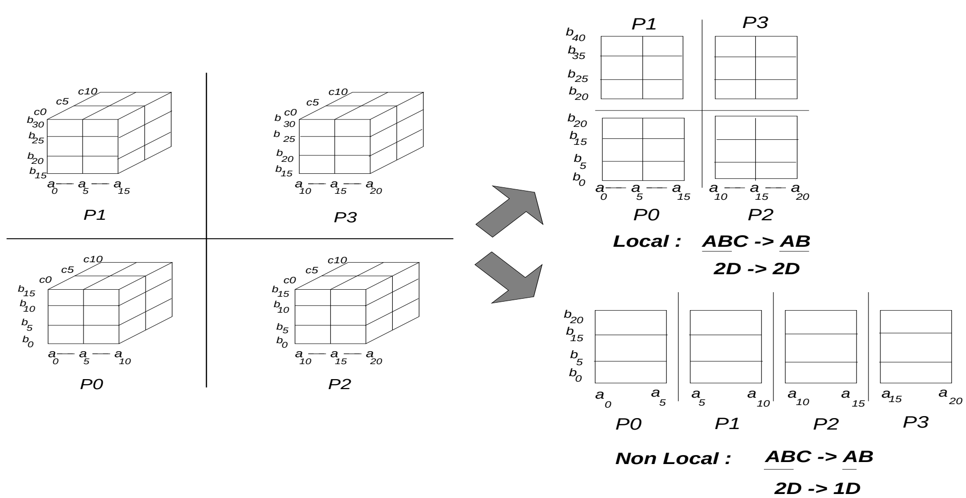


Figure 11: Local and Non Local aggregation calculations

The distribution of the sub-cubes is ascertained by the cardinalities of its two largest dimensions. A sub-cube can be 2D, 1D distributed or be on a single processor. To generate the schedule of data cube calculations, the lattice structure is used, exploiting the various optimization discussed earlier in the section on schedule generation. Computation and communication costs for aggregation are used by the scheduler by analyzing the chunk aggregation operations and the resulting partitioning. The cost analysis of the various aggregations between 1D, 2D and uniprocessor aggregations is not included in this article.

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. Figure 12 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.

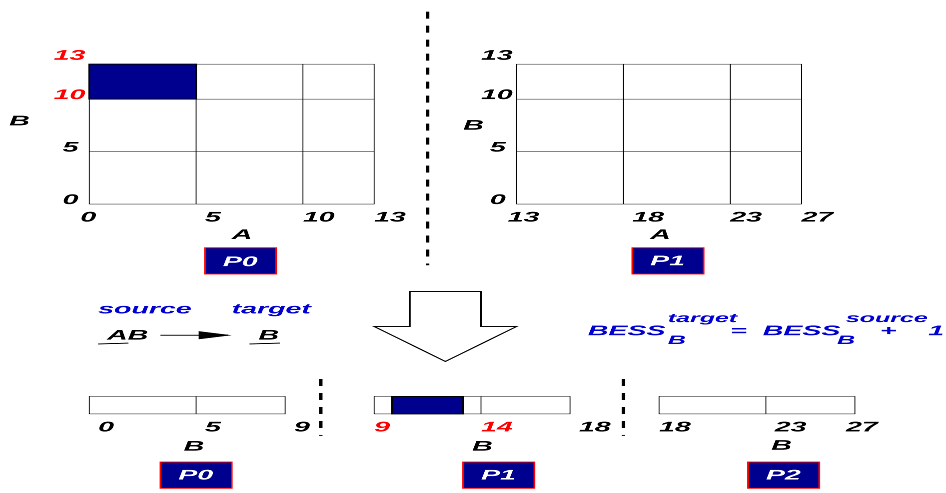


Figure 12: Modification of BESS indices while mapping a non-split chunk

5.3 Sparse - Sparse Chunk Aggregation

Figure 13 describes the aggregation algorithm of a sparse chunk with a sparse chunk. Sparse chunks store BESS+value pairs in minichunks. Sparse to sparse aggregations involve accessing these minichunks. The BESS values are kept sorted in the minichunks to facilitate the aggregation calculations by using sort and scan operations used in relational processing. Figure 14 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.

Buffer management for this aggregation depends on the order in which chunks to be aggregated are accessed.

Algorithm 2 *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 access, get the next chunk along dimension dim_agg_p */
                if(chunk_access = DIM_ORIENTED)
                        if((nextchunk ← Get_next_chunk(dim_agg_p, nextchunk)) = FALSE)
                                done = TRUE
                else
                        done = TRUE
                visited[nextchunk] = TRUE
                } while (! done)
end
```

Figure 13: Sparse - sparse chunk aggregation algorithm

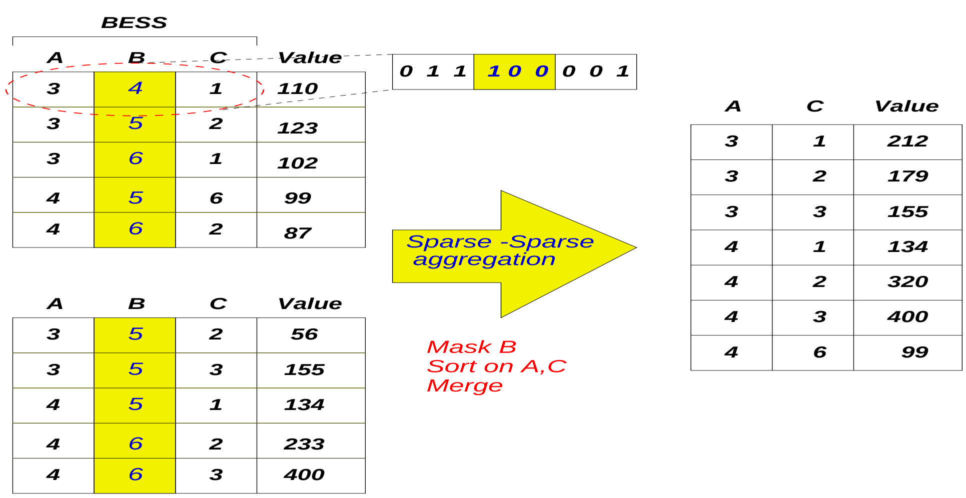


Figure 14: Sparse - sparse chunk aggregation

There are two alternative ways to access chunks, *dimension oriented* in which the chunks are accessed along the dimension to be aggregated. The other method, *chunk numbering*, accesses chunks in the order in which they are laid out in memory. Figure 15 illustrates these two choices. In the dimension oriented access, a buffer can be filled with the source chunks that map to a single target chunk and the sort and scan operations are done on the buffer in memory. For the chunk order access, the consecutive chunks accessed are mapped to different chunks for all dimensions except the innermost dimension. This results in sort of each individual chunk and merge with the sorted values calculated earlier mapping to the same chunk. This involves a disk read to get the previous result from the minichunk of the target chunk, which was written to disk if it was full.

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

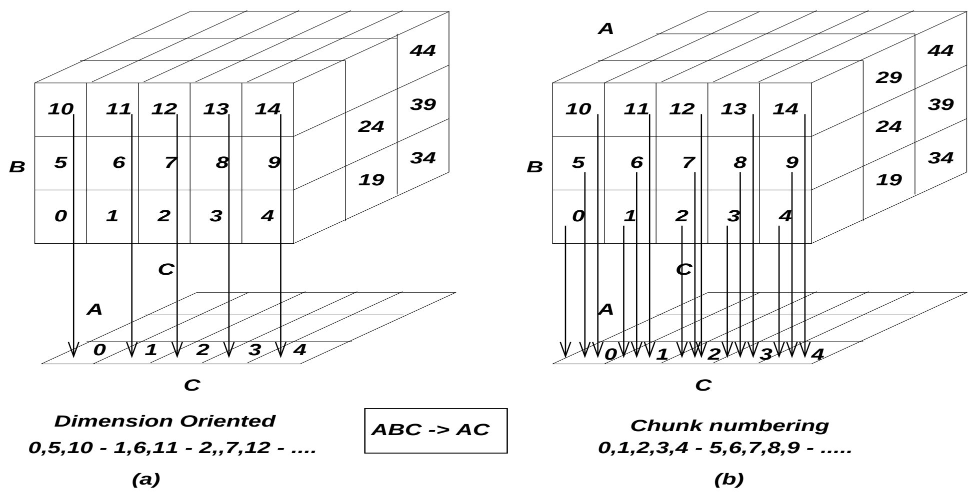


Figure 15: Dimension oriented vs. Chunk numbering access of chunks

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. Figure 16 describes the algorithm in more detail.

5.6 Cube and chunk level reuse and I/O

Calculating simultaneous multi-dimensional aggregates involves calculations for multiple cubes that are either chunked or stored as multidimensional arrays. Since a large number of cubes are calculated and disk storage is used for minichunks, chunk structures and data cube structures, strategies to reduce the number of scans for an intermediate calculation should reuse data in memory effectively. This is determined by the order of calculation of sub-cubes in the lattice structure. Figure 17 illustrates the order by a depth first and a breadth first traversal of the lattice structure. In Figure 17(a), the order of computation is $ABC \rightarrow AB \rightarrow A$, reusing AB to calculate A before calculating AC from ABC. The result of an aggregation can be reused in this case. On the other hand, in Figure 17(b) the source ABC can be reused by calculating AB, AC, BC from ABC before calculating the next level.

Only a minichunk is in memory for a particular chunk, the remaining minichunks (if any) are stored on disk. A result of an aggregation results in a disk write for the target minichunk when it gets full. A depth first traversal, which reuses the resultant chunk, will still result in disk reads for chunks which have many minichunks. This is a consequence of having only a minichunk size memory buffer for each chunk for scalability reasons. A larger minichunk size would have data of more resultant chunks in memory to be reused.

In case of the breadth first traversal, the source minichunks are read for a chunk in a memory buffer and aggregated with the target. Once this is done, the source buffer is released, again to accommodate many such calculations. At a cube level reuse, if the minichunk size can fit the values of the source chunk, they will remain in memory after the $ABC \rightarrow AB$ calculation. This can be reused to calculate the other aggregates from the source. In case there are multiple minichunks for a chunk not much reuse is possible at the cube level. However, at a chunk level reuse, the memory buffer allocated for ABC's chunks aggregation can be reused to calculate the target of AB, AC and BC by doing the appropriate maskings. Again, this will hold true only in the case of chunk accesses by chunk number and not for dimension oriented chunk access. This is because the chunks are read in dimension order and this order is different for $ABC \rightarrow AB$ (dimension order is C), $ABC \rightarrow AC$ (dimension order is B) and $ABC \rightarrow BC$ (dimension order is A). Thus, reuse at chunk-level does not take advantage of the dimension-oriented chunk access optimizations.

5.7 Interprocessor communication

The processor with the least load is assigned to a uni-processor 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

Algorithm 3 *Chunked - multidimensional array cube aggregation*

```
for i ← 1 to N_p
    for j ← 1 to ndim, j ≠ dim_agg_p
        local_start_index_j^p = chunks_p[i].cidx_j * chunk_extent_j
        local_stop_index_j^p = local_start_index_j^p + chunks_p[i].cdim_j
        /* Calculate the global index in dimension */
        if(j ∈ D_p^d)
            global_start_index_j^p = local_start_index_j^p + E_j^p[myid]
            global_stop_index_j^p += local_stop_index_j^p + E_j^p[myid]
        if(j ∈ D_c^d)
            j' = Rank(j, D_c^d)
            proc_j^b = BinarySearch(global_start_index_j^p, E_j^p[], P_c^{j'})
            local_start_index_j^c = global_start_index_j^p − E_j^c[proc_j^b]
            proc_j^e = proc_j^b /* (e)end and (b)beginning processors */
            while(global_stop_index_j^p > E_j^c[proc_j^e])
                proc_j^e ← proc_j^e + 1
            if(proc_j^b ≠ proc_j^e)
                local_stop_index_j^c -= E_j^c[proc_j^e]
        else /* For non-distributed dimension j in target */
            local_start_index_j^c = global_start_index_j^p
    /* Check if source chunk maps to the same processor or different processors */
    if(proc_b = proc_e) /* Same processor */
        for j ← 1 to ndim, j ≠ dim_agg_p
            target_offset += local_start_index_j^c * F_j^c
        if(proc_b ≠ myid) /* Off-processor value */
            Copy target_offset+value pairs into communication buffer B_{proc_b}
            (If B_{proc_b} is full, do an Asynchronous SEND to proc_b and reuse buffer)
        else /* Local processor value */
            Copy target_offset+value pairs into local buffer B_myid
            (If B_myid is full, aggregate buffer and reuse)
    else
        for each BESS value in chunk
            target_offset = 0
            for j ← 1 to ndim, j ≠ dim_agg_p
                global_index_j = global_start_index_j^p + BESS_j
                if(j ∈ D_c^d)
                    proc = BinarySearch(global_index_j, E_j^p[], P_c^{j'})
                    local_index_j = global_index_j − E_j^p[proc]
                else
                    local_index_j = global_index_j
                target_offset += (local_index_j / dimension_size_j^proc) * F_j^c
            if(proc = myid) /* Local processor value */
                if(local buffer B_myid is full)
                    Aggregate local buffer B_myid with target_chunk_id of target
                else
                    Copy target_offset+value to local buffer B_myid
            else /* Off-processor value */
                Copy target_offset+value to communication buffer B_proc
                (If B_proc is full, do an Asynchronous SEND to proc and reuse buffer)
    Asynchronous RECEIVE from other processors in R_{proc_i} for 0 ≤ i < |P_p|, i ≠ myid, and aggregate to local address offsets.
end
```

Figure 16: Chunked - multidimensional array cube aggregation

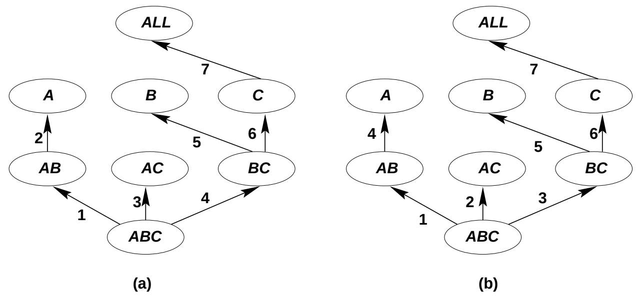


Figure 17: (a) Depth first traversal of lattice (b) Breadth first traversal of lattice

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 18 shows the communication phase on each processor.

Algorithm 4 *Communication in source → 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 18: Communication in source → target aggregation

6 Data Mining

Data mining can be viewed as an automated application of algorithms to detect patterns and extract knowledge from data [8]. An algorithm that enumerates patterns from, or fits models to, data is a data mining algorithm. Data mining is a step in the overall concept of knowledge discovery in databases (KDD). Large data sets are analyzed for searching patterns and discovering rules. Automated techniques of data mining can make OLAP more useful and easier to apply in the overall scheme of decision support systems. Data mining techniques like *Associations, Classification, and Clustering* [8] can be used together with OLAP to discover knowledge from data. In the next few sections we present an integration of association rule mining and classification with the parallel OLAP and multidimensional analysis infrastructure presented in this article.

6.1 Association Rule Mining

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 [2] [18] [19] to data mining are data-driven and can use this summary information to discover association rules. Transaction data can be used to mine association rules by associating *support* and *confidence* for each rule. 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 and ALL. 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 [2] and other applications. Earlier applications of this technique were to software process engineering [3]. 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.

Consider a 3 attribute data cube with attributes A, B and C, defining $E_3 = ABC$. For showing 2-way associations, we will calculate the interestingness function between A and B, A and C and finally between B and C. When calculating associations between A and B, the probability of E, denoted by $P(AB)$ is the ratio of the aggregation values in the sub-cube AB and ALL. Similarly the independent probability of A, $P(A)$ is obtained from the values in the sub-cube A, dividing them by ALL. $P(B)$ is similarly calculated from B. The calculation $|P(AB) - P(A)P(B)| > \delta$, for some threshold δ, is performed in parallel. Assume that the cubes AB and A are distributed along the A dimension, no replication of A is needed. However, since B is distributed in sub-cube B, and B is local on each processor in AB, B needs to be replicated on all processors. AB and A cubes are distributed, but B is replicated on all processors. Figure 19 shows a sample calculation of P(AB), P(A) and P(B) on three processors. A sample calculation is, highlighted in the figure, $|0.03 - 0.22 \times 0.08| = 0.0124$ which is greater than δ values from 0.001 to 0.01, and the corresponding attribute values are associated within that threshold.

Aggregates with m-attributes and below, present in the data cube can be used to calculate m-way associations by looking at sub-cubes for all combination of m dimensions. Since these cubes are pre-computed, calculations are very efficient. We do not discuss this further in this article. Next, we describe algorithms for

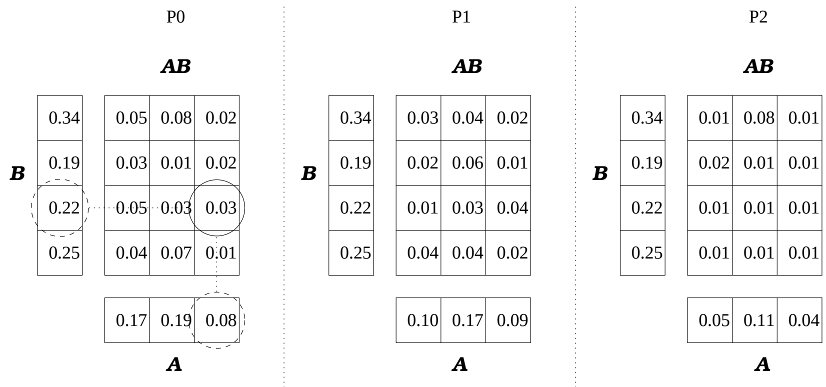


Figure 19: Use of aggregate sub-cubes AB, A and B for calculating "interestingness" on three processors

classification on the multidimensional structure.

6.2 Classification

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.

Several classification models have been used in the past, notably neural networks [25], genetic algorithms [16], decision trees [4] [27] and some others. Among these models, the decision tree models are considered to be the most useful in the domain of data mining because they are relatively inexpensive to construct, easy to interpret and easy to integrate with data base systems. Also, for a variety of problem domains they yield comparable or better accuracy as compared to the other models [25].

A decision tree recursively partitions the training set until each partition consists entirely or dominantly records from one class. Each *non-leaf* node of the tree contains a split point which is a test on an attribute and determines how the data is partitioned. Once the decision tree is built from the training set it can be used to classify future instances. The decision-tree based classifiers that can handle large data sets are important because use of larger data sets improves the classification accuracy [25].

Previous work in classifying large data sets has been to use sampled data sets or multiple partitions of the data set [25] [5]. Recent work has focused on using the entire data set, in classifiers like SLIQ [24] and SPRINT [30]. A parallel classifier in the same spirit has been developed in ScalParC [21]. Classifiers like CART [4] and C4.5 [27] perform sorting at every node of the decision tree, which makes them expensive for large data sets since disk-based sorting has to done. 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 5: 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

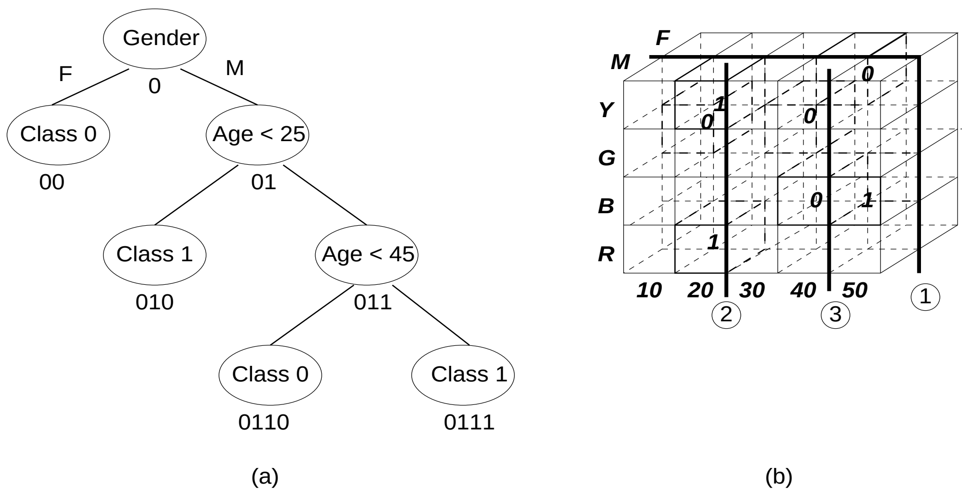


Figure 20: (a) Classification tree for training set (b) Classification tree embedded on the base cube

Table 5 is an example training set with three attributes, Age, Car color and Gender, and a class attribute. Figure 20(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 [4]. 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. The *gini* index calculations and the node splits for the example above are given in Figure 21. At Node 0, the attribute *Gender* yields the optimal $gini_{split}$ value of 0.214. This creates a split with one partition with M values for gender and another with F values. After this split is made, two child nodes are created. The record values need to be partitioned consistently between the two nodes for the *split-attribute* and the *non-split attributes.* Splitting the split-attribute is straightforward by adjusting pointer values. The challenge is to split the non-split attributes

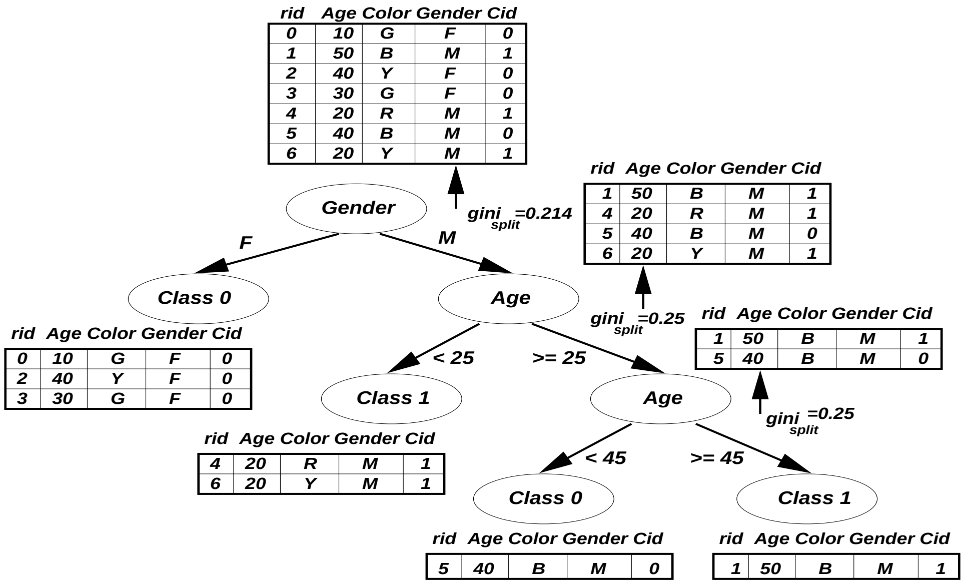


Figure 21: Gini index calculation for the attributes and node splitting

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.

6.2.1 Classification on the multidimensional 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.

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 22(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 22(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

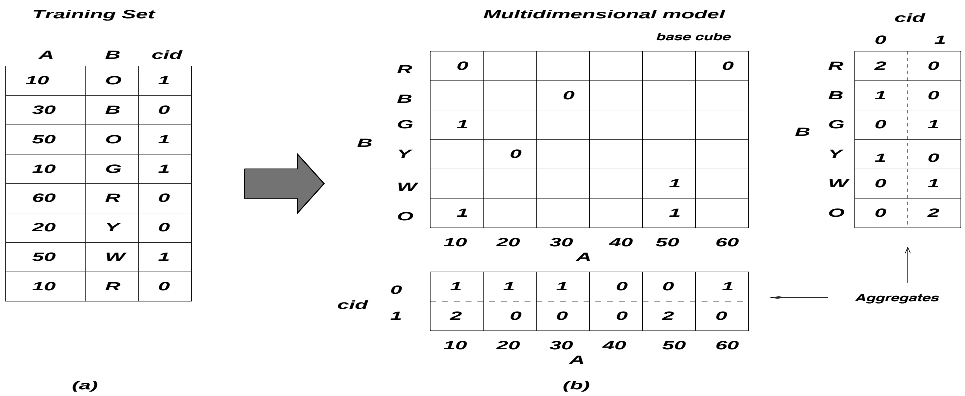


Figure 22: (a) Training set records (b) corresponding multidimensional model

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 22(b). For a parallel version, the count matrices are computed in parallel and each processor maintains the classification tree nodes, identifying the portion of the hypercube present locally. Each processor computes the count matrices locally for each active node of the tree. Collective communication is used to consolidate the global count matrices. Communication is $O(\mathcal{N}_n^k|d|)$ at a level k, where $\mathcal{N}_n^k$ is the number of active nodes to classify at level k, and $|d|$ is the size of the largest dimension. This is much less than the communication time of $O(N/p)$ ($O(N)$ worst case) at each level for each attribute in the attribute-list approach discussed earlier. Further details can be found in [14].

Figure 23 illustrates the classification tree building process using the multidimensional model and the aggregates maintained at the highest level of the cube structure, one for each dimension.

6.3 Partial Cubes

Partial cube calculations are done to calculate all the aggregate sub-cubes at level 1, 2 or 3 of the cube lattice as needed in data mining. This is done by maintaining the minimum count of intermediate calculations. In a full data cube, the number of aggregates is exponential. In the cube lattice there are n levels and for each level i there are $\begin{pmatrix} n \\ i \end{pmatrix}$ aggregates at each level containing i dimensions. The total number of aggregates, thus is 2^n. For a large number of dimensions, this leads to a prohibitively large number of cubes to materialize.

Next, we need to determine the number of intermediate aggregates needed to calculate all the aggregates for all levels below a specified $m, m < n$, for m-way associations and for meta-rules with m predicates. The number of aggregates at level 2 is $\sum_{j=1}^{n-1} \begin{pmatrix} j \\ 1 \end{pmatrix}$. For level 3 it is $\sum_{j=1}^{n-2} \begin{pmatrix} j \\ 1 \end{pmatrix}$. At each level k, the first $k-1$ literals in the representation are fixed and the others are chosen from the remaining. The total number of aggregates is then $\sum_{i=1}^{n-1} \sum_{j=1}^{i} \begin{pmatrix} j \\ 1 \end{pmatrix}$. After simplification this becomes $\frac{n(n-1)(2n+2)}{12}$. For example, when $n = 10$, this number is 165, a great reduction from $2^{10} = 1024$. The savings are even more significant for $n = 20$, where this number is 1330, much smaller than $2^{20} = 1048576$. Further details are in [13]. A similar analysis of the number of cubes needed for calculating all the one-attribute aggregates at level 1 of the cube lattice is

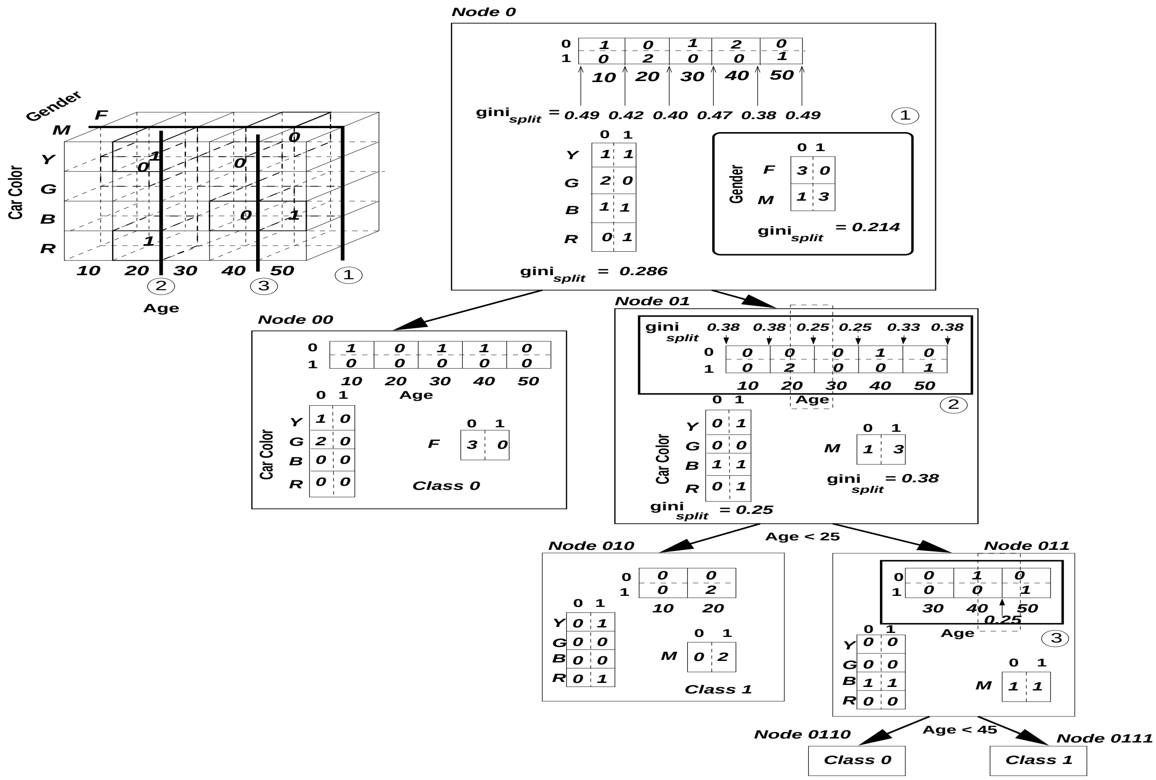


Figure 23: Gini index calculation for the attributes and node splitting with multidimensional aggregates

given in [14].

7 Performance Results and Discussion

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 7) 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 4 data sets, one each of dimensionality 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, skewed data sets and the OLAP council benchmark [26], which models a realistic business situation with

Table 6: Description of datasets and attributes, (N) Numeric (S) String

Data set	No. of dimensions	Cardinalities (d_i)	Product ($\prod_i d_i$)	Tuples
Dataset I	3	1024(S),256(N), 512(N)	2^{27}	10 million
Dataset II	5	1024(S),16(N),32(N),16(N),256(S)	2^{31}	1 and 10 million
Dataset III	10	1024(S),16(S),4(S),16,4,4,16,4,4,32(N)	2^{37}	5 and 10 million
Dataset 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

similar results.

Table 6 illustrates the data sets for our experiments. The 5 dimensional data set has chunk sizes in each dimension of 32, 4, 8, 8, 16 for a total chunk dimension product of 2^{17}. The number of chunks is 2^{14} for the base cube and are calculated for each sub-cube depending on the dimensions it has, again getting distributed across processors. Finally, for the 10 dimensional data set the chunk dimension sizes are 64, 4, 2, 4, 2, 2, 4, 2, 2, 4 for a total chunk dimension product of 2^{19} and total number of chunks distributed across processors as 2^{18} for the base cube. The other sub-cube chunk sizes and chunk structure sizes are calculated from these.

The number of sub cubes in the datacube for Dataset I is $2^3 = 8$, Dataset II is $2^5 = 32$ and for 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.

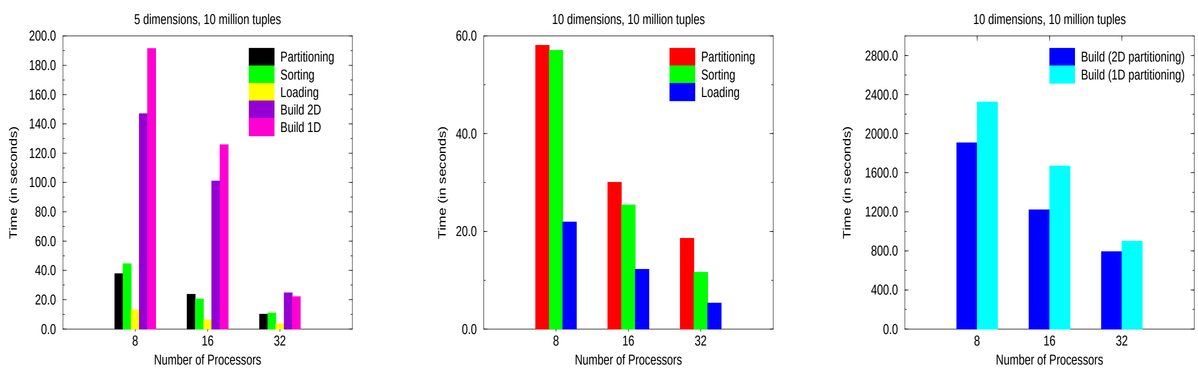


Figure 24: Time taken by various phases of the data cube construction algorithm for 5 (32 sub cubes) and 10 (1024 sub cubes)

Figure 24 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 25 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

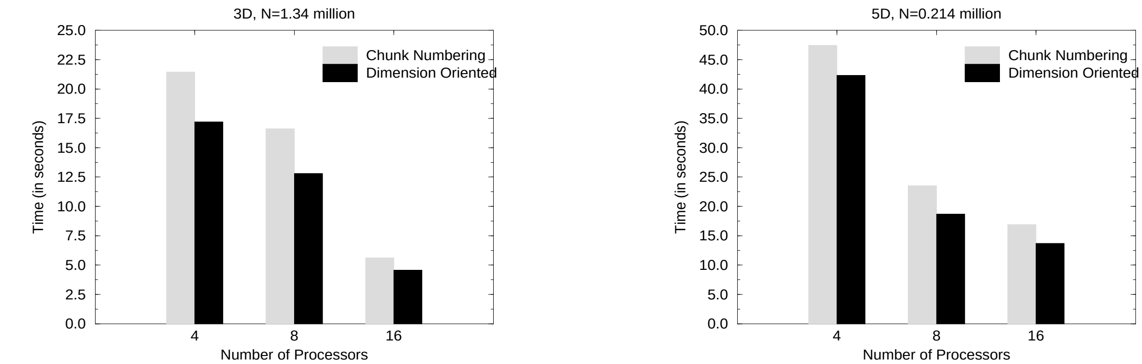
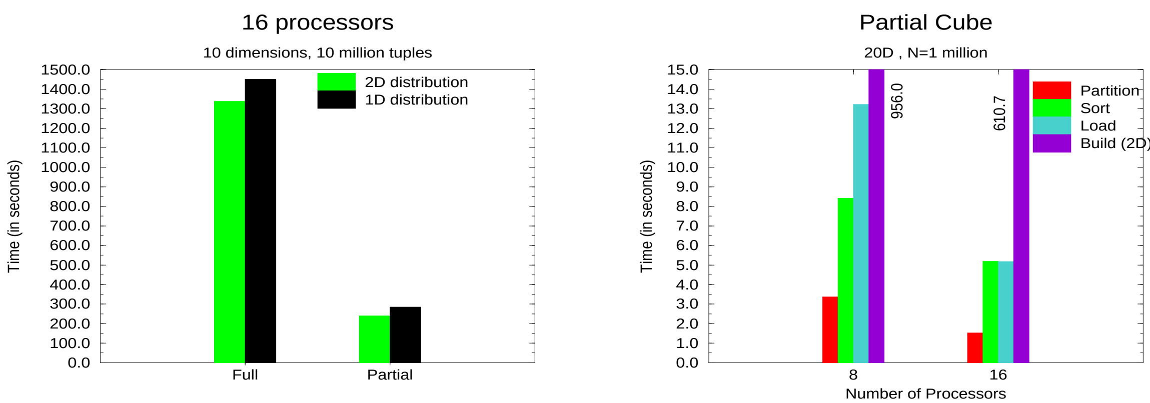


Figure 25: Comparison of the two chunk access methods, chunk numbering and dimension oriented for a dataset with 3 and 5 dimensions

dimension to be aggregated in these cases. This effect will be more pronounced with higher data densities.



(a) (b)

Figure 26: Comparing full and partial cubes (a) 10D, 10 million tuples on 16 processors, (b) partial cubes for 20D on 8 and 16 processors

Figure 26 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. This meant that for mining of 2-way association rules the partial cube can be constructed fairly quickly. Similarly, 3-way associations can also be mined eficiently. These two are the more important ones in terms of using support and confidence measures to find useful rules from large databases.

Table 7 shows the time for calculating various subcubes for two different chunk sizes. A larger chunksize means lesser number of chunks and hence lesser chunk traversal and aggregation overhead. Hence we observe lower numbers for it indicating better performance. 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.

Table 8 shows the parameters that are indicators of the computation communication and I/O parameters and performance for dataset I on 4 processors. With a larger chunksize we see that all these reduce, notably

Table 7: 3D dataset with Chunksize I (32, 8, 16) and Chunksize II (64, 8, 32) (a) Aggregation time (in seconds) on 4 processors (b) Speedup performance

Operation	Chunksize I	Chunksize II
$ABC \rightarrow AB$	7.446	2.067
$ABC \rightarrow AC$	37.41	12.40
$ABC \rightarrow BC$	103.26	47.71
$AC \rightarrow A$	0.148	0.084
$BC \rightarrow B$	0.194	0.045
$BC \rightarrow C$	0.042	0.032
Total	143.78	62.34

(a)

	Processors		
	4	8	16
Chunksize I	143.78	44.95	30.43
Chunksize II	62.34	11.59	9.65

(b)

Table 8: Count of each parameter for 3D dataset with Chunksize I (32, 8, 16) and Chunksize II (64, 8, 32) on 4 processors

Parameter	Chunksize I	Chunksize II
Split chunks	16,928	4,384
BESS Communicated	708,489	642,724
Offsets Communicated	45,014	19,174
Minichunks write	68,944	58,087
Minichunks read	309,292	221,544
Merge BESS write	99,356	45,915
Merge BESS read	102,521	47,176

the number of split chunks, and the BESS merge read and write operations. This are reflected in the improved times seen in Table 7.

Finally, Figure 27 shows the performance of 100 random range queries on the base cube at level n and the $n-1$ dimension aggregates for $n = 5$ and $n = 10$. We see that the performance scales very well as the number of processors is increased. Ranges are defined both on the distributed and non-distributed dimensions. Queries on hierarchies, roll-ups, also scale well with increasing number of processors as they are also range queries on the lowest level. These queries reflect the queries in the OLAP council benchmarks where variable ranges are specified for various attributes.

8 Conclusions

In this article 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* 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

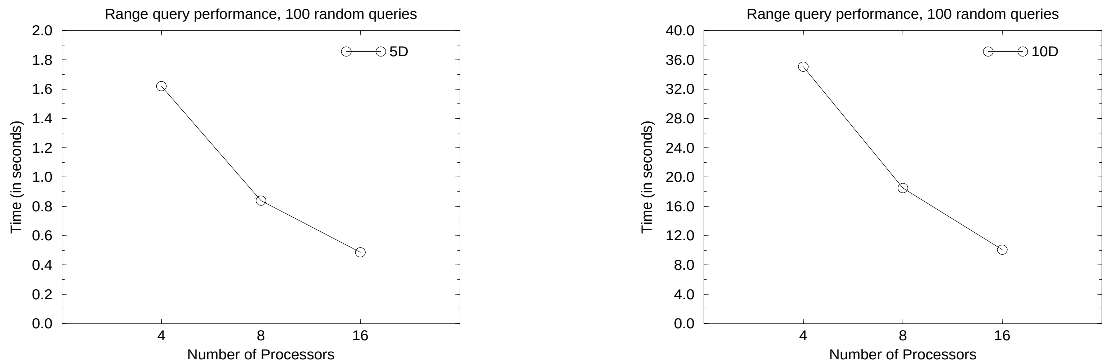


Figure 27: Range query performance for 100 random queries on the last 2 levels of the cube lattice for Datasets II (5D) and III (10D)

cube size is below a specific threshold. Operations between chunked and multi-dimensional array cubes are supported.

The data structures to track the different cubes in a data cube, the chunk structures of each cube and the chunks themselves (using minichunks) use paging to support a large number of cubes, a large number of chunks per cube and a large chunk size. A combination of these results in supporting a large number of dimensions and large data sizes. For coarse grained parallelism, the cubes can either be partitioned using one largest dimension or a combination of the two largest dimensions. The choice depends on the cardinality of the two largest dimensions and the number of processors. Further, a cube can be allocated on a single processor if its size is small and the overhead of parallelization is high. Operations between these distributions are supported in our framework.

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 while 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] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *Proc. 22th Int'l Conference on Very Large Databases, Mumbai, India*, September 1996.

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

[3] I. Bhandari and et. al. A case study of software process improvement during development. *IEEE Transactions on Software Engineering*, 19(12), 1993.

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

[5] P.K Chan and S.J Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. Intl. Workshop on Multistrategy Learning*, 1993.

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

[7] P.M. Deshpande, K. Ramaswamy, A. Shukla, and J.F. Naughton. Caching multidimensional queries using chunks. In *Proc. ACM SIGMOD Conference on Management of Data*, June 1998.

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

[9] 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.

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

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

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

[13] S. Goil and A. Choudhary. Partial Data Cubes for Parallel Data Mining. Technical Report CPDC-9803-019, Northwestern University, March 1998.

[14] S. Goil and A. Choudhary. Parallel classification using the multidimensional data model (under preparation). Technical Report CPDC-9901-001, Northwestern University, January 1999.

[15] Sanjay Goil. *High Performance On-Line Analytical Processing and Data Mining on Parallel Computers.* PhD thesis, Northwestern University, 1999.

[16] D.E Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Morgan Kaufmann, 1989.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] Kenan Software, http://www.kenan.com/acumate/mddb.htm. *An Introduction to Multi-dimensional Database Technology*, 1997.

[23] 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.

[24] 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.

[25] D. Michie, D.J. Spiegelhater, and C.C. Taylor. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

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

[27] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

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

[29] P. Scheuermann, J. Shim, and R. Vingralek. Watchman : A data warehouse intelligent cache manager. In *Proc. 22th Int'l Conference on Very Large Databases, Mumbai, India*, June 1998.

[30] 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.

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

[32] 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.

[33] 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.