NORTHWESTERN UNIVERSITY

High Performance On-Line Analytical Processing and Data Mining on Parallel Computers

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

Sanjay Goil

EVANSTON, ILLINOIS

June 1999

ABSTRACT

High Performance On-Line Analytical Processing and Data Mining on Parallel Computers

Sanjay Goil

Decision support systems are important in leveraging the information present in large scale data repositories in many scientific and business applications. Data analysis and data mining on these warehouses pose new challenges for traditional database systems. On-Line Analytical Processing (OLAP) and data mining operations require summary information on these data sets. Query processing for these applications require different views of data for analysis and effective decision making. The multi-dimensional data model is a natural and intuitive approach for such applications. Data mining techniques can be applied in conjunction with OLAP for an integrated solution. As data warehouses grow, parallel processing techniques need to be applied to enable the use of larger data sets and reduce the time for analysis, thereby enabling evaluation of many more options for decision making.

In this dissertation we focus on parallel processing techniques for scalable OLAP and data mining. A scalable parallel multi-dimensional infrastructure for OLAP integrated with data mining techniques like association rules and classification is designed and implemented. Multidimensional OLAP systems store data in a multidimensional structure on which analytical operations are performed. For large data sets and a large number of dimensions, multidimensional arrays are impractical and other efficient sparse data structures and techniques are required. We introduce a Bit-encoded sparse structure (BESS) for storage compression which allows aggregate operations on the compressed data. Precomputed aggregate calculations in a *Data Cube* can provide efficient query processing for OLAP applications and data mining. We address the issues involved in parallel construction and maintenance of partial and full data cubes and answering OLAP queries and data mining tasks using them. In particular, issues relating to handling of large data sets, a large number of dimensions, sparse data structures, and parallelism are investigated. Algorithms are presented for our techniques which have been currently implemented on the IBM-SP2 parallel machine and can be ported to another parallel platform with minimal effort. Results show that our algorithms for OLAP and data mining on parallel systems are scalable to a large number of processors, large dimensions, and large data sets, providing a high performance platform for such applications.

Acknowledgements

I express my gratitude to the people who have made this thesis possible. Foremost, I would like to thank Alok Choudhary for giving me this opportunity, and for being a constant source of advice and encouragement. I would like to thank my committee members Prithviraj Banerjee, Peter Scheuermann and Valerie Taylor for their help and guidance. Thanks are also due to Sanjay Ranka and Geoffrey Fox who funded my early years of graduate work at Syracuse. A special thanks to Srinivas Aluru, who provided invaluable guidance, and from whom I learnt many things. I would like to thank my friends who made my graduate experience worthwhile with interesting discussions and diversions.

This thesis would not have been possible without the encouragement and support of my family. I thank my father Ravindra Nath Goil for sharing his experiences and wisdom with me, and being a good friend, while I was growing up. My mother Rita Goil provided me with encouragement, love, and support when I needed them the most. My wife Pradnya made this a wonderful experience with her love and companionship.

I dedicate this thesis to my grandmother, (Late) Lilavati Goil, from whose stories I learnt many things about life in general, and who has been a model of strength and resilience for me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large amount of data are being collected from every day transactions in many application domains. Businesses store and update information on inventories, sales, customer information, among many others. Multi-gigabyte databases are being generated in the health-care industry, satellite imagery and different types of scientific databases. There is a need to analyze this data to gather information that is valuable. *Knowledge Discovery in databases* (KDD) is the process of nontrivial extraction of implicit, previously unknown and potentially useful information from large databases [FPSSU94]. While the overall process is referred to as KDD, the application of algorithms for extracting patterns from data is referred to as *data mining*. Most successful applications have been in target marketing, and analysis and selection of stocks and other financial instruments in the business domain, and applied to astronomy, molecular biology, global climate change modeling in the scientific domain [FPSSU94]. Data analysis of large databases is an important function that has not been addressed in conventional database systems. On-line Analytical Processing (OLAP) is a term used for ad-hoc multi-dimensional analysis of data, querying and retrieving data in various combinations of domain attributes, to unearth relationships and patterns within them. OLAP and data mining are complimentary activities, and if used in conjunction can provide a viable data analysis and knowledge discovery solution.

1.1 Motivation

Conventional relational database technology is designed and well suited for On-Line Transaction Processing (OLTP), in which transactions usually retrieve and update a small number of records. It requires support for millions of small transactions, data integrity and fault tolerance. The main requirements for such systems are availability (24 hours/day for 7 days a week), reliability and a high transaction throughput. Another category of applications which has recently emerged is On-Line Analytical Processing (OLAP) which usually involves large amounts of data, performing complex operations on data items to analyze and gain information of the underlying system. OLAP systems enable analysts to gain insight into the performance of the enterprise through a wide variety of views of data organized to reflect the multi-dimensional nature of the enterprise. Relational databases in their present form are not designed for, and cannot provide good performance for analytical processing of data.

OLTP is designed for day-to-day operations and decision support applications can interfere in the performance of an OLTP system. Generally, OLAP systems are designed differently, using a star or a snowflake schema to reflect the dimensionality of the system. It stores summarized information which is mostly read-only. Many records are accessed at a time to perform the analysis and quick response times are essential to enable many ad-hoc queries, typical of an analysis application. Table 1.1 compares OLTP and OLAP on various parameters [CD97]. Given that operational databases are finely tuned to support known OLTP workloads, executing complex OLAP queries against an operational database would result in unacceptable performance. Decision support might require data that is missing from the operational database, like historical data, since operational databases store only current data. Supporting the multi-dimensional models and operations typical of OLAP requires special data organization, access methods, and implementation methods, not generally provided by commercial DBMSs targeted for OLTP. For these reasons data warehouses are implemented separately from operational databases.

Large scale data warehouses are becoming increasingly available, acting as data

Table 1.1: OLTP vs. OLAP

Parameter	OLTP	OLAP
Function	Day to day operations	Decision support
DB design	Application oriented (ER)	Subject oriented (Star, snowflake)
Data	Current, isolated	Historical, Consolidated
View	Detailed, Flat relational	Summarized, multi-dimensional
Usage	Structured, repetitive	Ad-hoc
Unit of work	Short, simple transaction	Complex query
Access	Read/write	Read
Operations	Index/hash on primary key	Lots of scans
No. records accessed	Tens	Millions
No. users	Thousands	Hundreds
Database size	100MB-GB	100 GB-TB
Metric	Transaction throughput	Query throughput, response

repositories for a large number of applications in various areas. Data available in data warehouses is already integrated, consistent, cleaned and ready for use. Modern information systems can benefit from the knowledge and insight from this data to perform effective decision support. A data warehouse stores data from many different databases and is typically based on relational database technology. OLAP systems make use of this data for multi-dimensional analysis, typically using multi-dimensional databases since they are more suitable for such analysis. Data mining, is another word for automated extraction of knowledge and insight from data to help in better decision making. OLAP integration with data mining helps users to determine what kinds of knowledge to be mined. It provides the flexibility for users to select the desired data mining function after analyzing the data through OLAP.

Table 1.2 shows the evolution of information system in the past few decades. Our motivation is derived from the new challenges posed by the requirements for data navigation and data mining.

Figure 1.1 illustrates the data warehousing and OLAP architecture. A data warehouse contains large amounts of transactional data gathered by an OLTP system and other sources. This data is used by OLAP servers to give a dimensional

Table 1.2: Evolution of Information systems

Step	Question	Technology	Characteristics
(60s) Data Collection	What was average revenue in past 5 years ?	Computers, Tapes, Disks	Retrospective static data delivery
(80s) Data Access	Unit Sales in NY last March ?	RDBMS & SQL	Retrospective dynamic data delivery (records)
(90s) Data Navigation	Unit Sales in NE in March ? Drill down to Boston.	OLAP, Data warehouses, Multi-dimensional databases	Analysis at multiple levels, Trend Analysis
(2000) Data Mining	What's likely to be unit sales next month ?	High Performance Computers Algorithms, Massive databases	Prospective Information delivery

view of the data in an efficient manner to ease the task of ad-hoc query processing. Data marts are specialized repositories for a domain and these may also be used by OLAP servers.

1.2 The Need for High Performance

Typically, large amounts of data are analyzed by OLAP and data mining applications. Ad-hoc analytical queries are posed by analysts who expect the system to provide real-time performance. Parallel computers can be used in such a scenario for a variety of reasons. Scalable solutions can build a highly accurate data mining model more quickly. Mining large databases and constructing complex models take a large amount of computing power which can take hours of valuable time. Scalable parallel systems can reduce this wait time to minutes or even seconds, thus increasing productivity and better understanding of the knowledge discovery process. The use of many processors enables the use of more memory and a larger database can be handled in the main memory attached to the processors. However, disk-based computing is required to handle larger data sets even on parallel machines.

Figure 1.1 illustrates the data warehousing and OLAP architecture and our focus on parallel algorithms, high performance and parallel I/O for OLAP and data mining tasks.

Multi-dimensional databases provide fast response times for OLAP queries as

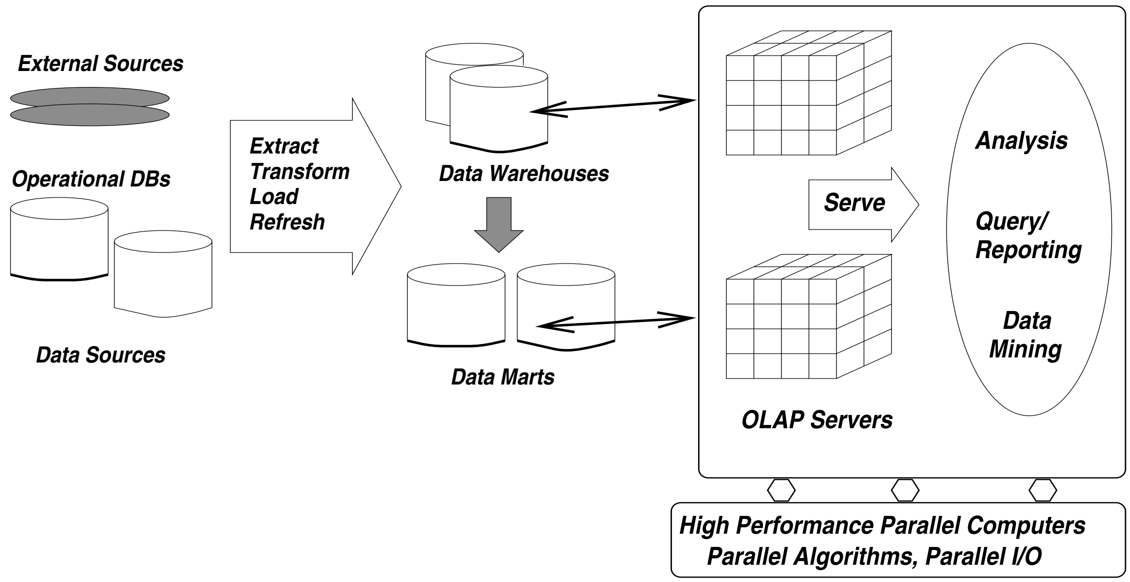


Figure 1.1: Data Warehousing and OLAP Architecture

can be seen from the commercial success these products have attained [Cog98, Pil98, Hyp98, Ora98, Inf98]. These are limited to low dimensions (5 to 10) and small data sizes (order of several Megabytes). However, current methods for multi-dimensional databases have not proven to be scalable in data sizes and the number of dimensions. In this thesis we present a parallel scalable infrastructure with algorithms and techniques for OLAP and integrate it with data mining using the multi-dimensional data model. Results show that our implementations are scalable to a large number of processors. We show the performance of these algorithms on the OLAP council benchmark [OLA97] which models a real OLAP environment on a IBM SP2 parallel machine. Synthetic data sets are also created to evaluate performance for a large number of dimensions.

Data mining techniques are integrated with this parallel OLAP infrastructure which uses summary data present in the data cube. A parallel algorithm for attribute-oriented data mining is presented which uses the computations from the data cube. Decision-tree based classification are constructed efficiently in parallel by using the multi-dimensional model. Performance is achieved by reducing

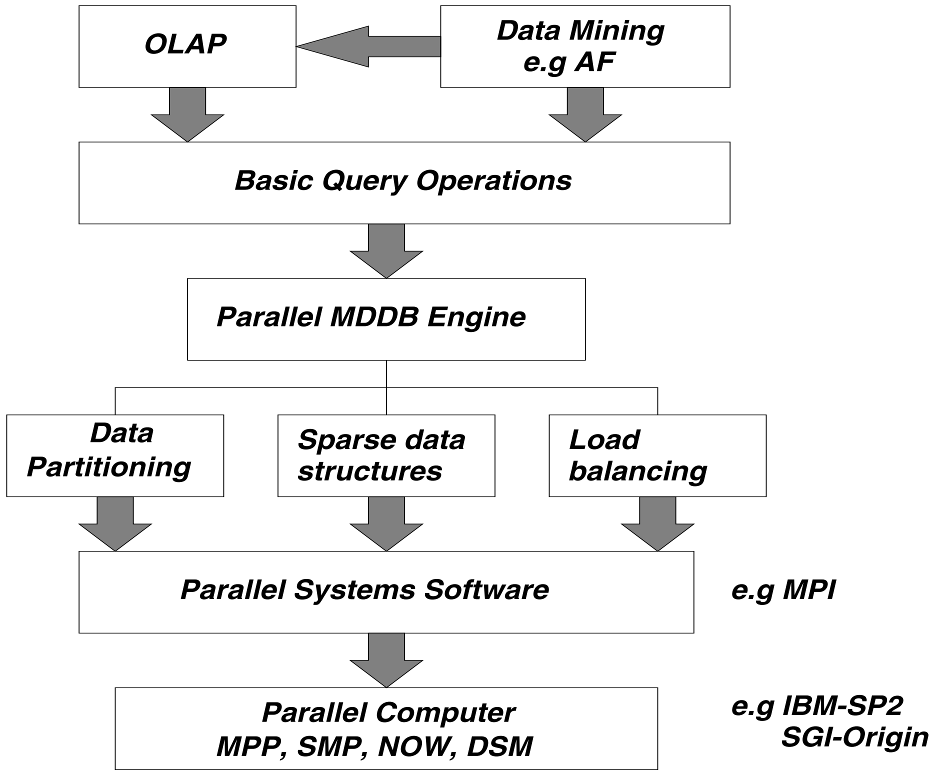


Figure 1.2: Overall schematic

the communication costs associated with maintaining the classification tree at each level.

Figure 1.2 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 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.

In this thesis our implementations are on the IBM SP2 machine. The IBM SP2 is a network of RS/6000 workstations connected together on a high speed communication switch. Our implementations are platform independent and can be ported with little or no effort on networks of workstations or other suitable parallel platforms.

1.3 Contributions

In this thesis we introduce a framework for scalable parallel OLAP and data mining using the multi-dimensional data model. Scalability is addressed in terms of the number of dimensions and the size of the datasets on a distributed memory parallel machine. For a large number of dimensions, the data sets are extremely sparse. Dimensional operations, although performed most efficiently on a multi-dimensional array are no longer possible for large datasets on arrays. Efficient sparse structures are required for storage and operational compression. We introduce a bit encoded sparse structure which provides scalability by allowing dimensional operations on compressed data. The multi-dimensional space is represented as a collection of chunks, and a datum in this space is represented as a bit encoding of the dimensional indices within the chunk space. Aggregation operations are performed on the bit encodings using bit operations without decompressing it into an array. We compute data cubes to develop a parallel multi-dimensional framework for OLAP. Costs for partitioning, load balancing, communication optimizations are modeled and used in a schedule for simultaneous calculations of the cube aggregates. We introduce communication optimizations in schedule generation for parallel cube construction for full and partial cubes, the latter needed for data mining tasks of attribute-oriented association rules. Paged multi-level data structures are designed to provide scalability in the number of data cubes and the number of chunks, which in turn provide the ability to handle a large number of dimensions and large data sizes. We design and implement parallel algorithms for attribute-oriented associations and decision tree based classification on the multi-dimensional model. These use the efficiency of the multi-dimensional operations, resulting in faster algorithms. Decision tree based classification takes advantage of the multi-dimensional model by evaluating the 1-dimensional aggregates in parallel for calculating the split point at each node. A single communication message is exchanged at each level for each node using the idea of *concatenated* parallelism. We have developed PARSIMONY - PARallel Scalable Infrastructure for Multidimensional ONline AnalYtical processing, which is an integrated framework for data mining and OLAP.

1.4 Organization

This thesis is organized as follows. Chapter 2 provides an overview of OLAP, data cubes, and data mining highlighting the related work. Physical data organization for the multi-dimensional data model is described that allows efficient dimensional operations. Chapter 3 presents a new sparse structure called the Bit-Encoded Sparse Structure and an analysis of its performance and comparisons to other existing sparse schemes. Chapter 4 presents schedule generation for parallel construction for partial and complete data cubes. Chapter 5 describes the data cube construction process and algorithms with a main-memory array based approach. Cost models for the schedule are also described for arrays. Chapter 6 presents a scalable disk-based parallel data cube construction infrastructure which supports large data sets in a large number of dimensions. Data structure management for paged data structures and algorithms are presented for disk based data cubes. Chapter 7 describes query processing for OLAP using the materialized aggregates in the data cube. Hierarchical queries and range queries as modeled by the OLAP council benchmark are used for evaluation. Consolidation queries are supported using hierarchies that group attribute values. Chapter 8 presents parallel algorithms for integrating data mining, notably association rules and classification trees, with our multi-dimensional data cube infrastructure. Attribute oriented association rules use the pre-aggregated calculations from the data cube to calculate the various probabilities. Decision-tree classification uses the efficiency of the multi-dimensional data model to compute the 1-dimensional aggregates that are repeatedly used to compute the split points at each node. Chapter 9 presents a summary of contributions and future directions.

Chapter 2

OLAP, Data Cubes and Data Mining

This chapter provides an introduction to OLAP, data cubes and data mining and their inter-relationships. Our work in the area of scalability, parallelism and performance is covered in subsequent chapters, where we present efficient parallel algorithms, memory and communication optimizations, implementations and results.

2.1 On-Line Analytical Processing

On-line Analytical Processing (OLAP) [Cod93] systems enable analysts and managers to gain insight into the performance of an enterprise through a wide variety of views of data organized to reflect the multi-dimensional nature of the enterprise data. OLAP gives insight into data through fast, consistent, interactive access to a wide variety of possible views of information. In contrast to traditional databases, it answers questions like "*what if ?*" and "*why ?*" in addition to "*who ?*" and "*what ?*". OLAP is used to build decision support systems which help in extracting knowledge from data.

OLAP is used to summarize, consolidate, view, apply formulae to, and synthesize data according to multiple dimensions. Queries posed on such systems are quite complex and require different views of data. A relational approach (relational OLAP) can be taken to build such systems. Relational databases are used to build and query these systems. However, a complex analytical query is multi-dimensional in nature, and cumbersome to express in SQL. A schema has to impose a multi-dimensional nature to the dimension-less relational model. Alternatively, multi-dimensional database techniques (multi-dimensional OLAP) have been applied to decision-support applications. Data is stored in multi-dimensional arrays 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. An intuitive view of data provides a sophisticated analytical functionality and support for complex queries. Data relationships are modeled more naturally and intuitively.

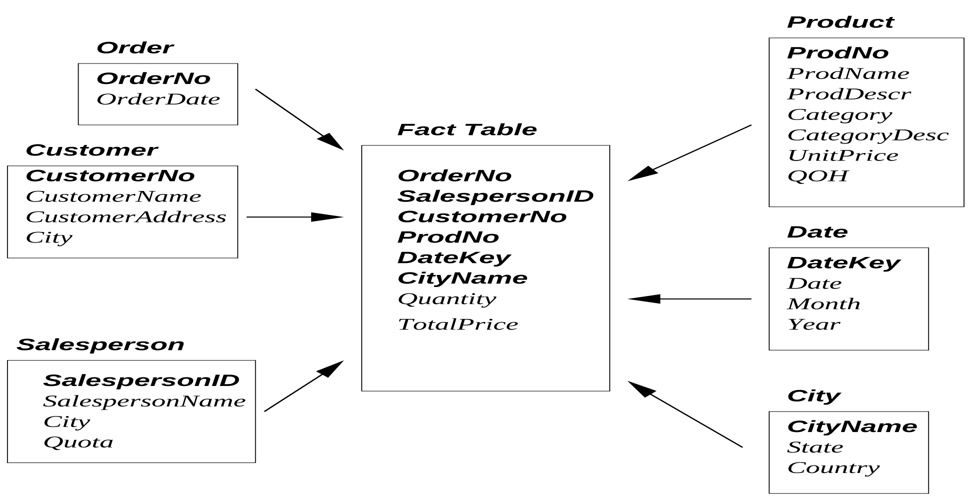


Figure 2.1: Star Schema

2.1.1 ROLAP - Relational OLAP

From an OLAP perspective there are two types of columns in the denormalized data. *Dimension* columns contain information used for aggregations. These columns take a fixed number of values, which is often a part of a hierarchy. For example, location code that supports zip-code, state and region. *Aggregate* columns are calculated amounts like counts averages and sums. Data is represented in a denormalized way called the *star* schema. For each dimension there is a *dimension* table which contains all the information that is specific only to the dimension itself. A *fact* table correlates all dimensions and contains information on the attributes of interest for the intersection of all the dimensions. Each row in the central fact table contains some combination of keys that make it unique. Keys in the central fact table are indexes into the dimension tables. Aggregation calculations are performed on the central fact table and aggregated rows are also kept there. Figure 2.1 illustrates a star schema and the associated fact table and the dimension tables. Each tuple in the fact table consists of a pointer (foreign key) to each of the dimensions that provide its multi-dimensional coordinates, and stores the measure value for the coordinates. Each dimension table consists of columns that correspond to attributes of the dimension. There is one dimension table each for Order, Customer, Salesperson, Product, Date and City in Figure 2.1. Every entry in the fact table points to one tuple in each of the dimensions which contains additional attributes. The star schema does not capture attribute (dimension) hierarchies directly. Alternatively, a *snowflake* schema represents dimensional hierarchies directly by normalizing the dimension tables. Figure 2.2 shows a snowflake schema, in which the Product, Data and City dimensions are normalized. It is easy to maintain the dimension tables in this schema. However, it reduces the effectiveness of browsing, since a entry in the fact table needs to access multiple dimension tables, and hierarchies need to be resolved for each dimension with a hierarchy [Kim96].

Any query on the central fact table requires multiple joins with the dimension tables. Relational databases use novel indexing techniques to execute these queries. Several index structures have been used to exploit indexing to reduce the scanning of data. *Bitmap* indices and *Join* indices are primarily used for ROLAP. A leaf page in an index structure corresponding to a domain value d, traditionally contains a list of the record ids (RIDs) of records that contain the value d. Bit map indices use an alternative representation of the above RID list as a bit vector that has one

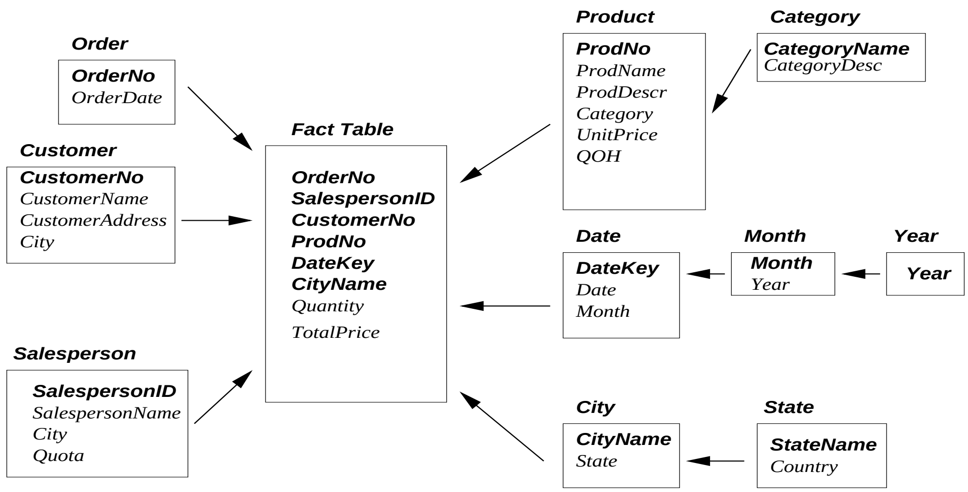


Figure 2.2: Snowflake Schema

bit for each record, which is set when the domain value for that record is d. The bit vector representation of the RID list can speed up index intersection, union, join and aggregation [OQ97]. For example, if there is a query of the form attribute $1 = c$ & attribute $2 = c$, then the qualifying records can be identified by taking the AND of two bit vectors. These are very effective in low cardinality domains (e.g gender) and can be made effective in higher cardinality domains through compression of bitmaps. Figure 2.3 shows an example of bitmaps for region (cardinality of 4)and rating (cardinality of 3). The query about customers in $Region = W$ and with $Rating = L$ can be found by extract the bit map indices for the attribute Region (rows 3, 4 and 5) with the bit W set as 1, and the bit maps for Rating (rows 3, 5 and 6) with the bit for L set as 1. An AND operation for the bits of W and L in the two indices will give the result of the query as rows 3 and 6.

In addition to indices on single table, the specialized nature of star schemas make *join indices* especially attractive for decision support. While traditional indices map the value in a column to a list of rows with that value, a join index maintains the relationships between a foreign key with its matching primary keys. In the context of the star schema, a join index can relate the values of one or more attributes of a dimension table to matching rows in the fact table. For example in Figure 2.1,

Base Table

Cust	Region	Rating
C1	N	H
C2	S	M
C3	W	L
C4	W	H
C5	S	L
C6	W	L
C7	N	H
C8	N	M

Region Index

RowID	N	S	E	W
1	1	0	0	0
2	0	1	0	0
3	0	0	0	1
4	0	0	0	1
5	0	1	0	0
6	0	0	0	1
7	1	0	0	0
8	1	0	0	0

Rating Index

RowID	H	M	L
1	1	0	0
2	0	1	0
3	0	0	1
4	1	0	0
5	0	0	1
6	0	0	1
7	1	0	0
8	0	1	0

Query: Customer where Region = W **and** Rating = L

Figure 2.3: Bitmap indices

there can be a join index on City that maintains, for each city, a list of RIDs of the tuples in the fact table that correspond to the sales in that city. A join index essentially precomputes a binary join in this case. Multi-key joins can represent n-way joins. Join indices can be used with bitmap representations for the RID lists for efficient join processing [OG95].

2.1.2 MOLAP - Multi-dimensional OLAP

A multi-dimensional structure like a multi-dimensional array can store the aggregate values. Dimension columns can be treated as the dimensions of the array. Aggregate values are also stored in multi-dimensional arrays stored separately. Queries are performed by accessing the appropriate data in the array with positions in a dimension acting as an index. A multi-cube architecture stores the aggregates in different arrays. Sparse data is stored either by using sparse data structures or by using array compression.

In our work multi-dimensional data structure are used for storage and manipulation of data. The multi-dimensional data model is discussed later in this chapter. MOLAP will be covered in detail there.

Table 2.1: Categories of aggregate functions

Category	Examples
Distributive	Sum(), Count(), Minimum(), Maximum()
Algebraic	Average(), Standard_Deviation(), MaxN()(N largest values), MinN() (N smallest values), Center_of_Mass()
Holistic	Median(), MostFrequent()(i.e. Mode()), Rank()

2.2 Data Cube

Many queries in a data warehousing environment require summary data, by computing aggregates. In addition to indices, materializing summary data can help to accelerate many queries. The *data cube* operator was introduced by Gray et. al [GBLP96] to support multiple aggregates. Data cube computes aggregates along all possible combinations of dimensions. This operation is useful for answering OLAP queries which use aggregation on different combinations of attributes. Data can be organized into a data cube by calculating all possible combinations of group bys. For a data set with k attributes this would lead to 2^k group by calculations.

The cube treats each of the k aggregation attributes as a dimension in k-space. An aggregate of a particular set of attribute values is a point in this space. The set of points form a k-dimensional cube. Aggregate functions are classified into three categories as shown in Table 2.2. *Distributive* functions allow partitions of the input set to be aggregated separately and later combined. *Algebraic* functions can be expressed in terms of other distributive functions, e.g. *average()* can be expressed as the ratio of *sum()* and *count()*. *Holistic* functions, such as *median()* cannot be computed in parts and combined.

2.2.1 Operations on the Data Cube

Data Cube operators generalize the histogram, cross-tabulation, roll-up, drill-down and sub-total constructs required by financial databases. The following operations can be defined on the data 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.
- Trend analysis over sequential time periods

Figure 2.4 shows a multi-dimensional representation of data with *product*, *date*, *supplier* as *dimensions* and *sales* as a *measure*. Partitioning a data set into dimensions and measures is a design choice. Dimensions usually have a hierarchy associated with them, which specify aggregation levels and the granularity of viewing data. For example $day \rightarrow month \rightarrow quarter \rightarrow year$ is a hierarchy on date.

An aggregate of an attribute is represented by introducing a new value ALL. The right hand side table in Figure 2.5 illustrates a data cube for the set of tuples given in the relation [GBLP96]. It also shows a multi-cube for multi-dimensional OLAP. The 3 dimensional base cube is the original table and the aggregates are in the other arrays calculated from the base cube. The multi-dimensional representation is an efficient storage organization and offers efficient access to elements. Dimensional operations can be implemented much easily, since dimensions reflect the storage organization. A table in a relational organization has no such order, and does

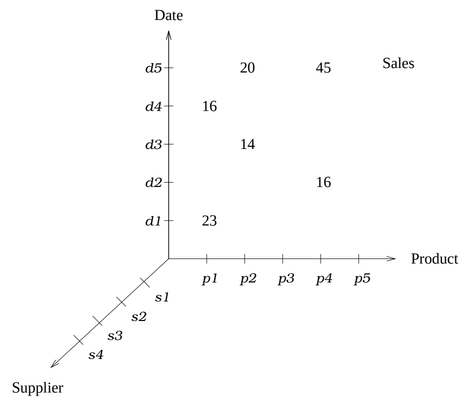


Figure 2.4: Example database of sales data

not have the concept of dimensional storage. Hence, dimensional operations are unnatural and cumbersome to implement and are not efficient.

2.2.2 OLAP Queries

OLAP queries can be modeled as point or region queries in a multi-dimensional space at the lowest level. A roll-up or drill-down operation involve aggregation on a hierarchy of a dimension and these are expressed as a range at the appropriate level of the hierarchy. Below are a few typical OLAP queries involving aggregate computation and can easily be answered using the data cube operator.

1. Give the total sales for each product in each quarter of 1998.

2. For suppler "ABC" and for each product, give the fractional increase in the sales in January 1998 relative to the sales in January 1997.

3. Select the top 3 suppliers for each product category for last year, based on total sales.

4. For each product category, select total sales this month of the product that had highest sales in that category last month.

5. Select suppliers for which the total sale of every product category increased in each of last 5 years.

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

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

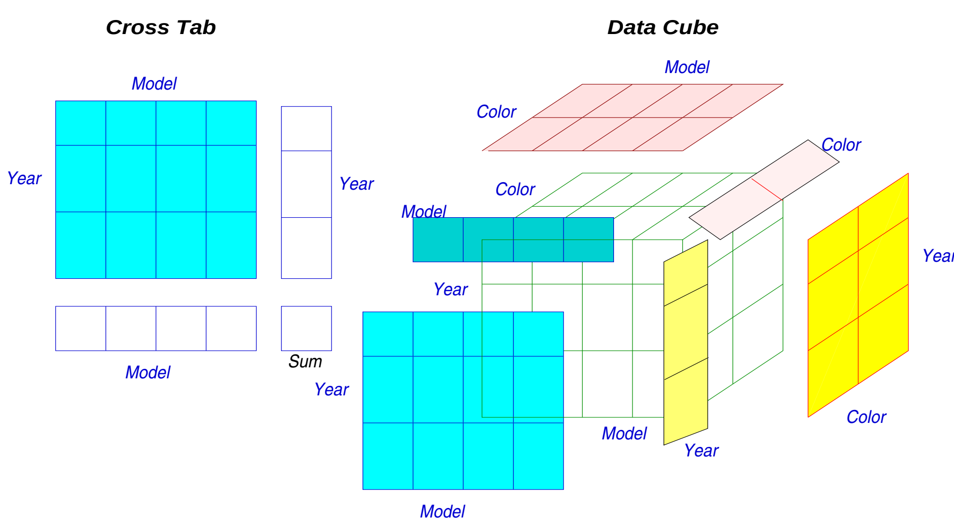


Figure 2.5: Data cube illustration in ROLAP (table) and MOLAP (array)

2.2.3 Cube Computation

In this section we briefly describe the optimizations applied in the ROLAP domain for cube computation. Several algorithms in the ROLAP domain have been proposed to optimize the simultaneous calculation of aggregates. Sorting and hashing is applied for a pipelined construction of the cube from a cube lattice, which is a graph with vertices representing a group by of a cube and a edge between two nodes i and j representing that j can be computed from group by i, and j has exactly one less attribute than i. We also describe the only MOLAP algorithm available in literature. This is shown to perform better than most ROLAP computations for the data cube.

PipeSort and PipeHash

These algorithms use the two basic methods of computing a group by - sorting and hashing in the ROLAP domain. Algorithms *PipeSort* and *Pipehash* [AS96] minimize the cost by computing all group bys in a pipeline from a single sort input stream. These use optimizations to compute multiple group bys by using the optimizations of **smallest parent** (a group by is constructed from the smallest group by available from which it can be computed), **cache-results** (reusing cached results) and **amortize-scans** (reduce disk scans cost). For example, if raw data is sorted on attribute order $ABCD$, then group bys $ABCD, ABC, AB$ and A can be computed without additional sorts. A global planning algorithm using pipelining is used to determine the order of sorting and the group by calculations for PipeSort. Every time a tuple of $ABCD$ is computed it is propagated to compute ABC and from this to AB and so on. Each pipeline is a list of group bys all of which can be computed in a single scan of the sort input stream. PipeSort does not scale very well with respect to the number of attributes. One sort operation is performed for the pipelined evaluation of each path. With k attributes, a lower bound for the number of such sorts is $\binom{k}{\lceil k/2 \rceil}$, which is exponential in k. External sorts are required for large datacubes which make its I/O cost very high.

The PipeHash algorithm uses hashing and careful allocation of multiple hash-tables by partitioning data on one or more attributes. Using a lattice, it first chooses for each group by, the parent group by with the smallest estimated total size. This constructs a minimum spanning tree (MST) where each vertex is a group by and an edge shows a group by computation. Usually, the available memory is not sufficient to compute all the group bys in the MST, hence it needs to be determined which group bys to compute together and how to manage allocation and deallocation of memory. PipeHash computes all group bys from the same partition that contain the partitioning attribute. When computing a subtree, all hash-tables of group bys in the subtree (except the root) are maintained in memory until all its children are created. For each group by, all children are created in one scan of the group by.

An *Overlap* method is presented in [AAD$^+$96] which overlaps the computation of different group by operations using the least possible memory for each computation. It tries to minimize the number of disk accesses by overlapping the computation of the sub-cubes, by making use of partially matching sort orders to reduce the number of sorting steps performed. At the beginning, the tuples in the base data are sorted in some order of attributes. The attributes at each node of the cube lattice are then ordered to be subsequences of this order. For a node A, among the various *parent* nodes, (nodes from which A can be computed) the node which shares the longest prefix in the attribute ordering with A is chosen as the parent. The lattice is labeled with estimated memory requirements, and set of sub-cubes are chosen which can be computed concurrently with the available memory constraints. For some sub-cubes, the estimated memory is allocated ("Partition") and for some others only a single page of memory is allocated ("SortRun"). Memory allocation is done in a breadth-first order, giving priority to sub-cubes with smaller partition sizes. The tuples in Partition are available for pipelining and the tuples in SortRun are written to disk for subsequent merging. The amount of I/O done depends on the partition sizes and the number of sorted runs that have been written to disk. The I/O for this algorithm is at least quadratic in k for sparse data sets [RS97].

Partitioned-cube

Partitioned-cube algorithm uses a divide and conquer approach that divides the relation into partitions that can fit in memory [RS97]. These partitions are used to construct the data cube recursively. It uses the idea from PipeSort of using pipelined paths to reuse the sort order of tuples. A data cube over a relation with T tuples is broken down into $n+1$ smaller sub-datacube computations for a large n. The first n of these sub-datacubes, each has approximately T/n tuples and k cubing attributes. The final sub-datacube contains no more than T tuples and has $k-1$ cubing attributes. The contention is that in the absence of significant skew the I/O cost is *typically* proportional to the number of attributes k. The choice of n is arbitrary in the algorithm and it is not clear how the analysis is affected when data is skewed. A in-memory algorithm, using the idea of pipelining in PipeSort is used to compute the data cubes of partitions that fit in memory.

Cubetrees

A "cubetree" has been defined in [RKR97] as a storage abstraction of the cube. It is realized using packed R-trees and creation and maintenance of the cube is done with sorting and bulk-incremental merge-packing of cubetrees. Multi-dimensional B-trees indices are independent of each other even when they share some dimensions, which results in wastage of space. Also, relational storage typically stores unsorted data which prohibits efficient merge operations during the updates. In a cubetree multiple materialized views map to the same dimensional space. Same index is used for storing aggregate data for different attributes that are necessarily not correlated, to reduce space requirements and increase the buffer hit ratio. Semantics of the index space are defined dynamically on the view to be focused on. A set $\mathcal{V} = \{V_1, V_2, \ldots, Vn\}$ of views is mapped to a collection of Cubetrees $\mathcal{R} = \{R_1, R_2, \ldots R_m\}$ such that each cubetree $R_i = R_{x_1, x_2, \ldots, x_{maxi}}$ is packed having points first sorted by x_{maxi} order, then by $x_{maxi} - 1$ and so on.

Multi-way cubing

A *Multi-way cubing* algorithm [ZDN97] visits the dimension values in the right order to make the computations more efficient in a array-based MOLAP scenario. The data is partitioned and processed in an order that requires only fragments of the array to be present in memory at any one time. Since the array representation provides direct access to the needed cells, the algorithm performs particularly well. It performs better than the algorithms based on sorting, overlap and pipelining, developed in the ROLAP domain and described above. Data compression is used to speed up the I/O by using a sparse representation for chunks below a certain density threshold. Operations are performed on arrays by decompressing sparse data into multi-dimensional arrays. Memory allocation is controlled by keeping only parts of group-by in memory, and scanning the input in an order which minimizes the memory requirements. An optimal dimension order is defined, increasing order of the cardinalities of the dimensions in the group-by, dictated by a minimum memory spanning tree (MMST) of cube computations. This algorithm is not very scalable in a large number of dimensions and large data sizes since it relies on operations on the array chunks in memory and requires several chunks in memory.

2.2.4 Selective Materialization

Efficient implementation of data cubes is discussed in [HRU96, Gup97]. Aggregates are materialized selectively based on certain parameters, such as space available for aggregates, reuse of aggregates, and overall benefit of materialized aggregates to a query stream. In a datacube, the values of the aggregate cells are computable from those of other cells in the same data cube. A cell containing ALL in one of its attributes is an aggregate and is called a *dependent* cell. A subset of all views is materialized when it is too expensive (in either space or time) all views. A lattice framework is used to represent dependencies among views. For queries Q_1 and Q_2, $Q_1 \preceq Q_2$, if and only if Q_1 can be answered using the results of Q_2. Q_1 is dependent on Q_2. For example, a query `(part)` can be answered from the results of a query `(part, customer)` and `(part, customer)` $\preceq$ `(part)`. The $\preceq$ operator imposes a partial ordering on the cube and can be used to organize views in a lattice structure, which is denoted by a list of elements (queries or views) L and a dependence relation $\preceq$. The *top* element , a view on which all other views are dependent is the raw data, containing all attributes. A linear cost model is used, which calculates the cost of answering a query Q from its ancestor Q_A as a function of the size of the table Q_A. This function is the number of rows in the table for query Q_A. The problem is to minimize the average time to evaluate a view with the constraint that only a fixed number of views can be materialized. A greedy algorithm using the idea of *benefit* of a view is computed by considering how it can improve the cost of evaluating views, including itself. For each view w that v covers in a set S of views, the cost of evaluating w using v is compared, and using whatever view from S that offers the lowest cost evaluation of w. The total benefit is the sum over all views w of the benefit of using v to evaluate w. The benefit of the greedy algorithm is shown to be within 63% of the optimal. The problem of materializing both views and indices has also been addressed using the greedy approach [GHRU97].

2.3 Data Mining

Data mining techniques are used to discover patterns and relationships from data which can enhance our understanding of the underlying domain. Data mining can be viewed as an automated application of algorithms to detect patterns and extract knowledge from data [FPSSU94]. 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, Clustering and Trend analysis* [FPSSU94] can be used together with OLAP to discover knowledge from data.

No one technique solves all data mining problems. Familiarity with a variety of techniques is necessary to provide the best approach to solving data mining problems. An overview of data mining techniques is presented next.

2.3.1 Association Rules

Attribute Focusing

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 [Bha95, HCC93, HF95] to data mining are data-driven and can use this summary information to discover association rules.

Since data cubes have aggregation values on combinations of attributes already calculated, the computations of attribute focusing are greatly facilitated by data cubes. An approach to data mining called Attribute Focusing targets the consumer 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 [Bha95] and other applications. Earlier applications of this technique were to software process engineering [Bea93]. We present a parallel algorithm to calculate the "interestingness" function used in attribute focusing on the data cube in the chapter on data mining.

An event, E is a string $E_n = x_1, x_2, x_3, \ldots x_n$; in which x_j is a possible value for some attribute and x_k is a value for a different attribute of the underlying data. E is interesting to that x_j's occurrence depends on the other x_i's occurrence. The "interestingness" measure is the size $I_j(E)$ of the difference between:
(a) the probability of E among all such events in the data set and,
(b) the probability that $x_1, x_2, x_3, \ldots x_{j-1}, x_{j+1}, \ldots x_n$ and x_j occurred independently. The condition of interestingness can then be defined as $I_j(E) > \delta$, where δ is some fixed threshold.

Meta-rule Guided Mining of Association Rules

Metarule-guided mining is an interactive approach to data mining done by specifying hypotheses in the form of metarules for data analysis [KHC97]. These hypotheses may be based on the user's experience, expectations, or intuition regarding the data. The task of a data mining system is then, to confirm the hypothesis by searching for patterns that match the given hypothesis. The use of meta-rules increases the likelihood of finding rules that are of interest to the user and can make the discovery process more efficient by using the metarules to constrain the search space. A *metarule* is a rule template of the form $P_1 \wedge \ldots \wedge P_m \Rightarrow Q_1 \wedge \ldots \wedge Q_l$ where the Ps and Qs are predicate variables and $p = m + l$ is the number of predicates in the metarule. Let X and Y consist of a set of conjunctive predicates. The *support* of a rule $X \Rightarrow Y$ in a data set is the probability that the tuples contain both X and Y. The *confidence* of $X \Rightarrow Y$ is the probability that a tuple contains Y under the condition that it contains X, i.e *Prob(Y—X)*. A rule $X \Rightarrow Y$ is strong if it satisfies both minimum confidence and minimum support thresholds.

By using metarules containing p predicates, rules can be derived by either using a pre-computed n-dimensional data cube or constructing multiple p-dimensional data cubes for a data set with n attributes. The latter approach is more feasible when the number of dimensions is large. Optimizations for construction of multiple cubes can be applied for such a construction by generating an appropriate schedule. More details are presented in the chapter on schedule generation.

Market Basket Analysis

Market basket analysis is a form of clustering used for finding groups of items that tend to occur together in a transaction (market basket). The models it builds give the likelihood of different products being purchased together and can be expressed as rules. It is particularly useful in the retail industry, where information on the products purchased together may be the only data available for mining customer patterns because transactions are anonymous. Mining of association rules over market baskets was introduced in [AIS93b] which gives a formal definition of the problem as: Let $\mathcal{I} = i_1, i_2, \ldots, i_m$ be a set of literals, called items. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq \mathcal{I}$. Each transaction has a unique TID associated with it. A transaction T contains X, a set of some items in $\mathcal{I}$, if $X \subseteq T$. An association rule is an implication of the form

$X \Longrightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. The rule $X \Longrightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. The rule has *support* s in the transaction set $\mathcal{D}$ if $s\%$ of transactions in $\mathcal{D}$ contain $X \cup Y$.

Association rules are generated over a set of transactions $\mathcal{D}$, that have support and confidence greater than the user specified thresholds for them.

Apriori algorithm for mining association rules for market basket data has been developed in [AS94]. Parallelization of Apriori is presented in [AS96]. The first pass of the algorithm counts item occurrences to determine the *frequent* (having minimum support) 1-itemsets. In each subsequent pass, a seed set of large itemsets found in the previous pass is used to generate *candidate* itemsets, which are potentially large itemsets. Actual support for these itemsets is then calculated to determine which ones actually are large itemsets and these are then used for the next pass. This process continues till no new large itemsets are found. Parallel data mining for association rules which use the Apriori algorithm as their basis and use different data distributions is presented in [HKK97].

Correlations

The chi-squared test for correlations for measuring significance of associations has been used in [BMS97]. Negative associations are also identified since both the presence and absence of items are used as a basis of generating rules. The statistical definition for dependence of two sets A and B, $\frac{P(A \wedge B)}{P(A)P(B)}$ is used. The definition of the chi-square statistic used specifies whether all k items in the itemset are k-way independent. Let R be the set of all possible basket values, and r be a single basket value. Each value of r denotes a cell, viewing R as a k-dimensional table, called a contingency table. To test whether a given cell is dependent, we must determine if the actual count in cell r differs sufficiently from the expectation. The chi-squared statistic is defined as $\chi^2 = \sum_{r \in R} \frac{(O(r)-E[r])^2}{E[r]}$, where $O(r)$ denotes the number of baskets falling into cell r, $E[i_j] = O_n(i_j)$ for a single item, and $E[r] = n \times E[r_1]/n \times \ldots \times E[r_k]/n$. This represents the normalized deviation from expectation. If all variables are really independent, the chi-squared value would be

0. Otherwise cut-off value is used to measure the significance of the association. The idea of measurement of the association from the cell value is similar to the one used in attribute focusing. However, the dimensions are the items present in the contingency table. As a limitation of the applicability of the chi-squared test, it can only be applied if all the cells in the contingency table have an expected value greater than 1 and 80% of the cells have a value greater than 5. As a workaround, in [BMS97], cells with small expected values are ignored.

Generalized Associations

A taxonomy present over items can be used to find association between items at several levels. This problem of mining generalized association rules is addressed in [SA95]. A taxonomy is generally a *is-a* hierarchy on items. At the lowest level of the taxonomy are the items in the transaction list. Rules at lower levels may not have minimum support. Many significant associations may not be discovered if the rules are restricted to the items on the leaves of the taxonomy. For example, if Jackets and Raincoats are a part of Outerwear, few people might buy hiking boots with Jackets but many people buy Outerwear with hiking boots. Further, taxonomies can be used to prune uninteresting or redundant rules. An Apriori style algorithm is used, and ancestors in the taxonomy are added to the items by replacing the item by its ancestor at the highest level.

Calculations in a data cube can be used to determine the large itemsets by calculating the appropriate weights from the available count values in each cell. Hierarchies on dimensions are used to represent taxonomies, and generalized association rules are supported at various levels of the cube through the traversal of these hierarchies, aggregating appropriately.

2.3.2 Classification

Classification is a learning function that classifies a data item into one of several predefined classes. It is a key data mining technique in which database tuples are considered as training samples to produce a model of the given data. Each tuple is assumed to belong to a predefined class, as determined by one of the attributes, called the *classifying attribute*. Once this is derived, the classification model can be used to categorize future data samples.

Each sample of a solved problem (available data) consists of observations and the corresponding correct class membership. Usually, a single conclusion or class will result for a given pattern of observations. There might be applications where multiple class membership is important, these can often be decomposed into several subproblems involving individual class membership.

Techniques from Statistics, Neural Networks, Machine Learning, Expert Systems and Genetic Algorithms have been applied to classification with differing amount of success. A well-accepted method is using decision-trees for classification. Decision tree classifiers obtain similar and sometimes better accuracy when compared with other classification methods.

A decision tree is a tree like structure in which the internal nodes represent a decision or a test on a data attribute, and each outgoing branch corresponds to a possible outcome of a test. Each leaf node represents a class. In order to classify an unlabeled data sample, the classifier tests the attribute values of the data sample against the decision tree. The tree is traversed by tracing a path from the root to a leaf which holds the class prediction for that sample.

Data Cubes can be used to generalize data from a training set to a higher level on which tree induction can be performed. The *count* information attached with each generalized cell and a classification threshold can be used to determine whether a significant set of the examples of the node belongs to a single class.

Decision Tree Methods

Decision tree classifiers use an initial tree building phase and then apply pruning to it. A *divide and conquer* approach is taken to repeatedly partition the training data. The data is split into two or more partitions using an attribute. This process is repeated recursively until all the examples in each partition belong to one class. The tree growth phase is the most time consuming step of the two, and this phase is the target for parallelization.

Early work in decision-tree based classification was targeted to relatively small data sets. ID3 [Qui86] and CART [BFOS84] have been used quite often as efficient decision tree algorithms. Efficiency and scalability become important issues when these algorithms are applied to mine very large, real-world databases. Memory-resident algorithms are no longer efficient for such data sets. Some of these issues were addressed in the SLIQ [MAR96] decision tree algorithm which performed classification on large data sets. This relies on a memory resident data structure which limits its scalability. SPRINT [SAM96] is a parallel algorithm which has data structures that do not have to be memory resident and hence can handle data sets too large for SLIQ. Another scalable approach has been proposed by using generalization using attribute-oriented induction and inducing decision-trees for classification at different levels [KWC+97]. Another parallel formulation of classification on large data sets is presented in [JKK98].

Evaluation of Splits

A splitting index is used to evaluate the alternative splits for an attribute to pick up the best split. Several measure of "goodness" are used. *gini* index is defined for a data set T containing examples from n classes as $gini(T) = 1 - \sum_{j=1}^{n} {p_j}^2$, where p_j is the relative frequency of class j in T.

An alternative is to use an information theoretic approach [Qui93]. Let the generalized data be a set P of p data samples. Suppose the classifying attribute has m distinct values defining m distinct classes $P_i, 1 \leq i \leq m$. Suppose P contains p_i samples for each P_i, then the probability that an arbitrary sample belongs to class P_i is p_i/p. The expected information needed to classify a given sample is given by $I(p_1, p_2, \ldots, p_m) = -\sum_{i=1}^{m} \frac{p_i}{p} \log_2 \frac{p_i}{p}$. An attribute A with values $a_1, a_2, \ldots, a_k$ can be used to partition P into $C_1, C_2, \ldots, C_k$, where C_j contains those samples in C that have the value a_j of A. Let C_j contain p_{ij} samples of class P_i. The expected information based on the partitioning by A is given by, $E(A) = \sum_{j=1}^{k} \frac{p_{1j}+\ldots+p_{mj}}{p} I(p_{ij}, \ldots, p_{mj})$. Thus, the information gained by branching on A is $gain(A) = I(p_1, p_2, \ldots, p_m) - E(A)$.

The attribute chosen at a decision-tree node to branch on will have the highest information gain.

2.3.3 Other Techniques

Some techniques used in data mining are based on artificial neural networks and genetic algorithms. For details refer to [BL97]. We briefly overview these techniques here without going into any further detail.

Neural Networks

Neural networks are modeled on neural interconnections present in the brain. A training set is usually created and the network is made to learn from it. Patterns inside the network are generalized for classification and prediction. Neural networks consist of basic units modeled on biological neurons. Each unit has many inputs that it combines into a single output value. These units are connected together, so that outputs from some units are used as inputs to some other. The combination function of a unit is called the *activation function*. Activation functions have non-linear behavior, consisting of a *combination function* (e.g weighted sum, AND, OR etc.) that combines the inputs and the *transfer function* (e.g. sigmoid, linear and hyperbolic), which transfers the value of the combination function to the output of the unit. A one-way flow of the outputs of one unit to the input of another where there are no cycles is called the *feed-forward* network.

Neural networks are a good choice for most classification and prediction tasks. A good training set consisting of records, whose prediction or classification values are already know,n is chosen. A neural network is trained by back-propagation which has the following steps:

1. Using the existing weights in the network of the training set it calculates the outputs

2. Error is calculated by taking the difference between the calculated result and the expected result.

3. The error is fed back to the network and the weights are adjusted to minimize the error.

Neural networks can handle a wide range of problems and can potentially produce good results even in complicated domains. But they also have drawbacks since they require inputs in the range of 0 to 1, and they may converge to an inferior solution prematurely.

Genetic Algorithms

Genetic Algorithms applies the mechanics of genetics and natural selection to a search for finding the optimal sets of parameters that describe a predictive function. It is used for predictive data mining. Genetic algorithms uses the selection, crossover, and mutation operators to evolve successive generations of solutions. As the generations evolve, only the most predictive survive, until the functions converge to an optimal solution.

Although genetic algorithms can handle a wide variety of data types, are applicable to optimization and produce explainable results, they have their weaknesses. It is difficult to encode many problems in genetic algorithm terms. They are computationally expensive and there is no guarantee of optimality.

2.4 Multi-dimensional data model

The information industry has undergone a steady evolution in database power and flexibility, starting from flat files, hierarchical to relational and distributed relational technologies, data structures have evolved to match more closely the way humans visualize and work with data. A recent development in the trend has been the recognition of the usefulness of multi-dimensional databases for data analysis applications. Multi-dimensional database technology facilitates flexible, high performance access and analysis of large volumes of complex and interrelated data [Sof97].

It is more natural and intuitive for humans to model a multi-dimensional structure. Figure 2.6 shows the rows of a sales database table with Model, Color and

Model	Color	Sales
Mini Van	Red	6
Coupe	Blue	5
Sedan	Red	4
Mini Van	Blue	5
Mini Van	White	4
Coupe	Red	3
Coupe	White	5
Sedan	Green	3
Sedan	Blue	3
Mini Van	Green	4
Coupe	Green	7
Sedan	White	2

Sales Volumes

Model \ **Color**	Red	Blue	White	Green
Mini Van	*6*	*5*	*4*	*4*
Coupe	*3*	*5*	*5*	*7*
Sedan	*4*	*3*	*2*	*3*

Figure 2.6: Sales volume relation and a two dimensional array representiion

Sales as the attributes. An alternative way to represent this table-based data is an array shown in Figure 2.6. This data matrix is known as a "cross tab" view. The matrix is an example of a two dimensional array which is the fundamental structure of a multi-dimensional database. In an array each axis is called a *dimension* and each element within it is called a *position*. In Figure 2.6, the dimensions are MODEL and COLOR, and the positions for them are {Mini Van, Coupe, Sedan} and {Red, Blue, White, Green} respectively. The sales volume figures, called *measures* are located at the intersections of the dimension positions. The intersections are called *cells* and are populated with measures.

There are several reasons for an array to be an efficient and effective means of presenting data over a relational table. Firstly, a great deal of information is gleaned immediately upon direct inspection of an array. Secondly, the array conveniently groups like information in columns and rows. This shows that a multi-dimensional array structure represents a higher level of organization than the relational table. The structure contains valuable information regarding the relationships between the data elements because our "perspectives" are embedded directly in the structure as dimensions as opposed to being placed as fields. All possible values of each dimension are presented as positions along the dimension. This structure apart from presenting information, also has quantifiable performance advantages. As an example, if sales volume for all blue sedans is needed from the example in Figure 2.6 then we must scroll through each individual record to identify all possible matching records. On the other hand, the multi-dimensional array structure presents the data for blue sedans in a pre-organized way. In the above discussion we have considered a 2 dimensional example for the ease of presentation. This discussion will naturally extrapolate to higher dimensions.

2.4.1 Physical Data Organization

Multi-dimensional arrays have been traditionally used to store multi-dimensional data. A cell position in a k-dimensional array is given by an index containing an offset in each of the k dimensions. Access to an element is done by using these index values which makes the calculation of the position of the data element very fast. A multi-dimensional array can be stored in memory in multiple ways. Row major or column major organizations are commonly used. Figure 2.7 illustrates row major and column major storage in memory. These organizations store the elements of a dimension contiguously, striding the other dimensions appropriately. Consider a k-dimensional array, with dimensions $d_0, d_1 \dots d_{k-1}$ and sizes $n_0, n_1 \dots n_{k-1}$. Let there be an ordering of the dimensions with d_0 being the inner most dimension and d_{k-1} be the outermost. For such an organization of data in memory, d_0 is stored contiguously. Adjacent elements in the multi-dimensional array will be stored with a stride of size n_0 for dimension d_1. Similarly, the other dimensions will be strided with increasing stride lengths. If the multi-dimensional array is stored on disk, data accesses to access consecutive elements in a higher dimension will require more disk reads. This will also result in poor cache performance if the data is already in memory. This effect will worsen as we access data for the outer dimensions, increasing from d_1 to d_{k-1}, as stride lengths increase. This is only true in the case where one dimension is given preference in contiguous storage over another. However, it is important to note that the stride lengths of the dimensions are known from the dimension sizes. On the other hand, chunking, where blocks of multi-dimensional data is stored together in chunks, does not favor any particular dimension for the storage of a dimension.

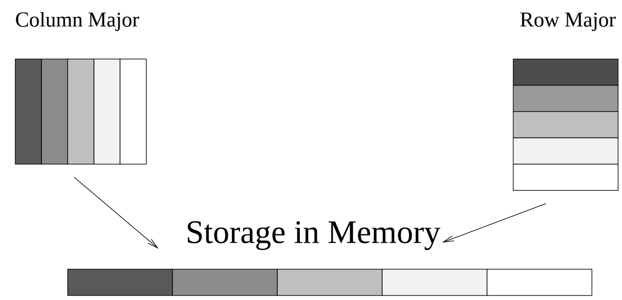


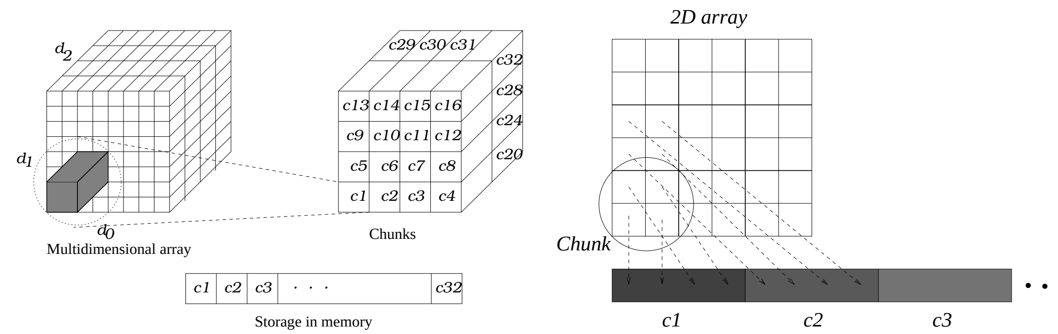
Figure 2.7: Row major and column major storage

2.4.2 Chunking

A *chunk* is defined as a block of data from the multi-dimensional array which contains data in all dimensions. A collection of chunks would then define the entire array. Figure 2.8(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. Chunk storage for a two dimensional array is shown in Figure 2.8(b). Accesses for a particular dimension, say d_{k-1} in the row-major storage, do not require strided accesses of stride size $\prod_{i=0}^{k-2} d_i$. Instead, contiguous elements in that dimension will be present in each chunk strided by small amounts since the chunk stores the elements in some dimension order.

2.4.3 Data Access Operations in OLAP and Data Mining

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



(a) Chunking for a 3D array (b) Storage in memory for a 2D chunked array

Figure 2.8: Storage of data in chunks

1. **Retrieval of a random cell element**

   A single cell is retrieved by specifying the values of all dimensions. For a k-dimensional cube, the dimensions are specified where d_1 takes value val_1, d_2 takes value val_2 and so on, represented by $(val_1, val_2, \ldots val_k)$ dimension vector.

2. **Retrieval along a dimension or a combination of dimensions**

   If a dimension or a combination of dimensions is unspecified, all values are returned for that dimension for the combination of the specified values. As an example, consider the case when d_1 and d_2 are unspecified and the rest of the dimensions are assigned values, represented by $RETRIEVE(*, *, val_3, \ldots val_k)$. A 2 dimensional plane, with all values in dimensions d_1 and d_2, is retrieved for the specific values of the other dimensions. Any m of the k dimensions may remain unspecified and this will return values for a m dimensional hypercube. Figure 2.9 is an illustration of retrieval along dimensions d_1 and d_2 done by fixing a value for dimension d_0 to 5, specified as $RETRIEVE(5, *, *)$.

3. **Retrieval for values of dimensions within a range (Range Queries)**

   Often values are needed between a range in several dimensions. If d_1 is constrained between val_{11} and val_{12} and the rest of the dimensions are specified

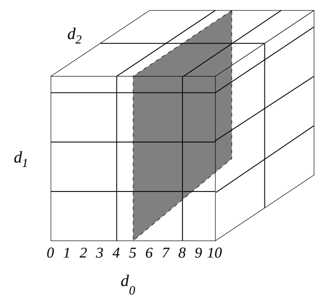


Figure 2.9: Retrieval along d_1 and d_2 by restricting dimension $d_0 = 5$

at a certain value, represented by $(val_{11} : val_{12}, val_2, \ldots, val_k)$. Any m of the k dimensions can have a range specified and values will be retrieved within that range. This is a m-dimensional range query. Figure 2.10 illustrates a range query in 3 dimensions, specified as $RETRIEVE(3 : 6, 2 : 10, 0 : 6)$.

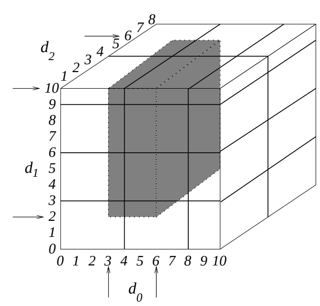


Figure 2.10: Range query in 3 dimensions, defining a range in all dimensions

4. **Aggregation operations on dimensions**

   Summarization of data along a particular dimension or combination of dimensions is often required. The aggregation can be for a single dimension or a combination of dimensions. Typically, in the construction of a data cube, a single dimension is aggregated from one level to another. To maintain a full data cube, aggregation of a single dimension is done for each dimension for the cubes present at a level. For any specified dimension j, values of cells along dimension j are aggregated using an aggregation operation. Each cell along dimension j is aggregated resulting in a $k-1$ dimensional cube.

This access pattern is different from the one in which retrieval is done along dimension j, because here all values are accessed to apply the aggregation operation whereas in the previous operation only the particular values for dimension j are accessed in the remaining $k-1$ dimensions. Figure 2.11 illustrates aggregation for a 3 dimensional cube along dimension d_0, represented as $AGGREGATE(*, d_1, d_2)$.

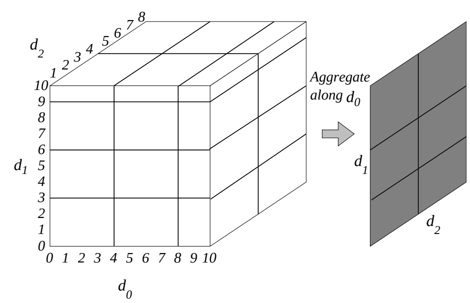


Figure 2.11: Aggregate along a single dimension (d_0) for a 3 dimensional cube

5. **Multi-dimensional Aggregation (Generalization/Consolidation: lower to higher level in hierarchy)**

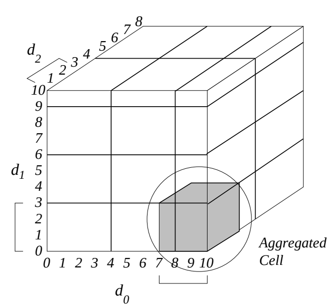


Figure 2.12: Aggregate along all dimensions

Another type of aggregation is one that aggregates values in all dimensions. This is required for ascending a generalization hierarchy in which a k dimensional section is aggregated to get a single value for a cell at a higher level in the hierarchy. Alternatively, a subset of dimensions can be aggregated to reduce dimensions and to answer specific queries. This is a generalized case of the previous operations, where the aggregation was on a single dimension. The operational complexity can perhaps be reduced if multiple dimensions are aggregated together. This aggregation can be represented as $AGGREGATE(d_0 : level, d1 : level, \ldots, d_n : level)$, where $level$, denotes the level of aggregation for the dimension. Figure 2.12 illustrates an aggregation in 3 dimensions, for a single cell, denoted by $AGGREGATE(d_0 : 1, d_1 : 1, d_2 : 1)$.

In Chapter 3 we study the implications of a sparse data structure on each of these operations and provide an analysis for each operation.

Chapter 3

Bit-Encoded Sparse Structure

Multi-Dimensional arrays are good structures for dense data, in which most cells are occupied by valid data elements. However, to store sparse data, space is wasted since many cells will remain unused. In the scientific computing community, considerable work has been done to represent sparse matrices in 2 and 3 dimensions using suitable sparse data structures. To the best of our knowledge, higher dimension sparse data structures have not received much attention in the scientific domain.

Most sparse data may not be uniformly sparse. Dense clusters of data can be found and stored as multi-dimensional arrays. Sparse data structures are needed to store the sparse portions of data. Data in a multi-dimensional database can be divided into chunks for storage. These chunks can then either be stored as dense arrays or stored using an appropriate sparse data structure as illustrated in Figure 3.1. Chunks also act as an index structure which helps in extracting data for queries and OLAP operations.

In this chapter we introduce a sparse data structure for multi-dimensional data useful for OLAP workloads in terms of storage requirements and data access performance. Typically, sparse structures have been used for advantages it provides in terms of storage, but operations on data have been performed on a multi-dimensional array which is constructed from it. 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 multi-dimensional structures for a large number of dimensions,

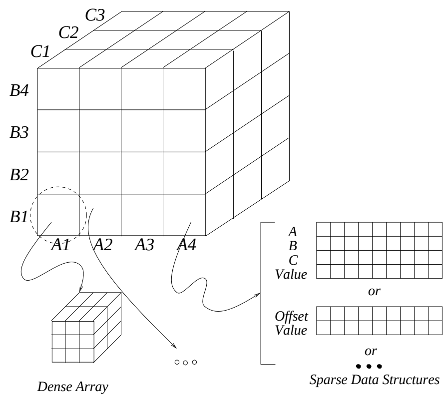


Figure 3.1: Dense and sparse chunk storage for the datacube

we will be 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 BESS.

Some indexing schemes have been reported in the literature for support of multi dimensional sparse data. Array storage issues for sparse data have been addressed in [ZDN97]. Data compression and storage of chunk-offsets have been used within chunks of data. A multi-level organization is used in Essbase [Col96], in which dimensions are split into a sparse set and a dense set. The sparse dimensions are stored in an index structure that has pointers to the corresponding multi-dimensional array for the remaining dimensions. Extensions of R-trees have been considered in [Sar97]. The issue of sparse and dense clusters for storage in multiple dimensions is raised there. The tree searches for data on a R-tree may become cumbersome for some OLAP queries if the search is not very restrictive, leaving some of the dimensions unspecified. Moreover, there have been no efforts to show the use of R-trees in OLAP using a high number of dimensions.

Table 3.1 describes the parameters used in the analysis presented in the following sections. These parameters control the amount of memory and the layout of data in memory for the data storage scheme under consideration. k and n_i together control the number of cells in the data. The valid data among these is controlled by the density D. c, the cell size is the amount of data stored with each combination of dimensions.

In what follows, N_{total} is the total number of valid cells in the data. This can be controlled by adjusting the density factor D. N_{chunks} is the average number of valid cells contained in a chunk.

Table 3.1: Parameters for analysis

Parameter	Description
k	Number of dimensions
n_i	Size of dimension, $0 \leq i < k$
d_i	Dimension numbers, $0 \leq i < k$
D	Density
c	Cell size

3.1 Bit-Encoded Sparse Storage (BESS)

A sparse data structure that treats dimensions symmetrically will be really helpful for OLAP queries which are ad-hoc and not any dimension specific. Typical operations need fast access to index an element in each dimension to perform summary operations similar to what is implicit in a multi-dimensional array.

Bit encoded sparse storage maintains indices for a mapping of each dimension in bit encodings. Unique values in each dimension d_i can be represented by an integer value between 0 and n_i. This will require $\lceil \log n_i \rceil$ bits to represent dimension d_i. Since the dimension sizes are known, we can assign the appropriate number of bits to represent each dimension. The total storage required in this representation is $(c + \lceil \frac{\sum_{i=0}^{k-1} \lceil \log n_i \rceil}{wordsize} \rceil) N_{total}$. Figure 3.2 shows a storage of the indices for 3 dimensions, with $|A| = 100, |B| = 1000, |C| = 10000$, requiring 7, 10 and 13 bits respectively. These can be stored in a 4-byte integer and represents a 3-dimensional cube having its product of dimensions as 10^9. If each cell contains an integer the cube will require 3.72 GB.

The storage requirement for this method can be improved if chunking is used. Sizes of chunks in the various dimensions are much smaller than the total dimensions

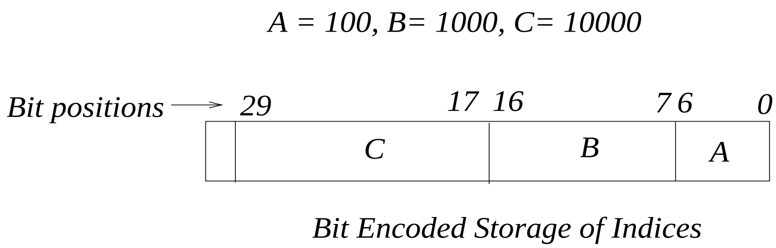


Figure 3.2: Bit Encoded Sparse Storage

sizes, and hence can be represented in fewer number of bits. If h_i, $0 \leq i < k$, is the size of a chunk in each dimension, the number of chunks is $\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil$. The storage required is $(\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil) * ((c + \lceil \frac{\sum_{i=0}^{k-1} \lceil \log h_i \rceil}{wordsize} \rceil) N_{chunk})$

Figure 3.3 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.

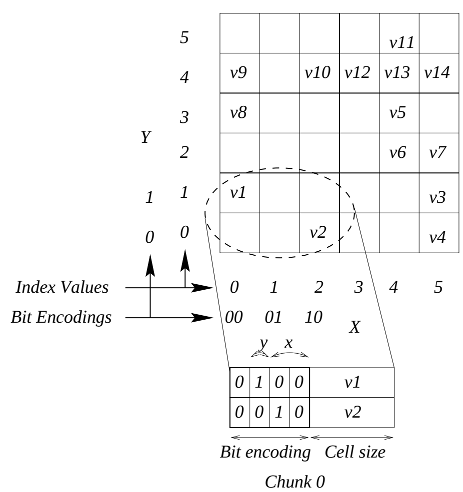


Figure 3.3: An example chunk storage with Bit Encoded Sparse Storage

A chunk index structure stores the chunk offsets for each chunk. Chunks are stored in memory in some order of dimensions. Without loss of generality let us assume that the order of storage is in the order X followed by Y in Figure 3.4. 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 by the number of bits and using a bit-mask to extract the bits that are used to encode each dimension.

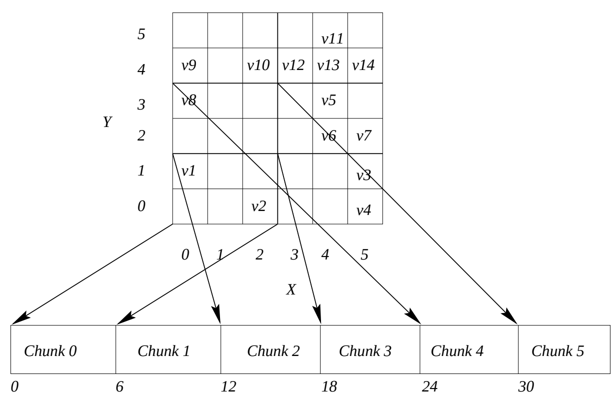


Figure 3.4: Storage of chunk offsets for index retrieval

Operations on BESS are described in a later section comparing them with other sparse storage schemes.

3.2 Other Sparse Storage Schemes

Next, we look at sparse data structures reported in the literature for multi-dimensional data and compare them in terms of storage requirements and data access performance with the bit-encoded sparse storage structure introduced in the previous section. The analysis parameters are the same as described in Table 3.1.

3.2.1 Index-Value pairs

The simplest scheme to store sparse data is the index of a position in a multi-dimensional array together with data. Figure 3.5(a) shows this form of sparse storage for a 2-dimensional array. Each entry stores a value of each of the dimensional index of the array along with the value. Without imposing any restriction on the values for the sizes of the dimensions an integer value is needed for each dimension to store its index. Hence $(c + k)$ storage is required for each cell. The total storage is hence $(c + k)N_{total}$.

The performance of this storage scheme depends on the density D, and the relation between c and k. For a low D, and if $c \geq k$, no more than twice the space required for the cells is needed. As D increases and with $k > c$, the effects of storing k integer values for the dimension indices will result in poor storage.

Storage by chunks can be exploited for this method, by storing the sparse chunks as Index-Value pairs. This would require $(c+k)N_{chunk}$ storage for each sparse chunk.

3.2.2 Offset-Value pairs

Offset-Value pair stores calculates an offset value from the start of memory location where the multi-dimensional array is stored. This offset value is stored along with each data cell (Figure 3.5(b)). Hence only one 32-bit integer value is required to store the offset. The data storage requirement is $(c + 1)N_{total}$.

The Offset-Value pair storage scheme linearizes the multi-dimensional representation by calculating the offset. Accesses to data values usually require the values of the various dimensional index in multi-dimensional operations. These can be obtained from the offset value by performing suitable operations. These would typically involve a division, a multiply and a subtract operation using the dimension sizes.

Chunking can be used to store sparse data using this scheme similar to the one described for BESS. This would require $(c + 1)N_{chunk}$ storage per chunk.

This method has the lower overhead in terms of storage costs. Since it stores an offset from the start of the chunk as an index, computation needs to be done on the offset to retrieve each of the dimension index values. This is often required in queries which perform operations along a dimension or a combination of dimensions. An operation involving a divide, multiply and a subtract is done to retrieve each dimension.

3.2.3 Compressed Sparse Dimensions

Compressed Sparse Row (CSR) storage has been extensively used for operations on sparse matrices in scientific computing. This is an extension of CSR for higher dimensions. Each data element is stored contiguously in some order of its dimensions. See Figure 3.5(c). Suppose the order of dimensions is from $d_0 \dots d_{k-1}$ from innermost to outermost. The valid cells of the array are stored contiguously in this dimension order. k levels of index values are maintained to dereference these values. All dimension indices are stored for d_0. Indices into this array are stored for d_1, pointing to the offsets for the values in this dimension. An example is illustrated in Figure 3.5. In this scheme the order of the dimensions affect the storage requirements for these dimension indices. The storage required for the indexes is calculated as follows. The storage for d_0 is N_{total} elements. For d_1, $\prod_{i=1}^{k-1} n_i$ entries are needed. For each subsequent d_j, $\prod_{i=j}^{k-1} n_i$ entries are stored. Data size is cN_{total}. Hence, the total storage for this scheme is $(c+1)N_{total} + \sum_{j=1}^{k-1} \prod_{i=j}^{k-1} n_i$.

Consider the dimensions to be sorted in non-decreasing order of their sizes, $n_0 \leq n_1 \leq \ldots \leq n_{k-1}$. Now consider two extreme cases, one with dimension ordering from $d_0, d_1 \dots d_{k-1}$ and another in the order $d_{k-1}, d_{k-2} \dots d_0$. The storage requirements for the index when d_0 is the innermost dimension and d_{k-1} is the outermost is $\sum_{j=1}^{k-1} \prod_{i=0}^{k-j-1} n_i$. On the other hand, the storage required for a dimension ordering with d_{k-1} as the innermost dimension and d_0 as the outermost is $\sum_{j=1}^{k-1} \prod_{i=j}^{k-1} n_i$. Clearly, $\sum_{j=1}^{k-1} \prod_{i=0}^{k-j-1} n_i \leq \sum_{j=1}^{k-1} \prod_{i=j}^{k-1} n_i$ for $n_0 \leq n_1 \leq \ldots \leq n_{k-1}$. Consider an example with $k = 4, n_0 = 10000, n_1 = 1000, n_2 = 100, n_3 = 10$. For the dimension order d_{k-1} (innermost) to d_0 (outermost) the storage for the indexes is $n_0 \times n_1 \times n_2 + n_0 \times n_1 + n_0$. Thus, $10^9 + 10^7 + 10^4$ entries are needed for this. We are not considering the storage for the innermost dimension because an index is kept for each data element which is the same for both dimension orderings. For a dimension

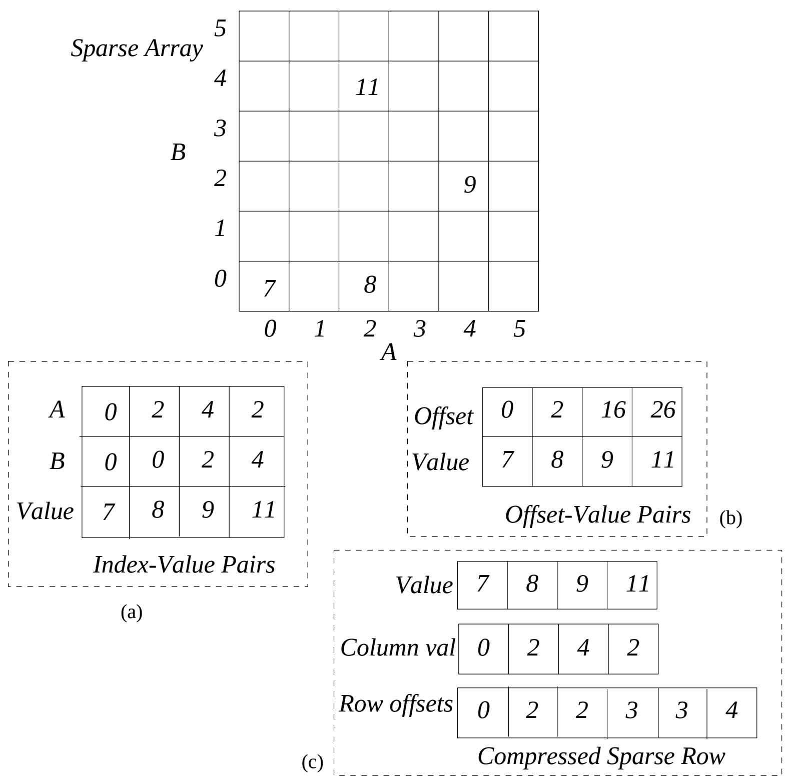


Figure 3.5: Sparse data structures for a sample sparse array: Index-Value pairs, Offset-Value pair and Compressed Sparse Dimensions format

ordering of d_0 (innermost) and d_{k-1} (outermost) the number of entries required are $n_1 \times n_2 \times n_3 + n_2 \times n_3 + n_3$, which is $10^6 + 10^3 + 10$. Keeping the largest dimension as the innermost dimension and progressively smaller ones as outer dimensions is the better choice for this scheme. Data access performance for this structure is presented in a later section.

3.2.4 Sparse-Dense split storage

This method has been used in *Essbase*, a multi-dimensional OLAP product from ArborSoft [Col96] [Ear94]. Dimensions are split into dense and sparse dimensions. The dense dimensions are stored as a multi-dimensional array blocks. The sparse dimensions act as an index into the dense blocks. Among the many possible ways

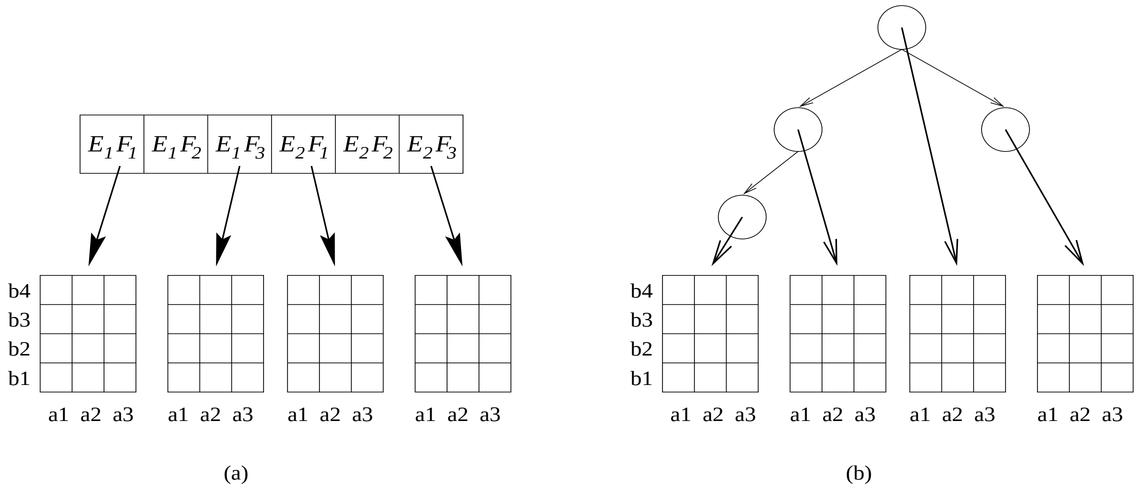


Figure 3.6: Splitting sparse and dense dimensions (a) block pointer array structure (b) binary tree index structure

to store the sparse dimensions two methods have been elaborated as illustrated in Figure 3.6. Either each combination of the sparse dimensions is stored, each valid combination having a pointer to its data in the dense array or a binary tree like index structure is maintained, each node of which has pointers to the corresponding dense block. We will consider the first scheme to store sparse dimensions which stores all possible combinations. Without loss of generality, consider the first j dimensions $d_i, 0 \leq i < j$ to be dense. The storage required for this is $c\prod_{i=0}^{j-1} n_i$. The remaining $k - j$ dimensions are sparse and are stored in an index structure.

One method stores pointers from all the possible combinations of the sparse dimensions to data blocks of the dense dimensions. Storage for a pointer is required for all combinations, but only those combinations that point to data are actually useful. Storage for the sparse dimensions when all combinations are stored is $\prod_{i=j}^{k-1} n_i$. Let D^* is the density of the chosen sparse dimensions. Then $D^* \prod_{i=j}^{k-1} n_i$ have pointers to dense data. The storage requirement for this structure is $(D^* \prod_{i=j}^{k-1} n_i) * c\prod_{i=0}^{j-1} n_i$ $= (cD^* \prod_{i=0}^{k-1} n_i)$.

The second method stores the sparse dimensions in a binary tree. Each node of the tree points to a dense data block. This has the advantage of having entries for those combinations that have data associated with them. However, now a pointer for the section number, left child, right child and the data block have to be stored. Additionally, a search over the binary tree has to be made to access the right data block. Periodic balancing of the binary tree might be required during insertion of data. The storage requirements for it are $(D^* \prod_{i=j}^{k-1} n_i) * (4 + c \prod_{i=0}^{j-1} n_i)$.

We will consider method 1 in our analysis of this structure. It is not always easy to split dimensions into sparse and dense dimensions. Some statistical techniques may have to be employed to get a good split. Also, this structure treats the dimensions asymmetrically. It is easier to find elements if the sparse dimensions are specified and the dense dimensions are left unspecified. This is true because an index structure is present for the sparse dimensions. However, a search made with a few dense dimensions specified and the sparse ones left unspecified, will be expensive.

Table 3.2 summarizes the storage requirements for each sparse data structure. A comparison of the storage requirements shows that the Offset-Value pair and BESS are the best in terms of storage requirements for higher values of k. The growth rate of storage for other structures is much higher for an increasing k.

Table 3.2: Storage required for the various sparse storage formats

Method	*Storage*
Multi-Dimensional Array	c $\prod_{i=0}^{k-1} n_i$
Compressed Array	$\alpha Dc \prod_{i=0}^{k-1} n_i, 0 < \alpha < 1$
Bit-Encoded Sparse Storage	$(\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil) * (c + \lceil \frac{\sum_{i=0}^{k-1} \lceil \log h_i \rceil}{wordsize} \rceil) N_{chunk}$
Index-Value pair	$(\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil) * (c + k) N_{chunk}$
Offset-Value pair	$(\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil) * (c + 1) N_{chunk}$
Compressed Sparse Dimensions (extension of CSR)	$(c + 1) N_{total} + \sum_{j=1}^{k-1} \prod_{i=j}^{k-1} n_i$
Sparse-Dense Split Storage	$(D^* c \prod_{i=0}^{k-1} n_i)$

3.3 Data Access Performance

Values of indices in the various dimensions are implicit in a multi-dimensional array but are not so in a sparse representation of data. Sparse data schemes need to be evaluated for their efficiency to perform these data access operations. The various cost parameters required for such an analysis are summarized in Table 3.3. The cost of the operations are from a 132 MHz, 64MB main memory, HP B132L and a 120 MHz, 96MB main memory, IBM RS6000/E30 workstation. We used the native C compiler on each machine without any loop optimizations to run a million iterations of each operation and factored out the loop cost. This gives us an idea on the relative performance of the operations and where we can look for optimizations in terms of operations.

Table 3.3: Typical costs of operations on a workstation

Cost	Operation	HP B132L	RS6000/E30
t_{mem}	Memory access	101 ns	123 ns
t_{add}	Addition (integer)	45 ns	8 ns
t_{comp}	Comparison	23 ns	8 ns
t_{mul}	Multiply (integer)	129 ns	16 ns
t_{div}	Division (integer)	648 ns	166 ns
t_{mod}	Modulo	701 ns	111 ns
t_{shift}	Shift	30 ns	16 ns
t_{and}	And	45 ns	16 ns
t_{or}	Or	45 ns	16 ns
t_{op}	Aggregate Operation	–	–

3.4 Operations

In the following sections each basic operation is analyzed with respect to BESS and sparse data structures discussed previously.

3.4.1 Retrieval of a random cell element

A random element can be retrieved if we have its index. For k-dimensional data, this is a specification of the type $(d_0 = val_0, d_1 = val_1, \ldots, d_{k-1} = val_{k-1})$. Table 3.4 presents the costs of retrieving an element for each type of structure.

For a multi-dimensional array, an offset is created from the start of the array by using each val_i and n_i, $0 \leq i < k$ to get the correct offset in each dimension. $k-1$ multiply and add operations are needed.

Table 3.4: Costs for retrieval of a random element

Structure	Cost
Multi-dimensional array	$(k-1)(t_{mul} + t_{add})$
BESS	$\sum_{i=0}^{k-1} O(\log\lceil\frac{n_i}{h_i}\rceil) + \frac{1}{2}[t_{mem} + k(t_{shift} + t_{and} + t_{comp})]N_{chunk}$
Index-Value	$\sum_{i=0}^{k-1} O(\log\lceil\frac{n_i}{h_i}\rceil) + \frac{1}{2}[k(t_{mem} + t_{comp})]N_{chunk}$
Offset-Value	$\sum_{i=0}^{k-1} O(\log\lceil\frac{n_i}{h_i}\rceil) + \frac{1}{2}[t_{mem} + (k-1)(t_{div} + t_{mul} + t_{sub} + t_{comp})]N_{chunk}$
Compressed Sparse Dimension	$k(t_{mul} + t_{mem}) + t_{comp} \log n_0$
Sparse-Dense Split storage	$(k-1)(t_{mul} + t_{add})$

For BESS, the appropriate chunk has to be found in each dimension. This can be obtained by k binary searches, one for each dimension. For n sorted elements, a binary search takes $O(\log n)$ time. In the desired chunk, the dimension index values are decoded and compared with the needed val_is. This requires k bit and operations, k shift operations and comparisons for each element. On an average half the elements in the sparse chunk are searched.

For the Index-Value pair, the cost for searching the chunk is the same as BESS since both use the chunk index structure. However, within the sparse chunk, dimension index values are available and can be directly compared with val_i. Hence, k fetch and comparison operations are needed for each element.

The Offset-Value pair stores an offset in the chunk. To get the values of each dimension index $k-1$ division, multiply and subtract operations are required for each element. As can be seen from Table 3.3, this can be an expensive operation and has to be done for each entry in the chunk.

The index values in each dimension are already known in the compressed sparse dimension. Starting at the index of the outermost dimension, the indices are followed to the innermost dimension. Here a search for the index in the innermost dimension has to be made. The split storage scheme behaves like the multi-dimensional array for this access. The correct pointer to the dense block in the sparse dimensions index can be found by add and multiply operations on the dimension values.

Comparing costs in Table 3.4, retrieval of a random element is the fastest for multi-dimensional array and the split storage schemes. This is because elements are stored at regular strides and it is easy to index into the desired element. Compressed sparse dimension also has pointers set for all dimensions and following the appropriate k level pointers a random element can be retrieved. For the other three schemes, the decoding of the dimension index values is not needed for the Index-Value pairs, BESS provides with fast decoding of the dimension index values and Offset-Value pair is the slowest.

3.4.2 Retrieval along a dimension

Let $j, 0 \leq j < k$ be the dimension specified. Retrieval of values will restrict dimension j with a value val_j. This will return a $k-1$ dimensional hypercube if the operation is performed on a k dimensional hypercube. Table 3.5 shows the costs for this operation. For a multi-dimensional array, a binary search over the sorted list of the unique values in the dimension j is needed. This takes $O(\log n_j)$ time. Using this base address, the elements are retrieved using a multiply and an add over the remaining $k-1$ dimensions.

When using BESS, the chunk containing val_j is found by searching the sorted chunk list for dimension j using binary search. A chunk's base offset is converted into a dimensional index for each dimension. This is added to the dimension index within the chunk which is retrieved from BESS by using k bit and operations and k bit shift operations. This is done for each chunk in the $k-1$ dimensional plane. A similar analysis holds for the Index-Value pair, except that the dimension index values are already available. For the Offset-Value pair, the dimension index have to be calculated from the offset. This takes $k-1$ division, multiply and subtract operations for each element. These are computationally expensive operations when compared to the bit shift used by BESS.

The compressed sparse dimension stores pointers in a dimension ordering from the outermost to the innermost. These can be used to retrieve the data and requires traversing k levels of pointers. There are two cases. First, when j is not the innermost dimension. In this case a value for an intermediate dimension is known and only pointers from that value need to be followed. n_{avg} is the average number of elements found in the innermost dimension. In the second case, j is the innermost dimension. Each element found in the innermost dimension has to be compared with the value specified in this case.

The split storage structure can either have j belong to the sparse dimensions or in dense dimensions. In the former case, it is easy to calculate which index elements will belong to the set of possible block pointers that need to be followed to the dense blocks. When j belongs to the dense dimensions, each index element in the dense dimension is followed and the restriction is made for each dense block. Addition and multiply are the operations to do the pointer math. Again, D^* is the density in the sparse dimensions.

From the cost expressions in Table 3.5, multi-dimensional array has the best performance. However, the cost for accesses will increase as the strides of accesses increase, that is as the dimension order to be retrieved increases from innermost to outermost. Compressed sparse dimension is efficient when j is not innermost and so is the split storage scheme. For the other three schemes which operate on chunks, Index-Value has the least overhead of processing within a chunk, followed by BESS and then Offset-Value pair. For low values of N_{chunk}, the operation count is much less for the sparse data structures and hence might provide better retrieval performance than the multi-dimensional array.

Table 3.5: Costs for retrieval along a dimension $j, 0 \leq j < k$

Structure	Cost
Multi-dimensional array	$(t_{mem} + t_{comp}) \log n_j + (t_{mem} + t_{mul} + t_{add}) \prod_{i=0,i\neq j}^{k-1} n_i$
BESS	$(t_{mem} + t_{comp}) \log\lceil \frac{n_j}{h_j} \rceil + (\prod_{i=0,i\neq j}^{k-1} \lceil \frac{n_i}{h_i} \rceil)[k(t_{div} + t_{mul} + t_{sub})$ $+(t_{mem} + t_{shift} + t_{and} + t_{comp}))N_{chunk}]$
Index-Value	$(t_{mem} + t_{comp}) \log\lceil \frac{n_j}{h_j} \rceil +$ $(\prod_{i=0,i\neq j}^{k-1} \lceil \frac{n_i}{h_i} \rceil)[(t_{mem} + t_{comp})N_{chunk}]$
Offset-Value	$(t_{mem} + t_{comp}) \log\lceil \frac{n_j}{h_j} \rceil +$ $(\prod_{i=0,i\neq j}^{k-1} \lceil \frac{n_i}{h_i} \rceil)[(t_{mem} + t_{comp} + 2(t_{div} + t_{mul} + t_{sub}))N_{chunk}]$
Compressed Sparse Dimension	
(j is not innermost)	$t_{mem} n_{avg} \prod_{i=1,i\neq j}^{k-1} n_i$
(j is innermost)	$(t_{mem} + t_{comp}) n_{avg} \prod_{i=1}^{k-1} n_i$
Sparse-Dense ($j \in s$)	$(D^* \prod_{i=0,i\neq j}^{s-1} n_i) * (t_{mem} + t_{add} + t_{mul}) \prod_{i=s}^{k-1} n_i$
split storage ($j \notin s$)	$(D^* \prod_{i=0}^{s-1} n_i)(t_{mem} + t_{add} + t_{mul}) \prod_{i=s,i\neq j}^{k-1} n_i$

3.4.3 Retrieval for values of dimensions within a range (Range Queries)

Range queries on multi-dimensional arrays require retrieving values between the upper limit u_i and the lower limit l_i of the dimensions in which the range is specified. Let r include the dimensions in which the range is specified. Clearly, $|r| \leq k$. In the other structures, the operation is not so straightforward. In BESS, the chunk containing u_i and the chunk containing l_i in all specified range dimensions are identified from the chunk index structure. Values are then retrieved from the chunks within this range and for all chunks in the unspecified dimensions. Checks for ranges are needed for elements in the boundary chunks only. A similar analysis hold true for the Index-Value and the Offset-Value structures. The compressed sparse dimensions will use the pointers within the range specified for the dimensions. For the split storage structure, if the range is specified only in the sparse dimensions then the index structure is restricted to follow pointers only in the range selected and elements from the dense block are retrieved for all the valid pointers in the

Table 3.6: Costs for range query between l_i and u_i, $i \in r$.

Structure	Cost
Multi-dimensional array	$(t_{mem} + t_{add}) \prod_{i \in r}(u_i - l_i) * \prod_{i \in k-r} n_i$
BESS	$2(t_{mem} + t_{comp}) \sum_{i \in r} \log\lceil \frac{n_i}{h_i} \rceil +$ $\prod_{i \in r}(\lceil \frac{u_i - l_i}{h_i} \rceil \prod_{i \notin r} \lceil \frac{n_i}{h_i} \rceil * [k(t_{div} + t_{mul} + t_{sub}) +$ $(2t_{mem} + k(t_{shift} + t_{and} + t_{comp}))N_{chunk}]$
Index-Value	$2(t_{mem} + t_{comp}) \sum_{i \in r} \log\lceil \frac{n_i}{h_i} \rceil +$ $\prod_{i \in r} \lceil \frac{u_i - l_i}{h_i} \rceil \prod_{i \notin r} \lceil \frac{n_i}{h_i} \rceil * [k(t_{div} + t_{mul} + t_{sub}) + k(t_{mem} + t_{comp})N_{chunk}]$
Offset-Value	$2(t_{mem} + t_{comp}) \sum_{i \in r} \log\lceil \frac{n_i}{h_i} \rceil +$ $\prod_{i \in r} \lceil \frac{u_i - l_i}{h_i} \rceil \prod_{i \notin r} \lceil \frac{n_i}{h_i} \rceil * [k(t_{div} + t_{mul} + t_{sub}))(N_{chunk} + 1)]$
Compressed Sparse Dimensions	$\prod_{j=1, j \in r}^{k-1}(u_i - l_i) \prod_{j=1, j \notin r}^{k-1} n_i * (t_{mem} + t_{comp})n_{avg}$
Sparse-Dense Split storage (range in sparse dimensions)	$D^* \prod_{i \in s}(u_i - l_i)[t_{comp} + \prod_{i \in k-s} n_i * (t_{mem} + t_{add})]$
(range in dense dimensions)	$D^* \prod_{i \in s} n_i[t_{comp} + \prod_{i \in k-s}(u_i - l_i) * (t_{mem} + t_{add})]$

range. Otherwise, if the range is specified in the dense dimension then all the valid pointers to the dense blocks are used and the values are retrieved within the range from the dense blocks. s is the set of sparse dimensions and the set $k - s$ contains the dense dimensions.

Comparison of cost expression from Table 3.6 show us that the range query is the most efficient on the multi-dimensional array. Range search for split-storage searches for the valid pointers in the sparse structure and then restricts the dense blocks on the ranges specified in the dense dimension. For operations within chunks, the Index-Value pair is better than BESS which is better than the Offset-Value pair storage. However, for low densities, the sparse structure can perform better than the multi-dimensional array.

3.4.4 Aggregation of a dimension

Each element is accessed and aggregated to the resultant structure with the same dimension value in the appropriate dimension. For a multi-dimensional array, the resultant structure is an array without the dimension which is to be aggregated. The result is initialized with zeros and each element is added to the result in the appropriate dimension. For a k dimensional cube aggregation of a single dimension results in a $k-1$ dimensional cube. Table 3.7 compares the cost of aggregating a dimension.

For the BESS, operations are performed for each chunk. First the chunk dimension index values are calculated from its offset. For each element in the chunk, the appropriate dimension is retrieved from the encoded structure by bit shifting. This is aggregated to the result chunk which is also stored in BESS format. The dimension which is being aggregated is removed from the bit representation by appropriately shifting the remaining bits. The resultant bit representation has $k-1$ dimensions and values prior to the aggregation. The list of values in the chunk is then sorted on this bit representation. This gets the values that need to be aggregated adjacent to each other. A traversal of the list is then made, aggregating adjacent values which have the same bit representation. This takes $O((t_{shift}+t_{and}+t_{op}+\log N_{chunk})N_{chunk})$. Index-Value pair has the dimension index readily available which is directly used. For the Offset-Value pairs, k mod and subtract operations gets the correct index. A strategy similar to the one described for BESS is then used for these.

For storage in compressed sparse dimensions, the index structure is traversed from the outermost to innermost dimension. For the dimension to be aggregated, its index is traversed in a strided manner and values retrieved from following the pointers are aggregated if they match on the innermost dimension values.

In sparse-dense split storage if a sparse dimension is to be aggregated, then the aggregate operator is applied to the corresponding dense block pointers along the dimension to be aggregated. For aggregating a dense dimension, the aggregation operator is applied to the dense blocks and the sparse index is unaffected.

Table 3.7: Costs of aggregating a dimension j

Structure	Cost
Multi-dimensional array	$(t_{mem} + t_{add} + t_{op}) \prod_{i=0}^{k-1} n_i$
BESS	$\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil * [k(t_{div} + t_{mul} + t_{sub}) +$ $(t_{mem} + t_{shift} + t_{and} + t_{op} + \log N_{chunk}) N_{chunk}]$
Index-Value	$\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil * [k(t_{div} + t_{mul} + t_{sub}) + (kt_{mem} + t_{op} + \log N_{chunk}) N_{chunk}]$
Offset-Value	$\prod_{i=0}^{k-1} \lceil \frac{n_i}{h_i} \rceil * [k(t_{mem} + t_{div} + t_{mul} + t_{sub}) + t_{op} + \log N_{chunk}] N_{chunk}$
Compressed Sparse Dimensions	$\sum_{i=0}^{k-1} \prod_{l=i}^{k-1} n_l * (t_{mem} + t_{comp} + t_{op}) n_{avg}$
Sparse-Dense Split storage	$(D^* \prod_{i \in s} n_i)[t_{comp} + \prod_{i \in k-s} n_i * (t_{mem} + t_{add} + t_{op})]$

3.5 Choosing a sparse structure

From the storage perspective, the multi-dimensional array is the most inefficient for sparse data. Index-Value pair and Compressed sparse dimension structures are next, since they store index values for each dimension. For large data sets and high number of dimensions these structures are not efficient. However, these structures provide good access performance as can be seen from the analysis in the previous section.

BESS, Offset-value and Split storage structures give good storage performance for sparse data. The restriction with the split storage, however is that dimensions should be partitioned into groups of sparse and dense dimensions which might not always be possible or be inefficient for a large number of dimensions. Among these, performing dimension oriented operations discussed in earlier sections are the most expensive on Offset-Value structure. BESS provides good performance for such operations. The split storage scheme is asymmetric and provides good performance only when the operations restrict the upper level sparse structure. When comparing BESS and Offset-Value pair, BESS stores the bit encoded dimension indices in $\sum_{i=0}^{k-1} \lceil \log n_i \rceil$ bits. Offset-Value pair stores the offset from the start of the chunk in memory in $\log \prod_{i=0}^{k-1} n_i$ bits. The difference is at-most k bits in the representation and is much lower on an average. For operations requiring dimension index values, BESS uses k bit shift operations and k bit and operations for k dimensions and Offset-Value uses $k-1$ division, multiply and subtract operations. The cost of a bit shifting operation is an order of magnitude less than the cost of a division operation as can be seen from Table 3.3.

As has been seen in the previous sections, OLAP workloads need a sparse storage mechanism with the following criteria. First, dimensions should be treated symmetrically. Secondly, it should optimize on storage requirements and finally it should give good accesses for data retrieved along dimensions for multi-dimensional analysis. The last two requirements are conflicting and a good trade-off between the two is desired. We believe, the bit-encoded sparse structure (BESS) provides a good balance between the two. In the next section, we provide an experimental comparison of BESS with multi-dimensional arrays and the Offset-Value pair storage.

3.6 Performance Evaluation

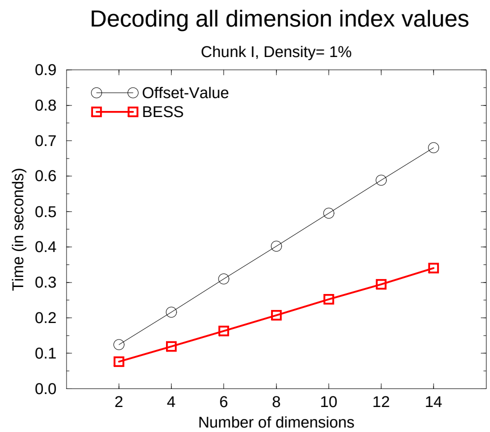
We consider chunked data storage and perform data operations on a chunk. Data in a chunk can either be stored a multi-dimensional array, Offset-Value pairs or using BESS. Performing operations on a data set will involve performing operations on a collection of chunks and will have the same comparative performance as that for a chunk. Results presented in this section are on uniformly distributed random data. The number of dimensions are varied while keeping the chunk size constant. The first data set, has the product of all the chunk dimensions as 10 million. Each cell in a multi-dimensional array contains a 4 byte integer, resulting in about a 38 MB dense chunk (Chunk I). Similarly, another data set fixes the product of all dimensions as 20 million to make a 76MB dense chunk (Chunk II). Queries are run by using varying densities of a chunk. For BESS and Offset-Value storage chunk densities of 1%, 5% and 10% are used. Offset-Value pair stores 8 bytes, a 4 byte measure value and a 4 byte offset, in the data sets chosen. BESS also contains 8 bytes, a 4 byte measure value and a 4 byte bit encoding. This makes the actual chunk sizes to be 0.76MB, 3.8MB and 7.6MB for the first chunk and 1.52MB, 7.6MB and 15.2MB for the second chunk. The number of dimensions, k, is varied from 2 onwards for each query. The dimension sizes, i.e the number of unique values in each dimension, vary accordingly. Again, k is the number of dimensions and D is the density of the chunk. In the following sections performance results are presented for the various queries.

3.6.1 Decoding dimension index values

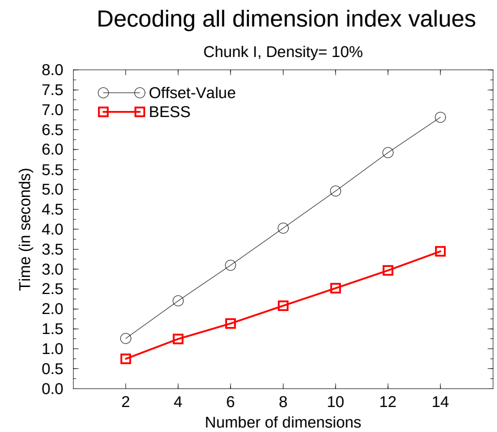
Figure 3.7 presents results for decoding all k dimensions from the value stored as a cell index for the first chunk on the IBM machine. k is varied from 2 to 14 in steps of 2. For example, when $k = 8$, a 8-dimensional cube is considered for dimension decoding. Offset-Value decodes an offset into the individual dimension index values, whereas BESS decodes the bits stored as the index. Figure 3.8 shows the performance for the second chunk on the IBM machine. Here, k is varied between 2 and 20 in steps of 2. BESS performs much better than Offset-Value storage for all dimensions and the difference gets more pronounced as we increase the number of dimensions. Similar results are observed on the HP machine. Figure 3.9 shows the performance using the first chunk on the HP machine.

3.6.2 Retrieval along a dimension and Range Queries

Figure 3.10 shows the performance of retrieving values along dimensions 3, 9 and 14 in chunk I with 14 dimensions. As can be observed from the figures, the retrieval of the outer dimensions is more costly than the inner dimensions. In multi-dimensional array this is due to the increase in strides of access and the resulting affect on the cache usage. For BESS and Offset-Value pairs, the processing performed to retrieve the dimension index increases as the dimension to be retrieved increases. This can also be observed from the analysis in the previous section, as k increases. The densities considered are 1%, 5% and 10%. The performance of the multi-dimensional array is not affected by the density as expected. The costs of BESS and Offset-Value pair storage increase as density increases since the number of elements to

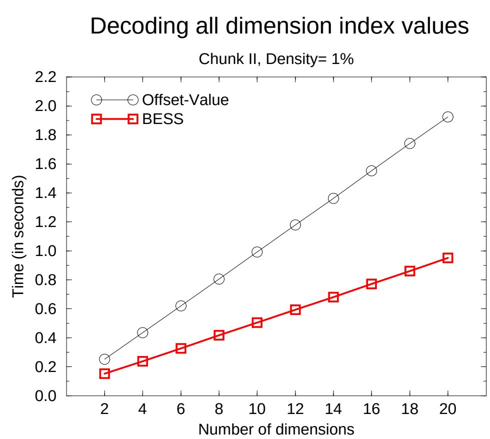


(a)

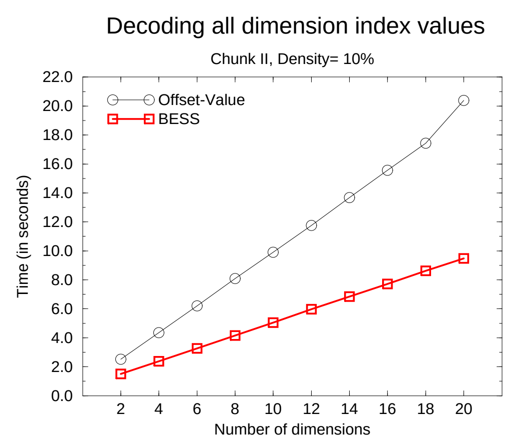


(b)

Figure 3.7: Decoding all the dimension index values for chunk I (run on IBM) (a) Chunk density = 1% (Data size = 0.76MB)(b) Chunk density = 10% (Data size = 7.6MB)



(a)



(b)

Figure 3.8: Decoding all the dimension index values for chunk II (run on IBM) (a) Chunk density = 1% (Data size = 1.52MB)(b) Chunk density = 10%(Data size = 15.2MB)

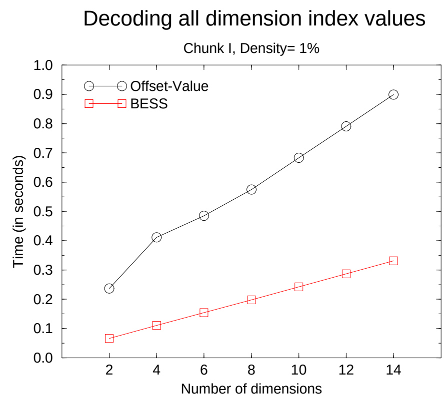
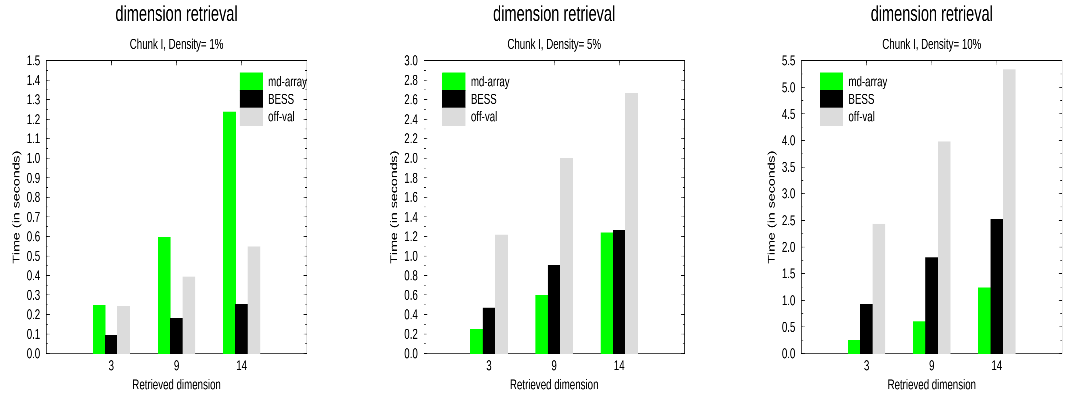


Figure 3.9: Decoding all the dimension index values for chunk I with density 1% (Data size = 0.76MB) (run on HP)

operate on within the chunk increase. It can be observed that BESS provides the best performance for density of 1% for retrieval of any dimension. For 5% density, the retrieval of the outermost dimension performs as well as the multi-dimensional array. Otherwise, for 5% and 10% densities, multi-dimensional array gives the best performance. However, BESS performs much better than Offset-Value storage.

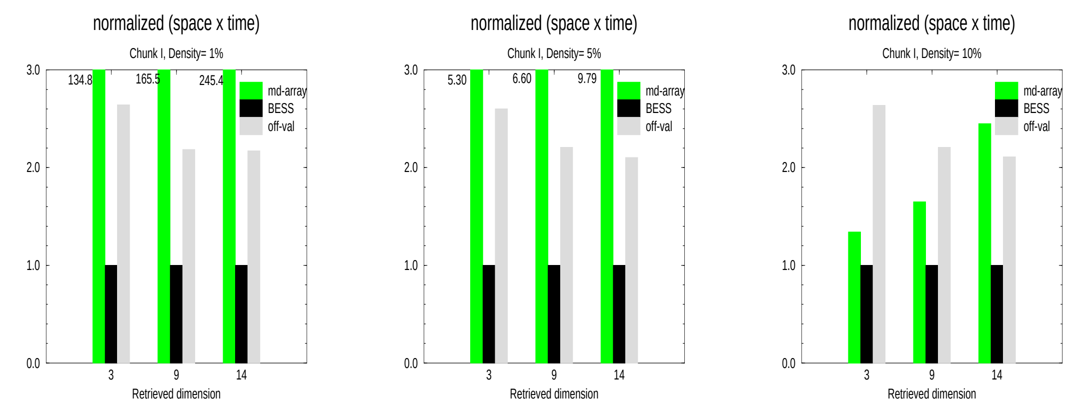
Next, we compare the costs of the three structures in terms of both time and space. A *time-space* metric is calculated as the product of time taken for the operation and the space requirements of the structure. Figure 3.11 shows this metric for dimension retrieval on a 14 dimensional chunk of size 40MB. This is normalized with respect to BESS. Again, chunk density of 1%, 5% and 10% are considered. It can be observed that BESS performs much better for the time-space metric than the other two structures in all dimensions.

Range queries retrieve data by restricting one or more dimensions. Range queries can be thought of as an extension of retrieval along one dimension. The operations performed for retrieval along a dimension are used for range queries for all three structures. Dimension restrictions can be applied easily to multi-dimensional arrays. For BESS and Offset-Value storage, the dimensions along which the range is specified are retrieved from the encoding and checked if they fall within the range. Performance of dimension decoding and dimension retrieval have been seen in this



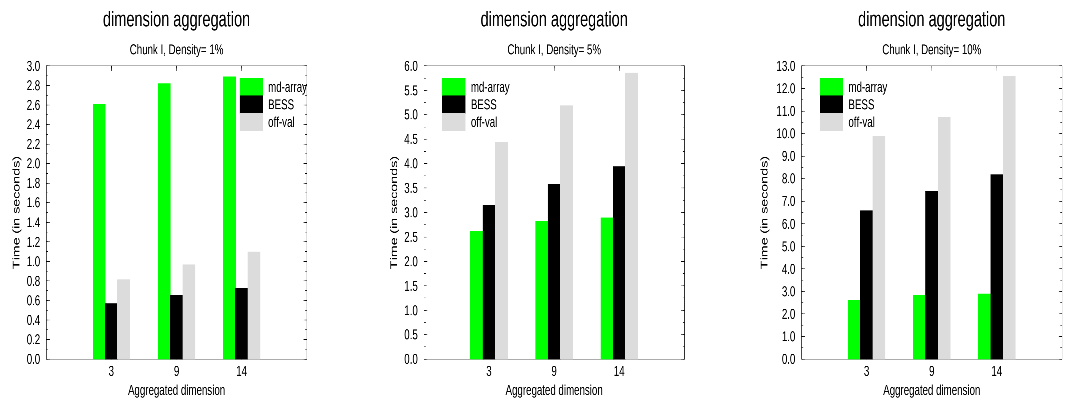
(a) (b) (c)

Figure 3.10: Time for dimension retrieval for a chunk I with 14 dimensions (run on HP) (a) density = 1% (Data size = 0.76MB) (b) density = 5% (Data size = 3.8MB) (c) density = 10% (Data size = 7.6MB)



(a) (b) (c)

Figure 3.11: Normalized (space $\times$ time) for dimension retrieval with respect to BESS for chunk I with 14 dimensions (run on HP) (a) density = 1% (Data size = 0.76MB) (b) density = 5% (Data size = 3.8MB) (c) density = 10% (Data size = 7.6MB)



(a) (b) (c)

Figure 3.12: Time for dimension aggregation for chunk I with 14 dimensions (run on HP) (a) density = 1% (b) density = 5% (c) density = 10 %

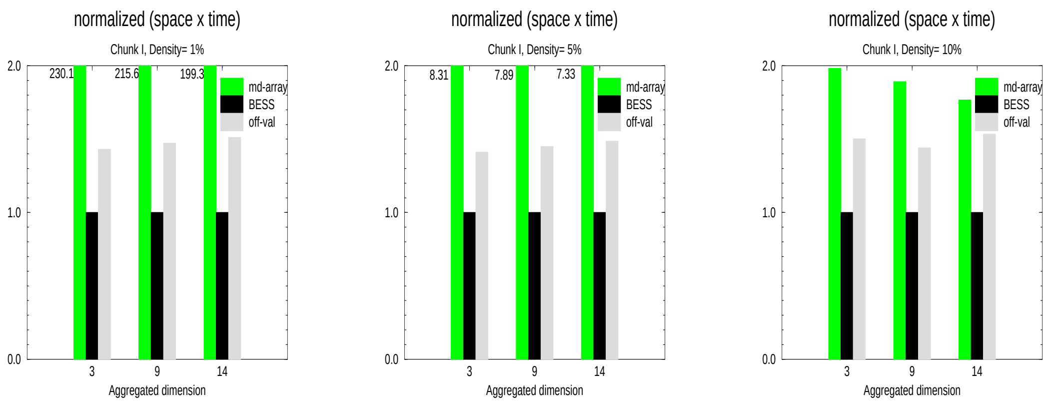
section. We expect similar results for these structures for range queries.

3.6.3 Aggregation along a combination of dimensions

Figure 3.12 shows the performance for aggregation along dimensions 3, 9 and 14 for chunk I with 14 dimensions. Densities of 1%, 5% and 10% are considered. It can be observed from the figures that BESS performs much better than multi-dimensional array and Offset-Value storage for 1% density. For 5% and 10% density the multi-dimensional array has a better performance. The number of elements increase with these densities and the aggregation operation contains a $O(n \log n)$ sort operation for both BESS and Offset-Value storage. BESS is better than Offset-Value for this aggregation operation.

However, if we compare the space-time metric in Figure 3.13, it can be observed that BESS performs much better than the other two structures.

Multi-dimensional aggregation queries can be thought of as an extension of aggregation along one dimension. The operations used for aggregation for all three structures are the same between the two. Aggregation can be done on one or more dimensions. We expect similar results for multi-dimensional aggregation as seen for



(a) (b) (c)

Figure 3.13: Normalized (space $\times$ time) for dimension aggregation with respect to BESS for chunk I (run on HP)(a) density = 1% (Data size = 0.76MB) (b) density = 5% (Data size = 3.8MB) (c) density = 10% (Data size = 7.6MB)

aggregation done along a dimension.

3.7 Summary

In this chapter we introduced a bit-encoded sparse structure for storing multi-dimensional sparse data. A comparison with other sparse data structures was presented in terms of storage efficiencies and data access times for the typical accesses required for OLAP and data mining workloads. The gain from reduced storage requirements is usually offset by overhead required in data accesses. We presented an analysis of some typical OLAP operations using sparse storage schemes for higher dimensional data. This helped in highlighting the benefits of BESS. Performance results show better performance for OLAP queries using BESS over other storage schemes. We evaluated BESS with a space $\times$ time metric and find that its performance is the best overall.

Furthermore, analysis and performance results presented in this chapter are in terms of design parameters : number of dimensions, dimension sizes, sparsity of data, cell size, which can serve to compare and contrast the storage versus data access performance trade-offs in choosing a suitable multi-dimensional structure for a given application.

Chapter 4

Schedule Generation for Parallel Computation

A schedule is required for calculating and maintaining aggregate calculation to answer queries and to perform data mining tasks. There are several alternatives for aggregate calculations and a schedule is required to minimize data input sizes, number of operations and inter-processor communication costs. We would like to handle large data sets, and since main memory on processors of the parallel machine is not sufficient to handle such large data sizes, we have to generate a schedule of operations that takes disk I/O costs into account.

In this chapter we discuss various cost parameters that need to be considered in parallel executions of the OLAP and data mining operations and optimizations that result in a better schedule of execution.

4.1 Model of Parallel Processing

Distributed Memory Parallel Computers (shared-nothing machines) consist of a set of processors (tens to a few hundred) connected through an interconnection network. The memory is physically distributed across the processors. Interaction between processors is through message passing. Popular interconnection topologies are 2D meshes (Paragon, Delta), 3D meshes (Cray T3D), hypercubes (nCUBE), fat tree (CM5) and multistage networks (IBM-SP2).

Further, disks can either be attached to a single processor or attached on a network and accessible from any processor. In the first case, if processor P_i needs data from a disk attached to processor P_j, then communication is required between P_i and P_j. In the second case, P_i can read over the network without any intervention of P_j. On the IBM-SP2 configuration we are currently using, we observe that the data access performance in the first case is much better in comparison to the second case. This is due to the fact that in our set up, the fast communication switch is used for message passing whereas ethernet is used for network access. Figure 4.1 shows a schematic of distributed memory machine organization.

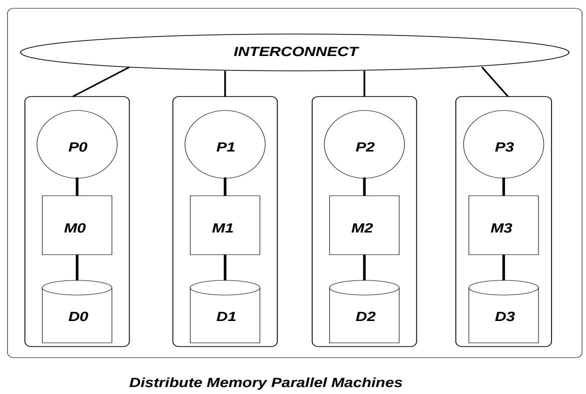


Figure 4.1: Distributed memory parallel machines

4.1.1 Communication Costs

Most of these machines have cut-through routed networks which will be used for modeling the communication cost of our algorithms. For a lightly loaded network, a message of size m traversing d hops of a cut-through (CT) routed network incurs a communication delay given by $T_{comm} = t_s + t_h d + t_w m$, where t_s represents the handshaking costs, t_h represents the signal propagation and switching delays and t_w represents the inverse bandwidth of the communication network. The startup time t_s is often large, and can be several hundred machine cycles or more. The per-word transfer time t_w is determined by the link bandwidth. t_w is often higher (an order

Table 4.1: Running times of various parallel primitives on a multi-staged network

Primitive	Running time on p processors
Broadcast	$O((t_s + t_w m) \log p)$
Reduce	$O((t_s + t_w m) \log p)$
Parallel Prefix	$O((t_s + t_w) \log p)$
Gather	$O(t_s \log p + t_w mp)$
All-to-All Communication	$O((t_s + t_w m)p + t_h p \log p)$

to two orders of magnitude is typical) than t_c, the time to do a unit computation on data available in the cache. On the IBM SP2 with 120 MHz thin nodes that we have used for our implementation using the message passing interface (MPI), the latency is 29.3 milliseconds and the bandwidth is 102 MB/sec (uni-directional) and 113 MB/sec (bi-directional) using the SP switch [May98]. The uni-directional bandwidth is the *point-to-point* bandwidth and the bi-directional bandwidth is the exchange bandwidth, measured by sending and receiving messages between processors. This translates to $t_s = 29.3$ milliseconds and $t_w = 8.85$ nanoseconds.

4.1.2 Primitive Operations

Parallelization of applications requires distributing some or all of the data structures among the processors. Each processor needs to access all the non-local data required for its local computation. This generates aggregate or collective communication structures.

Several algorithms have been described in the literature for these primitives and are part of standard textbooks [KGGK94]. The use of collective communication provides a level of architecture independence in the algorithm design. It also allows for precise analysis of an algorithm by replacing the cost of the primitive for the targeted architecture. Table 4.1 provides a complexity analysis of these operations on a multi-staged network. These costs are used in the analysis of the algorithms presented in the next section.

1. **Broadcast.** In a Broadcast operation, one processor has a message of size m

to be broadcast to all other processors.

2. **Reduce.** Given a vector of size m on each processor and a binary associative operation, the Reduce operation computes a resultant vector of size m and stores it on every processor. The i^{th} element of the resultant vector is the result of combining the i^{th} element of the vectors stored on all the processors using the binary associative operation.

3. **Parallel Prefix.** Suppose that $x_0, x_1, \ldots, x_{p-1}$ are p data elements with processor P_i containing x_i. Let $\otimes$ be a binary associative operation. The Parallel Prefix operation stores the value of $x_0 \otimes x_1 \otimes \ldots \otimes x_i$ on processor P_i.

4. **Gather.** Given a vector of size m on each processor, the Gather operation collects all the data and stores the resulting vector of size mp in one of the processors.

5. **All-to-All Communication.** In this operation each processor sends a distinct message of size m to every processor.

4.2 Data Partitioning

Data is distributed on processors to distribute work equitably. In addition, a partitioning scheme for multidimensional has to be *dimension-aware* and for dimension-oriented operations have some regularity in the distribution. A dimension, or a combination of dimensions can be distributed. In order to achieve sufficient parallelism, it would be required that the product of cardinalities of the distributed dimensions be much larger than the number of processors. For example, for 5 dimensional data ($ABCDE$), a 1D distribution will partition A and a 2D distribution will partition AB. We assume that dimensions are available that have cardinalities much greater than the number of processors in both cases. That is, either $|A_i| \gg p$ for some i, or $|A_i||A_j| \gg p$ for some i, j, $1 \leq i, j \leq d$, and d is the number of dimensions. We will use these partitioning in order to study the communication requirements for data movement in the intermediate aggregate calculations in the data cube. Cost parameters for communication in schedule generation will include the data movement involved in aggregate calculation. Figure 4.2 illustrates an example of 1D and 2D partitioning for a 3 dimensional cube on 4 processors.

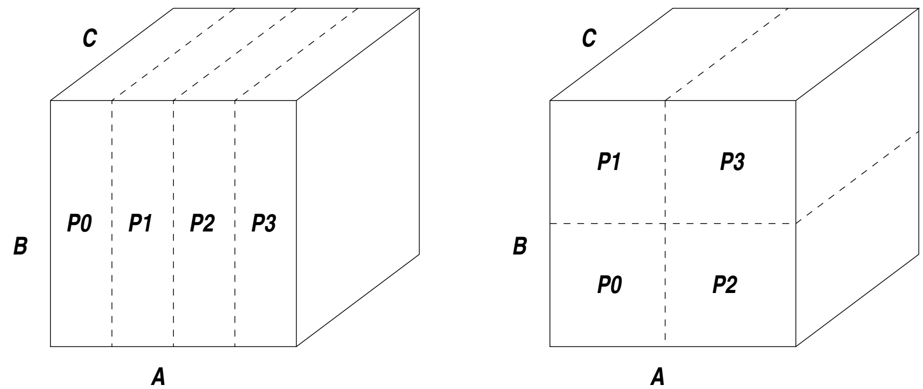


Figure 4.2: 1D and 2D partition for 3 dimensions on 4 processors

4.3 Schedule Generation for Data Cubes

The data cube operator is the n-dimensional generalization of the group-by operator. Consider the following query which uses the cube operator.
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);

$2^N - 1$ aggregate calculations are needed for a N-dimensional data cube. For example, $2^3 = 8$ group-bys are calculated for the above query: {Model, Year, Color}, {Model, Year}, {Model, Color}, {Year, Color}, {Model}, {Year}, {Color} and ALL, as shown in Figure 2.5.

4.3.1 Optimizations

Several optimizations can be performed over the naive method of calculating each aggregate separately from the initial data [GBLP96]. Optimizations 1 and 2 are normally considered for a uniprocessor model. Optimization 3 is an added and important consideration for a parallel implementation to reduce the overheads from communication costs.

1. **Smallest Parent:** For computing a group-by this selects 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 actual choice will be made by picking the smaller of ABC and ABD.

2. **Effective use of Cache:** This aims at using the cache effectively by computing the group-bys in such an order that the next group-by calculation can benefit from the cached results of the previous calculation. This can be extended to disk based data cubes by reducing disk I/O and caching in main memory. For example, after computing ABC from $ABCD$ we compute AB followed by A. This leads to reduction in disk I/O scans. In MOLAP systems the sizes of the intermediate levels are fixed in an array based implementation and the order can be determined easily. However, for sparse representations, estimations have to be made for intermediate sub-cube sizes.

3. **Minimize inter-processor Communication:** Communication is involved among the processors to calculate the aggregates. The order of computation should minimize the communication among the processors because inter-processor communication costs are typically higher than computation costs. For example, $BC \rightarrow C$ will have a higher communication cost to first aggregate along B and then divide C among the processors in comparison to $CD \rightarrow C$ where a local aggregation on each processor along D will be sufficient.

A lattice framework to represent the hierarchy of the group-bys was introduced in [HRU96]. This is an elegant model for representing the dependencies in the calculations and also to model costs of the aggregate calculations. A scheduling algorithm can be applied to this framework substituting the appropriate costs of computation and communication. A lattice for the group-by calculations for a 5-dimensional cube ($ABCD$) is shown in Figure 4.3. Each node represents an aggregate and an arrow represents a possible aggregate calculation which is also used to represent the cost of the calculation.

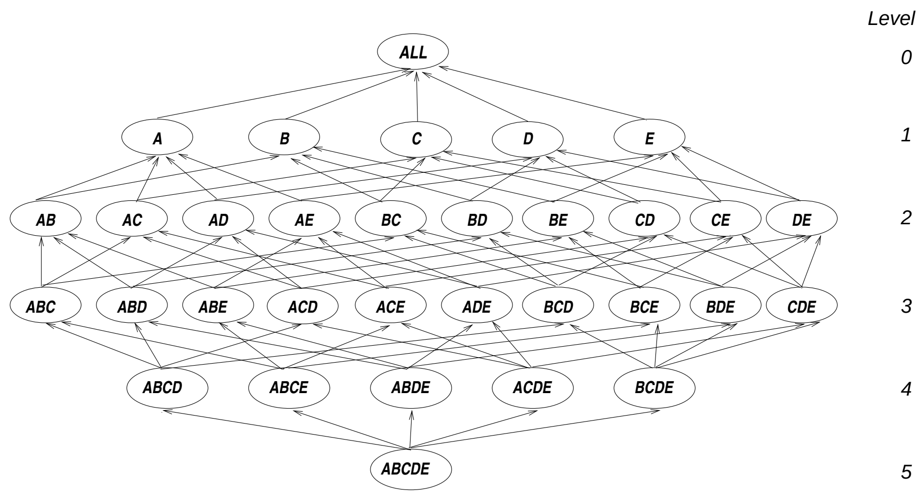


Figure 4.3: Lattice for cube operator

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. This is described later in the section on aggregation.

In the uniprocessor domain, optimizations mentioned above have been used in a sort based algorithm *PipeSort*, a hash based algorithm *PipeHash*, and an *Overlap* algorithm overlaps the calculation of cubes from their parents [AAD$^+$96]. A recursive sparse data cube computation algorithm called *Partitioned-Cube* is presented in [RS97]. These use the relational database model, optimizing the sorted order of tuples to perform aggregations, or using hashing to do the same. An array based algorithm which uses a schedule to optimize memory use and disk scans for I/O is proposed in [ZDN97]. This has been shown to outperform the algorithms and optimizations used in the relational database model.

4.3.2 Size Estimation

A cost model is needed for the schedule to be generated in order to minimize the total time using the optimizations listed above. For a parallel implementation using multidimensional arrays the costs of computation and communication are in terms of the array sizes. A n-dimensional array, with dimension sizes $|d_1|, |d_2|, \ldots |d_n|$, has the size $\prod_{i=1}^{n} d_i$, and this is used to determine the cost of computation and communication operations. The detailed model for arrays is given in Chapter 5, which deals with in-memory construction of datacubes using multidimensional arrays and describes the various operations and algorithms.

However, with chunking and presence of sparse chunks the cube size cannot be taken for calculating computation and communication costs. Size estimation is required for sparse cubes to estimate computation and communication costs when dimension aggregation operations are performed. We use a simple analytical algorithm for size estimation in presence of hierarchies presented in [SDNR96]. If the data is assumed to be uniformly distributed, it is possible to mathematically approximate the number of tuples that will appear in the result of the cube computation using the result from [Fel57]: If r elements are chosen uniformly and at random from a set on n elements, the expected number of distinct elements obtained is $n - n(1 - 1/n)^r$. This result can be applied to find the upper bound on the size of the cube, with one dimension aggregated. The number of unique values is known in each dimension, this is the size of the dimension, $|d_i|$ for dimension d_i. The size of a group by on any subset of attributes can be estimated using the above formula. As an example, consider a 4 dimensional cube, $ABCD$, with dimension sizes d_A, d_B, d_C and d_D. To compute the size of the group by on attributes A, B and C, computing the cube ABC, with the number of elements as $d_A \times d_B \times d_C$. This is n in the above formula. r is the number of elements in the cube $ABCD$. This is similar to what is done in relational group by estimation. This is a simple and fast method for size estimation. Sampling based methods and an algorithm based on probabilistic counting have also been proposed for size estimation [FM85, SDNR96]. A simple sampling based method works by taking a random subset of the database and compute the cube on that subset. Quality of the estimates depend on the size of the sample. The analytical algorithm is shown to perform well for uniformly distributed random data and also works well for some amounts of skew in the data. It has been shown in [SDNR96] that for Zipfian data, with a Zipf factor 0.8 the analytical algorithm estimates within 20% of the actual size. Since we need reasonable estimates to select the materialization of a sub-cube from a sub-cube at a higher level, this works well and we use it due to its simplicity. Detailed costs of operations and algorithms used are presented in Chapter 6, which details the construction of the disk based data cubes.

4.4 Schedule for Attribute-oriented Associations

Attribute Focusing techniques for m-way association rules and meta-rule guided mining for association rules with m predicates in the rule require that all aggregates with m dimensions be materialized. Let us call these the *essential* aggregates. In this case the whole data cube need not be calculated. Since the number of aggregates is a fraction of the total number (2^n), a different schedule is needed for the intermediate, non-essential aggregate calculations, that will help materialize the essential aggregates.

4.4.1 Left and Right Schedules

For this discussion, let us assume that the dimensions are ordered in an non-increasing order of their cardinalities, represented by the ordering of the dimensions in the aggregate. For example, in ABC, $|A| \geq |B| \geq |C|$. To obtain the essential aggregates, we traverse the lattice (Figure 4.3), level by level starting from level 0. For a level k, each entry has to be matched with the an entry from level $k+1$, such that the cost of calculation is the lowest and the minimum number of non-essential aggregates are calculated. We have to determine the lowest number of non-essential intermediate aggregates to be materialized. This can be done in one of two ways. For each entry in level k we choose an arc to the leftmost entry in level $k+1$, which it can be calculated from. This is referred to as the *left schedule*. For example, ABC has arcs to $ABCD$ and $ABCE$, and the left schedule will choose $ABCD$ to calculate ABC. Similarly, a *right schedule* can be calculated by considering the arcs from the right side of the lattice.

Figure 4.4 shows the left and right schedules for a 2D partitioning (AB is partitioned) for 5 dimensions. Shaded entries are the ones that generate communication. The two schedules have different communication requirements. For a 2D partitioning, $ABCD \rightarrow ABC$ will be a *local* calculation, since D is to be aggregated and each processor can do it on the portion of $ABCD$ it owns, resulting in ABC also being partitioned on AB. However, $ABCD \rightarrow ACD$ requires data movement since one of the distributed dimension is involved in the aggregation, namely B. The resulting aggregate, ACD will be distributed on AC. We maintain 2D distribution for all the aggregates with 2 or more dimensions. 1D partitioning is used for one dimensional aggregates.

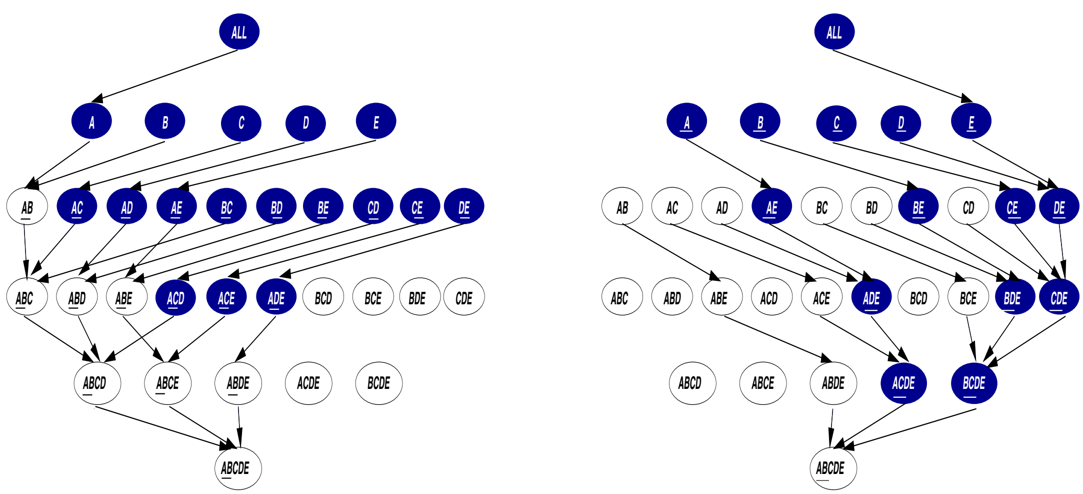


Figure 4.4: Left and Right schedules for 5 dimensions, where dimensions are ordered in a non-increasing order of their cardinality

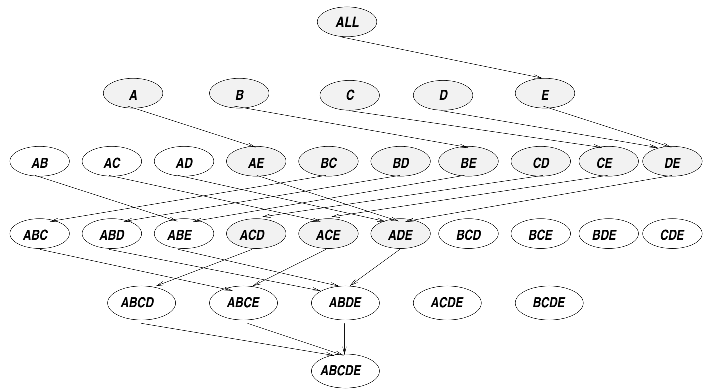


Figure 4.5: Left schedule for 5 dimensions, reduced communication

Communication in the left schedule can further be reduced by using conforming partitioning between the source (parent) and target (child) aggregates. For example, AC is calculated from ABC in the left schedule which generates communication. Since ACD is also present, it can be used to calculate AC without any communication. The left schedule reduces communication at the higher levels (levels with larger number of dimensions) by preserving conforming partitioning for calculations. The right schedule results in data movement earlier on (at higher levels) which leads to reduced communication at lower levels, i.e with aggregates with lower number of dimensions. Figure 4.5 shows the schedule with reduced communication, by using conformal partitioning in the left schedule (Figure 4.4).

Consider a set of 5 dimensions, ordered in a non-increasing order. Let $|A| = 10^5, |B| = 10^4, |C| = 10^3, |D| = 10^2, |E| = 10$. Table 4.2 shows the aggregates and their sizes that require communication. After canceling terms, it can be seen that the sizes of aggregates involved in communication in the left schedule are less than that for the right schedule. This is shown in the "terms left" row in the table, after canceling like terms at all levels to calculate the aggregate sizes. However, the sizes of the aggregates depend on the ordering of the dimensions by their cardinalities.

Table 4.2: Aggregates and sizes requiring communication

Level	Left	Right
1	A, B, C, D, E	A, B, C, D, E
2	AE, BC, BD, BE, CD, CE, DE $10^7 + 2 \times 10^6 + 2 \times 10^5 + 10^4 + 10^3$	AE, BE, CE, DE $10^6 + 10^5 + 10^4 + 10^3$
3	ACD, ACE, ADE $10^{10} + 10^9 + 10^8$	ADE, BDE, CDE $10^8 + 10^7 + 10^6$
4		ACDE, BCDE $10^{11} + 10^{10}$
Terms left	$10^9 + 10^5$	10^{11}

4.4.2 Dimension Ordering

In Table 4.2 the dimensions were in a decreasing order of their cardinalities. Communication generated by the left schedule was lesser than that generated by the right schedule. However, the sizes of the aggregates created in the left schedule are much higher if this ordering of the dimensions is considered. This is because more dimensions with higher cardinalities appear in the materialized aggregates. Consider, a dimension ordering $|D_i|, i = 0, \ldots k-1$, where $|D_0| > |D_1| > \ldots |D_{k-1}|$. A left schedule materializes cubes which have dimensions from the highest end of the dimension sizes in such an ordering, i.e $|D_0|, |D_1|....$ At lower levels (lesser number of dimensions), the sizes of the aggregates materialized are higher since the dimensions with higher cardinalities are involved in non-local computations. But, the aggregates at higher levels (more dimensions) needing communication are from the right side of the lattice and hence have smaller sizes. An overall analysis of the sizes involved in communication shows that the left schedule with this ordering results in lesser communication.

Now, consider the dimensions ordered in an increasing order of their cardinalities, it can be seen that the sizes of the aggregates are much lower. Here the left schedule will have lower cardinality aggregates materialized but the sub-cubes involved in communication have larger sizes. Table 4.3 shows the sizes for a 5 dimensional example with $|A| = 10, |B| = 10^2, |C| = 10^3, |D| = 10^4, |E| = 10^5$.

Table 4.3: Aggregates and sizes requiring communication (Dimension ordering in non-decreasing order)

Level	Left	Right
1	A, B, C, D, E	A, B, C, D, E
2	AE, BC, BD, BE, CD, CE, DE $10^9 + 10^8 + 2 \times 10^7 + 2 \times 10^6 + 10^5$	AE, BE, CE, DE $10^9 + 10^8 + 10^7 + 10^6$
3	ACD, ACE, ADE $10^{10} + 10^9 + 10^8$	ADE, BDE, CDE $10^{12} + 10^{11} + 10^{10}$
4		ACDE, BCDE $10^{14} + 10^{13}$
Terms left	$10^9 + 10^8 + 10^7 + 10^6 + 10^5$	$10^{14} + 10^{13} + 10^{12} + 10^{11}$

There are several combinations of dimensions that can be chosen to be intermediate aggregates to result in all the cubes at level 2 to be computed. However, when communication is considered, the solutions lie in two extremes, one with large sized sub-cubes to be computed, but minimizing communication, and the other is to compute small sized sub-cubes but the ones involved in communication are large. The choice of the left schedule, with the dimensions sorted in a non-decreasing order of their cardinalities provides a good compromise in minimizing the communication costs. Since communication costs are at least an order of magnitude more than computation costs, we would like to optimize the communication costs, we pick this schedule. As a further constraint, to obtain sufficient parallelism for 2 dimensional partitioning, in a general case, the partitioning dimensions should be chosen such that they provide parallelism for a given number of processors, p. The rest of the dimensions can then be ordered in a non-increasing order of their cardinalities.

4.4.3 Number of aggregates to materialize

In a full data cube, the number of aggregates is exponential. In the cube lattice there are d levels and for each level i there are $\binom{d}{i}$ aggregates at each level containing i dimensions. The total number of aggregates, thus is 2^d.

Table 4.4: Aggregates materialized for the data cube and for the case where m=2, for d=20

Level	Data Cube	m=2 (Partial)
1	20	20
2	190	190
3	1140	171
4	4845	153
5	15504	136
6	38760	120
7	77520	105
8	105970	91
9	127960	78
10	184756	66
11	127960	55
12	105970	45
13	77520	36
14	38760	28
15	15504	21
16	4845	15
17	1140	10
18	190	6
19	20	3

Next, we need to determine the number of intermediate aggregates needed to calculate all the aggregates for all levels below a specified $m, m < d$, for m-way associations and for meta-rules with m predicates.

Table 4.4 gives the number of aggregates at each level for a data cube and for a partial cube which has all 2-dimensional aggregates materialized. The number of aggregates at level 2 is $\sum_{j=1}^{d-1} \binom{j}{1}$. For level 3 it is $\sum_{j=1}^{d-2} \binom{j}{1}$. At each level k, the first $k-1$ literals in the representation are fixed and the others are chosen the remaining. The total number of aggregates is then $\sum_{i=1}^{d-1} \sum_{j=1}^{i} \binom{j}{1}$. After simplification this becomes $\frac{d(d-1)(2d+2)}{12}$. For $n = 20$, this number is 1330, a great reduction from $2^{20} = 1048576$. Similarly, calculations can be done to compute the number of intermediate cubes for calculating all sub cubes at a level m, for some

$m > 2$ to perform m-way association rule mining.

The base cube is d-dimensional. As intermediate aggregates are calculated by traversing the lattice (converted into a DAG by the left schedule) the following happens: 1) Dimensionality decreases at each level, 2) number of aggregates increase and 3) sizes of each aggregate decrease. Also, for a large number of dimensions it is impractical to calculate the full data cube. A subset of the aggregates can be materialized, and queries can be answered from the existing materializations, or some others can be calculated on the fly.

4.5 Summary

Calculations of simultaneous multidimensional aggregates can be optimized by scheduling the various group by calculations to reduce computation effort and reuse data in memory. Calculation of an aggregate from the smallest parent reduces the computational effort. Reducing disk I/O scans by reusing data in memory and scheduling group by calculations to reuse the cache effectively are some other optimizations. In a multiprocessor environment, various sub-cubes are distributed on processors. These need to communicate for aggregate computations. Minimizing communication across processors is important and is achieved by choosing group by calculations at each level of the cube lattice which leads to a minimum communication cost. A cost model modeling computation and communication costs is used to associate costs to the various group by calculations in a cube lattice. A schedule is generated which minimizes the cost at each level, and generates a directed acyclic graph from the cube lattice. A computation algorithm then enumerates the various calculations on the DAG, either traversing it depth first or breadth first. Cube reuse is dependent on the order of the traversal and will be discussed in Chapter 6.

Chapter 5

Parallel Data Cube Computation

This chapter deals with the construction of data cubes on parallel processors. For in-memory construction, we assume that the total available memory on the processors is large enough to hold the data sets in memory. This is a reasonable assumption since most parallel machines these days have anywhere between 128MB to 1GB main memory per node. With 16 nodes we can handle databases of size 2GB to 16GB and larger data sets can be handled by increasing the number of processors or memory on each processor. Hence, it is important to develop scalable parallel algorithms to handle large databases. In this chapter we present in-memory algorithms to calculate the data cube [GC97]. Further scalability can be achieved by building disk based parallel algorithms. Disk-based algorithms using chunking and sparse data structures is the topic of the next chapter.

5.1 Parallel Data Cube Construction for MOLAP

We assume that data is provided as a set of tuples stored in a file and the number of distinct elements are given for each attribute. For illustration purposes, let A, B, C and D be the attributes in a data set with D_a, D_b, D_c and D_d as the number of their distinct values, respectively. We assume that the number of distinct values in each dimension is known. However, the values are determined from the database by the algorithm. Without loss of generality, let $D_a \geq D_b \geq D_c \geq D_d$. If this is not the case, it can be made true by a simple renaming of the attributes. Let p be the number of processors, numbered $P_0 \ldots P_{p-1}$, and N be the number of tuples.

Algorithm 5.1.0 shows the various steps in the data cube construction algorithm. Each step is explained in the next few subsections.

Algorithm 5.1.0 *Parallel data cube construction and operations*

Step 1. Partition tuples between processors (*Partitioning*)
Step 2. Load tuples into multidimensional array (*Loading*)
Step 3. Generate schedule for order of group-by calculations
Step 4. Perform aggregation calculations (*Aggregation*)
Step 5. Redistribute/Assign sub-cubes to processors for query processing
Step 6. Define *local* and *distributed* hierarchies on all dimensions

---

First, the tuples are partitioned on p processors in a partitioning step. The *partitioning* phase is followed by a *loading* phase in which a multidimensional array is loaded on each processor from the tuples acquired after the partitioning phase. This creates the *base cube*. Loading can either be done by a hash-based method or a sort-based method. We have implemented both and have compared their scalability properties. This is followed by a *aggregation* phase which calculates the various aggregate sub-cubes. We describe each phase in the next few subsections.

5.1.1 Partitioning

A sample based partitioning algorithm is used to partition the tuples among the processors. Attribute A is used in this partitioning. This is done to ensure the partitioning of data at the coarsest grain possible. This divides A nearly equally onto the processors and also establishes an order on A. If $A_x \in P_i$ and $A_y \in P_j$ then $A_x \leq A_y$ for $i < j$. In fact, in the partitioning scheme used here, tuples end up being sorted locally on each processor. The algorithm is illustrated in Algorithm 5.1.1 with communication steps shown in bold.

Algorithm 5.1.1 *Sample based partitioning*

---

p - Number of processors.

Step 1. Sort tuples on attribute A locally on each processor

Step 2. Pick up a sample of size p of attribute A from each processor. **Gather** the sample on processor 0 and sort it locally.

Step 3. Pick up $p - 1$ splitters from this list on processor 0 and **Broadcast** them.

Step 4. Using the splitters each processor divides it's tuples into p sorted partitions with all values of attribute A in a lower partition being lower than a higher numbered partition.

Step 5. Using **All-to-All communication** each partition is sent to its destination processor.

Step 6. Each processor merges the sorted partitions obtained from other processors resulting in a set of tuples sorted globally on attribute A.

Once tuples are partitioned on processors, they are loaded into a multidimensional array (md-array). The size of the md-array in each dimension is the same as the number of unique values for the attribute represented in that dimension. A tuple is represented as a cell in the md-array indexed by the values of each of the attributes. Hence, each *measure* needs to be loaded in the md-array from the tuples. We describe two methods to perform this task, a hash based method and a sort-based method.

5.1.2 Hash-based cube loading

Algorithm 5.1.2 describes the hash-based cube loading algorithm. Each attribute is hashed to get a hash table of unique values for it. A sort on the attribute's hash table will index the dimension of the base cube corresponding to that attribute. These hash tables are then probed to fill in the measure values in the base cube. Hashing techniques are known to provide good performance on the average, though it heavily depends on the choice of a good hash function.

Algorithm 5.1.2 *Hash-based data cube loading Algorithm*

---

k - number of attributes(dimensions)

Step 1. For each tuple, hash each of the attributes into a separate hash table, one for each attribute. k hash tables are created as a result of this. (**Hash** phase)

Step 2. Compress and sort each hash table.

Step 3. Update the index of the hash table with its order in the corresponding sorted list.

Step 4a. Pick up each tuple and probe the hash tables for each of its attributes to obtain the indices in each dimension. (**Probe** phase)

Step 4b. Update the cell at the index in the md-array with the measure values of the tuple.

5.1.3 Sort-based cube loading

Algorithm 5.1.3 describes the sort-based cube loading algorithm. This method provides regularity of access over the sorted hash tables since the attributes probing them are sorted, unlike the hash-based method, where accesses in the hash table have no order. The sorted hash-tables are scanned only in one direction for all the tuples which have the same value for all the attributes in dimensions which come earlier, i.e have more unique values, since the order in which the attributes are sorted is with respect to their number of unique values. For example, for two consecutive records (a_1, b_1, c_1, d_1) and (a_1, b_1, c_1, d_4), hash table for D is scanned from the current position to get the index. However, for (a_1, b_1, c_1, d_4) and (a_1, b_2, c_1, d_1), hash tables for both C and D need to be scanned from the beginning.

5.1.4 Aggregation

An example of a base cube is shown in Figure 5.1 with $p = 4, D_a = 8, D_b = 4, D_c = 2, D_d = 2$. Hence each processor has $\frac{D_a}{p}$ portion of A. We illustrate the costs of calculating the various GROUP-BYs from the base cube for a three attribute (or 3D) cube in Table 5.1. Let t_{op} be the cost of an addition, t_{copy} the cost of copying a byte. Communication is modeled by collective communication operations as described in the previous section. These costs are then used by a scheduling algorithm, which

Algorithm 5.1.3 *Sort-based data cube loading Algorithm*

---

k - Number of attributes(dimensions)

Step 1. Sort tuples locally on each processor in such a way that B is sorted for each unique value of A, C is sorted for each unique value of B and so on. Note that A is already sorted on each processor.

Step 2. For each tuple hash each of the attributes into a separate hash table, one for each attribute. k hash tables are created as a result of this. (hash phase)

Step 3. Compress and sort each hash table.

Step 4. Pick up each tuple and calculate the indices of each attribute in the md-array in the following way. (Assume that the sorting order is A, B and C)

Step 4a. Pick up the attribute at the innermost level of the sort and linearly scan the sorted hashed list in for it to pick up the index.

Step 4b. As long as the values at higher levels are the same, scan past the point where the scan stopped last, to find the index. If a attribute value changes, scan the sorted hash lists from the start for all inner attributes.

Step 4c. Update the cell at the index in the md-array with the measure values of the tuple.

---

generates a schedule for calculating the various group-bys.

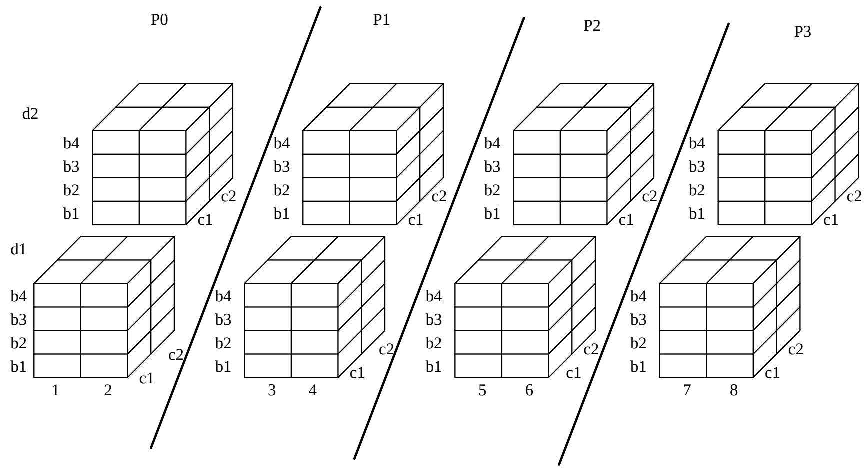
Some calculations are *local* and some are *non-local* and need multiple processors to exchange data leading to communication among processors. An example of a local aggregate calculation, ABCD $\rightarrow$ ABD, is shown in Figure 5.2. Even with local calculations the costs can differ depending on how the calculation is made. Calculating AC from ABC requires summing on the B dimension and calculating AC from ACD requires aggregation on D. Depending on how the multidimensional array is stored these costs could be different since the stride of the access can affect the cache performance. From the cost calculations shown in Table 3, we see that the cost of calculating aggregates from a parent are lower if the order of the attributes in the aggregate is a prefix of the parent. Calculating ABC $\rightarrow$ AB $\rightarrow$ A is a local calculation on each node. Each cube is distributed across the processors since dimension A is distributed. The intermediate cubes, resulting from aggregation of parent cubes are also distributed among the processors. This results in good load balancing among the processors. Calculations involving multiple cubes can also be distributed across processors as a result of this. The first attribute in the aggregate cube is always distributed among processors. As a result, A is distributed in ABCD, ABC, AB and A, B is distributed in BCD, BC and B, and C is distributed in CD and C and D is distributed. Figure 5.1 shows A as distributed in ABCD.

Source	Target	Cost
ABC $\rightarrow$	AB	$(\frac{D_a}{p}D_bD_c))t_{op}$
	AC	$(\frac{D_a}{p}D_bD_c)t_{op}$
	BC	$(\frac{D_a}{p}D_bD_c)t_{op} + (D_bD_c)t_{reduce} + (\frac{D_b}{p}D_c)t_{copy}$
AB $\rightarrow$	A	$(\frac{D_a}{p}D_b)t_{op}$
	B	$(\frac{D_a}{p}D_b)t_{op} + D_bt_{reduce} + \frac{D_b}{p}t_{copy}$
AC $\rightarrow$	A	$(\frac{D_a}{p}D_c)t_{op}$
	C	$(\frac{D_a}{p}D_c)t_{op} + D_ct_{reduce} + \frac{D_c}{p}t_{copy}$
BC $\rightarrow$	B	$(\frac{D_b}{p}D_c)t_{op}$
	C	$(\frac{D_b}{p}D_c)t_{op} + D_ct_{reduce} + \frac{D_c}{p}t_{copy}$
A $\rightarrow$	ALL	D_at_{op}
B $\rightarrow$	ALL	D_bt_{op}
C $\rightarrow$	ALL	D_ct_{op}

Table 5.1: Calculation of GROUP-BYs for a three attribute data cube $(D_a \times D_b \times D_c)$, on p processors

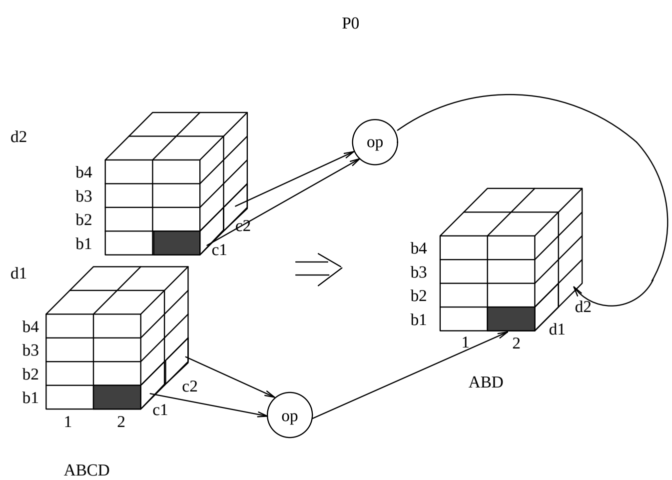
Figures 5.3 and 5.4 illustrate a global aggregate calculation of calculating $ABCD \rightarrow BCD$. First a local aggregation is done along dimension A (Figure 5.3). This is followed by a **Reduce** operation on BCD (Figure 5.4). Each processor then keeps the corresponding portion of BCD, distributing B evenly among processors.

Clearly, as can be observed from the aggregation calculations shown in the table above, there are multiple ways of calculating a GROUP_BY. For calculating the aggregate at a level where aggregates can be calculated from multiple parents, we need to pick up an assignment such that the cost is minimized. In [SAG96], this is solved by posing it as a graph problem and using minimum cost matching in a bipartite graph. We have augmented it by our cost model and the optimizations



A: 1,2,3,4,5,6,7,8
B: b1,b2,b3,b4
C: c1,c2
D: d1,d2

Figure 5.1: Basecube for 4 attributes on 4 processors



A: 1,2
B: b1,b2,b3,b4
C: c1,c2
D: d1,d2

Figure 5.2: *Local* aggregation calculation, ABCD $\rightarrow$ ABD on processor P0

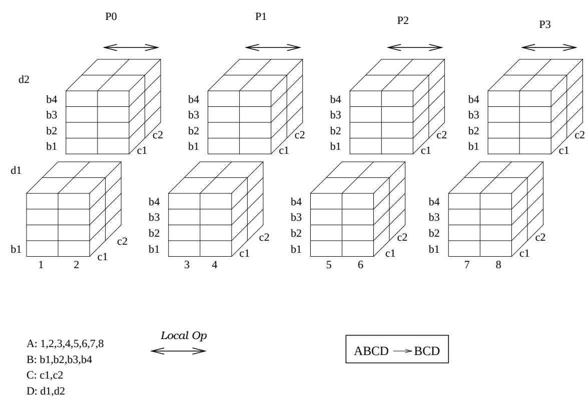


Figure 5.3: *Global* aggregation calculation, local phase, ABCD → BCD

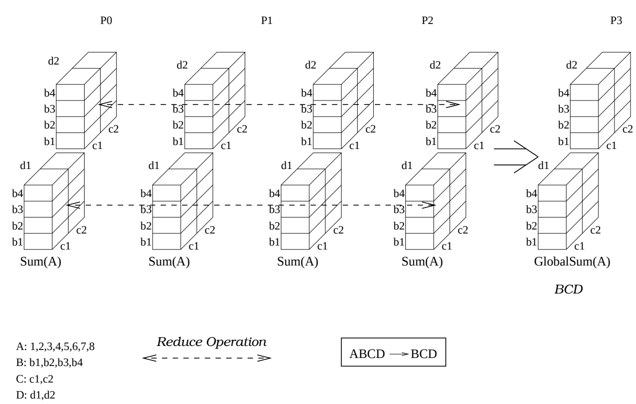


Figure 5.4: *Global* aggregation calculation, global phase, ABCD → BCD

needed. Again, refer to Figure 4.3 in Chapter 4. Suppose we want to calculate aggregates at a level k from the parent at level $k+1$. A bipartite graph $G((V = X \cup Y), E)$ is defined as follows. A group of nodes X is the nodes at level k. Clearly $|X| = 2^k$. Another group of nodes $|Y|$ is the nodes at level $k+1$ and $|Y| = 2^{k+1}$. The edges connecting the nodes at level k and $k+1$ belong to E. The edge weights are the costs of calculating the particular aggregate at level k from the parent at level $k+1$. Costs described in Table 5.1 are used here. A node at level $k+1$ can possibly calculate $\begin{pmatrix} k+1 \\ k \end{pmatrix}$ aggregates at level k. Hence the nodes are replicated these many times at level $k+1$ and these are added to Y. We seek a match between nodes from level $k+1$ and level k which minimizes the total edge costs. Figure 5.5 shows an example of calculating level 2 from level 3 of the lattice of Figure 4.3. This is done for each level and the resulting order of calculations at each level is picked up. This creates a directed acyclic graph which is then traversed, from the base cube as the root, to each child in a depth first search manner, calculating each aggregation. This results in preserving all of the three optimizations for multiple group-bys described in an earlier section.

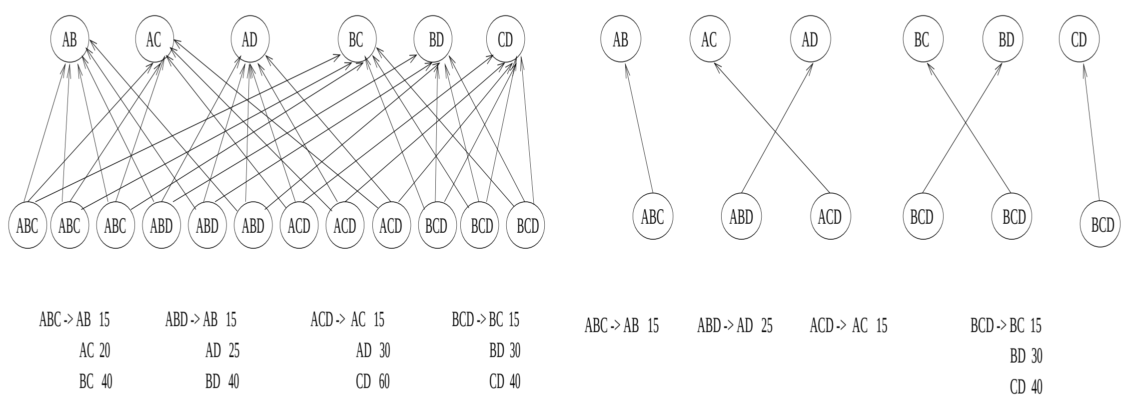


Figure 5.5: A minimum cost matching for calculation of Level 2 aggregates from Level 3

5.2 Performance Evaluation

We implement the algorithms presented in the previous section using 'C' and message passing using the Message Passing Interface (MPI) on the IBM SP-2 parallel machine. Each node of the SP-2 is a RS/6000 processor, with 128MB memory. The interconnection network is a multistage network connected through a high speed switch. The use of C and MPI makes the programs portable across a wide variety of parallel platforms with little effort.

5.2.1 Data sets

We use the OLAP council benchmark [OLA97] which simulates a realistic On-Line Analytical Processing business situation. The data sets have attributes taken from the following set of attributes: Product (9000), Customer (900), Time (24), Channel (9) and Scenario (2). The values in brackets following each attribute are the number of unique values for that attribute. These are taken to be the size of the dimensions in the resulting multi-dimensional arrays in the sub-cubes of the data cube. For the History Sales data, we have picked out 1000, 101, 100 and 10 distinct values for its 4 attributes from the benchmark data and generated the records by picking out each attribute randomly from the set of distinct values for that attribute. Product, Customer and Channel are character strings with 12 characters each. Time is an integer depicting year and month (e.g. 199704 is April 1997) and Scenario is a 6 character string showing if it is an "Actual" or a "Budget" value. The *measure* stored in each cell of the array is a **float** value depicting the sales figure for the particular combination of the attributes. This can potentially contain many more values. We have used the *Sum()* function to compute the aggregates in the results presented here. The characteristics of the data sets are summarized in Table 5.2.

5.2.2 Data cube Construction

We present results for the different phases of data cube construction for all the data sets described above. Figure 5.6 shows the performance of hash-based and sort-based methods for the Shipping Cost data with 27000 records. We observe that the

Data Set	Dimensions	Records	Base cube dimensions	Data size
Shipping Cost	Customer, Time, Scenario	27000	$900 \times 24 \times 2$	1MB
Production Cost	Product, Time, Scenario	270,000	$9000 \times 24 \times 2$	10MB
Current Sales	Product, Customer, Channel	81000, 608866	$9000 \times 900 \times 9$	320MB
Budget	Product, Customer, Time	951,720	$9000 \times 900 \times 12$	420MB
History Sales	Product, Customer, Time, Channel	1,010,000	$1000 \times 101 \times 100 \times 10$	500 MB

Table 5.2: OLAP benchmark data sets used for data cube construction

algorithm scales well for this small data size. Each individual component scales well also. The two methods take almost the same time for the data cube construction.

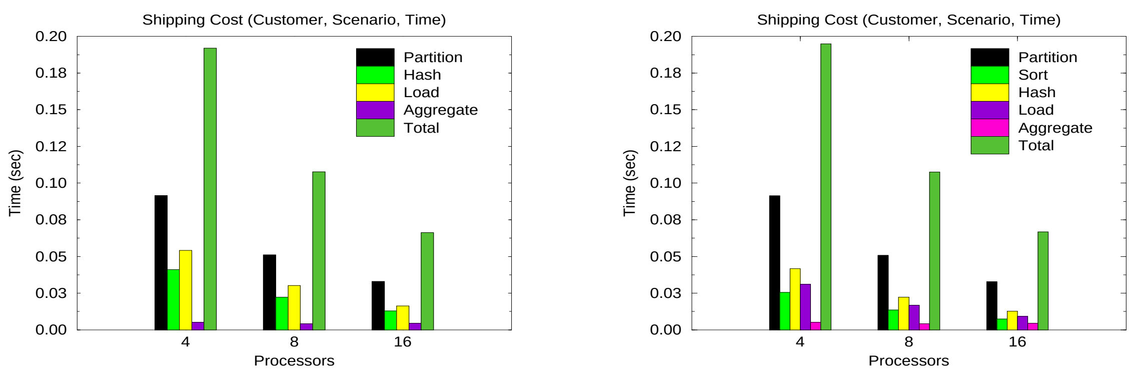


Figure 5.6: Various phases of cube construction using (a) hash-based (b) sort-based method for Shipping Cost data (N = 27000 records, p = 4, 8 and 16 and data size = 1MB)

Production Cost data contains 270,000 records and the performance of the two methods on this data set is shown in Figure 5.7. Again, as we increase the number of processors, each of the component in the construction algorithm scales well resulting in good overall scalability for both methods.

Figure 5.8 shows the performance of hash-based and sort-based methods for the Current Sales data with 81000 records. This data has a density of 0.11%.

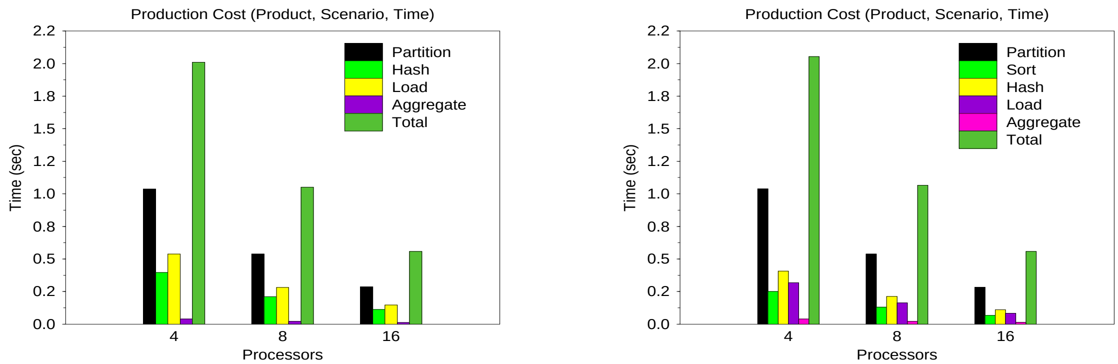


Figure 5.7: Various phases of cube construction using (a) hash-based (b) sort-based method for Production Cost data (N= 270,000 records, p = 4, 8 and 16 and data size = 10MB)

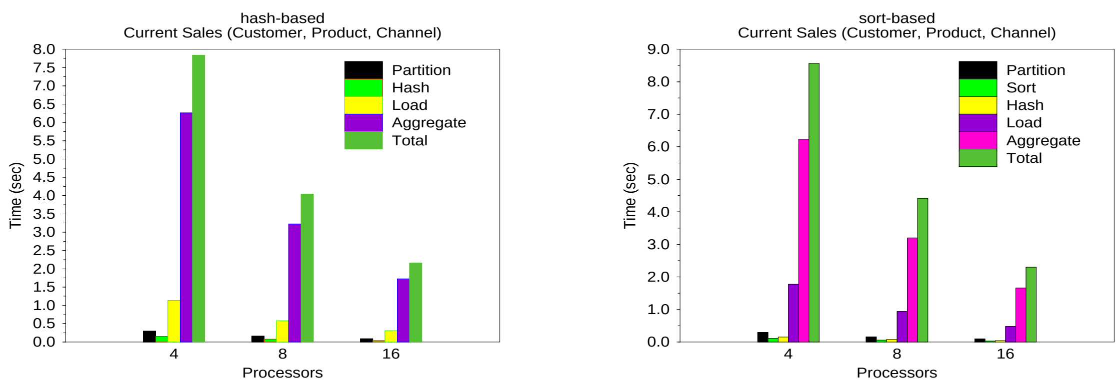


Figure 5.8: Various phases of cube construction using (a) hash-based (b) sort-based method for Current Sale data (N = 81,000 records, p = 4, 8 and 16 and data size = 320MB)

The aggregation time is the main component in the total time because of the large cube size for this data set. Figure 5.9 shows the performance of hash-based and sort-based methods for the Current Sales data with 608,866 records. This data has a density of 0.84%. All the individual components have increased except the aggregation component. The number of tuples has increased in this data set which affects the components till the loading phase. Aggregation times are dependent on the cube size, which has not changed.

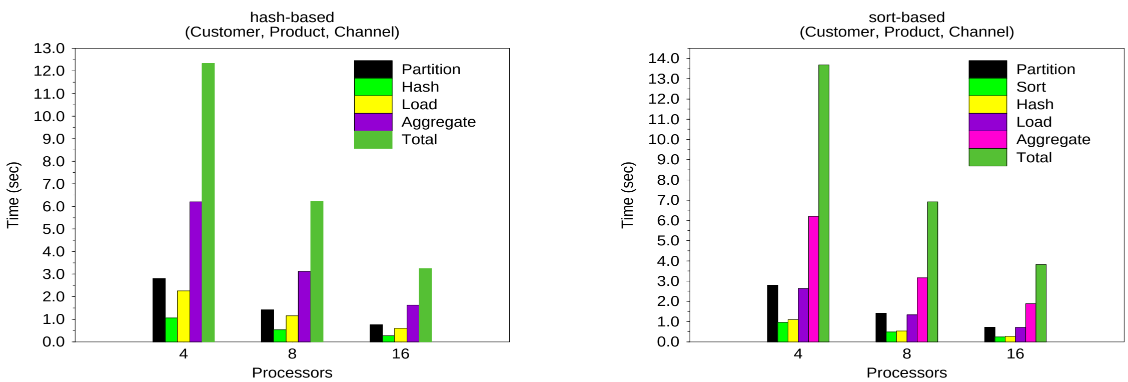


Figure 5.9: Various phases of cube construction using (a) hash-based (b) sort-based method for Current Sale data (N = 608,866 records, p = 4, 8 and 16 and datasize = 320MB)

Figure 5.10 gives the performance on Budget data with nearly a million records for 8, 16, 32, 64 and 128 processors. It can be observed that for large data sets the methods perform well as we increase the number of processors.

Figure 5.11 gives the performance on History Sales data containing 4 attributes (4 dimensional base cube) with more than a million records for 8, 16, 32 and 64 processors. The density of the cube is 1%. The hash-based method performs better than the sort-based method for this case. The addition of a dimension increases the stride factor for accesses for the sort-based loading algorithm, deteriorating the cache performance. The hash-based method has no particular order of reference in the multidimensional array and benefits from a better cache performance. It can

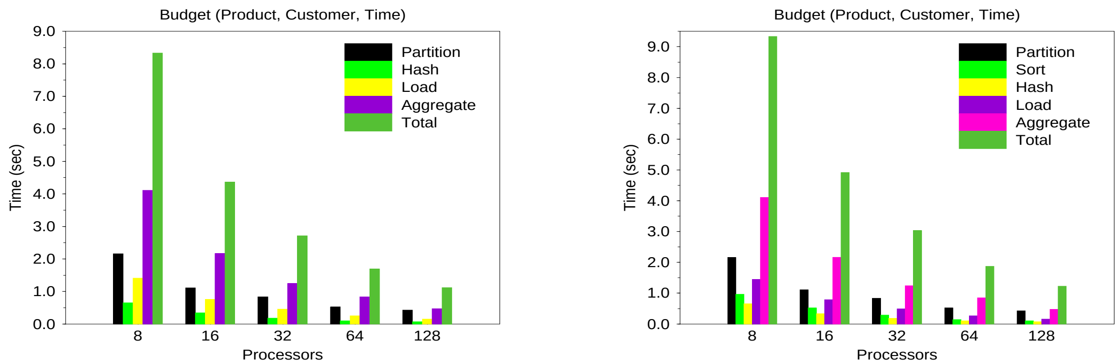


Figure 5.10: Various phases of cube construction using (a) hash-based (b) sort-based method for Budget data (N = 951,720 records, p = 8, 16, 32, 64 and 128 and data size = 420MB)

be observed that for large data each component in cube construction scales well as we increase the number of processors.

Next, we compare the effect of density of the data sets on the data cube construction algorithm. The size of the multi-dimensional array is determined by the number of unique values in each of the dimensions. Hence the aggregation costs, which use the multi-dimensional array are not affected by the change in density as can be seen from Figure 5.12 and Figure 5.13. The other phases of the data cube construction deal with the tuples and the number of tuples increase as the density of the data set increases. The costs for partition, sort, hash and the load phases increase because the number of tuples on each processor increases. The scalability for both densities, and both hash-based and sort-based methods is good as observed from the figures.

Comparing the sort-based methods and the hash-based methods, we observe that the hash-based method performs slightly better for 3D cubes but a lot better for the 4D case. We have used a multi-dimensional array and the outermost dimension of the array (the one changing the slowest) does not match the inner-most sorted-dimension. The inner-most dimension is used for the attribute with the largest

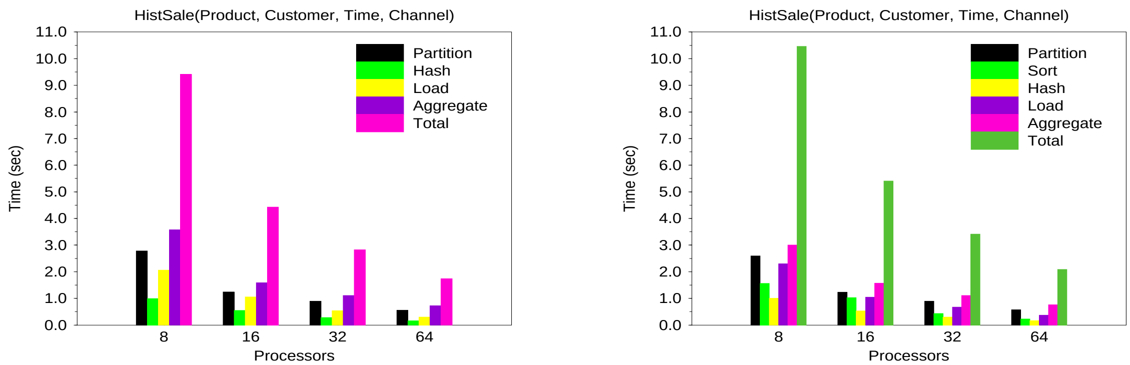


Figure 5.11: Various phases of cube construction using (a) hash-based (b) sort-based method for History Sales data (N = 1,010,000 records, p = 8, 16, 32 and 64 and data size = 500MB)

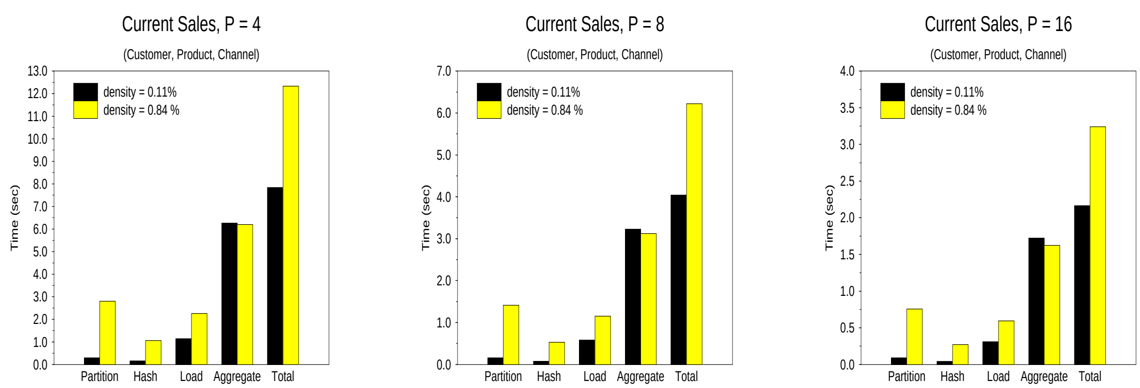


Figure 5.12: Comparison of cube construction components for two different densities for the hash based method

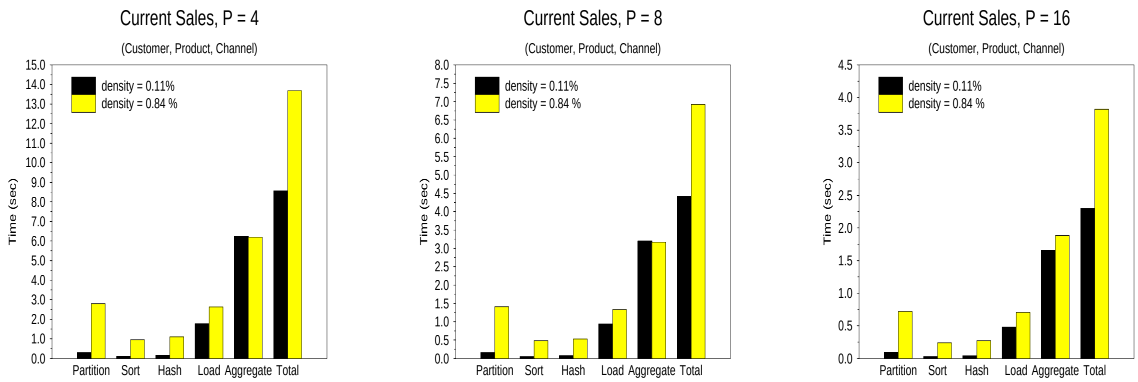


Figure 5.13: Comparison of cube construction components for two different densities for the sort based method

number of unique values to make the aggregate calculations efficient. The sort-based loading algorithm starts with the inner-most attribute in the sorted order which is also the smallest dimension. This also happens to be the outermost dimension of the array having the maximum stride. This is done to make the accesses during the aggregate calculation regular in the largest dimension, which is the innermost dimension. Chunking of arrays [SS94] can make the memory accesses for the sort-based method more regular since it does not favor any particular dimension for contiguous storage.

5.3 Summary

We presented a parallel multidimensional array based data cube computation framework in this chapter. Product of the dimensions in the cube are used to represent the cube size. Data is partitioned in a single dimension, each cube being partitioned in the largest dimension it contains. Dimensions are sorted in a non-increasing order to order the dimension of the multidimensional array, in order to exploit coarse grained parallelism. Minimum cost matching is used to obtain the lowest cost of group by computation at each level of the cube lattice. Performance evaluation for an implementation on the IBM SP2 is done using the OLAP council benchmark. This benchmark replicates a real life business scenario for OLAP. Each component of the cube construction process parallelizes well.

Chapter 6

Disk-based Parallel Cube Computation

In this chapter we describe our design for a parallel and scalable disk based infrastructure for data cubes 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. We address the issues of storage of cubes, parallelism at different levels, aggregation calculation orderings and the chunk access for aggregations. Some portions of the material in this chapter has appeared in [GC98b, GC99]. Moreover, a partial cube can be constructed if the number of dimensions is large or a specific level of the cube is needed. For example, in 2-way attribute-oriented data mining of associations, all cubes at level 2 are materialized by using the base cube and the minimum materializations of sub-cubes at the intermediate levels between 3 and $n-1$ [GC98a].

6.1 Data Partitioning and Parallelism

Data is partitioned on processors to distribute work equitably. 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 $|D_i| \gg p$ for some i, or $|D_i||D_j| \gg p$ 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, D_i is used to choose among the k partitions, and then D_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 6.1 shows the base cube loading for a 3 dimensional cube (ABC) distributed on 4 processors in a 2D distribution, where dimensions A and B are distributed in a (2, 2) grid. Clearly, when $K = 1$, a 1D partitioning is done with $k = p$.

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

Table 6.1 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

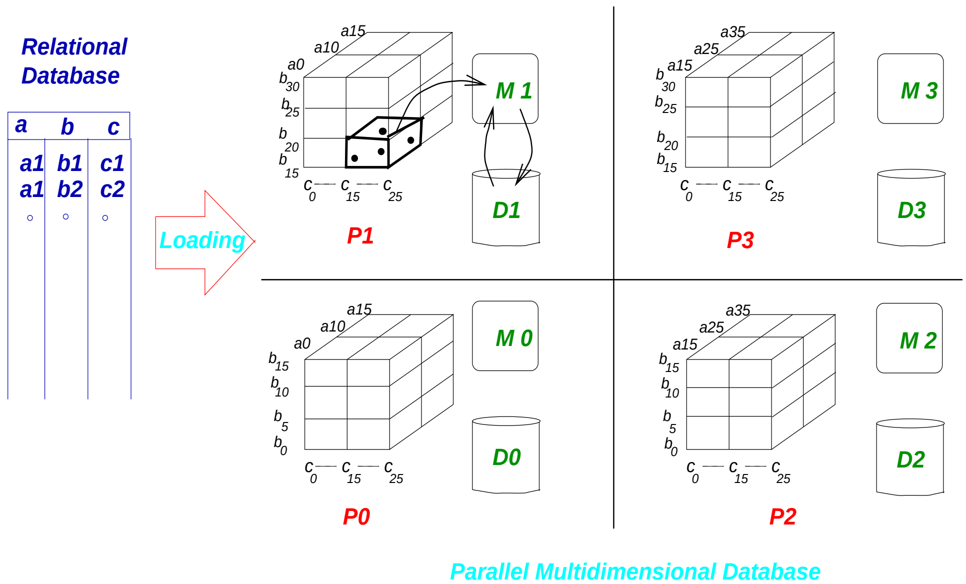


Figure 6.1: Base cube loading for a 3 dimensional cube on 4 processors

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 6.2, 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.

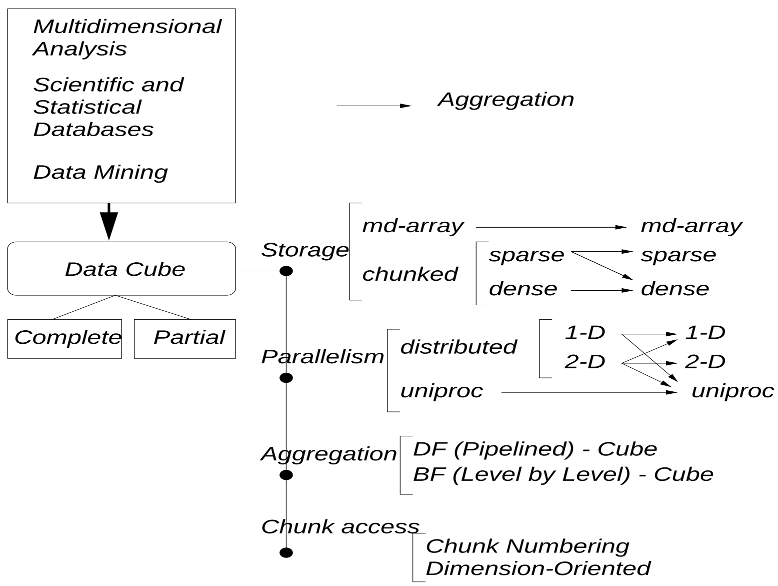


Figure 6.2: Scheduling for Data Cube

Table 6.1: Partitioning of sub-cubes following aggregation calculations

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

6.2 Data Structure Management

For large data sets the sizes of the cubes and the number of cubes will not fit in main memory of the processors. A scalable parallel implementation will require disk space to store results of computations, often many of them intermediate results. This is similar to a *paging* based system which can either rely on virtual memory system of the computer or perform the paging of data structures to the needs of the application. We follow the latter approach. Figure 6.3 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

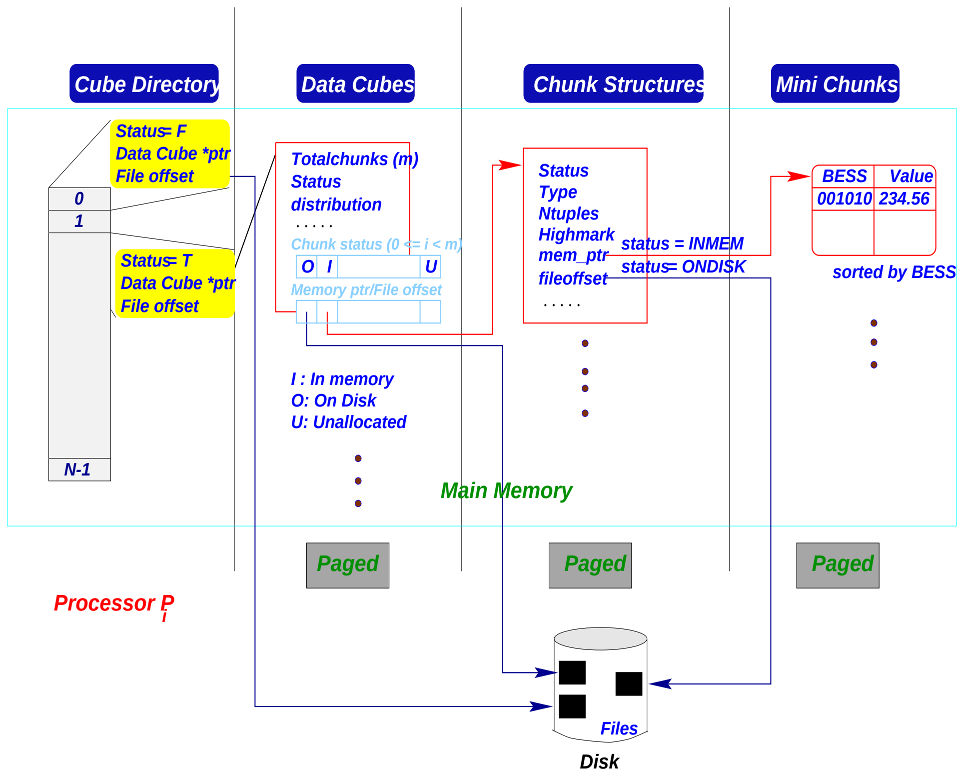


Figure 6.3: Data structure management for disk based data cubes

cstatus is maintained to keep track of chunk structure paging. The chunk address is a pointer to the chunk structure in memory which stores information for each chunk. This is when cstatus is set to INMEM. Otherwise, cstatus can either be UNALLOCATED or ONDISK. In the latter case the chunk address will be a file offset value. For a multidimensional array, the size of the array and the dimension factor in each dimension are stored to lookup for the calculations involving aggregations instead of calculating them on the fly every time.

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

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

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

6.3 Cost Model

As discussed in Chapter 4, Section 4.3.2, 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 6.1. 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 6.2 summarizes the cost estimations for the various distributions for the aggregation of the first distributed dimension is aggregated. Table 6.3 summarizes the cost estimation when dimension 2 is aggregated under the various distributions.

Table 6.2: 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 6.3: Cost Model for parallel aggregation calculations for aggregating dimension 2

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

[ZDN97] 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.

6.4 Algorithms and Analysis

Since chunks can either be sparse or dense, we need methods to aggregate sparse chunks with sparse chunks, sparse with dense chunks and dense with dense chunks. The case of dense chunks to sparse chunk does not arise since a dense chunk does not get converted to a sparse chunk ever. Also, a chunked organization may be converted into a multi-dimensional array. The various options are illustrated in Figure 6.2. In this section we discuss the algorithms for cube aggregations and chunk mappings. Table 6.4 describes the notation used in the algorithm description.

Table 6.4: 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

6.4.1 Chunk mapping to processors

Algorithm 6.4.1 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.

Algorithm 6.4.1 *Chunked cube mapping*

---

for $i \leftarrow 1$ *to* N_p
  for $j \leftarrow 1$ *to* $ndim,\ j \neq dim\_agg_p$
    $local\_start\_index_j^p = chunks_p[i].cidx_j * chunk\_extent_j$
    $local\_stop\_index_j^p = local\_start\_index_j^p + chunks_p[i].cdim_j$
    / Calculate the global index in dimension */*
    if($j \in \mathcal{D}_p^d$*)*
      $global\_start\_index_j^p = local\_start\_index_j^p + \mathcal{E}_j^p[myid]$
      $global\_stop\_index_j^p = local\_stop\_index_j^p + \mathcal{E}_j^p[myid]$
    if($j \in \mathcal{D}_c^d$*)*
      $j' = Rank(j, \mathcal{D}_c^d)$
      $proc_b = BinarySearch(global\_start\_index_j^p, \mathcal{E}_j^p[], \mathcal{P}_c^{j'})$
      $local\_start\_index_j^c = global\_start\_index_j^p - \mathcal{E}_j^c[proc_b]$
      $proc_e = proc_b$ */* (e)end and (b)beginning processors */*
      while($global\_stop\_index_j^p > \mathcal{E}_j^c[proc_e]$*)*

```
            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 $j \leftarrow 1$ *to* $ndim$, $j \neq dim\_agg_p$
  $global\_index_j = global\_start\_index_j^p + BESS_j$
  if$(j \in \mathcal{D}_c^d)$
    $proc = BinarySearch(global\_index_j, \mathcal{E}_j^p[], \mathcal{P}_c^{j'})$
    $local\_index_j = global\_index_j - \mathcal{E}_j^p[proc]$
  else
    $local\_index_j = global\_index_j$
  $target\_chunk\_id \mathrel{+}= (local\_index_j / chunk\_extent_j) * \mathcal{F}_j^c$
if$(proc = myid)$ /* *Local processor value* */
  if$(target\_chunk\_id \neq prev\_target\_chunk\_id)$
    Aggregate local buffer $\mathcal{B}_{myid}$ *with* $target\_chunk\_id$ *of target*
  else
    Copy BESS value to local buffer $\mathcal{B}_{myid}$
  $prev\_target\_chunk\_id = target\_chunk\_id$
else /* *Off-processor value* */
  Copy (BESS+value, $target\_chunk\_id$*) pairs into communication buffer* $\mathcal{B}_{proc}$
  (If $\mathcal{B}_{proc}$ *is full, do an* **Asynchronous SEND** *to proc and reuse buffer)*
Asynchronous RECEIVE *from other processors in* $\mathcal{R}_{proc_i}$ *for* $0 \leq i < |\mathcal{P}_p|, i \neq myid$*, and aggregate to local chunks.*
end

Each chunk in the source cube is processed to map its values to the target chunk. The chunk structure carries information about the chunk's dimensional offsets in *cidx*. This along with *cdim*, the chunk extents, is used to calculate the local value in each dimension. For distributed dimensions we need to add the start of the processor range to calculate the global value. This is then used to calculate the target start and stop values. This is used to determine the destination target processor and the target chunk. The source can map to the same target chunk on the same processor, same target chunk on another processor, split among chunks on the same processor or split among chunks on different processors. These cases are illustrated in Figure 6.4 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

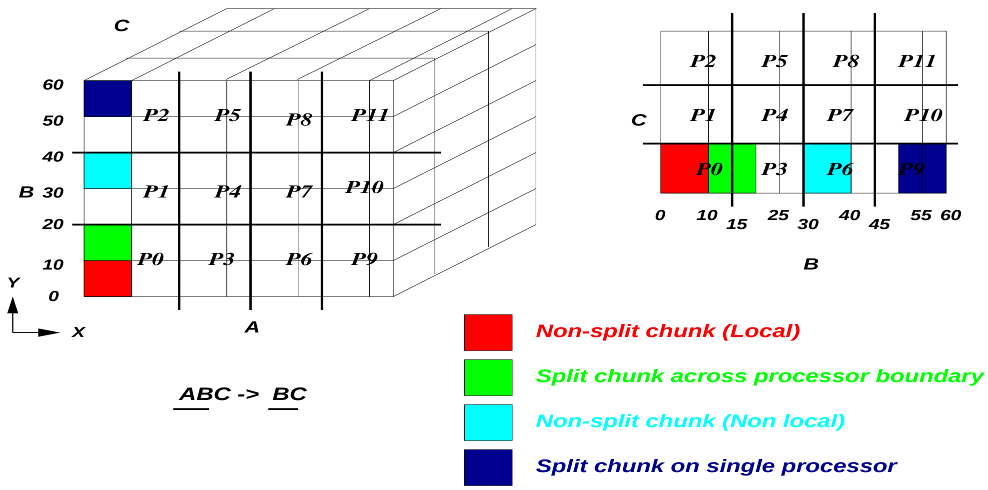


Figure 6.4: Chunk mapping after aggregation for a 2D cube distribution

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.

6.4.2 Local - Non Local Aggregation

Figure 6.5 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 6.1.

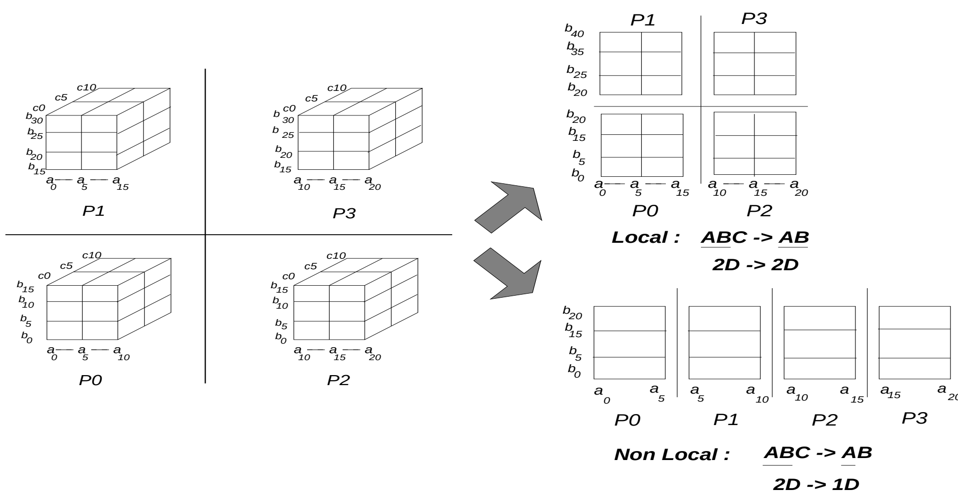


Figure 6.5: Local and Non Local aggregation calculations

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 6.6 illustrates a case for $\underline{A}B \rightarrow \underline{A}$ where the chunks with A extents of $10-13$ on processor P0 map to the chunk with extents $9-13$ on processor P1. The BESS encoding of A needs to be incremented by 1 to correctly reflect the target BESS encoding. If the chunk is overlapping over a target chunk boundary, then each BESS value has to be extracted to determine its target chunk. This is computationally more expensive than the direct case. It is to be noted that a 2 dimensional distribution may result in more overlapped chunks than a 1 dimensional distribution, because the former has more processor boundary area than the latter.

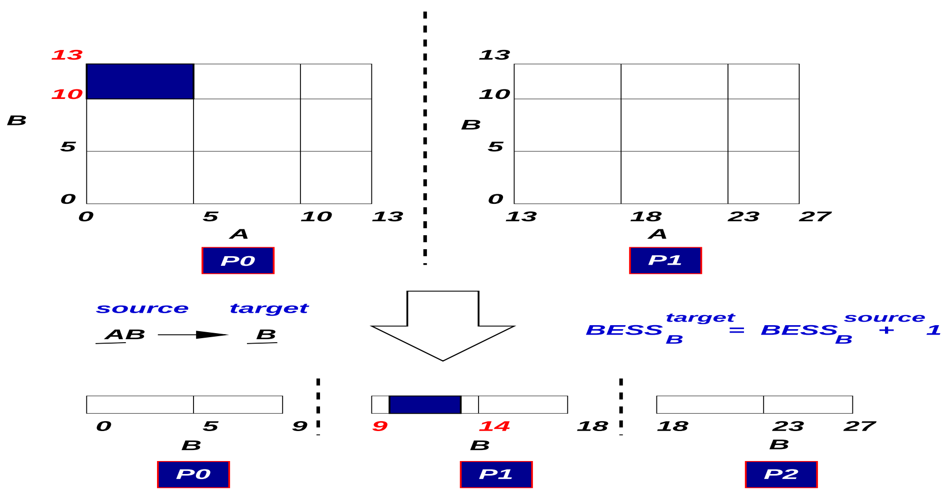


Figure 6.6: Modification of BESS indices while mapping a non-split chunk

6.4.3 Sparse - Sparse Chunk Aggregation

Algorithm 6.4.3 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 6.7 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. 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 6.8 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.

6.4.4 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.

Algorithm 6.4.3 *Sparse - sparse chunk aggregation algorithm*

---

for $i \leftarrow 1$ to N_p
    visited[i] = FALSE;
$done = FALSE$
Initialize $prev\_target\_chunk\_id$
for $i \leftarrow 1$ 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 $j \leftarrow 1$ 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(($nextchunk \leftarrow Get\_next\_chunk(dim\_agg_p, nextchunk)) = FALSE$)
                $done = TRUE$
        else
            $done = TRUE$
        $visited[nextchunk] = TRUE$
        } while (! $done$)
end

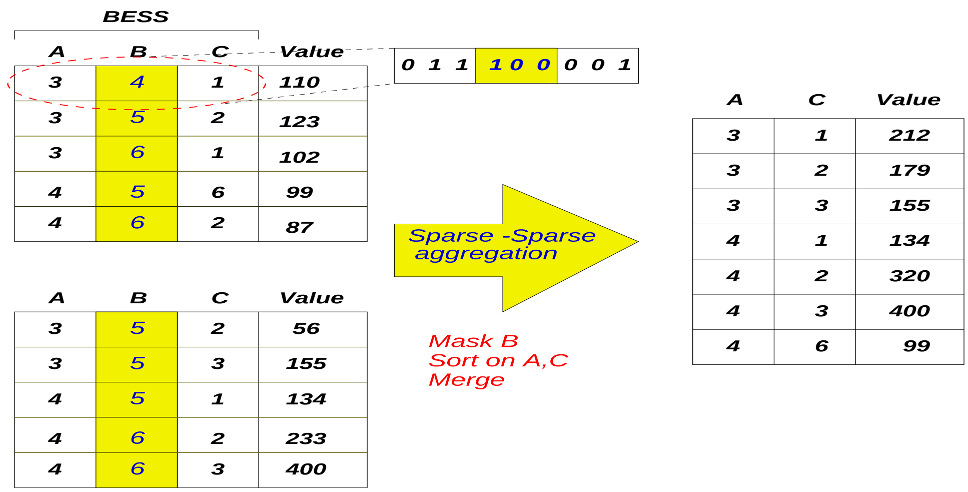


Figure 6.7: Sparse - sparse chunk aggregation

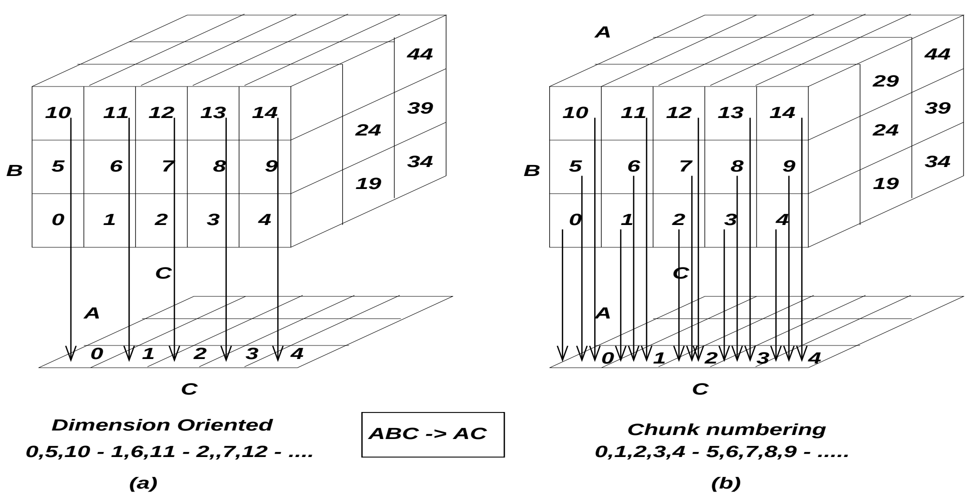


Figure 6.8: Dimension oriented vs. Chunk numbering access of chunks

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.

6.4.5 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. Algorithm 6.4.5 describes the algorithm in more detail.

Algorithm 6.4.5 *Chunked - multidimensional array cube aggregation*

---

for $i \leftarrow 1$ to N_p
    for $j \leftarrow 1$ to $ndim$, $j \neq dim\_agg_p$
        $local\_start\_index^p_j = chunks_p[i].cidx_j * chunk\_extent_j$
        $local\_stop\_index^p_j = local\_start\_index^p_j + chunks_p[i].cdim_j$
        /* Calculate the global index in dimension */
        if($j \in \mathcal{D}^d_p$)
            $global\_start\_index^p_j = local\_start\_index^p_j + \mathcal{E}^p_j[myid]$
            $global\_stop\_index^p_j + = local\_stop\_index^p_j + \mathcal{E}^p_j[myid]$
        if($j \in \mathcal{D}^d_c$)
            $j' = Rank(j, \mathcal{D}^d_c)$
            $proc^b_j = BinarySearch(global\_start\_index^p_j, \mathcal{E}^p_j[], \mathcal{P}^{j'}_c)$
            $local\_start\_index^c_j = global\_start\_index^p_j - \mathcal{E}^c_j[proc^b_j]$
            $proc^e_j = proc^b_j$ /* (e)end and (b)beginning processors */
            while($global\_stop\_index^p_j > \mathcal{E}^c_j[proc^e_j]$)
                $proc^e_j \leftarrow proc^e_j + 1$
            if($proc^b_j \neq proc^e_j$)

$local\_stop\_index^c_j$ -= $\mathcal{E}^c_j[proc^e_j]$

else /* For non-distributed dimension j in target */

$local\_start\_index^c_j = global\_start\_index^p_j$

/* Check if source chunk maps to the same processor or different processors */

if($proc_b = proc_e$) /* Same processor */

for $j \leftarrow 1$ to $ndim$, $j \neq dim\_agg_p$

$target\_offset \mathrel{+}= local\_start\_index^c_j * \mathcal{F}^c_j$

if($proc_b \neq myid$) /* Off-processor value */

Copy $target\_offset$+value pairs into communication buffer $\mathcal{B}_{proc_b}$

(If $\mathcal{B}_{proc_b}$ is full, do an **Asynchronous SEND** to $proc_b$ and reuse buffer)

else /* Local processor value */

Copy $target\_offset$+value pairs into local buffer $\mathcal{B}_{myid}$

(If $\mathcal{B}_{myid}$ is full, aggregate buffer and reuse)

else

for each BESS value in chunk

$target\_offset = 0$

for $j \leftarrow 1$ to $ndim$, $j \neq dim\_agg_p$

$global\_index_j = global\_start\_index^p_j + BESS_j$

if($j \in \mathcal{D}^d_c$)

$proc = BinarySearch(global\_index_j, \mathcal{E}^p_j[], \mathcal{P}^{j'}_c)$

$local\_index_j = global\_index_j - \mathcal{E}^p_j[proc]$

else

$local\_index_j = global\_index_j$

$target\_offset \mathrel{+}= (local\_index_j / dimension\_size^{proc}_j) * \mathcal{F}^c_j$

if($proc = myid$) /* Local processor value */

if(local buffer $\mathcal{B}_{myid}$ is full)

Aggregate local buffer $\mathcal{B}_{myid}$ with $target\_chunk\_id$ of target

else

Copy $target\_offset$+value to local buffer $\mathcal{B}_{myid}$

else /* Off-processor value */

Copy $target\_offset$+value to communication buffer $\mathcal{B}_{proc}$

(If $\mathcal{B}_{proc}$ is full, do an **Asynchronous SEND** to $proc$ and reuse buffer)

Asynchronous RECEIVE from other processors in $\mathcal{R}_{proc_i}$ for $0 \leq i < |\mathcal{P}_p|, i \neq myid$, and aggregate to local address offsets.

end

6.4.6 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 6.9 illustrates the order by a depth first and a breadth first traversal of the lattice structure. In Figure 6.9(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 6.9(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.

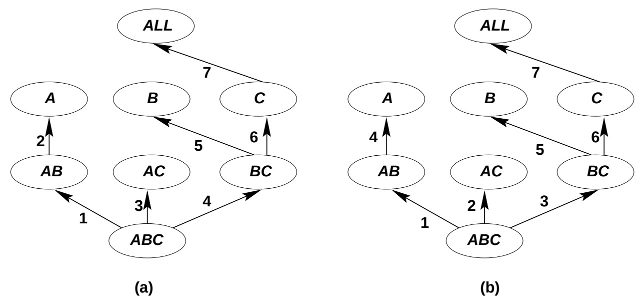


Figure 6.9: (a) Depth first traversal of lattice (b) Breadth first traversal of lattice

6.4.7 Data cube Updates

Data warehouses are updated on a timely basis as more data becomes available. OLAP data analysis is usually on historical data and the need for frequent updating of the cube is generally not an issue. With an update to the data warehouse, as more data is added, either the data cube can be computed again or the data cube can be updated depending on the cost of the computation. Deletions from the data warehouse require cube re-computation. Parallelism decreases the time of computation of the data cube computation and larger data cubes can be computed again fairly quickly. This is necessitated in the multi-dimensional model every time the schema changes, i.e either the number of dimensions change or the dimension cardinality changes. If new tuples are added then the updates can be propagated to the aggregate computations without requiring data cube re-computation. A set of tuples to be added is converted to a set of chunk numbers and BESS offsets within the chunk on each processor. BESS offsets are then added to the appropriate chunk, which is an insert into a sorted list of BESS offsets, aggregating the value for duplicate indices. This is done for each level of the cube masking the appropriate dimension encoding for the new tuples.

6.5 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 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. Algorithm 6.5.0 shows the communication phase on each processor.

6.6 Performance Evaluation

In this section we present performance results for our disk-based 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

Algorithm 6.5.0 *Communication in source $\rightarrow$ target aggregation*

---

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

---

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 $D_0 \rightarrow D_1 \rightarrow D_2 \ldots D_{n-1}$, is conformant to the layout of chunks where D_0 is the outer most dimension and D_{n-1} is the inner most, for loading a sorted run of values.)

The base cube is loaded on each processor (Figure 6.1) 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 chose 4 data sets, one each of dimensionality 5, 10 and 20 to illustrate performance. Random data with a uniform distribution is used in these performance figures. We have evaluated other types of data, skewed data sets and the OLAP council benchmark [OLA97], which models a realistic business situation with similar

Table 6.5: Description of datasets and attributes, (N) Numeric (S) String

Data set	Dimension	Cardinalities (d_i)	$\prod_i d_i$	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

results.

Table 6.5 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 6.10 shows the time taken by the various phases of the data cube construction algorithm. Each phase of the cube construction process shows good speedup for all the data sets when a single dimension is partitioned in the base cube. A two dimensional partitioning performs better than a one dimensional partitioning because there are more chunks in the partitioned dimension that allows for better sparse-sparse aggregation performance.

Figure 6.11 shows the time taken by the two different methods of accessing chunks for aggregation calculations. Dimension oriented method is better than the access that follows chunk numbering because of better buffer use in the sparse aggregation calculations. In all cases we observe that the dimension ordering method works better than the access pattern that follows chunk numbering. This effect is

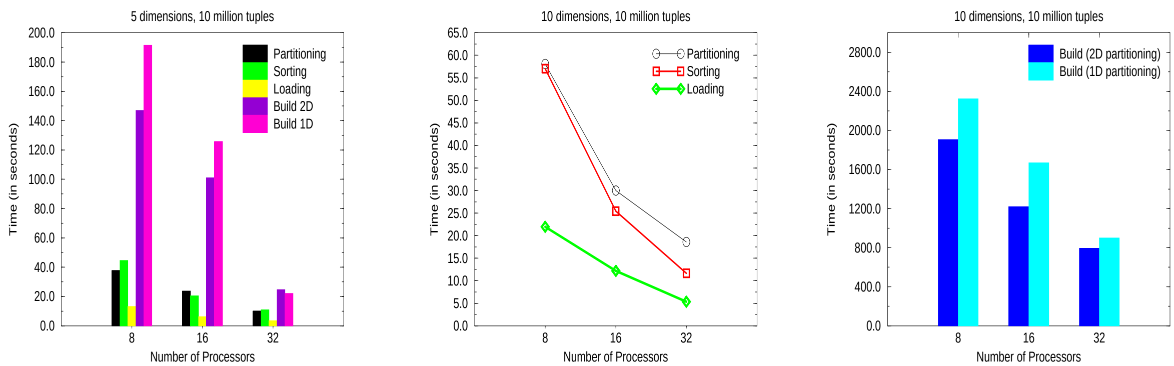


Figure 6.10: Time taken by various phases of the data cube construction algorithm for 5 (32 sub cubes) and 10 (1024 sub cubes)

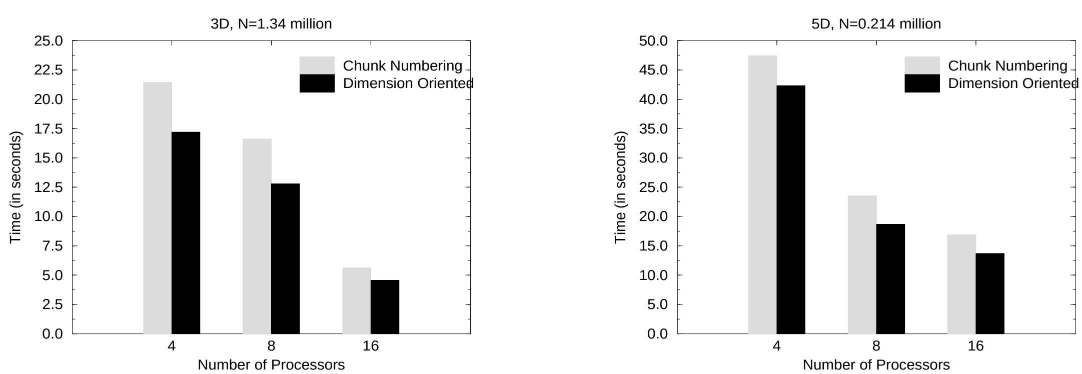
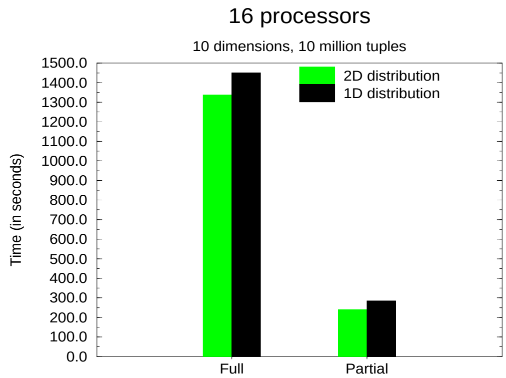
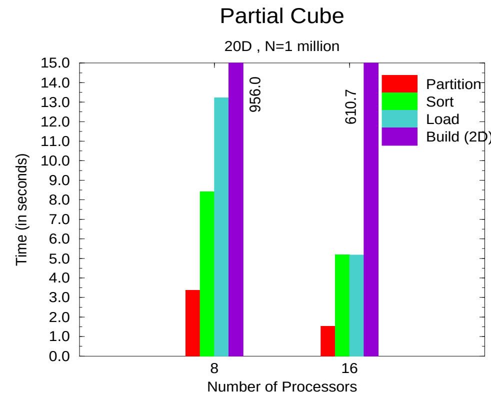


Figure 6.11: Comparison of the two chunk access methods, chunk numbering and dimension oriented for a dataset with 3 and 5 dimensions

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



(a)

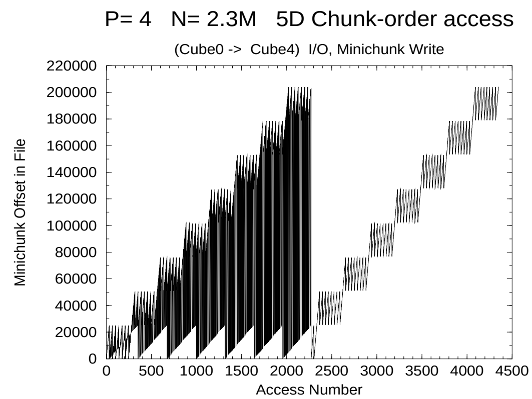


(b)

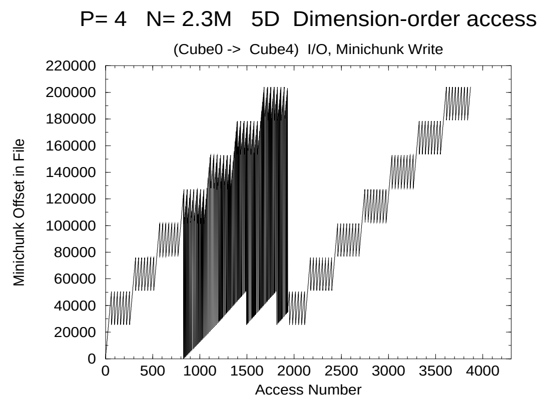
Figure 6.12: Comparing full and partial cubes (a) 10D, 10 million tuples on 16 processors, (b) partial cubes for 20D on 8 and 16 processors

Figure 6.12 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 efficiently. These two are the more important ones in terms of using support and confidence measures to find useful rules from large databases.

Figure 6.13 compares the minichunk write performance for the two chunk access orders for a 5 dimensional data set computing cube 4 from cube 0 ($ABCDE \rightarrow ACDE$). A minichunk write occurs to flush the aggregation buffer while performing the sparse aggregation. This can happen if the buffer is full or when the target mapping of the source chunks change. We observe that there are lesser writes for dimension order access and the file offsets that are written have better locality than the chunk-order access mechanism. Similar performance is observed for 8 processors



(a)



(b)

Figure 6.13: Minichunk writes to Cube 4 on 4 processors (a) 5D, 2.37M, Chunked-order access (b) 5D, 2.37M, Dimension-order access

as shown in Figure 6.14

Figure 6.15 shows the write performance for a 10 dimensional data set for computing all the 9-dimensional aggregates. Cube 1 aggregates the innermost dimension and cube 10 aggregates the outermost dimension of cube 0. The strides of the chunks increase as the various aggregations are computed from cube 0. The figure shows the write offsets of the minichunks, cube 1 accessing the chunks in a linear manner and the other cubes revisiting some of the offsets, the stride increasing as we go from cube 2 to cube 10. This is reflected in higher write times for the computations that aggregate the outermost dimension, as is seen even in the case of multidimensional arrays. Experiments on other data sets with different dimensionalities give similar results.

Figure 6.16 plots the file offsets accessed for the minichunk reads for reading cube 0 for the 3-dimensional data set (ABC) on 4 processors. This reflects on the calculation costs of the aggregates from the base data. Two different chunk sizes are used in the figure. There is a locality of offsets depicted for each of the reads for cubes 1, 2 and 3. The steps in the figure for cubes 1 and 2 represent the change of the chunk plane of the outermost dimension. Since the cardinality of the outermost dimension (A) chosen is 1024, and the partitioning is in two dimensions,

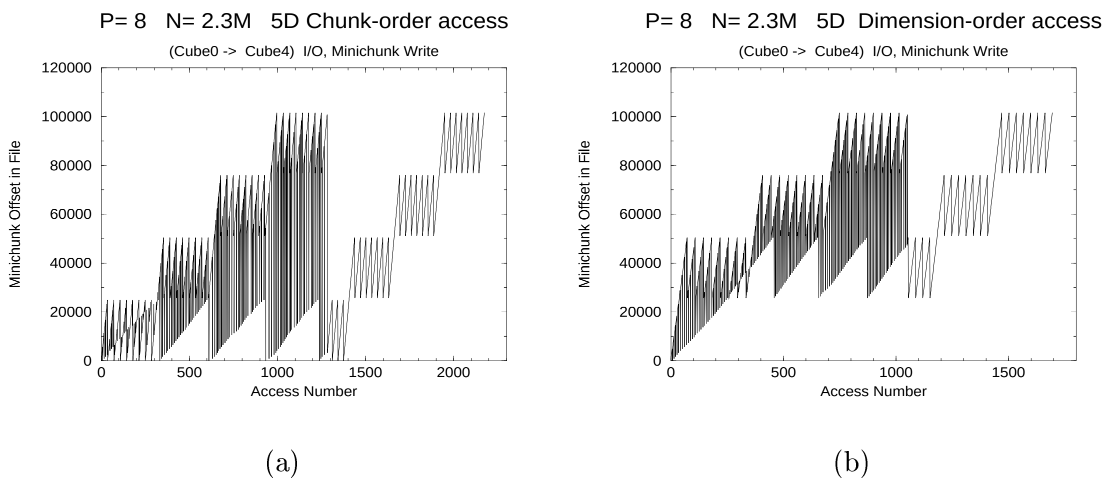


Figure 6.14: Minichunk writes to Cube 4 on 8 processors 5D, 2.37M (a) Chunked-order access (b) Dimension-order access

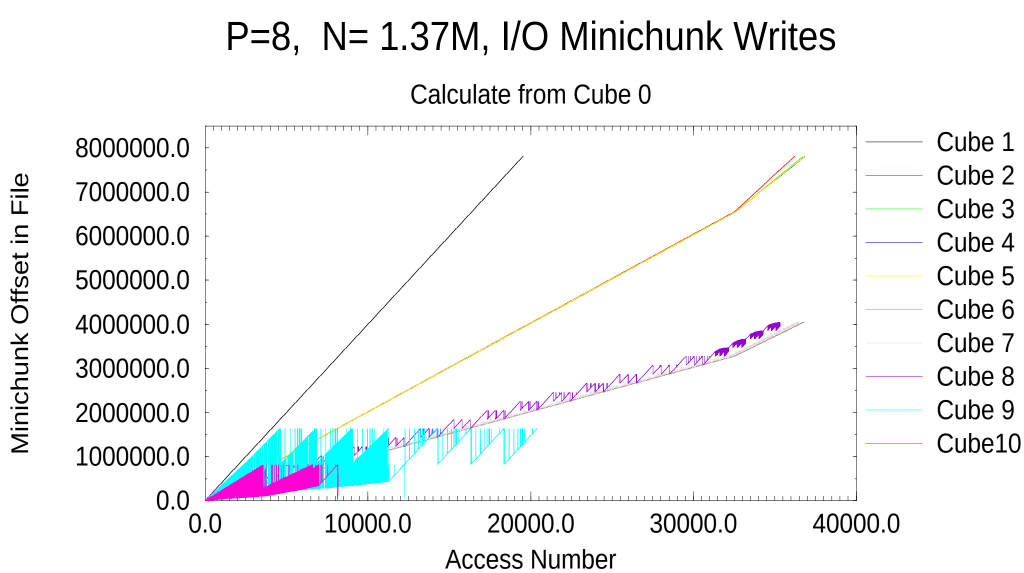


Figure 6.15: Minichunk writes to level 9 cubes calculating from Cube 0 aggregating innermost (Cube1) to outermost (Cube10) dimensions

each processor has nearly 512 attributes of A. Choosing a chunk size of 32 results in 16 intervals, and a chunk size of 64 results in 8 intervals, shown as steps in the figures. Thus, as the chunk size is increased the number of reads decreases and the locality in the file increases. However, for the outermost dimension A, the chunk reads have the accesses covering the entire range of the file offsets. Experiments on other data sets with different dimensionalities and dimension cardinalities show similar results.

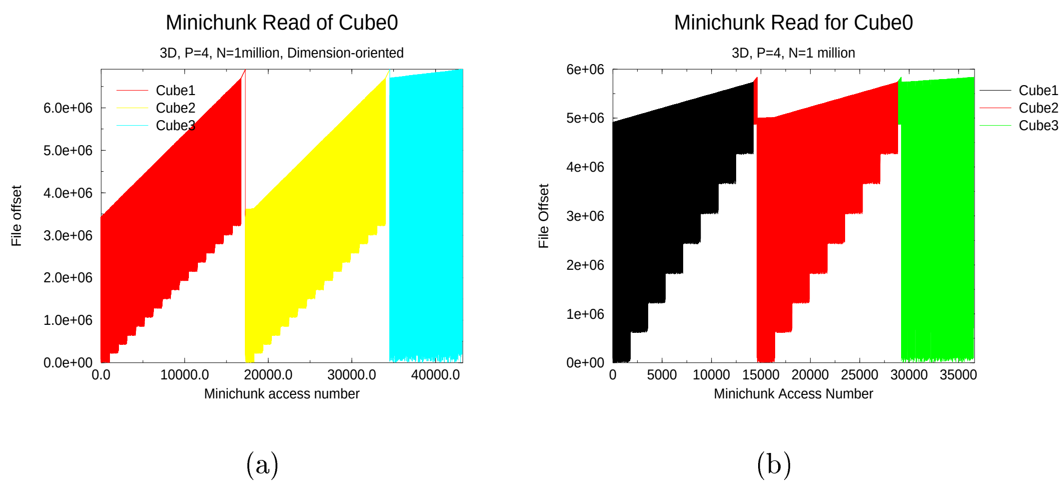
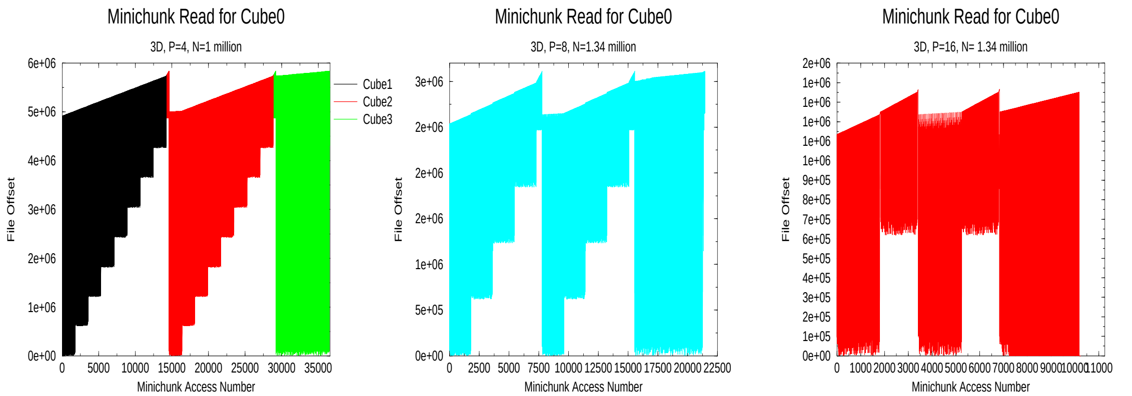


Figure 6.16: Minichunk read for Cube0: P=4, N=1.34M (a) chunksize= [32, 8, 16] (b) chunksize= [64, 8, 32]

Similarly, when the number of processors is increased, the chunk number of reads decrease. Also file offsets locality for all the dimensions except the outermost dimension improves. Figure 6.17 shows this behavior for a 3 dimensional data set for 4, 8 and 16 processors.

Figure 6.18 analyzes the write performance of minichunks for cubes AB, AC and BC being computed from cube ABC in a depth first traversal. Dimension order traversal is used which allows for locality in writing of minichunks. As can be seen from the figures the accesses to file offsets for cube 1 and 2 exhibit locality. Cube 3 aggregates the outermost dimension and the writes are with a higher stride. As chunk sizes are increased, the number of writes decrease and the locality is much better reflected in better running time. Experiments with chunk-ordering



(a) (b) (c)

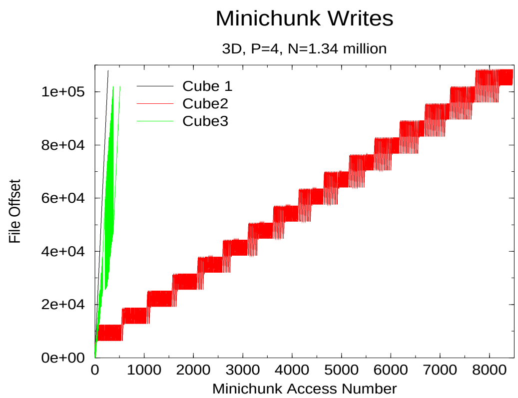
Figure 6.17: Minichunk read for Cube0: N=1.34M, chunksize=[64, 8, 32] (a) P=4 (b) P=8 (c) P=16

access show that the write performance does not exhibit locality regardless of the dimension being aggregated. Similar results are obtained for other combinations of dimensions and aggregation calculations for the other data sets.

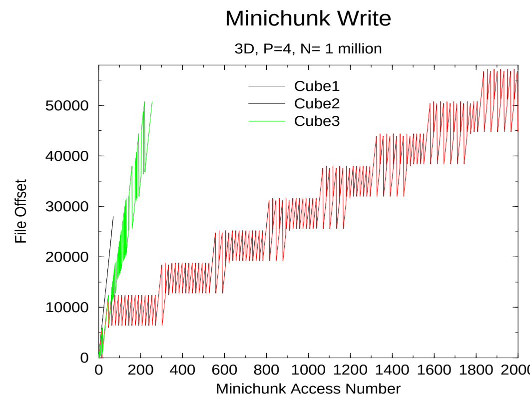
Aggregate calculations are also performed by using cube 2 (AC) and 3 (BC) as sources and the read access performance is shown in Figure 6.19. It can be seen that there is considerable amount of locality of reference in the file offsets for cube 2 since it is referenced in units of chunks for which the minichunks had locality. However, for cube 3 the minichunks are scattered in the file and hence the read accesses are also spread over the file. This results in higher read time from cube 3. As chunk size increases the number of reads decrease and locality further increases.

Table 6.6 shows the time for calculating various sub-cubes 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 6.7 shows the parameters that are indicators of the computation communication and I/O parameters and performance for dataset I on 4 processors. With

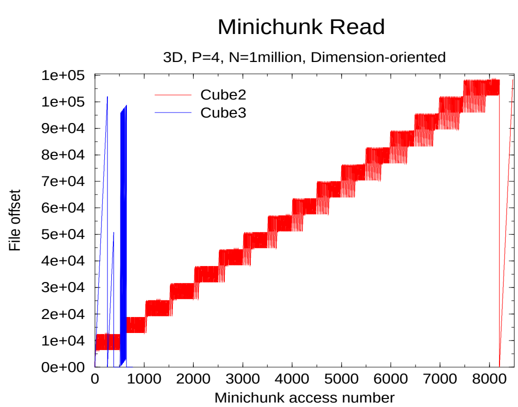


(a)

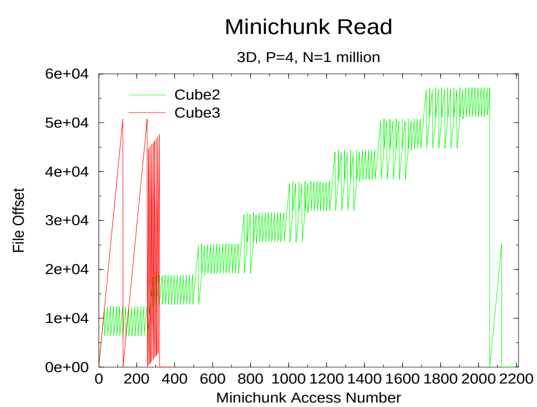


(b)

Figure 6.18: Minichunk write for cubes 1, 2 and 3: P=4, N=1.34M (a) chunksize=[32, 8, 16] (b) chunksize=[64, 8, 32]



(a)



(b)

Figure 6.19: Minichunk read for Cube2 (Magnified) and Cube3: P=4, N=1.34M (a) chunksize=[32, 8, 16] (b) chunksize=[64, 8, 32]

Table 6.6: 3D dataset with Chunksize I (32, 8, 16) and Chunksize II (64, 8, 32) (a) Aggregation time (in seconds) on 4 processors (b) Aggregation time (in seconds) for two chunk sizes on 4, 8 and 16 processors

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)

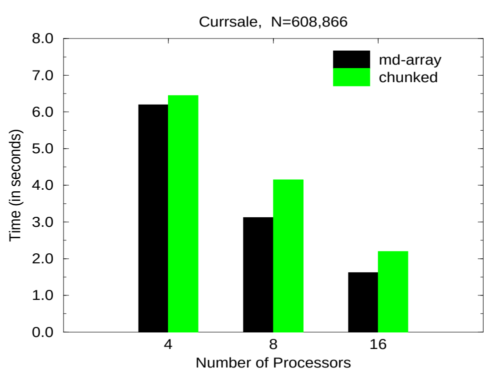
a larger chunksize we see that all these reduce, notably the number of split chunks, and the BESS merge read and write operations. This are reflected in the improved times seen in Table 6.6.

A comparison of the performance of aggregate operations using BESS and other sparse structures was presented in Chapter 3. It showed that for lower densities BESS performed as well as the multi-dimensional array and as the densities increased its aggregate performance degraded but was much better than the competing offset-value representation. Figure 6.20 compares the time taken for data cube construction using multi-dimensional arrays and the chunked cube representation using BESS. The data density is 1%. The performance of the chunked implementation is very close to that of the array. Data cube aggregation is performed on a 5 dimensional data set with dimension cardinalities $256, 16, 32, 16, 64$, with data densities of 0.16% and 1.6%, with 214,487 and 2,144,873 tuples respectively. The performance of the lower density data set for the chunked implementation is close to that provided by the multi-dimensional array. However, with around 2 million tuples, disk I/O costs increase with minichunk read and writes and this increases the

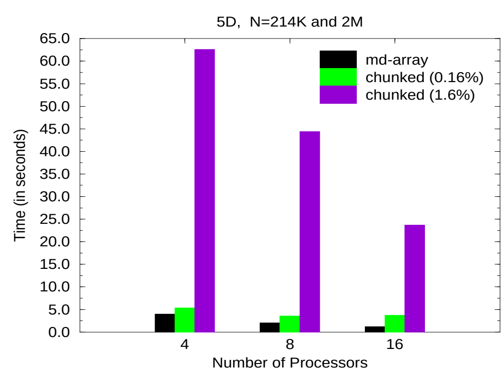
Table 6.7: Count of each parameter for 3D dataset with Chunksize I (32, 8, 16) and Chunksize II (64, 8, 32) on 4 processors

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

overall time. The multi-dimensional implementation is completely memory based and does not involve disk I/O. As densities are increased for higher dimensional data sets, I/O costs increase and overall time is much worse than the multi-dimensional array implementation. However, larger data sets and large dimensionalities cannot be executed using multi-dimensional arrays.



(a)



(b)

Figure 6.20: Comparison of multi-dimensional array and chunked cube implementations for data cube build (a) Currsale dataset, 608,866 tuples (b) 5D data set, 214K and 2.14M tuples

6.7 Summary

The design and implementation of a scalable disk based infrastructure for OLAP is described in this chapter. A bit encoded sparse structure introduced in Chapter 3 is used as the sparse data structure for storing data in chunks, providing storage compression and allowing operations to be performed on the compressed data. This allows for scalability in terms of data sizes and the number of dimensions. Data structure management is done to provide a disk-based method to allow numerous cubes to be materialized and provide optimizations for data reuse. Communication is asynchronous to overlap computation and communication. The algorithms are implemented on the IBM SP2 and performance results are shown for synthetic data sets containing up to 20 dimensions. The choice of chunk size determines the I/O cost and a larger chunk size results in less number of disk read and write operations, also reducing the chunk maintenance overhead.

Chapter 7

Query Processing

7.1 OLAP Query processing

OLAP operations such as roll-up, drill-down, pivot, slice and dice are the ways to interact with the data cube for multi-dimensional data analysis. Presence of pre-computed aggregates in the data cube improves query performance for queries based on summarizations. Presence of hierarchies, and calculations in the cube on hierarchies, provide better query performance for queries based on generalizations and consolidations. However, there is a trade-off between storage space required for aggregates and improved query performance. Clearly, for higher dimensional data calculating the full data cube is infeasible. Hence, it is essential to determine what aggregates are to be pre-computed and use optimizations to compute them. This depends on the nature and frequency of queries in a dynamic query environment. Efficient query evaluation techniques need to be developed for the OLAP environment. Techniques from query evaluation techniques in OLTP [Gra93] will be useful. In what follows we perform queries assuming the materialization of a complete data cube. 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 [HRU96, DRSN98, SSV96]. We believe these can be easily incorporated in our framework.

7.2 Aggregate Query Processing

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. Hierarchies can be used for data mining at multiple levels in attribute oriented approaches as discussed in a later section. Attribute-oriented induction [HCC93] is performed for association rule discovery by ascending a predefined concept hierarchy. 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.

7.2.1 Hierarchies

Figure 7.1 shows an example of a hierarchy for a 3D base cube with (Location, Time, Person) as the attributes (dimensions). Assume that hierarchies (City, Quarter, Group) are defined on these dimensions. Each dimension represents an attribute and each value in that dimension is a distinct value for that attribute. A hierarchy describing the grouping of some of the attribute values is defined in a *dimension hierarchy*. A hierarchy can have several levels. Figure 7.1 defines a first level hierarchy on each dimension where $(a_1, a_2, a_3) \in C1$, $(b_1, b_2) \in Q1$, $(c_1, c_2) \in G1$ and so on. The distributed dimension can have its hierarchy distributed over a processor boundary. A global index of the hierarchy is maintained at each processor. For example, cities in **C3** are distributed across P_0 and P_1 in the figure, containing the indices for (a_7, a_8, a_9). A consolidation operation would need a inter-processor communication if the consolidated value of $C3$ is required. Hierarchy at higher levels can be defined similarly. Support for roll up operations using consolidation can then be provided along each dimension. Drill downs can use them to index into the appropriate dimension index to provide a higher level of detail.

A hierarchy is defined over a dimension in the base cube. Hence the sub-cubes of the data cube can use the same hierarchy, since the dimensions in the sub-cubes are

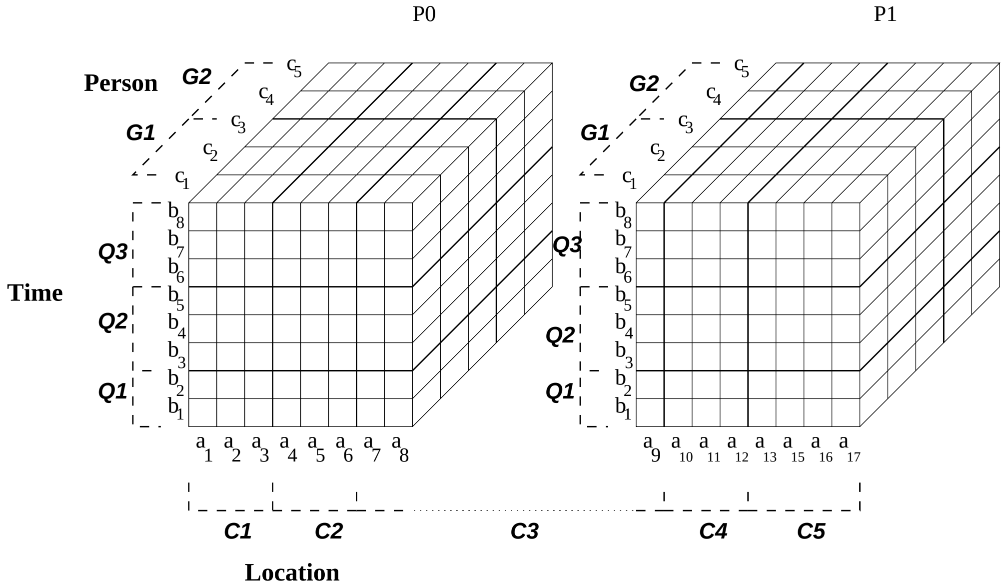


Figure 7.1: An example hierarchy defined on the dimensions for a two processor case

a subset of the dimensions in the base cube. Additionally, more dimensions can get distributed over the processors, and a distributed hierarchy needs to be maintained for these dimensions as well. In our example, in the cube (Time, Person) a 2D cube calculated from (Location, Time, Person), Time is a distributed dimension on which a distributed hierarchy needs to be maintained. As a result, each dimension has two hierarchies for it, one *local* and one *distributed*. Queries will choose the appropriate hierarchies when performing operations on these cubes.

Algorithm 7.2.2 and Algorithm 7.2.3 describe the algorithms for range queries and hierarchical consolidation query in the sparse chunked method of representing the multidimensional space.

7.2.2 Range Queries

Queries are answered by the appropriate materialized sub-cube in the data cube which has the correct set of dimensions. This is determined by checking the cube representations at the appropriate level of the cube lattice. The hierarchies are translated to the base level dimensions for retrieval of tuples in a global range space. Chunks that belong to the range on each processor are locally reported as the output. Figure 7.2(a) illustrates the range query in two dimensions, A and B, whose ranges have been specified. 16 chunks overlap the range specified, numbered 0 through 15. Chunks 5, 6, 9 and 10 are inclusive in the range and all tuples contained in them are retrieved. The other chunks are overlapping the range boundaries and hence each tuple needs to be retrieved to get its BESS offsets within the chunk to check for its inclusion in the output.

Algorithm 7.2.2 *Range query Algorithm on the Chunked data cube*

Input: Query(Dimensions $\mathcal{D}$, Ranges $\mathcal{R}$, Hierarchy $\mathcal{H}$)
Output: Set of tuples belonging to $\mathcal{R}$
On each processor
    Determine the appropriate cube $\mathcal{C}$, for dimension set $\mathcal{D}$
    for each hierarchy h_i in $\mathcal{H}$ for dimension i
        Determine the set of ranges at the base level of the hierarchy ($\mathcal{R}_i^0$) for this processor
    for each chunk c_j in $\mathcal{C}$ on this processor
        Retrieve chunk extents in global space
        if chunk c_j belongs to $\mathcal{R}^0$
            Output tuples for Query
        else
            if chunk c_j overlaps with $\mathcal{R}^0$
                Extract BESS values of c_j and check for inclusion in $\mathcal{R}^0$
end

7.2.3 Consolidation Queries

Consolidation provides an aggregate for the hierarchies defined on a dimension. For example, if it is desired to report the value for group **G1** for a city **C1** in the quarter **Q1**, then the corresponding values are aggregated by using the *dimension hierarchy* defined in the previous section. Some consolidation operations are *local*, as in they can be performed at a single processor, like **C4** in the figure. However,

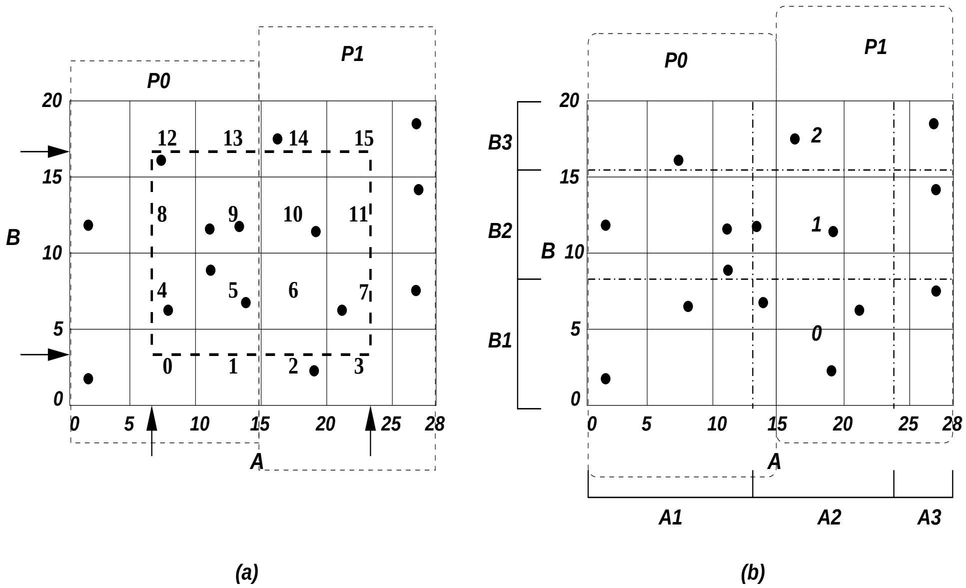


Figure 7.2: (a) Range query with chunks (b) Consolidation query illustrating the need for communication

some operations are *non-local*, like the values for city **C3** in any quarter will need the aggregates values from both the processors as shown in Figure 7.3. A inter-processor communication is needed to further aggregate the data to get the correct result.

Algorithm 7.2.3 describes the consolidation algorithm given a hierarchy on some dimensions. Each processor, does a mapping of its chunk ranges in each dimension to the appropriate hierarchy, aggregating the value to the cell. A hierarchy is defined in Figure 7.4 which shows the range definitions, which can either be uniformly defined or be non-uniform according to the hierarchy on the dimension. Chunk ranges identify the boundaries of chunk regions. We follow the uniform chunk range definition to keep a fast mapping of chunks on processors possible without the additional calculations of determining the chunk indices from indices stored for a non-uniform definition. A range can overlap chunk boundaries and consequently processor boundaries. For such overlapping chunks in the hierarchy, the BESS values are examined to determine if they belong to the query region. This happens only

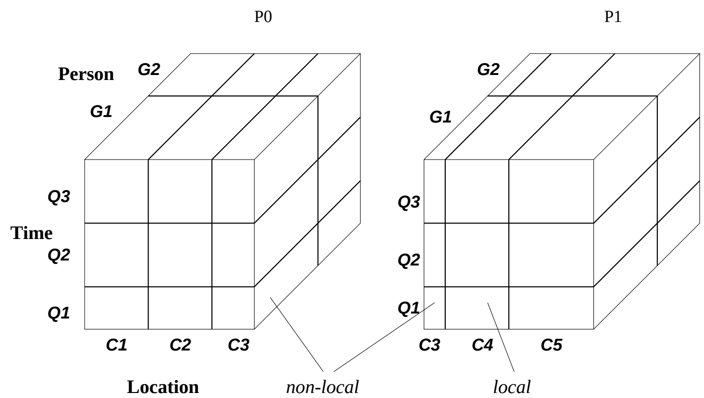


Figure 7.3: Consolidation for a level 1 hierarchy defined on the cube in Figure 7.1

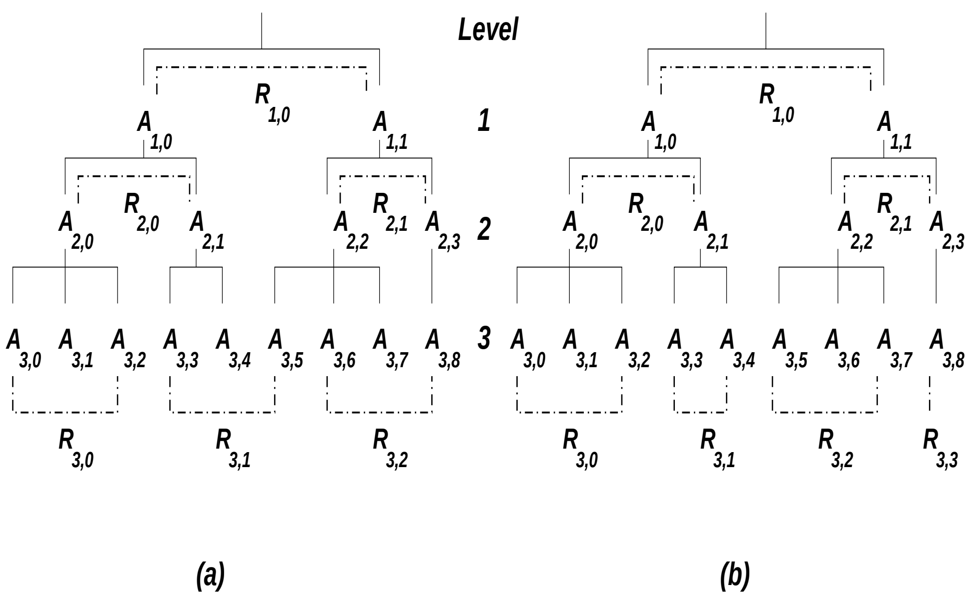


Figure 7.4: (a) Uniform ranges (b) Ranges according to hierarchy

at the lowest level of the hierarchy definition, when a range at a higher level of the hierarchy is transformed to dimension indices. Communication is generated since the target output cube at the desired hierarchy level is distributed across processors. Consider the hierarchy **A1, A2** and **A3** on dimension A in Figure 7.2(b). The hierarchies numbered 0, 1 and 2 for **A2** are distributed across processors and the tuples need to be communicated to the target processor where the resultant is stored. Hierarchies at multiple levels may not be contiguous in the layout in the attribute values at the lowest level. This is especially true with discrete valued dimensions where there is no ordering defined between attribute values and varied concepts are grouped together. This will translate into fetching the appropriate chunks included in the grouping and filtering the output as defined by the hierarchy.

Algorithm 7.2.3 *Consolidation Query Algorithm on the Chunked data cube*

Input: Query(Dimensions $\mathcal{D}$, Hierarchy $\mathcal{H}$)
Output: Set of tuples belonging to $\mathcal{R}$
On each processor
    Determine the appropriate cube $\mathcal{C}$, for dimension set $\mathcal{D}$
    for each chunk c_j in $\mathcal{C}$ on this processor
        Retrieve chunk extents in global space
        if chunk c_j belongs/overlaps dimension ranges of hierarchy at level i ($\mathcal{H}^i$)
            Aggregate the tuple values to the appropriate cell in the multidimensional space
            defined by $\cap(\mathcal{H}^i)$, communicating to the appropriate processor, if required.
end

7.3 Performance Evaluation

We again use the OLAP council benchmarks described earlier and define some sample hierarchies on each dimension. The queries in the benchmark represent the typical set of queries used in an OLAP scenario. These are encapsulated as point queries or range queries. A range is specified in a set of dimensions and values

Table 7.1: Sample level 1 and level 2 number of groups for hierarchies defined on the dimensions

Dimension	Hierarchy	
	Level 1	Level 2
Customer	9	4
Product	8	3
Channel	4	2
Time	8	2

are retrieved in the given range. We also used the synthetic data set (defined in Chapter 6) to evaluate the performance of queries in high dimensions.

Table 7.3 defines a sample hierarchy for the first two levels. For our experiments we have divided the size of a dimension equally among the number of groups defined for the hierarchy at that level. The allocation of actual attribute values to the groups at a higher level is contiguous for these experiments. However, these can be arbitrarily assigned, if desired.

Figure 7.5 shows the times for performing consolidation operation using the hierarchies on the dimensions. For Current Sales we have used the level 1 hierarchy on Product and Customer. For History Sales data, the hierarchy is used for all the dimensions, i.e. Product, Customer, Time and Channel. The figure for Budget also includes hierarchies along all dimensions, Product, Customer and Time. We see good scaling properties for each data set as the number of processors is increased.

Comparing the consolidation time for Current Sales and Budget, we observe that the time for Budget data is lower even though their base cubes are similar in size. The presence of a hierarchy along the Time dimension, which is the outermost dimension in the multi-dimensional array, improves the cache performance. Chunks of size $\lceil \frac{12}{8} \rceil \times \frac{900}{9} \times \frac{9000}{8}$, from outermost to innermost dimension, are consolidated for Budget, whereas chunks of size $9 \times \frac{900}{9} \times \frac{9000}{8}$ are consolidated for Current Sales. However, both scale well as the number of processors is increased.

Figure 7.6 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

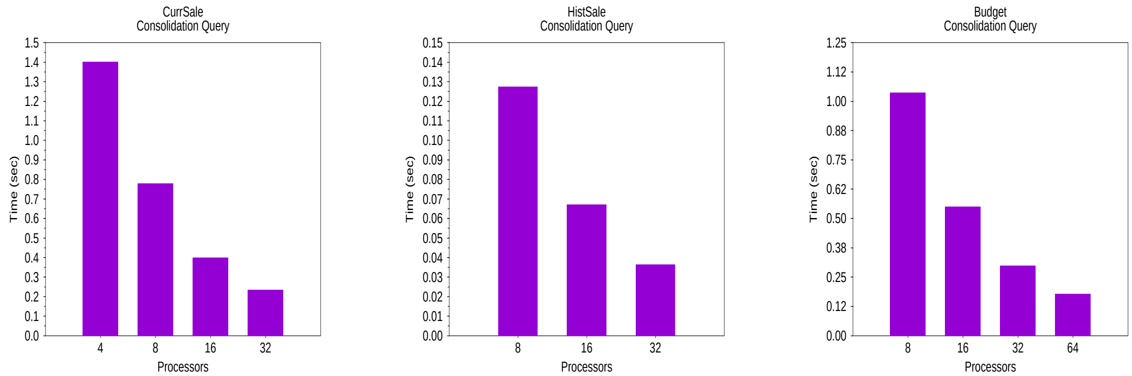


Figure 7.5: Time for consolidation query using the hierarchies defined in Table 7.3 for Current Sales, History Sales and Budget data sets

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.

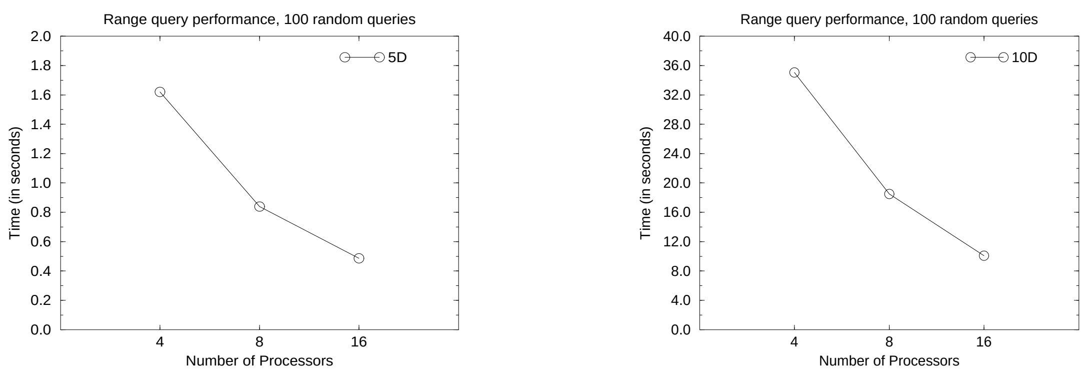


Figure 7.6: Range query performance for 100 random queries on the last 2 levels of the cube lattice for Datasets II (5D) and III (10D)

Hierarchical consolidation queries are expressed in terms of range queries with the appropriate aggregate operation. Performance of ad-hoc range queries gives an indication of how well the consolidation algorithm will perform.

7.4 Summary

OLAP queries are encapsulated as point or range queries in a multi-dimensional data space. Concept hierarchies defined on items can be translated into hierarchical definitions grouping items in a dimension together. A roll-up query uses the grouping information to aggregate data. Multiple chunks may be included at the base level of the hierarchy for a dimension, since uniform sized chunks are used. Appropriate, filtering of the dimensions must be done once chunks are retrieved for a query. Chunk overlap is exploited for queries that have overlapping regions in the multidimensional space. Memory reuse is at a finer grain, since chunks are cached in memory rather than whole cubes. Performance results are presented using OLAP style queries on both OLAP benchmarks and synthetic data sets in higher dimensions, showing good parallel performance.

Chapter 8

Data Mining on data cubes

Data mining techniques are used to discover patterns and relationships from data which can enhance our understanding of the underlying domain. A data mining tool can be understood along the following components : *data*, what is analyzed ; *statistics*, the kind of patterns that are searched for by the tool; *filtering*, how many of those patterns are to be presented to the user and the selection of those patterns; *visualization*, the visual representation used to present the patterns to the user; *interpretation*, what should the user think about when studying the pattern; *exploration*, in what order should the patterns be considered by the user; *process support*, how does one support data mining as a process as opposed to an exercise done only once.

Among the various techniques for data mining, two areas have received considerable attention in terms of analyzing large data sets and developing scalable algorithms. These are *Association Rules* and *Classification*. This chapter presents parallel algorithms for generating association rules and decision-tree based classification using the parallel OLAP infrastructure.

8.1 Association Rules

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 [Bha95] [HCC93] [HF95] 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. Support is defined as Prob$(A \cup B) = \frac{Number_of_transactions(A \cup B)}{Total_transactions}$ and confidence as Prob$(B|A)$ $= \frac{Number\_of\_transactions(A \cap B)}{Number\_transactions(A)}$. 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.

Attribute Focusing calculates associations between attributes by using the notion of percentages and sub-populations. The overall distribution of an attribute is compared with its distribution in various subsets of data. If a certain subset of data has a characteristically different distribution for the focus attribute, then that combination of attributes are marked as interesting.

An event, E is a string $E_n = x_1, x_2, x_3, \ldots x_n$; in which x_j is a possible value for some attribute and x_k is a value for a different attribute of the underlying data. E is interesting to that x_j's occurrence depends on the other x_i's occurrence. The "interestingness" measure is the size $I_j(E)$ of the difference between:
(a) the probability of E among all such events in the data set and,
(b) the probability that $x_1, x_2, x_3, \ldots x_{j-1}, x_{j+1}, \ldots x_n$ and x_j occurred independently. The condition of interestingness can then be defined as $I_j(E) > \delta$, where δ is some fixed threshold.

Another condition of interestingness, in attribute focusing depends on finding the optimal number of attribute values, n formally described as $I_j(E_n) > I_j(E_{n-1})$; $I_j(E_n) \geq I_j(E_{n+1})$; where $E_n = x_1, x_2, x_3, \ldots, x_n$. AF seeks to eliminate all but the most interesting events by keeping E only if the number of attribute values, n, is just right. Eliminate one or more x_i's and I_j decreases, include one ore more new x_i's to the string and I_j gets no larger. The convergence to n removes patterns like E_{n-1} and E_{n+1} which are less interesting than E_n and have information already contained by E_n. Hence, as a result of this the user does not have to drill down or roll up from a highlighted pattern, since the event descriptions returned are at their most interesting level.

Figure 8.1 shows an algorithm to compute the first measure of interestingness of the attribute focusing method using data cubes on the multidimensional array implementation which uses a one-dimensional partitioning. This algorithm can be extended to calculate the probabilities from the disk-based cube implementation, which also supports a two dimensional distribution. The appropriate 2D and 1D cubes need to be present at a processor to make the correct calculations. A higher dimensional calculation will involve more cubes and the appropriate probability calculation. Appropriate communication primitives have to be used for the various combinations of distributions for the sub-cubes to facilitate the local calculation on each processor.

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. Since 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

---

Algorithm 8.1.0 *Calculation of interestingness using data cubes*

Step 1. Replicate each single attribute sub-cube on all processors using a **Gather** followed by a **Broadcast**.
Step 2. Perform a **Reduce** operation of ALL (0D cube) followed by a **Broadcast** to get the correct value of ALL on all processors.
Step 3. Take the ratio of each element of the AB sub-cube and ALL to get $P(AB)$. Similarly calculate $P(A)$ and $P(B)$ using the replicated sub-cubes A and B.
Step 4. For each element i in AB calculate $|P(AB) - P(A)P(B)|$, and compare it with a threshold δ, setting AB[i] to reflect the association.

B is replicated on all processors. Figure 8.1 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.

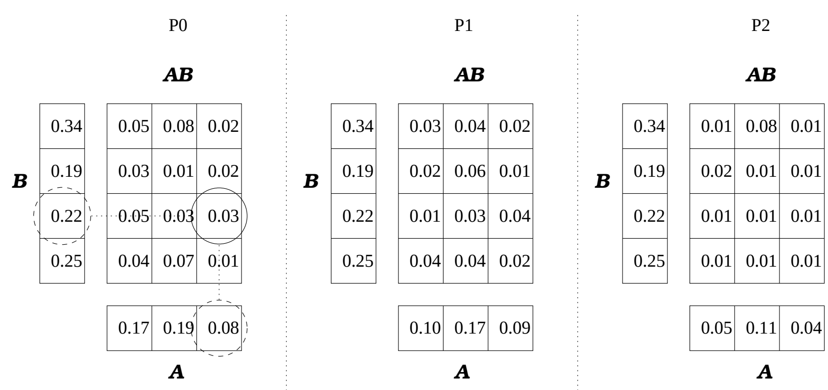
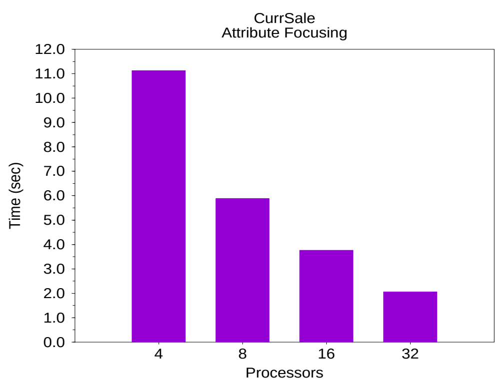


Figure 8.1: Use of aggregate sub-cubes AB, A and B for calculating "interestingness" on three processors

8.1.1 Performance Evaluation - Associations

We used a few data sets from the OLAP council benchmark. In particular, Current Sales (3D), Budget(3D) and History Sales (4D)(as described in previous sections) are used to illustrate the performance.

We perform 2-way association calculations by performing attribute focusing for all combinations of two attributes. For example, for a 3D cube, ABC, interestingness measure will be calculated between A and B, A and C and between B and C. Typically, a few different δ values are used for analysis of data, to vary the degree of association between the attributes. We run each calculation for 20 different δ values, ranging from 0.001 to 0.02 in steps of 0.001. Figure 8.2 shows the time for these calculations on Current Sales data on 4, 8, 16 and 32 processors. We observe good speedups as the number of processors is increased. Also, communication time increases as the number of processors is increased. Replication of the 1D cubes involve communication, and with larger number of processors, the terms involving p in the complexity of the collective communication operations (refer to Table 4.1) increase. However, since communication time is marginal compared to computation times, the increase does not affect the overall scalability.



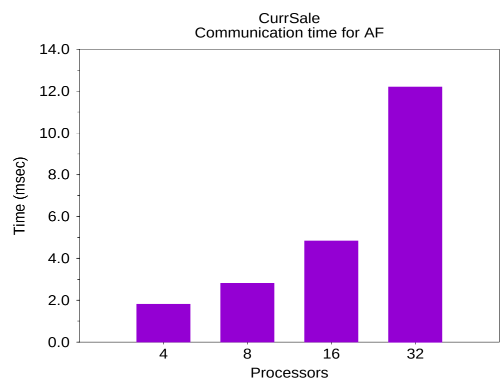


Figure 8.2: Time for (a) attribute focusing time for 20 different δ values for Current Sales data, (b) associated communication cost (in milliseconds) on 4, 8, 16 and 32 processors

Figure 8.3 shows the attribute focusing calculation time for the History Sales data. The associated communication time is slightly higher in this case since 4 1D arrays are replicated, instead of 3 in the 3D case. The communication time (which is shown in milliseconds) is still at least an order of magnitude smaller than the computation time. The size of the cubes is smaller than the Current Sales data and hence the magnitudes of time are much smaller. Hence, the speedups are more modest as we increase the number of processors.

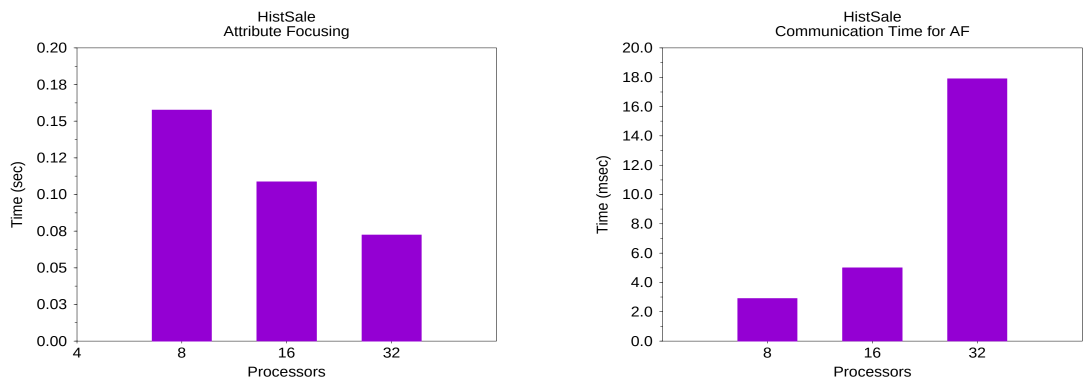


Figure 8.3: Time for (a) attribute focusing for 20 different δ values for History Sales data, (b) associated communication cost (in milliseconds) on 8, 16 and 32 processors

Figure 8.4 shows the speedups we observe for the Budget data for attribute focusing calculations. The communication costs are similar to the ones observed in the Current Sales data since they involve similar sized arrays.

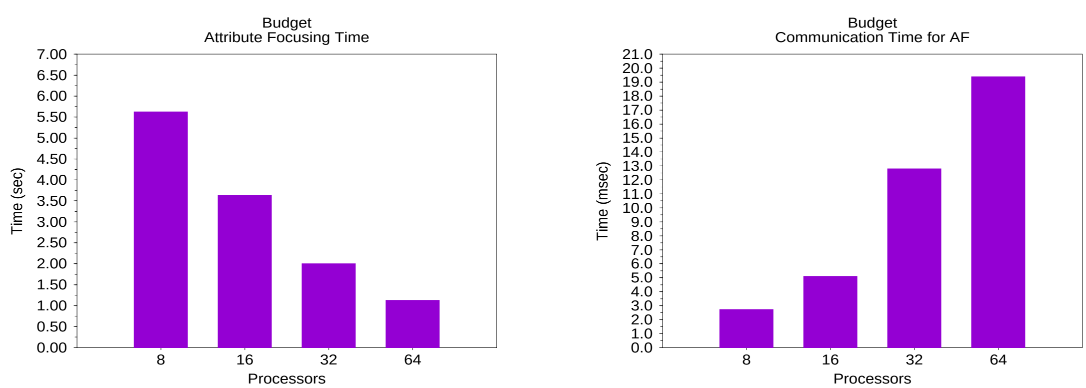


Figure 8.4: Time for (a) attribute focusing for 20 different δ values for Budget data, (b) associated communication cost (in milliseconds) on 8, 16, 32 and 64 processors

8.2 Classification

Numerous techniques have been developed in machine learning with the goal of deriving a concept description from pre-classified examples. This leads to inducing a generalized description of a class given a set of positive and negative samples from that class. Classification is an important problem in data mining. 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. Sample applications include medical diagnosis, financial analysis and forecasting, engineering fault diagnosis, information retrieval and classification of visual representations.

Several classification models have been used in the past, notably neural networks [MST94], genetic algorithms [Gol89], decision trees (CART [BFOS84], C4.5 [Qui93]) 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 [MST94].

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 [MST94].

Table 8.1 is an example training set with three attributes, Age, Car color and Gender, and a class attribute. Age is a continuous attribute, whereas both Car color and Gender are categorical attribute. For a categorical attribute having c

Table 8.1: 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

distinct classes it is assumed that the splitting decision forms c partitions, one for each of its values. Figure 8.5(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 [BFOS84]. 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 8.6. At Node 0, the attribute *Gender* yields the optimal $gini_{split}$ value of 0.214. This creates a split with one partition with M

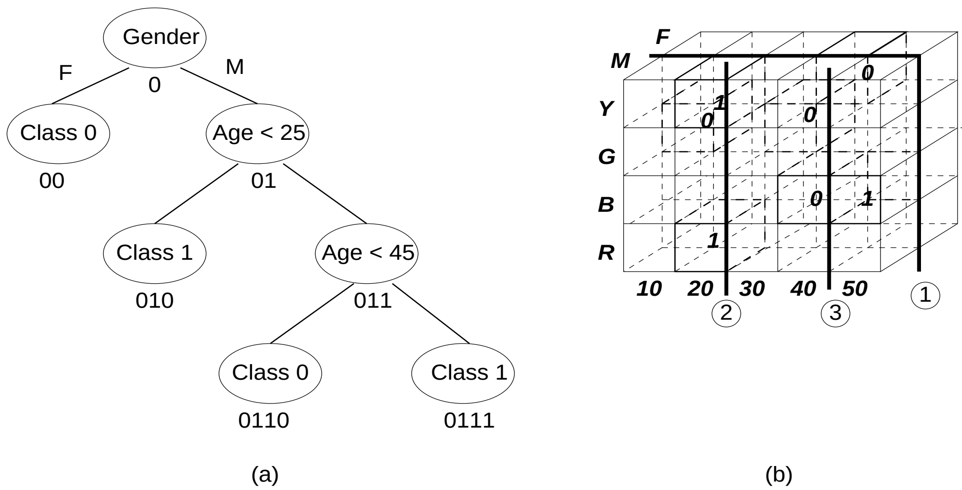


Figure 8.5: (a) Classification tree for training set (b) Classification tree embedded on the base cube

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 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.

Previous work in classifying large data sets has been to use sampled data sets or multiple partitions of the data set [MST94, CS93]. Recent work has focused on using the entire data set, in classifiers like SLIQ [MAR96] and SPRINT [SAM96]. A parallel classifier in the same spirit has been developed in ScalParC [JKK98].

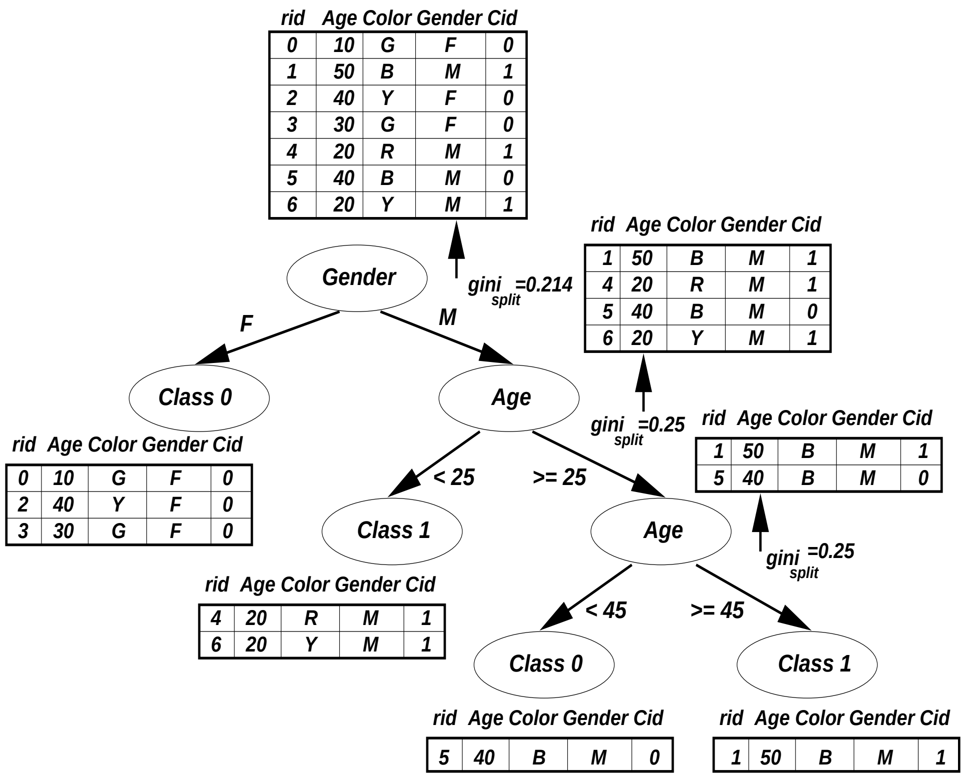


Figure 8.6: Gini index calculation for the attributes and node splitting

A data parallel approach has been described in [Kuf95] for parallel decision trees. A designated *host* processor coordinates the split point calculation using information gain measure from the *worker* processors Classifiers like CART [BFOS84] and C4.5 [Qui93] perform sorting at every node of the decision tree, which makes them expensive for large data sets since disk-based sorting is required. 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.

8.2.1 Classification on the multidimensional cube structure

The motivation comes from using a multidimensional chunk based infrastructure for OLAP and multidimensional analysis for higher dimensions which optimizes the building of the data cube operator. We observe that, gini index calculations are done on the 1-dimensional aggregates of the data cube structure. For a data set with k attributes (dimensions) the cube computes all aggregates with $k-1, k-2 \ldots 0$ attributes. A total of 2^k aggregates are calculated. Multidimensional analysis, OLAP queries and association rule mining are performed efficiently using the materialized aggregates in the data cube. In this model an attribute is treated as a dimension, and records are points in a multidimensional space. Dimensional operations can be performed more efficiently in such a model since a structure is imposed in the storage of data.

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 multi-dimensional 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 the number of records belonging to each class. Therefore for each aggregate we store the number of records in each class. Figure 8.7(a) gives an example training set with two dimensions, A, a continuous dimension and B a categorical dimension and two class values 0 and 1.

Figure 8.7(b) is the corresponding multidimensional model. The continuous dimensions A is stored in the sorted order. The aggregates store the number of records mapping to that cell for both classes 0 and 1. To calculate the $gini_{split}$ for the continuous attribute attribute A it is now easy to look at the A aggregate and sum the values belonging to both classes 0 and 1 on both sides of the split point under consideration to get the count matrix. Gini index calculation is done on an attribute list which in the case of a multidimensional model is a dimension. Count matrix is repeatedly calculated on the sorted attribute list which is readily available in the cube structure as a higher level one dimensional aggregate. Each dimension is sorted in the dimensional structure as shown in Figure 8.7(b).

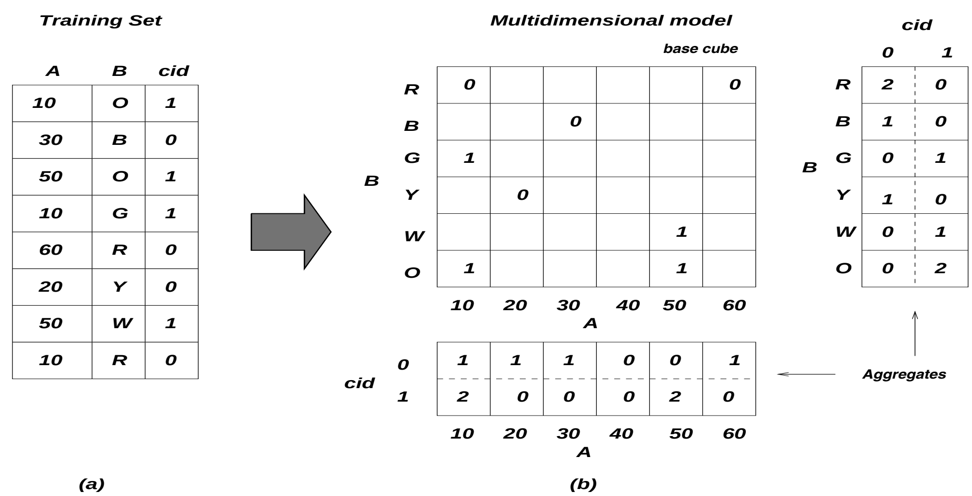


Figure 8.7: (a) Training set records (b) corresponding multidimensional model

Figure 8.8 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. The gini index is calculated using the 1-dimensional aggregate values in each dimension. For example, in node 0 shown in the figure, the gini value for the split 25 in the dimension Age is calculated as $(3/7)*[1-((2/3)^2+(1/3)^2)]+(4/7)*[1-((3/4)^2+(1/4)^2)] = 0.4$. Gender is picked up as the split attribute with minimum $gini_{split} = 0.214$. Two nodes are creates (00 and 01) and Gender is split between the nodes. However, the 1-dimensional aggregates are calculated taking into account the split at Gender. This is shown as the split 1 in the multi-dimensional representation. Node 00 calculates the aggregates with Gender as M for Car Color with Y, G, B, R and Age with $10, 20, 30, 40$ values. Similarly, node 01 computes the 1-dimensional values for Car Color and Age with Gender as F. Gini computations are done on each of these to find the split attribute for the node. Since node 00 has all records belonging to class 0, it is marked as classified and no computations are needed further on this node. However, node 01 gets split on the Age attribute with minimum $gini_{split} = 0.25$.

The challenge is to calculate the one dimensional aggregates efficiently and keep them updated to reflect the partitions after each split. Figure 8.9(a) shows the lattice for a 5D cube with attributes A, B, C, D and E. A *complete* cube calculates all the possible aggregates. Since for classification we need the 1 dimension aggregates with the class id counts, we can optimize the calculation by calculating some intermediate values. There are multiple paths for aggregate calculations as can be seen from the lattice in Figure 8.9(a).

The simple method of computing each level 1 aggregate is to do so from base data. Let us examine if we can do better, by calculating some intermediate values. Several optimizations have been used to compute the intermediate aggregates in the OLAP domain. An aggregate is calculated from the *smallest parent*. For example A can be calculated from any of AB, AC, AD or AE at the next level and should be computed with the least cost among them. Similarly, each aggregate can be computed from aggregate at the next level. A complete cube gives the opportunity to choose the best schedule if all the intermediate aggregates are also materialized as in some cases of OLAP. As we have observed earlier, classification requires only the top level aggregates we can optimize on the number of aggregates materialized. The number of *non-essential* aggregates must be minimized. To compute the one dimension aggregate values we observe from Figure 8.9(b) and Figure 8.9(c) that only a subset is required.

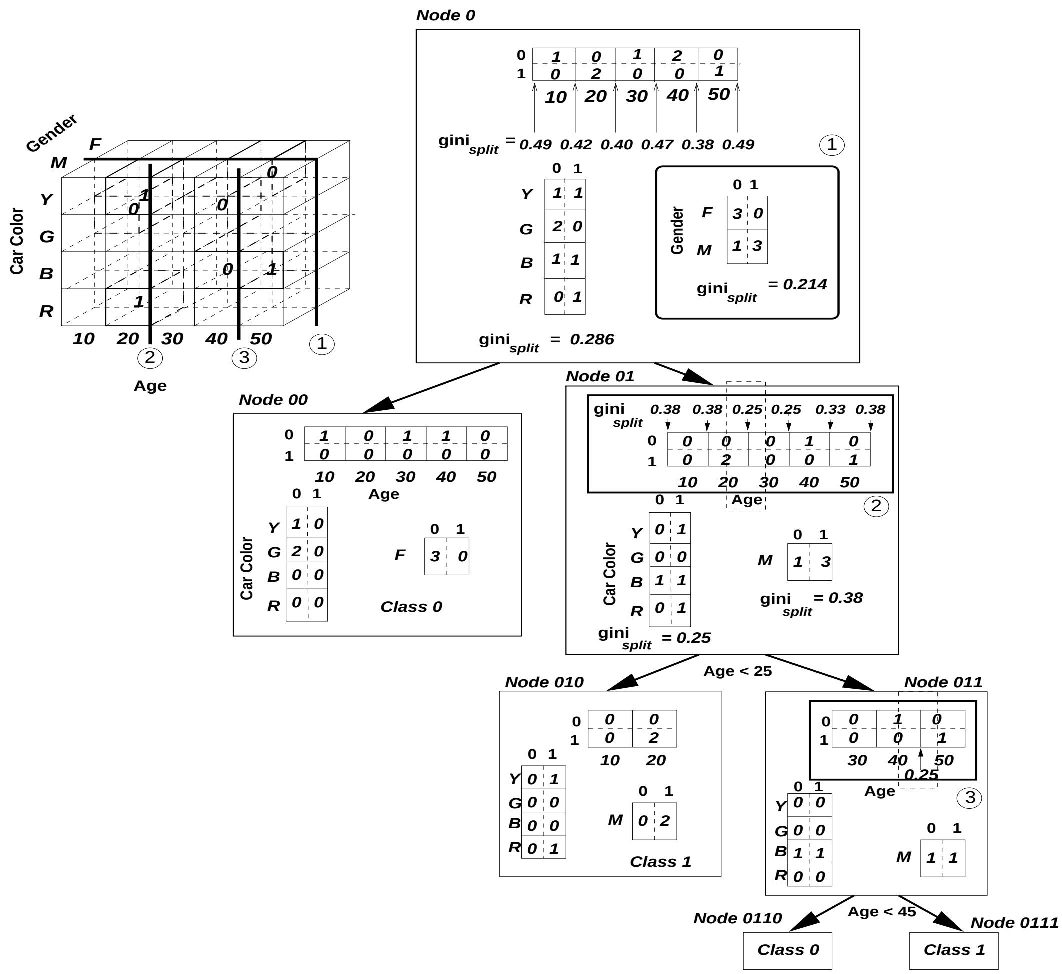


Figure 8.8: Gini index calculation for the attributes and node splitting with multidimensional aggregates

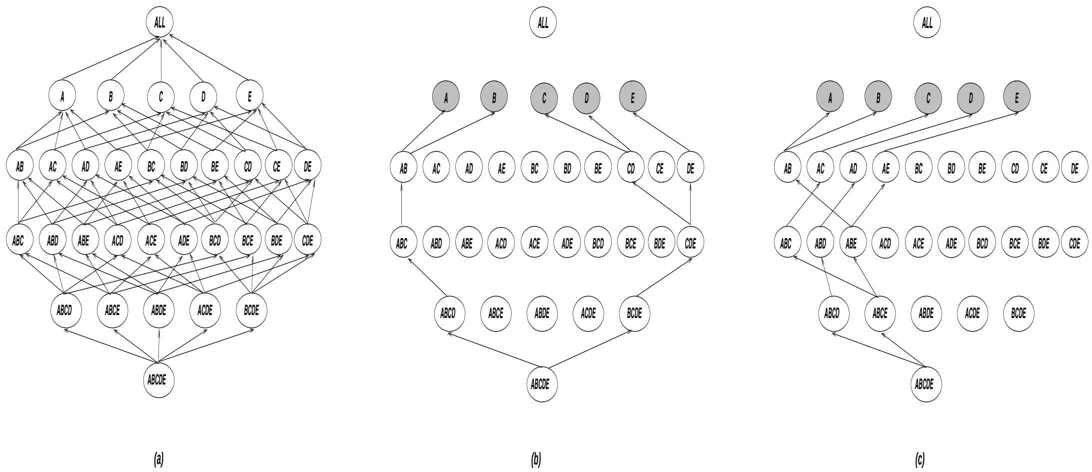


Figure 8.9: (a) Lattice for cube operator (b) Aggregates needed for classification (Scheme I) (c) Aggregates for classification (Scheme II)

Attribute-coverage **scheme**

Consider a k dimensional data set. The lattice has $k+1$ levels, numbered $0, 1 \ldots k$ with $k, k-1, \ldots 0$ attributes. We need to consider the first k levels for classification. At each level, a minimum number of aggregates must encapsulate all the attributes. This will lead to a overall minimum number of non-essential aggregates. At level 0, we have the base data set or the base cube. Level 1 has aggregates with $k-1$ attributes. To cover all k attributes of the data set at a level l, $\lceil \frac{k}{l} \rceil$ aggregates can do the job as illustrated in Figure 8.9(b). This leads to a total of $\sum_{l=2}^{k-1} \lceil \frac{k}{l} \rceil$ intermediate aggregates. We call this the **attribute-coverage** scheme.

Left-oriented **scheme**

For parallel computation, data is distributed across processors to distribute computation. Usually, for multidimensional systems a dimension or a combination thereof is distributed. Assume that a single dimension, perhaps the largest one, is distributed for each aggregate sub-cube, across P processors. A choice of intermediate cubes above, although the minimum, will lead to data redistribution since not all have the same first dimension. For example, $ABCDE \rightarrow BCDE$ will result in aggregation of A and distribution of B but $ABCDE \rightarrow ACDE$ will be a local aggregation on each processor. Typically, communication is much more expensive than computation on coarse-grained parallel machines, hence we would like to avoid redistribution, especially since these calculations need to be done several times. Another possibility which takes this into account is shown in Figure 8.9(c). More about parallel computation of aggregates will follow in a later section. At level 1, one of two attributes left by fixing the first $k-2$ attributes will represent all k attributes. This leads to $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$ aggregates at level 1. For example, with $ABCDE$ at level 5, we have $ABCD$ and $ABCE$ have all 5 attributes between them. Similarly, at level 2, the first $k-3$ attributes are fixed and two are chosen from the remaining 3 in $\begin{pmatrix} 3 \\ 2 \end{pmatrix}$ ways. Generalizing this, at each level l, there are $\begin{pmatrix} l+1 \\ l \end{pmatrix}$ aggregates

minimally. The total number of non-essential aggregates is then $\sum_{l=0}^{k-1} \begin{pmatrix} l+1 \\ l \end{pmatrix} = \sum_{l=0}^{k-1} l = (k-1)k/2$. We call this the **left-oriented** scheme. For $k=5$, we need 10 non-essential aggregates at levels higher than 1.

A split of an attribute partitions the dimension into two partitions for a continuous attribute and into m partitions for a categorical attribute with m values. This results in invalidation of some of the aggregation calculations. If a dimension l is partitioned, the aggregates containing attribute l can be partition it and the count values are still valid. However, for aggregates which do not contain the attribute l, that is the ones which have aggregated this attribute, the count values are no longer valid. They need to be recalculated. Recalculation depends on which attribute is partitioned in our scheme since we only maintain a subset of aggregates.

Figure 8.11 shows the recalculation aggregates in shaded boxes for a 5 dimensional data set with each of A, B, C, D and E attributes being partitioned using

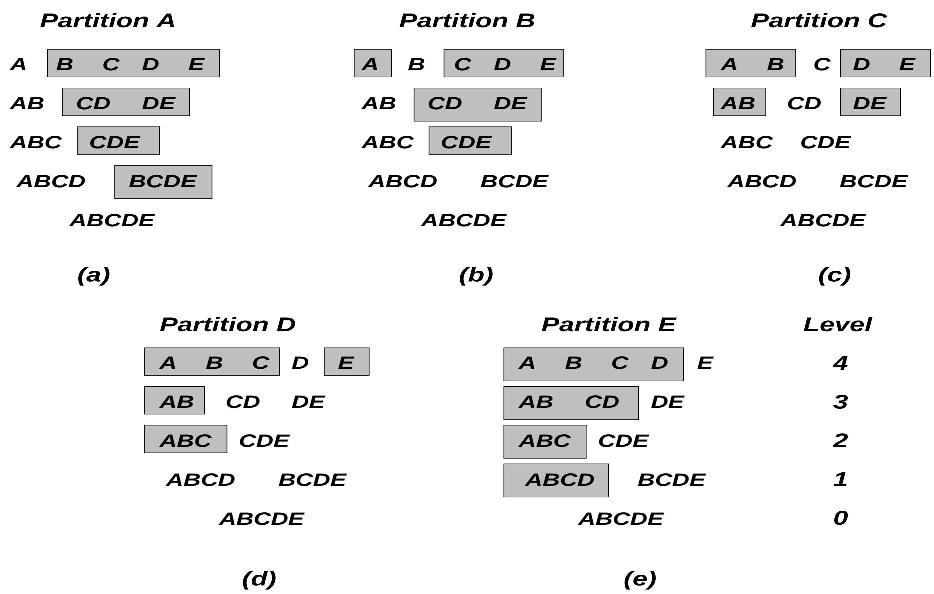


Figure 8.10: Recalculation is done for the aggregates in shaded boxes when the attribute split is (a) A (b) B (c) C (d) D (e) E for attribute-coverage scheme

the scheme in Figure 8.9(c). It can be observed that the aggregate recalculation depends on the attribute that is partitioned. Least recalculation is needed for A and the most for E. This is a result of the dimension order of the aggregates stored. All intermediate aggregates have the attribute A in them and hence can be partitioned at a node. The frequency of the other attributes decreases in the order of their appearance in the base cube. Figure 8.10 shows the recalculation needed for the the scheme in Figure 8.9(b). It treats attributes more uniformly for recalculations. However, if the dimensions on the extremes of the dimension order are split they involve more recalculation since the intermediate cubes are picked up from the extremities to fill in the full encapsulation of the attributes. For example, for $ABCDE$, A and E are on extremities and if A is split the aggregate sub-cubes chosen from the right sides of the lattice at each level $BCDE, CDE, CD, DE$ need to be recalculated. Similarly, when E is split, the aggregate sub-cubes from the

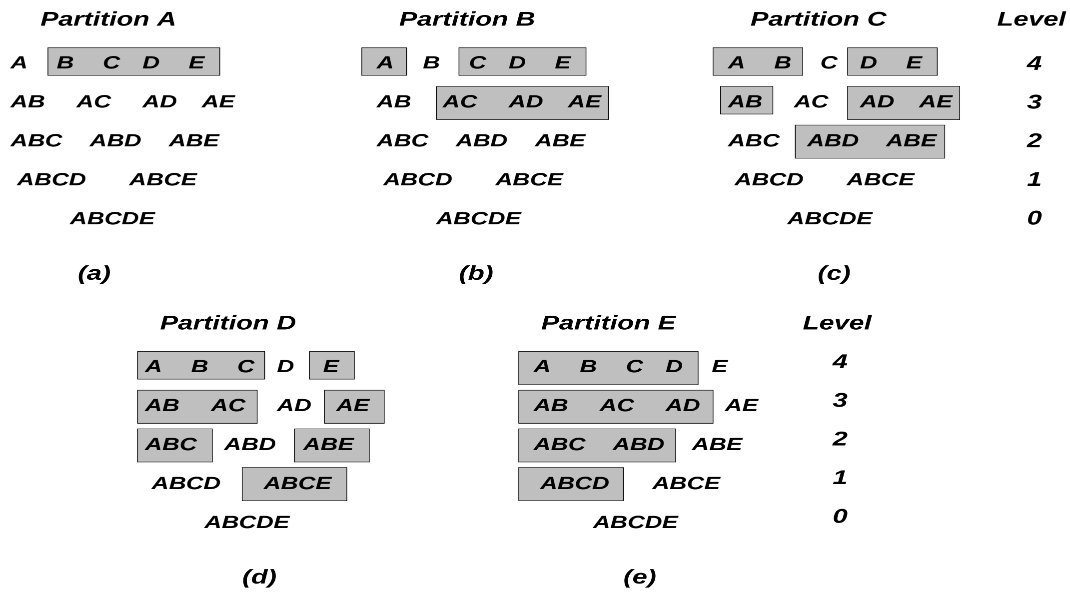


Figure 8.11: Recalculation is done for the aggregates in shaded boxes when the attribute split is (a) A (b) B (c) C (d) D (e) E for left-oriented scheme

left side $ABCD, ABC, AB, CD$ are recalculated. The least number of levels to be recalculated is for the middle attribute and it increases towards both ends. It can be seen from Figure 8.10 that C has to recalculate on level 3, B, D have to recalculate on level 3 and 2, and A, E have to recalculate on 3, 2 and 1. Since, the number of cubes is the minimum in this scheme it is preferable for sequential computation. However, for parallel computation, if dimension A is distributed across processors, this scheme results in more communication, whereas the earlier scheme will result in no communication even though a few more cubes are to be materialized. There are other combinations of sub-cubes that can be materialized but they fall in between the two cases presented above.

For each node of the classification tree there might be re-computation of multiple sub-cubes creating many additional cubes, one for each active node at the current level in the classification tree. Although, precomputation saves time to calculate the 1-dimensional aggregates, it occupies space since there need to be precomputed aggregates for each active node. We consider the simple method of computing the one dimensional aggregates from the base cube. Taking advantage of the data organization in a multidimensional system this can be done efficiently. Figure 8.12 illustrates the computations needed from the base cube $ABCDE$ to get the one dimensional aggregates at each node to be split. At node 0, all calculations are done. After gini index calculation, attribute A is selected to be split, hence at node 01, A' is straightforward to calculate by readjusting the extent of the dimension. However, the aggregate count values in each class for the other one dimensional aggregates, B, C, D, E are invalidated and need to be recalculated. Similarly, at node 01, C is used for the split and hence C' get calculated by readjusting the dimension and the other aggregates A', B, D, E have to be recomputed.

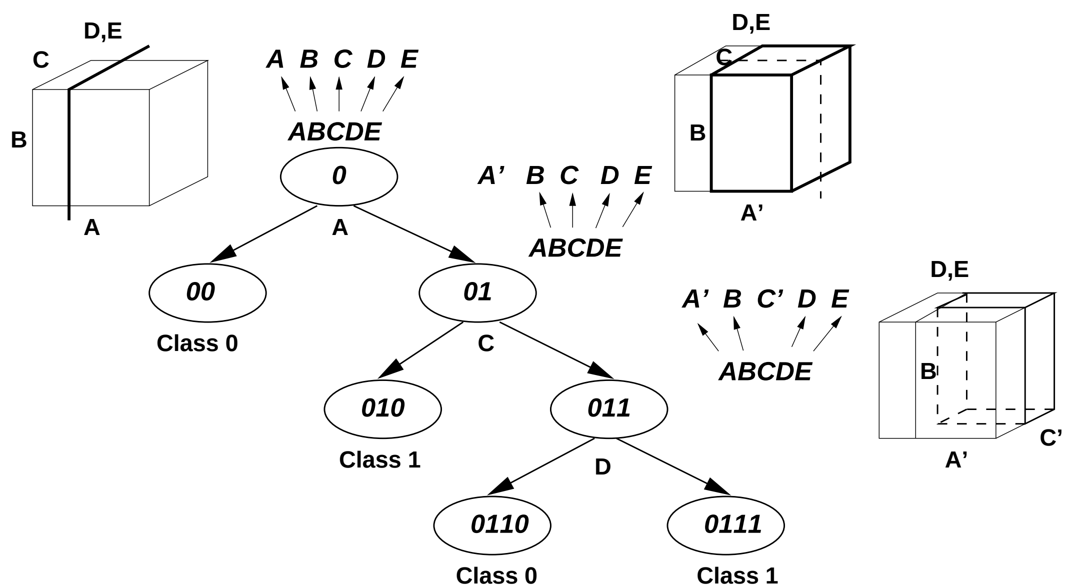


Figure 8.12: Computations done from the base cube $ABCDE$ at each split

8.2.2 Parallel Classification Algorithms

In this section we will briefly discuss the previous efforts in parallel classification [Fif92, SAM96, JKK98], significant of which are parallel SPRINT [SAM96] (on IBM SP2) and ScalParC [JKK98] (on Cray T3D) since they do not require resorting at each node. A parallelization of SPRINT on shared memory multiprocessors is presented in [ZHA99]. Both data parallel and task parallel versions of the algorithms are presented. Parallelization of tree methods, especially multidimensional binary search trees (k-d trees), quadtrees and octrees for hierarchical methods have received considerable attention in the areas of scientific processing [AfAGR96]. Classification trees fall under a similar paradigm.

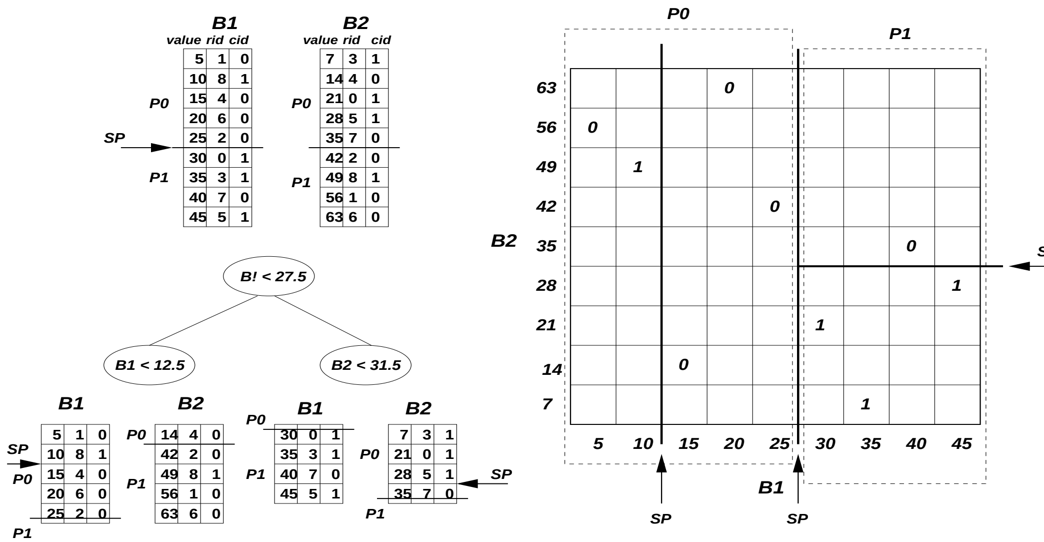
The parallel version of SPRINT partitions each attribute list by sorting each one using a parallel sort with probabilistic splitting. This gives approximately equal sections to each processor of each sorted attribute list. Split points are found by first doing a **prefix sum** operation for the count of values below and above the first split point on each processor followed by building the count matrices locally on each processor. After calculating the gini index locally, the processors communicate to determine which split point has the lowest value. Since there is only one count matrix for categorical attributes, they are constructed locally and one processor collects the global count and calculates the gini values.

Each processor splits its splitting attribute list locally. *rids* are collected from all processors to build the hash table on each processor. The non-splitting attribute lists then probe this table with *rids* to determine the split locally. This has a space complexity of $O(N)$ and makes it unscalable in memory requirements. ScalParC on the other hand maintains a distributed hash table for the splitting phase which has a space complexity of $O(\frac{N}{P})$. However, the splitting phase is slightly different in their case. A distributed hash table, called the *node table* is maintained by hashing a *rid* with a hash function $h(rid) = (p = rid\ div\ \frac{N}{P} l = rid\ mod\ \frac{N}{P})$, where the first field is the processor number and the second is the local index on that processor. After the splitting decision, each processor uses the *rids* of the split attribute to construct hash buffers with $(l, child)$ entry destined for all processors p calculated by the hash function. An **all-to-all personalized** communication phase exchanges these to update the distributed node table. For each non-split attribute list, the distributed node table is queried by filling out a communication buffer with *rids* (enquiry buffer) and sending it to the processor p which then fills it with the child label and sends it back. Two **all-to-all personalized** communication phases are needed to achieve this.

The size of the node table at each level is usually of size $O(\frac{N}{P})$. Also, each processor sends $O(\frac{N}{P})$ elements from each of the non-splitting attribute list. However, there are cases when a processor has to hash all global *rids* at some level and the other processors need to send $O(N)$ elements to be queried by the node table on a single processor. Figure 8.13(a) shows this scenario at level2. [JKK98] contends that such scenarios are rare and on an average the communication buffers required are $O(\frac{N}{P})$. However, the worst case complexity is $O(N)$. For d dimensions the worst case communication complexity is thus $O(Nd)$. Computation is needed to compute the count matrices for each dimension. The worst case complexity for this step is also $O(Nd)$. [JKK98] asserts that *no* parallelization scheme which sorts its elements once can avoid this unscalability. We contend that using our *multidimensional* scheme for classification this can be avoided. In Figure 8.13(b) the multidimensional method is shown which does not need to enquire the node table on other processors and hence avoids this potential unscalable possibility. Dimension $B1$ is partitioned equally on processors $P0$ and $P1$. At the first split at $B1 = 27.5$ two nodes are created and each processor performs the splits locally after the prefix operation on the count values to calculate the count matrix locally for the $B2$ dimension. The size of message for the communication is of the order of the dimension size. There is no communication needed for splitting the attribute $B1$.

8.2.3 Proposed Parallel Classification on the Data Cube

Parallel classification on the multidimensional base cube is similar to the sequential classification algorithm, except the fact that each processor calculates the aggregates locally and then needs to update counts for each partitioned dimension from other processors. This is done by a communication phase which calculates a **prefix sum** on the first split point for continuous attributes since each split point is evaluated



(a) Attribute List Method

(b) Multidimensional aggregate method

Figure 8.13: (a) A worst case scenario for sorted attribute lists at level 2 of the tree for ScalParC (b) Multidimensional approach does not need to probe the node table

by computing the $gini_{split}$. For categorical attributes, however, a processor can sum the counts for a value of the categorical dimension and calculate the gini index. Note that separate processors can compute the gini value for a categorical attribute.

Dimension partitioning and gini index calculation

Figure 8.14 shows a one dimensional partitioning of the multidimensional base cube with two dimensions, A and B, and the associated local aggregate calculations. Each processor builds the aggregate locally, and for each continuous attribute determines the gini index for the values that lie in its partition. For non-distributed dimensions, each processor locally calculates the aggregates and then does a **Reduce (sum)** on the aggregates for each such dimension. Each processor then works on a part of the aggregate list to calculate the gini index values. This results in computation being partitioned between processors. If the size of the dimension i is d_i, then each

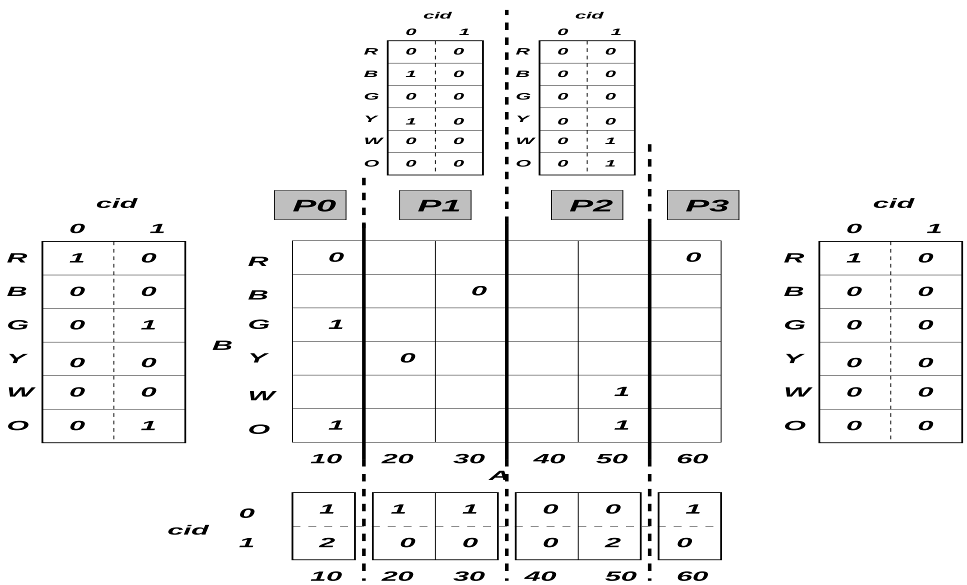


Figure 8.14: One dimensional partitioning using the largest attribute size

processor $P_j, 0 \leq j < P$, calculates the gini index from $j(d_i/P)$ to $(j+1)(d_i/P)$ for each dimension. The minimum gini index is computed at each processor and a **Reduce (minimum)** is done to get the minimum index and the split attribute on each processor. An alternative is to do the entire gini index computation on each processor, to remove the extra communication step. The complexity of computation of gini indices at a level k of the tree is $O(n_a \mathcal{N}_n^k)$, where n_a is the number of attributes and $\mathcal{N}_n$ is the number of active nodes at a level. For categorical attributes also the computation can be distributed similarly if a **Reduce (sum)** is performed on the aggregated dimensions. Otherwise a **Gather** operation gets the aggregates on a single processor which does the calculation. Since dimensions with categorical attributes are usually small we use the latter approach.

Another alternative is to perform a two dimensional partitioning by selecting the two largest dimensions and partitioning the multidimensional base cube on a 2-dimensional processor grid as shown in Figure 8.15. This may provide better load balancing in some cases. The communication pattern for the two distributed dimension changes but remains the same for the rest. The non-distributed dimensions get the aggregates locally and do a **Reduce (sum)** to sum each dimensional aggregate independently on each processor. Each processor then works at calculating the gini index in contiguous portions as in the one-dimensional partitioning case. For the distributed dimension A in the figure the reduce for the dimension on processor P_j is done on processor $(P_j/P_Y)P_Y$, where P the number of processors is divided into a two dimensional grid in the dimensions X and Y as $P_X \times P_Y$. The processors $P_0 + i * P_Y, i = 0, \ldots P_X$ then calculate the local sums of class ids and need a **Parallel Prefix (sum)** to get the values updated across processor boundaries with the number above and below in each class. Each of these processors then calculates the gini index for the local dimension values.

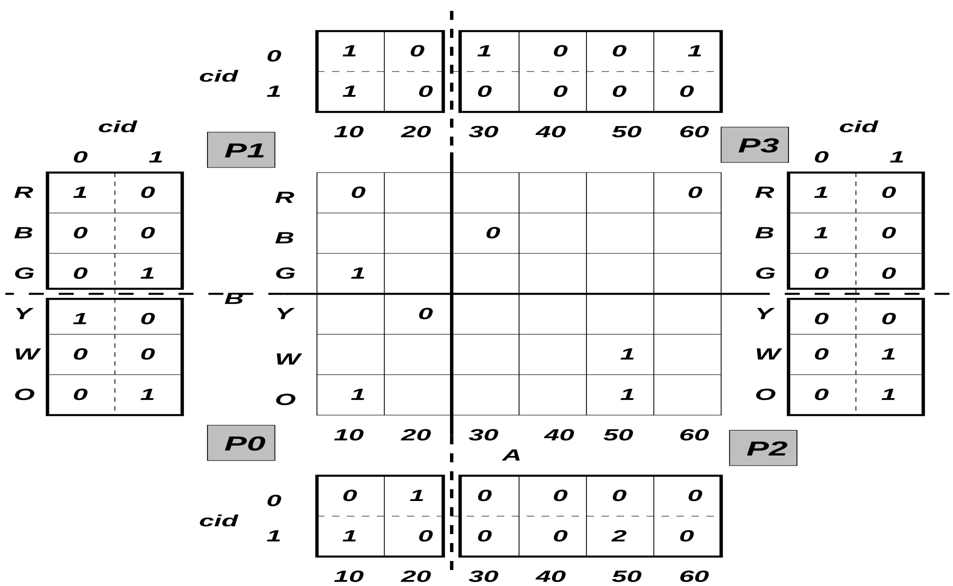


Figure 8.15: Two dimensional partitioning using the two largest attribute sizes

Similarly, for dimension B a **Reduce (sum)** operation is done on processor $P_j \% P_Y$ and again a **Parallel prefix (sum)** is done between processors $P_0 + i, i =$

$0, \ldots P_Y$ to calculate the gini index locally.

Node splits

Once the $gini_{split}$ is calculated for all attributes (dimensions), each processor picks out the minimum and the attribute it is related to. A **Reduce (minimum)** gets the minimum and the attribute which is the *split* attribute. Each processor then partitions the split attribute locally. Each processor maintains a mapping of dimension indices to the node they belong to, as they are split. In other words, a global tree representation is present on each node. Each node keeps the dimensional boundaries of the dimensions split, defining a hypercube the node encapsulates. The aggregates are then calculated within the node boundaries. The communication required is for each dimension, but now there are multiple count values, one for each active node which has not yet been classified.

Let n_a be the number of attributes (dimensions) divided into n_d, the set of distributed dimensions and n_u, the set of non-distributed dimensions. For non-distributed dimensions these need to be aggregated using a **Reduce (sum)** on all processors to distribute the computation of the gini index calculation. The amount of data communicated at a level k of the classification tree is $O(\mathcal{N}_n^k|D_u|)$, where $\mathcal{N}_n^k$ is the number of active nodes at a level k, and $|D_u|$ is the sum of dimension sizes of dimensions $d_i \in n_u$. For a distributed dimension a local aggregation is done for each dimension for each node and then a **Parallel prefix (sum)** is done. The data communicated for this operation at level k is $O(\mathcal{N}_n^k n_d)$. The total data communicated at level k of the classification tree is then $O(\mathcal{N}_n^k(|D_u| + n_d)) = O(\mathcal{N}_n^k|D_a|)$, $|D_a| = |D_u| + |D_d|$, because the number of split dimensions is usually one or two. This is the case for our implementation where the optimization of message-coalescing, reduces the number of communication messages by combining the messages as one buffer.

For the scalable record based classification algorithm (ScalParC) the complexity of communication at each level is $O(n_a \frac{N}{P})$ at each level and $O(n_a N)$ in the worst case, where N is the size of the training set. The number of active nodes at any level is much smaller than the size of the training set. Also, the sum of dimension sizes,

$|D_a|$, is also much smaller when compared to N. Thus, $\mathcal{N}_n^k \times |D_a| < N$, for a large N. This makes the overall communication requirement of our multidimensional classification algorithm better. Computation of the 1-D aggregates and the gini index calculations has the time complexity $O(\frac{N}{P} + \mathcal{N}_n D_a)$ in our method, whereas ScalParc requires $O(\frac{N}{P} n_a)$, to compute the gini indices and $O(N n_a)$ in the worst case.

Figure 8.16 shows the **Reduce (sum)** of the B array and the prefix sum of the aggregated A array on each processor.

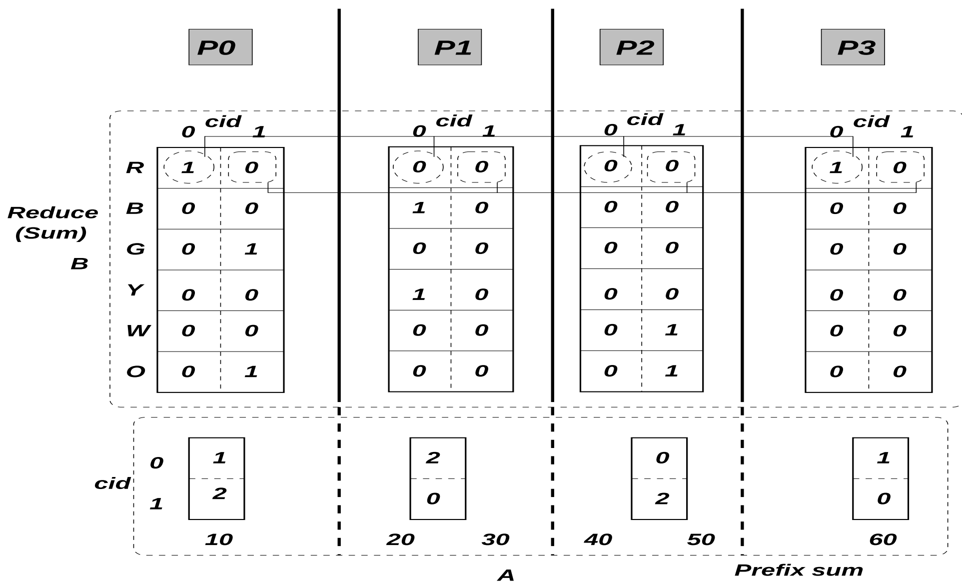


Figure 8.16: Communication of the dimension count values

Figure 8.17 shows the **broadcast** of the prefix sum from the last processor. This is followed by the local calculation of the gini index and a **Reduce (minimum)** for the gini value which determines the attribute used for the split. Notice that for a non-distributed categorical attribute B the calculation is distributed across processors by letting each processor work on a section since all information is available on each processor as a result of the reduce. Suppose the $gini_{split}$ results in selecting

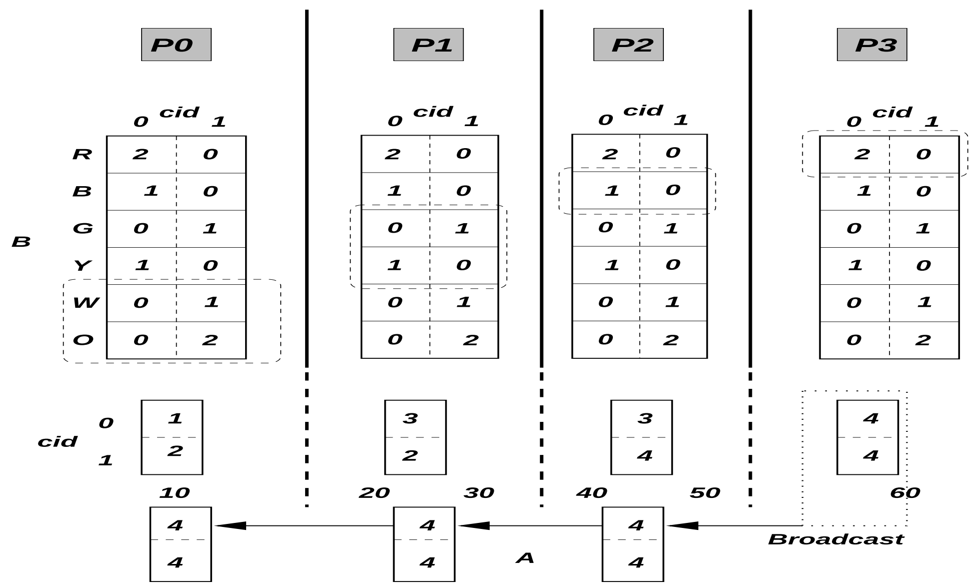


Figure 8.17: Computation of the gini indices locally

$A = 25$ as the split point in the example.

Figure 8.18 shows the split at each processor into L and R nodes labeled 00 and 01 respectively. The counts of each dimension are done for each node on each processor. The steps for parallel gini index calculations are done for each node on every processor. Communication can be concatenated for the nodes at each level following the idea of concatenated parallelism in [AfAGR96] for each active node.

8.2.4 Communication Optimizations

Only one collective communication operation is performed for all nodes at each level. This is due to the fact that information that needs to be communicated for all nodes is stored contiguously. Each node has the counts of class id. values for each dimension. This is combined using a **Reduce (sum)** operation across all nodes as described in a previous section. The count matrices for each dimension are allocated contiguously for each active node as shown in Figure 8.19. Each

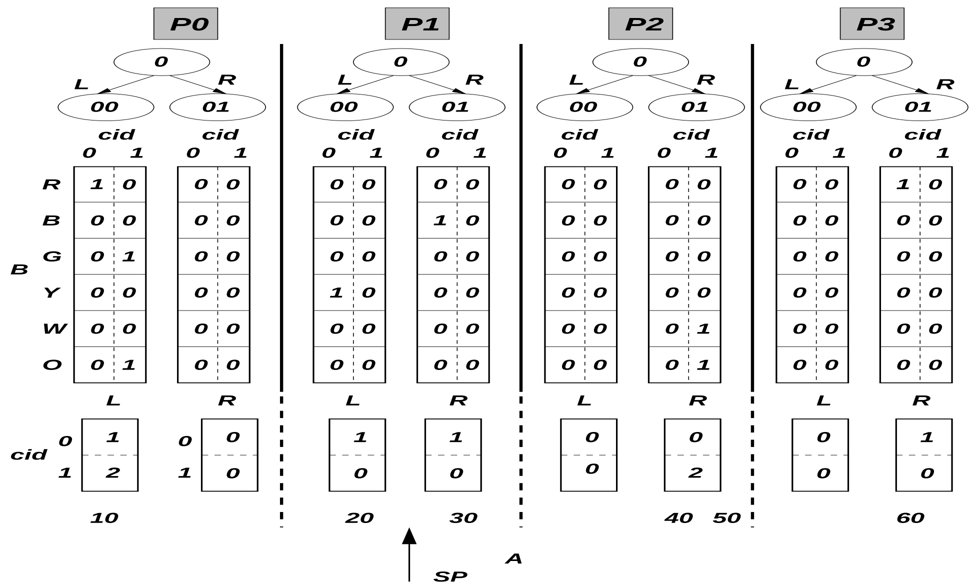


Figure 8.18: Node split of dimension $A = 25$, and the count arrays for each node

node allocates memory for each dimension, d_i, for each class. At each level only the unclassified nodes (active nodes) are represented and need to participate. Algorithm 8.2.4 describes the steps of the overall parallel algorithm for classification.

8.2.5 Performance Evaluation - Classification

We use the synthetic data generator introduced by Agrawal et al. in [AIS93a]. It is a widely used synthetic set used by many others, primarily because there are no large real data sets available. The largest dataset in the often used Statlog collection of training databases [MST94] has 57000 records and the most interesting data sets from the UCI machine learning repository [CBM98] have a few hundred to a few thousand training samples. The synthetic set has nine attributes as shown in Table 8.2. $\mathcal{U}(x : y)$ denotes the integer uniform distribution with values $v : x \leq v \leq y$. The data set can be generated with different classification functions that assign labels to the records produced. We use Function 1, which is the default,

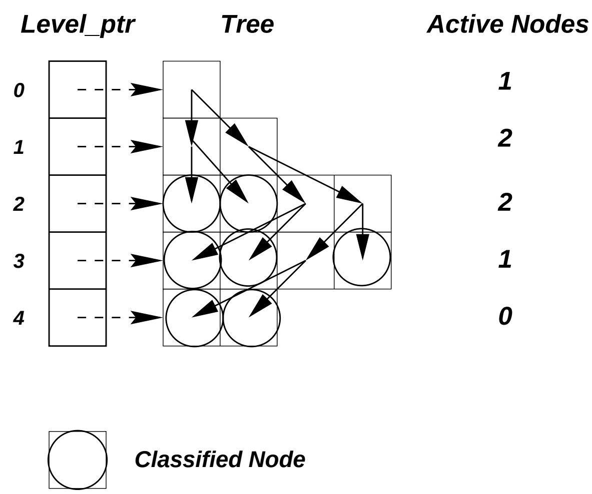


Figure 8.19: Tree structure at each processor, nodes being stored contiguously to coalesce messages for communication at each level

for our performance study. We have used various subsets of the data to evaluate the performance of our algorithm, varying the number of dimensions, the number of records and the classification threshold. We observe good speedup and scale-up performance for our experiments performed mainly on a 16 node IBM SP2 with thin nodes (120Mhz processor) running AIX and having 128 MB main memory per processor. Some experiments were performed on a larger IBM SP2 available to us on a restricted usage basis. We present a subset of our results in this section to illustrate performance.

Figure 8.20 (a) shows the classification time for a 8 dimensional set (without the Loan attribute) containing 100,000 records with classification threshold set to 0.6. As the number of processors are increased we observe a modest speedup. Figure 8.20 (a) shows the performance of the classification algorithm and the associated communication time for the same 8 dimensions with 1 million records. We observe that as the number of processors are increased in this case the cost of communication increases, but is still very small in comparison to the overall time. We observe good

Algorithm 8.2.4 *Multidimensional Classification Algorithm*

/* set the root node to represent the entire multidimensional space */
for $i \leftarrow 0$ to k
    $begin_i = 0$
    $end_i = |d_i|$
$n\_active\_nodes = 1$;
while($n\_active\_nodes > 0$)
    For each chunk c
        Get chunk boundary values in each dimension
        For each active node $j, (0 \leq j < n\_active\_nodes)$
            Determine if chunk c is contained within node j,$[begin_i, end_i), forall i$, else if it overlaps or else not in range.
    For each active node $j, (0 \leq j < n\_active\_nodes)$
        Compute the contribution of chunk c to 1-dimensional aggregate if the chunk is inclusive or overlapping to the node j.
    One **Reduce (sum)** communication operation for consolidating k 1-dimensional aggregates on all processors.
    For each active node $j, (0 \leq j < n\_active\_nodes)$
        Check if node j is already classified within the threshold level.
        If the node is *not* classified
            For each attribute $i, 0 \leq i < k$,
                Compute *gini* indices for each dimension i recording the minimum value and corresponding i ($split_i$) and the split point.
            Update for the child node, $begin_i$ and end_i for $split_i$ using the split point and copy the others from the parent node j.
            Add the number of child nodes to the count of active nodes.
    Update n_active_nodes as the count of child nodes.
end

Table 8.2: The sizes of the dimensions in data set used

Predictor Attribute	Distribution	Max. number of entries
Salary	$\mathcal{U}(20K, 150K)$	131
Commission	0, if(salary > 75K) else $\mathcal{U}(10K, 75K)$	66
Age	$\mathcal{U}(20, 80)$	61
Education	$\mathcal{U}(0, 4)$	5
Car	$\mathcal{U}(1, 20)$	20
Zip Code	$\mathcal{U}$(nine zip codes)	9
House Value	$\mathcal{U}(0.5 \times k \times 100K, 1.5 \times k \times 100K)$ k depends on ZipCode	1351
Home Years	$\mathcal{U}(0, 30)$	31
Loan	$\mathcal{U}(0, 500K)$	501

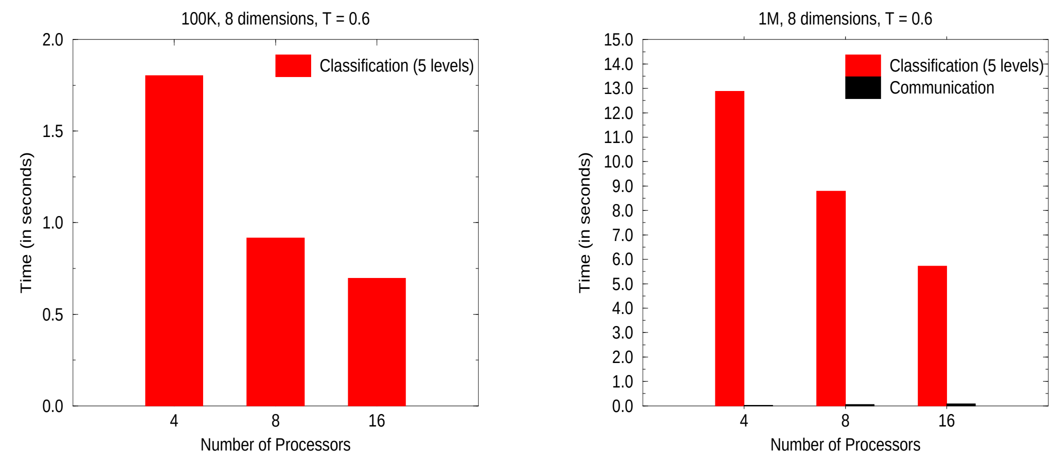
speedups for this case.

Figure 8.21 shows the time for classifying 1 million and 5 million records with 8 dimension (again without Loan attribute) and classification threshold value set to T = 0.8. The 5 million records data set is We observe that most of the time is taken for the 1 dimensional aggregate calculations and the gini computations, a small fraction taken by communication and maintenance of the classification tree.

Figure 8.22 shows the classification performance for a 9 dimensional data set with 1 million records. The major component of the computation is the 1-dimensional aggregate calculations for all nodes. This is the parallel component and we see good speedup as the number of processors is increased. The other phases of communication and gini index calculations as shown in the figure are small portions of the entire time. Communication increases slightly as number of processors increase, due to the increase in p term in the complexity of the **Reduce** global communication operation.

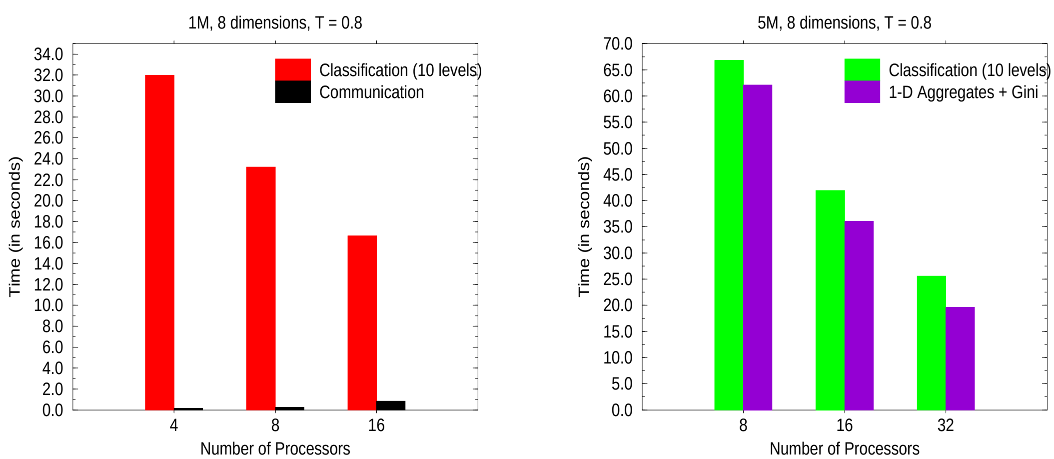
8.3 Summary

The data cube represents quantitative summary information for a subset of the attributes. Association rules can be discovered from the aggregates in the OLAP



(a) (b)

Figure 8.20: Classification of a 8 dimensional data set for 5 levels, with classification threshold T = 0.6 for (a) 100K records (b) 1M records



(a) (b)

Figure 8.21: Classification of a 8 dimensional data set for 10 levels, with classification threshold T = 0.8 for (a) 1M records (b) 5M records

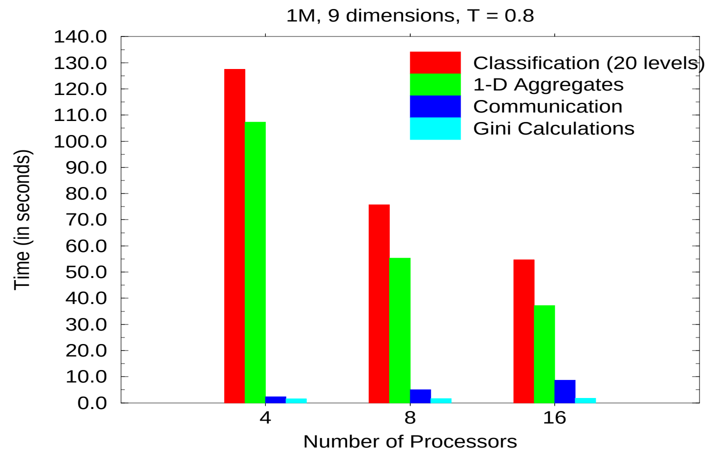


Figure 8.22: Classification of a 9 dimensional data set for 20 levels, with classification threshold T = 0.8 for 1M records

cube. A viable data analysis solution is to make data mining an interactive approach by combining it with OLAP. Multi-level associations between attributes determine the granularity of the interesting patterns in data, some rules not being identified at the lowest level of detail, or just too many rules being generated. The parallel algorithms for associations take the aggregate calculations in the cube to get the probabilities for support and confidence for attribute-oriented rules. A multi-dimensional storage allows for efficient computation of these measures.

Classification using decision trees is performed on the multi-dimensional storage of the OLAP cube. The algorithm can be applied to any cube in the cube lattice, depending on the attributes to be included in the data to be classified. The split point for a node in the classification tree is chosen after computing the counts of records belonging to each class for each attribute value, in each dimension. A multidimensional representation make a 1-dimensional computation for each dimension from the base cube very efficient. The parallel OLAP framework is used to parallelize this calculation and one round of communication is required at each level of the tree by concatenating (coalescing) communication for each node together. Results on a synthetic benchmark [AIS93a], widely used for classification algorithms performance, show good parallel performance.

Chapter 9

Summary and Future Work

Large scale data warehouses are becoming increasingly available, acting as data repositories for a large number of applications in various domains. Data available in data warehouses is already integrated, consistent, cleaned and ready for use. Modern information systems can benefit from the knowledge and insight from this data to perform effective decision support. A data warehouse stores data from many different databases and is typically based on relational database technology. OLAP systems make use of this data for multi-dimensional analysis, typically using multi-dimensional databases since they are more suitable for such analysis. Data mining is another word for automated extraction of knowledge and insight from data to help in better decision making. OLAP integration with data mining helps users to determine what kinds of knowledge to be mined. It provides the flexibility for users to select the desired data mining function after analyzing the data through OLAP.

Multidimensional databases provide fast response times for OLAP queries as can be seen from the commercial success these products have attained. However, these are limited to low dimensions (5 to 10) and small data sizes (Megabytes). However, current methods for multidimensional databases have not proven to be scalable in data sizes and the number of dimensions. In this thesis we have developed a parallel scalable infrastructure with algorithms and techniques for OLAP and integrated it with data mining using the multidimensional data model. Data mining techniques are integrated with this parallel OLAP infrastructure which uses summary data present in the data cube. Parallel algorithms for attribute-oriented data mining are presented which use the computations from the data cube. Decision-tree based classification trees are constructed efficiently in parallel by using the multidimensional model. Performance is achieved by reducing the communication costs associated with maintaining the classification tree at each level. Results show that our implementations are scalable to large data sets, large number of dimension, and a large number of processors.

The main contributions of this thesis can be summarized as follows.

- Introduced a framework for scalable parallel OLAP and data mining using the multi-dimensional data model.
- Introduced Bit encoded sparse structure which provides scalability by allowing dimensional operations on compressed data.
- Communication optimizations in schedule generation for parallel cube construction for full and partial cubes.
- Parallel algorithm design and implementation for attribute-oriented associations and decision tree based classification on the multidimensional model.
- Integration of data mining with the parallel OLAP infrastructure.
- Demonstrated scalability in terms of - data sizes, number of dimensions and number of processors.

9.1 Future Work

Pre-computed summary information available in the data cube is redundant information, which helps in answering queries that require some of the precomputations. A query stream may only access a portion of the data cube. The space-time tradeoff between space required for pre-computations and the benefit to query evaluation can be addressed by selective materialization of the aggregate cubes. Various approaches have been taken in the relational database domain. Greedy heuristics based on dominance among views has been used in [HRU96]. Query benefit analysis has been applied using the average execution time for previous queries and cost saving ratio metrics in [DRSN98]. Chunks as a unit of caching have also been suggested, which is shown to be better than table level caching for query streams. This work can be extended in the parallel domain with our chunked storage scheme and caching of minichunks. Communication costs need to be taken into account for a parallel version and will figure in the cost calculations of the various aggregates. Another area to investigate is the incremental maintenance of cubes. Periodical updates to the data warehouse, require the analysis to be done on the most current data in the warehouse. Larger data sets and larger number of dimensions will increase the cost of parallel cube construction even further. Hence, a strategy for incremental maintenance will be required for more efficient query processing. Optimization for evaluating related complex queries simultaneously can be investigated for improved query times. A query can use the chunks cached for multiple expressions simultaneously. Recently, a standard multidimensional interface for databases called OLE DB for OLAP has been proposed [Mic98]. This defines Multi-dimensional expressions (MDX), which allow clients to ask several related dimensional queries in a single MDX expression. Optimizations of simultaneous queries is an important area to be addressed for performance. Some optimization in the ROLAP domain have been addressed using MDX expressions in [ZDN98]. An appropriate multidimensional API for query expressions in parallel to give support to the query expression is an important area to be explored.

Parallel data mining for association rules on multiple concept levels can generate valuable information in many scenarios. With a well defined concept hierarchy on some attributes meaningful rules can be generated. High dimensional clustering is another area, where a multidimensional data model can make the calculations more efficient. An interactive approach to OLAP and association rule mining can make the rule generation process more focused and explore parts of the database that are more interesting. High dimensional classification and clustering require a large amount of computation. Dimension reduction techniques, like principal component analysis [Fuk90] can be applied to reduce the number of dimensions. These should be explored for their efficacy in OLAP and data mining to handle even larger number of dimensions by mapping the data space in some lower dimensional space. Methods for automatic subspace clustering [AGGR98] repeatedly use the projections of k-dimensional data into $k-1$ dimensional data, which can be organized in a chunk based representation. Support for dynamic chunk sizes is required as subspaces are merged and dense regions are grown.

Bibliography

[AAD⁺96] 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 International Conference on Very Large Databases*, September 1996.

[AfAGR96] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multidimensional binary search trees. In *Proc. International Conference on Supercomputing*, May 1996.

[AGGR98] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1998.

[AIS93a] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, December 1993.

[AIS93b] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 207–216, May 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th International Conference on Very Large Databases*, 1994.

[AS96] R. Agrawal and J. Schafer. Parallel mining of association rules: Design, implementation and experience. Technical Report RJ10004, IBM Almaden Research Center, 1996.

[Bea93] I. Bhandari and et. al. A case study of software process improvement during development. *IEEE Transactions on Software Engineering*, 19(12), 1993.

[BFOS84] L. Breiman, J.H Friedman, R.A Olshen, and C.J Stone. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.

[Bha95] I. Bhandari. Attribute focusing: Data mining for the layman. Technical Report RC 20136, IBM T.J. Watson Research Center, 1995.

[BL97] M. Berry and G. Linoff. *Data Mining Techniques*. John Wiley and Sons, 1997.

[BMS97] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proc. ACM SIGMOD Conference on Management of Data*, May 1997.

[CBM98] E. Keogh C. Blake and C.J. Merz. *UCI Repository of machine learning databases*. University of California, Irvine, http://www.ics.uci.edu/~mlearn/MLRepository.html, 1998.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1997.

[Cod93] E.F. Codd. Providing OLAP to user-analysts : An IT mandate. Technical Report White Paper, E.F. Codd and Associates, 1993.

[Cog98] Cognos, http://www.cognos.com/powerplay/ppes_wp.html. *Powerplay Enterprise Server Technical White Paper*, 1998.

[Col96] G. Colliat. OLAP, relational, and multi-dimensional database systems. In *SIGMOD Record*, volume 25(3), September 1996.

[CS93] P.K Chan and S.J Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. International Workshop on Multistrategy Learning*, 1993.

[DRSN98] 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.

[Ear94] R. J. Earle. Method and apparatus for storing and retrieving multidimensional data in computer memory. *U.S Patent 5359724*, 1994.

[Fel57] W. Feller. *An Introduction to Probability Theory and its Applications.* John Wiley and Sons, 1957.

[Fif92] D.J Fifield. *Distributed Tree construction from large data sets.* Bachelor's Honors Thesis, Australian National University, 1992.

[FM85] P. Flajolet and G.N. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[FPSSU94] U.M. Fayyad, G. Piatesky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in data mining and knowledge discovery.* MIT Press, 1994.

[Fuk90] K. Fukunaga. *Introduction to Statistical Pattern Recognition.* Academic Press, 1990.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. 12th International Conference on Data Engineering*, 1996.

[GC97] S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, 1(4), 1997.

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

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

[GC99] S. Goil and A. Choudhary. Design and Implementation of a Scalable System for Multidimensional Analysis and OLAP. In *Proc. International Parallel Processing Symposium*, April 1999.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J Ullman. Index selection for OLAP. In *Proc. International Conference on Data Engineering*, January 1997.

[Gol89] D.E Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Morgan Kaufmann, 1989.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), June 1993.

[Gup97] H. Gupta. Selections of views to materialize in a data warehouse. In *Proc. International Conference on Database Technology*, January 1997.

[HCC93] J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(1), February 1993.

[HF95] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. of International Conference on Very Large Databases*, 1995.

[HKK97] E. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proc. ACM SIGMOD International Conference on Management of Data*, May 1997.

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

[Hyp98] Hyperion Solutions Inc., http://www.hyperion.com/whitepapers.cfm. *The Role of OLAP in a Data Warehousing Solution*, 1998.

[Inf98] Information Advantage, http://www.infoadvan.com/products/. *MyEureka! White Paper*, 1998.

[JKK98] 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.

[KGGK94] V. Kumar, A. Grama, A.l Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings, 1994.

[KHC97] M. Kamber, J. Han, and J. Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proceeding of the 3rd International Conference on Knowledge Discovery and Data Mining*, August 1997.

[Kim96] R. Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley and Sons, 1996.

[Kuf95] R. Kufrin. Decision trees on parallel processors. In *IJCAI-95 Workshop on Parallel Processing for Artificial Intelligence*, August 1995.

[KWC+97] M. Kamber, L. Winstone, W. Cong, S. Cheng, and J. Han. Generalization and decision tree induction: Efficient classification in data mining. In *Proc. of 1997 International Workshop on Research Issues on Data Engineering*, April 1997.

[MAR96] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth International Conference on Extending Database Technology*, March 1996.

[May98] F. May. *SP Switch Performance.* IBM Corporation, http://www.rs6000.ibm.com/resource/technology/spswperf.html, 1998.

[Mic98] Microsoft, http://www.microsoft.com/data/oledb/olap. *OLE DB for OLAP design specification*, 1998.

[MST94] D. Michie, D.J. Spiegelhater, and C.C. Taylor. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

[OG95] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. In *SIGMOD Record*, September 1995.

[OLA97] OLAP Council, http://www.olapcouncil.org. *OLAP Council Benchmark*, 1997.

[OQ97] P. O'Neil and D. Quass. Improved query performance with variant indices. In *Proc. ACM SIGMOD Conference on Management of Data*, 1997.

[Ora98] Oracle Corporation, http://www.oracle.com/olap/collatrl/exp_eo.pdf. *Oracle Express: Enabling better decision making*, 1998.

[Pil98] Pilot Software, http://www.pilotsw.com/olap/olap.htm. *An Introduction to OLAP: Multidimensional Terminology and Technology*, 1998.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[RKR97] N. Roussopoulus, Y. Kotidis, and M. Roussopoulus. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proc. ACM SIGMOD Conference on Management of Data*, 1997.

[RS97] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *23rd International Conference on Very Large Databases*, September 1997.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 21st International Conference on Very Large Databases*, June 1995.

[SAG96] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RC 10026, IBM Almaden Research Center, San Jose, 1996.

[SAM96] J.C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. 22th International Conference on Very Large Databases*, September 1996.

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

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

[Sof97] Kenan Software. *An Introduction to Multi-dimensional Database Technology.* http://www.kenan.com/acumate/mddb.htm, 1997.

[SS94] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proc. of the Eleventh International Conference on Data Engineering*, February 1994.

[SSV96] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN : A data warehouse intelligent cache manager. In *Proc. 22th International Conference on Very Large Databases*, June 1996.

[ZDN97] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multi-dimensional aggregates. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 159–170, 1997.

[ZDN98] Y. Zhao, P. Deshpande, and J. Naughton. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1998.

[ZHA99] M. Zaki, C.T. Ho, and R. Agrawal. Scalable parallel classification for data mining on shared-memory multiprocessors. In *Proc. International Conference on Data Engineering*, March 1999.

Vita

Sanjay Goil was born in Bombay, India. He received a B.S in Computer Science from Birla Institute of Technology and Science, Pilani, India in 1990 and a M.S in Computer Science from Syracuse University, New York in 1995. From 1991 to 1993, he was a Research Associate in the Networked Computing Research Department at AT&T Bell Laboratories, Murray Hill, New Jersey. Prior to that he spent a year with Tata Consultancy Services in New Delhi, India. His research interests are in the areas of high performance computing and architecture, parallel algorithms and techniques for large scale applications, and performance analysis.