NOTE TO USERS

Parallel Computation of Data Cubes

By

Xinrong Huang

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

December, 2004

Your file Votre référence
ISBN: 0-494-00761-3
Our file Notre référence
ISBN: 0-494-00761-3

Abstract

As the information age explodes with data, On-line Analytical Processing (OLAP) has attracted a great deal of attention in the research communities. At the heart of OLAP is the data cube, a relational construct that supports true multi-dimensional data analysis.

Over the past decade, a number of sequential algorithms for efficient data cube construction have been presented. However, most of the algorithms are specifically designed for data set that fits in main memory. In order to reduce the expensive external memory sorting cost, Partitioned-Cube algorithm, which works with the Memory-Cube algorithm, is proposed. In this thesis, we are interested in computing the data cube efficiently when the data set is larger than the main memory. The adapted version of Partition-Cube algorithm and Memory-Cube algorithm are presented.

As noted, most of the existing algorithms are designed for implementation on sequential machines. Given the expense of such computation, another focus of our research is to compute the data cube in parallel. We present two approaches to parallelizing the Partitioned-Cube algorithm and the Memory-Cube algorithm.

Acknowledgements

I would like to thank my supervisor, Dr. Frank Dehne for introducing me to the area of data mining and parallel computing. Also I am particulary grateful for his many suggestions and constant support during this research.

I am also thankful to Dr. Todd Eavis for his guidance through this research and writing of my thesis.

I would like to express my gratitude to all the people in the data cube group of the school of Computer Science at Carleton University for their help and invaluable suggestions.

Many thanks to my friends, Anthony Leung, Haixia Yang, Ling Yang, Lijia Hao, Wei Chang, Zhan Ma, Ning Wang and Shu Zhang for their encouragement and patiently reading my thesis draft.

Finally, I am grateful to my parents and sister for their *constant support* and *endless love*. Without them this work would never have come into existence.

Ottawa, Ontario
December, 2004

Xinrong Huang

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, there has been tremendous growth in the data warehousing market. The sophistication and maturity of the global Internet and its graphical sibling, the World Wide Web, have lead to tremendous growth in the size of databases that corporations build, manage, and analyze. As data warehouses grow, parallel processing techniques have been applied to enable the use of larger data sets and reduce the time for analysis, thereby enabling evaluation of many more options for decision making [21].

On-line Analytical Processing (OLAP) is used to extract useful summary information from data warehouses. This summary information can be used in decision support systems (DSS) to assist knowledge workers (executive, manager, analyst) to make better and faster decisions [14]. Over the past two decades, OLAP has become a fundamental component of contemporary decision support systems. OLAP is the foundation for a wide range of essential business applications, including sales and marketing analysis, planning, budgeting, performance measurement and data warehouse reporting [25]. At the heart of OLAP is the data cube [24], a relational operator that supports true multi-dimensional analysis. The efficient computation of the cube

is a key issue in improving the response time of OLAP queries.

1.1 Motivation

This thesis explores efficient algorithms for data cube construction and the use of parallel approaches for computing data cubes.

In order to compute the data cube more efficiently, a number of efficient sequential algorithms for data cube construction have been presented in recent years. The most significant ones are the PipeSort and PipeHash algorithms proposed by Sarawari et.al.[35], Overlap algorithm proposed by Deshpande et. al.[17, 9], Partitioned-Cube algorithm proposed by Ross et. al.[34], Bottom Up Computation algorithm (BUC) proposed by Beyer et. al.[13] and the Array-Cube algorithm proposed by Zhao et. al.[39].

In the real world, the size of corporate databases is usually much larger than available main memory. Even though external memory sorting can be utilized by the underlying data cube algorithms, doing so produces significant I/O costs (i.e., intermediate files). Therefore, an important goal for the computation of large data cubes is the design of a more efficient mechanism for the manipulation of data sets that are too large for main memory.

The Partitioned-Cube algorithm was proposed for solving this problem. However, the original algorithm and implementation suffer from two important problems which prevent it from achieving optimal performance. In this thesis, we present an approach, based upon the sharing of sort costs, that attempts to solve these problems. Our experimental results demonstrate a significant performance improvement.

Moreover, all of these algorithms – including Partitioned-Cube – are designed for

implementation on sequential machines. However, computing a data cube can be a very expensive task. Application of parallel processing can speed up this process. Despite the popularity and importance of data cubes, very little research has been carried out on parallel cube computation. In [15], the authors present a general framework for the efficient parallelization of existing data cube construction algorithms, with a particular focus on the parallelization of the PipeSort algorithm for *shared disk* architectures.

In this thesis we build upon that research but specifically target a fully distributed shared nothing cluster architecture. Cluster machines are workstations connected via high speed interconnection networks. These low-cost machines can often approximate the performance of massively parallel machines on certain classes of algorithms. The widespread availability of these inexpensive machines makes it possible to use them effectively for the computation of data cubes, even in small size firms. Parallel cube construction algorithms would therefore be important in this context. We present two approaches to parallelize the Partitioned-Cube algorithm and the Memory-Cube algorithm in this thesis.

1.2 Thesis Organization

The thesis is organized as follows. Chapter 2 provides an overview of Online Analytical Processing and data cubes, including topics such as fundamental OLAP operations, server architectures and general data cube concepts. Chapter 3 presents the data cube operator and describes a number of the algorithms that have been designed for its efficient construction, mainly focused on the PipeSort algorithm and the Partitioned-Cube algorithm. Chapter 4 presents two approaches to improve the

performance of Partitioned-Cube and Memory-Cube. We also give an experimental analysis of the adapted version of the Partitioned-Cube algorithm and the Memory-Cube algorithm. In Chapter 5, we propose two methods to parallel the two algorithms. We provide out conclusions in Chapter 6.

Chapter 2

An introduction to OLAP and the Data Cube

2.1 Introduction

Over the past two decades, database and data management systems have played a vital role in the growth and success of corporate organizations. Decision support systems (DSS) are rapidly becoming a key to gain competitive advantages for businesses. DSS allow businesses to get data that is locked away in operational databases and turn that data into useful information. Many corporations have built or are building new unified decision-support databases called data warehouses on which users can carry out their analysis.

While operational databases maintain current state information, data warehouses typically maintain historical information. As a result, data warehouses tend to be very large and to grow over time. Users of DSS systems are typically interested in identifying trends rather than looking at individual records in isolation. Decision-support queries thus make heavy use of aggregations and are much more complex than OLTP (On-line Transaction Processing) queries. The size of data warehouses and the

complexity of queries can cause queries to take very long to complete. This delay is unacceptable in most DSS environments, as it severely limits productivity. The requirement of query execution times is usually a few seconds or a few minutes at the most. There are many ways to achieve such performance goals. Query optimizers and query evaluation techniques can be enhanced to handle aggregations better. Using different indexing strategies can improve the query times as well. A commonly used technique is to materialize (precompute) frequently-asked queries [26].

In this chapter, we examine the current trends, technologies, and terminology relevant to an understanding of On-line Analytical Processing or OLAP [10, 14, 18], mainly focusing on Data Cubes. We provide an introduction to the general area of decision support systems (DSS) in Section 2.2. Section 2.3 defines OLAP in terms of its core operations and functionality. Section 2.4 introduces the data warehouse. Section 2.5 discusses the data cube, which is the focus of this thesis. Section 2.6 concludes the chapter with a brief summary.

2.2 Decision Support Systems

Decision Support System (DSS) refers to an interactive computerized system that gathers and presents data from a wide range of sources, typically for business purposes. DSS applications represent systems and subsystems that help people make decisions based on data that is culled from a wide range of sources [3].

Since the early 1970s, DSS technologies and applications have evolved significantly. Many technological and organizational developments have exerted an impact on this evolution. DSS once utilized more limited database, modelling, and user interface functionality, but technological innovations have enabled far more powerful

DSS functionality. Traditionally, DSS supported individual decision-makers, but later DSS technologies were applied to work groups or teams, especially virtual teams. The advent of the Web has enabled inter-organizational decision support systems, and has given rise to numerous new applications of existing technology as well as many new decision support technologies themselves. It seems likely that mobile tools, mobile e-services, and wireless Internet protocols will mark the next major set of developments in DSS. With respect to the current environment, there are four primary decision support tools, including data warehouses, OLAP, data mining, and Web-based DSS. OLAP is the focus of our present research [36].

2.3 On-line Analytical Processing

On-line Analytical Processing (OLAP) is a category of software tools that provides analysis of data stored in a database. OLAP tools enable users to analyze different dimensions of a multidimensional data repository. For example, it provides time series and trend analysis views [6].

Typical OLAP operations include *roll-up* (increasing the level of aggregation) and *drill-down* (decreasing the level of aggregation or increasing detail along one or more dimension hierarchies), *slice* and *dice* (selection and projection), and *pivot* (re-orienting the multidimensional view of data).

2.4 The Data Warehouse

A data warehouse (DW) is a "subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management's decision-making process" [28].

In order to support OLAP functionality, data warehouses provide an effective alternative to the transaction-oriented environment of the operational database. They are organized around subjects, rather than atomic transactions. They represent aggregated or summarized information from a variety of sources. They house data collected over very long periods, typically years. And they are tuned for read-only access.[19]

In general, the Data Warehouse (DW) is a collection of data designed to support management decision making. Data warehouses contain a wide variety of data that presents a coherent picture of business conditions at a single point in time. Construction of a data warehouse includes development of systems to extract data from operational systems plus installation of a warehouse database system that provides managers flexible access to the data. The term data warehousing generally refers to the combination of many different databases across an entire enterprise [2].

2.4.1 Data Warehouse Architecture

Figure 2.1 shows a typical data warehousing architecture [14]. In general, data warehouses consists of three tiers (i.e. data source, data mart, front-end tools). Information is first extracted from operational sources and then cleaned, transformed and loaded into the data warehouse proper which, at this point in time, is almost always a relational DB. The data warehouse itself may be constructed as a monolithic enterprise-wide entity and/or a series of data marts, each containing some subset of the corporate data. After the data warehouse has been constructed, the OLAP server provides analytical functionality for the DSS system. In practice, there are two forms of OLAP servers, known as *Multidimensional OLAP (MOLAP)* and *Relational OLAP*

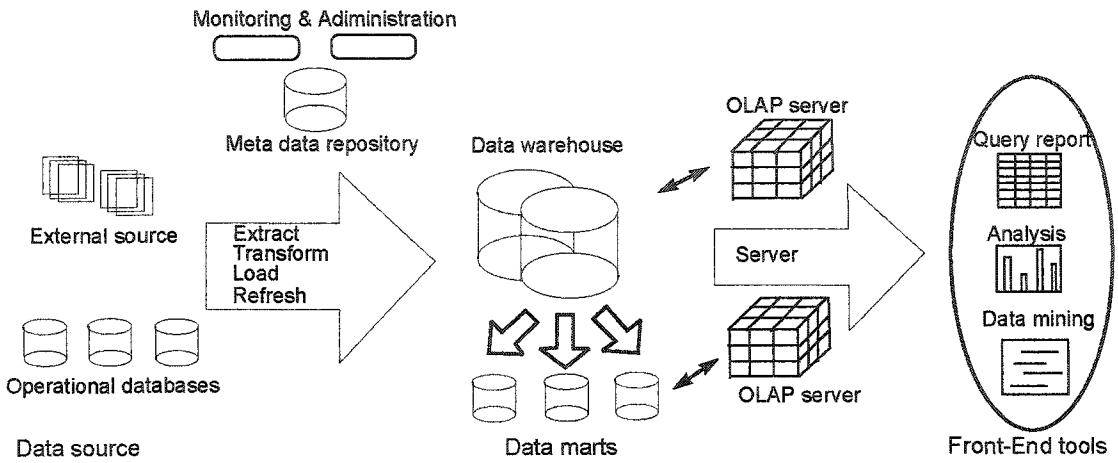


Figure 2.1: The Data Warehouse Architecture

(ROLAP). Finally, in the top tier, the front end tools provide a user-friendly (often graphical) interface for the knowledge workers who will exploit the system [14].

2.4.2 ROLAP and MOLAP

As stated above, there are two main forms of OLAP server, ROLAP and MOLAP. In ROLAP servers, data is stored in the tables of relational databases or extended-relational databases. The associated DBMS systems support extensions to SQL and special access and implementation methods to efficiently implement the multidimensional data model and operations. ROLAP servers are well-suited to large data sets because sparse data sets may be stored compactly in tables (there is no need to represent or index the empty regions of the cube space). Moreover, ROLAP servers can exploit the scalability and the transactional features of the relational systems [14].

In contrast, MOLAP servers directly store multidimensional data in special data

structures (e.g. arrays) and implement the OLAP operations over these data structures [14]. Because of their exploitation of multi-dimensional arrays, these MOLAP products often offer rapid response time on OLAP queries. Unfortunately, MOLAP solutions have not proven to scale effectively to large, high-dimensionality data sets [19]. Specifically, the array-based structures are not well suited to sparse hypercube spaces.

2.5 The Data Cube

The *data cube*, a relational construct that supports true multi-dimensional data analysis, was introduced by Jim Gray et al. in [24].

In order to more fully understand the data cube, we first introduce some of the basic concepts related to the topic. A standard OLAP analysis environment consists of a group of dimensions, each of which has been identified by the data warehouse designers as being of interest to the user community. In OLAP terminology, dimensions are also known as *attributes*; Attributes can be of two types. *Feature* attributes refer to those dimensions that represent entities or concepts central to the structure of the organization. Examples would be things such as customer and product. *Measure* attributes, on the other hand, refer to the items of interest, the values that will be aggregated in terms of the feature attributes. We note that although there may be many feature attributes, in most cases there will be a very small number of measure attributes (often just one). Measure attributes may be calculated using aggregate functions from one of the three distinct categories (i.e. distributive,algebraic,holistic). Aggregate functions will be discussed in Section 2.5.1.

In a d dimensional data set, each of the d attributes $\{A1, A2, .., Ad\}$ has a *cardinality* that identifies the number of unique values for that attribute. For instance, if one of the data cube dimensions is "Product", and there are 300 individual products in our database, then the cardinality of Product, denoted $|Product|$, is 300. We refer to the group of d cardinalities as the cardinality set C.

2.5.1 Categories of Cubes Aggregate Functions

Aggregation is a fundamental operation in decision support systems. OLAP applications often need to summarize data at various levels of detail and on various combination of dimensions. These aggregate functions can be classified into three categories [24]:

- **Distributive**. Distributive functions have the unique feature that they allow the input set to be partitioned into disjoint sets that can be aggregated separately and later combined. Examples are *sum*, *min*, and *max*.
- **Algebraic**. An algebraic function is one that can be produced by combining distributive functions. Examples would include *average* and *standard deviation*.
- **Holistic**. An aggregate function is holistic if there is no constant bound on the size of the storage needed to describe a sub-aggregate. *Median* and *rank* are common examples of holistic functions.

For simplification, we will utilize a single distributive measure attribute, namely *summation* throughout the thesis [19].

2.5.2 The Data Cube Lattice

In total, a d-dimensional data warehouse is associated with 2^d views. In OLAP terminology, views are also known as *cuboids* or *group-bys*. Each view or cuboid represents a distinct combination of feature attributes, and can be seen as depicting an aggregation of the measure attribute at a given level of granularity. For example, given the attribute set ABC, we say that the aggregated view A is of coarser granularity than the aggregated view AB. Note that we are interested in attribute combinations, not permutations, since the order of the attributes does not matter. In practice, we usually substitute letter labels for the attribute names. For example, Customer may be attribute "A", Product may be attribute "B", etc. The "Customer/Product" cuboid is therefore simply referred to as AB.

The relationship between the 2^d views in terms of common attributes is typically represented by a *lattice* [26]. See Figure 2.2 for a graphical illustration. Starting with the *base cuboid*, the finest granularity view containing the full complement of d dimensions, the lattice branches out by connecting every parent node with the set of child nodes/views that can be derived from its dimension list. In other words, the attributes of a parent view must be a superset of the attributes of a child view. A parent containing k dimensions can be connected to k views at the next level in the lattice, each of which contains $k-1$ attributes. Conversely, a child view can be associated with $d-k$ parents (if this is not obvious, note that because the lattice is perfectly symmetrical, the number of parents for a given view at level k is equivalent to the number of children for a given view at level $d-k$). Finally, it should be understood that parent/child relationships are not exclusive. Parents can share common children just as children may have common parents.

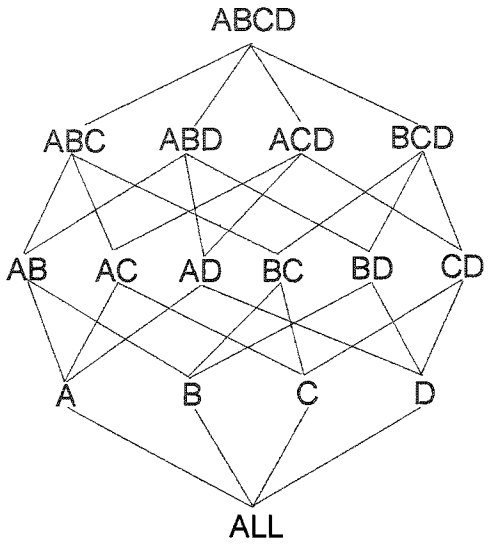


Figure 2.2: The data cube lattice consists of all possible attribute combinations. The "all" node represents the aggregation of all records.

Conceptually, the data cube consists of the base cuboid, surrounded by a collection of $2^d - 1$ sub-cubes/cuboids that represent the aggregation of the base cuboid along one or more dimensions. Since the base cuboid contains all feature attributes, it can be used to compute all of the other coarser cuboids by aggregating across one or more of its component dimensions. In other words, it may be possible to initially compute only a subset of all possible views, leaving the materialization of the remaining views to some later time (if necessary). As such, a data cube can be described as full if it contains all 2^d possible views, or partial if only a subset of views has actually been constructed [19].

2.5.3 The Data Cube Operator

The Group-By operator in SQL is typically used to compute aggregates on a set of attributes. For business data analysis, it is often necessary to aggregate data across many dimensions. For example, in a retail application, one might have a table Transactions with attributes Product (P), Data (D),Customer (C), and Sales (S). An analyst could then query the data for finding:

- sum of sales by P, C.

  For each product, give a breakdown on how much of it was sold to each customer.

- sum of sales by D,C.

  For each date, give a breakdown of sales by customer

- sum of sales by P.

  For each product, give total sales.

In 1995, Gray et al. introduced the CUBE operator for conveniently supporting multiple aggregates in OLAP databases [24]. The CUBE operator is the n-dimensional generalization of the Group-By operator. It computes group-bys corresponding to all possible combinations of a list of attributes. Returning to our retail example, the collection of aggregate queries can be conveniently expressed using the cube-operator as follows:

```
SELECT P, D, C, Sum(S)
FROM Transactions
CUBE-BY P, D, C
```

This query will result in the computation of $2^3 = 8$ group-bys: PDC, PD, PC, DC, D, C, P and *all*, where all denotes a single aggregate value. The straightforward way to support the above query is to rewrite it as a collection of eight Group-By queries and to execute them separately [9]. The goal of the data cube algorithms previously mentioned is, of course, to perform this computation much more efficiently.

2.6 Conclusion

Data warehousing and on-line analytical processing (OLAP) are essential elements of decision support, which has increasingly become a focus of the database industry [14]. Since data cube queries represent an important class of OLAP queries in decision support systems, the precomputation of the data cube is critical to improving response time. Over the past decade, numerous solutions for data cube generation have been proposed. In this chapter, we have introduced basic concepts relevant to OLAP and the data cube. We will discuss algorithms for data cube construction in the next chapter.

Chapter 3

Computing Full Data Cubes

3.1 Introduction

OLAP applications often require computation of multiple related group-bys. In order to improve the response time of the associated queries, precomputation of the date cube is critical.

In this chapter, we first introduce the basic techniques for computing aggregates. We then discuss efficient algorithms for the computation of the full data cube in the remainder of the chapter.

3.2 Basic Techniques for Computing Aggregates

As Graefe [23] points out, the basic techniques for computing aggregates are:

- To minimize data movement and consequent processing cost, compute aggregates at the lowest possible system level.
- If possible, use arrays or hashing to organize the aggregation columns in memory, storing one aggregate value for each array or hash entry.

- If the aggregation values are large strings, it may be wise to keep a hashed symbol table that maps each string to an integer so that the aggregate values are small. When a new value appears, it is assigned a new integer. With this organization, the values become dense and the aggregates can be stored as an N-dimensional array.
- If the number of aggregates is too large to fit in memory, use sorting or hybrid hashing to organize the data by value and then aggregate with a sequential scan of the sorted data.
- If the source data spans many disks or nodes, use parallelism to aggregate each partition and then coalesce these aggregates.

In practice, a number of these techniques have been applied in many of the most effective sequential data cube generation algorithms [24].

3.3 Algorithms for the Computation of Data Cubes

To date, a number of algorithms for fast computation of data cubes have been proposed and can be classified into three general categories: *top-down*, *bottom-up*, and *array-based*. In this section, we will review some well-known examples.

3.3.1 Top-Down Algorithms

Top-Down approaches for computing the data cube use more detailed group-bys to compute less detailed ones that contain subsets of attributes of the former. Examples include PipeSort, PipeHash [35], Partition-Cube, Memory-Cube [34], and Overlap [9].

For top down processing, Sarawagi [35] proposed a number of features that can optimize the computation of data cubes:

1. **Smallest Parent:** This optimization tries to compute a view from the smallest previously computed parent. Figure 3.1 shows a four-attribute cube. The dashed edges represents options for computing a view from parents. For example, AC can be computed from ABC, ACD or $ABCD$. Obviously, ABC or ACD are better choices for computing AC because their sizes are smaller than $ABCD$. Even among ABC and ACD, there may be big difference in terms of their sizes.

2. **Cache Results:** In order to reduce disk I/O, this optimization aims at caching (in memory) the results of a view from which other parents are computed in order to reduce disk I/O. For example, given the cube in figure 3.1, after computing ABC, we can keep it in the memory for computing view AB.

3. **Amortize Scans:** Amortize the disk I/O by computing as many views as possible from a given parent. For example, if the view $ABCD$ is stored on disk, then we can reduce disk read costs if all of ABC, ACD, ABD and BCD were computed in the one scan of $ABCD$.

4. **Share Sorts:** This is specific to the sort based algorithms and the objective is to share sorting cost across multiple views.

5. **Share Partitions:** This optimization is specific to the hash based algorithms. When the hash table cannot be fitted into memory, data is partitioned and aggregation is done for each partition that fits in memory. It can improve performance by sharing partition costs across views.

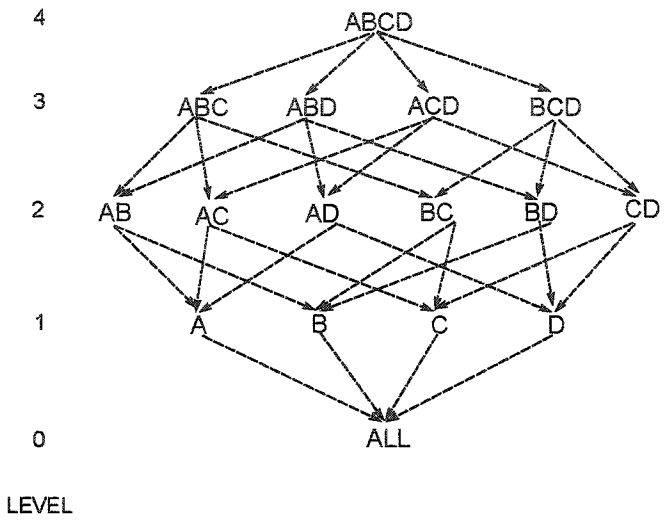


Figure 3.1: A top-down data cube lattice with 4 dimensions

We should note that the above optimizations are often contradictory. Therefore, it is extremely important to find effective trade-offs that are robust across variations in data set size and dimension count.

3.3.1.1 PipeSort

The Pipesort algorithm presented by Sarawagi et al. in [35] tries to minimize the number of sorts, while at the same time it seeks to compute a group-by from its smallest parent. The PipeSort algorithm is presented in Algorithm 1.

First, Pipesort annotates each edge of the search lattice of a data cube with two costs: *sort cost* and *scan cost*. Sort cost refers to the cost of computing views from a unsorted parent. Scan cost is the cost of computing views from a sorted parent.

Second, Pipesort proceeds level by level in the search lattice, starting from level $k = 0$ to level $k = d - 1$, where d is the total number of attributes, and ordering the

Algorithm 1 Pipesort

Input: A lattice of 2^d nodes augmented with "sort" and "scan" costs.
Output: A minimum cost spanning tree.
Method:

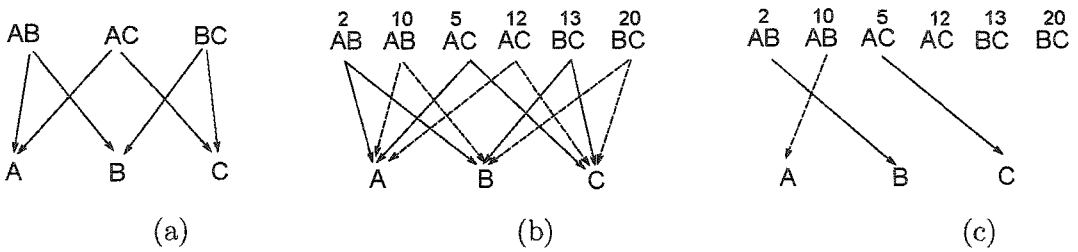
```
1: for level k = 0 to level d − 1 do
2:    Generate-Plan(k + 1 → k)
3:    for each group-by g in level k + 1 do
4:       Fix the sort order of g as the order of the group-by in level k that is connected
         to g by a "scan" edge
5:    end for
6: end for
```

attributes at each node/view and pruning edges to convert the search lattice into a *minimum cost spanning tree*(MCST). If the attribute order of a view is a prefix of the order of its parent in the tree, then this view can be computed from its parent without sorting, and the edge is marked "Sort" edge. Otherwise, the parent view has to be sorted to compute its child, and edge is marked "Scan" edge. The marking is constrained by the requirement that at most one edge out of any node can be marked "Scan" edge. PipeSort guarantees this using *weighted bipartite matching* technique [33]. This also minimizes the sum of edge costs at each level of the search lattice. The important step in Algorithm 1 is the Generate-Plan function, which uses the weighted bipartite matching algorithm to define the minimum cost mapping from level $k+1$ to level k. The algorithm is shown in Algorithm 2. Figure 3.2 illustrates how level 1 group-bys are generated from level 2. In order to use the weighted matching algorithm, we first make one additional copy of each level 2 group-by and associate these copies with the sort cost of the group-by. Once the weighted matching is performed, a minimum cost edge set connecting the two levels is added. The final step is to reorder the attributes of group-bys in level $k+1$ according to the order of attributes of its child in level k.

Algorithm 2 Generate Plan

Input: The nodes of level k and $k+1$.
Output: A minimum cost matching.
Method:

1: **for** $levelk = 0$ to $leveld - 1$ **do**
2: Create k additional copies of each level $k+1$ group-by
3: Connect each new vertex to the same set of child vertices as the original
4: Assign "Sort" costs to the new edges and "Scan" costs to the original.
5: Find the minimum cost matching on the augmented bipartite graph.
6: **end for**



(a) (b) (c)

Figure 3.2: (a) Possible Pathways. (b) Transformed Search Lattice. (c) Minimum Cost Matching.

Third, PipeSort adds a node corresponding to the relation R (raw data) to the tree, and adds an edge marked "Scan" edge from R to the root of the tree. PipeSort then converts the resulting tree into a set of paths such that every edge in the tree is presented in one and only one path, and all the edges in each path except the first edge are marked "Scan" edge.

When the PipeSort algorithm has completed, a minimum cost spanning tree is formed. Figure 3.3 provides a simple graphical illustration for a four dimensional data cube. Each pathway is called a *pipeline*. For example, $Rawdata \rightarrow CBAD \rightarrow CBA \rightarrow CB \rightarrow C \rightarrow All$ is a pipeline. Once prefix-ordered pipelines have been defined, we can compute all the views in a pipeline with a single sort and scan.

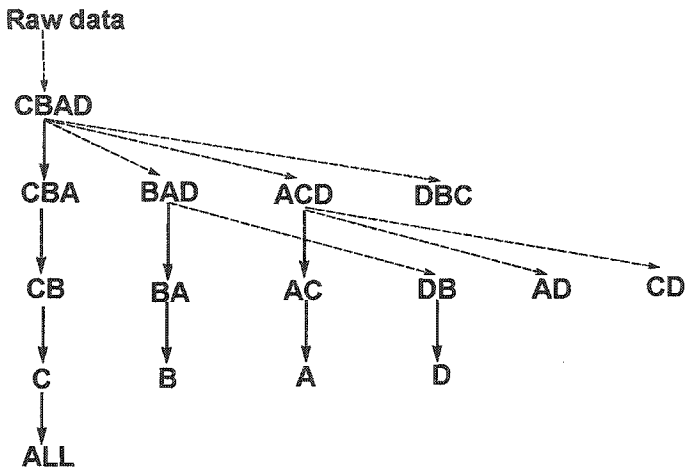


Figure 3.3: A four dimensional minimum cost pipesort spanning tree
Note: "scan" edges are dashed, while "sort" edges are solid.

3.3.1.2 Partitioned-Cube and Memory-Cube

When the relation is sparse and larger than main memory, the performance of PipeSort will be degraded significantly due to heavy I/O overheads in sorting and merging intermediate results. The Partitioned-Cube and Memory-Cube algorithms proposed by Ross in [34] attempt to solve this problem. The algorithm combination is based on two fundamental ideas that have been successfully used for performing complex operations such as sorting and joins over very large relations:

- Partition the large relations into fragments that fit in memory.
- perform the complex operation over each memory-sized fragment independently.

Partitioned-Cube Algorithm

Algorithm Partitioned-Cube is described in Algorithm 3. The algorithm utilizes a divide-and-conquer technique to compute data cubes. This algorithm works with

another algorithm called Memory-Cube that computes the data cube of a relation that fits in memory. The Memory-Cube algorithm will be explained in the next section.

The structure of Partitioned-Cube follows the recursive structure of the data cubes themselves. A data cube is obtained by fixing each possible value of a CUBE BY attribute B_j in turn and computing the tuples in the corresponding sub-datacube. B_j is the attribute chosen to partition relation R, and it is bounded by the number of buffers in memory and the domain cardinality of the attribute B_j. In subsequent invocations, the recursive algorithm will compute the data cube with the value ALL for the CUBE BY attribute B_j. Rather than re-reading the input relation of R for the ALL data cube, it reads the finest granularity cuboid F, which may be significantly smaller than R if there are many tuples in each partition, and is never larger than R.

We will demonstrate the algorithm by an example. Let's look at the figure 3.4, which shows the process of computing a data cube with four attributes $\{A, B, C, D\}$. The dashed arrows denote "Sort" edges, while solid arrows refers to "Scan" edges. Assume that the partition order is A, B, C, D.

First, the relation R is partitioned into fragments that fit in memory on attribute A. We then apply the Memory-Cube algorithm on each partition to compute the group-bys that have A as an attribute, e.g $ABCD, ABC, AB$ etc. When all the partitions have been processed, we are left with the finest granularity cuboid $F\{ABCD\}$.

Second, the finest granularity cuboid $F\{ABCD\}$ is read into memory and is partitioned on attribute B. Note that attribute A can be projected out during this phase. The partitions are processed to compute those cuboids that have attribute B, e.g. BCD, BD etc. Once all the partitions have been processed, the finest granularity

Algorithm 3 Partitioned-Cube(R,$\{B_1, ..., B_m\}$, A, G)

Input: A set of tuples R, possibly stored in horizontal fragments; CUBE BY attributes$\{B_1, ..., B_m\}$; attribute A to be aggregated; aggregate function $G()$.

Output: The data cube result for R over $\{B_1, ..., B_m\}$ in two horizontal fragments F and D on disk. F contains the finest granularity data cube tuples(i.e., grouping by all of $\{B_1, ..., B_m\}$, and D contains the remaining tuples. (F and D may themselves be further horizontally partitioned.)

Method:

1: **if** (R fits in memory) **then**
2: return Memory-Cube(R,$\{B_1, ..., B_m\}$, A, G)
3: **else**
4: { choose an attribute B_j among $\{B_1, ..., B_m\}$;
5: scan R and partition B_j into sets of tuples $R_1, ..., R_n$;
6: $/* n \leq card(B_j)$ and $n \leq$ number of buffers in memory */;
7: **for** $i = 1$ to n **do**
8: let(F_i, D_i) = Partitioned-Cube(R_i, $\{B_1, ..., B_m\}$, A, G);
9: **end for**
10: let(F', D') = Partitioned-Cube(F,$\{B_1, ..., B_{j-1}, B_{j+1}, B_m\}$, A, G);
11: D = the union of F', D' and the D_i's;
12: return(F, D); }
13: **end if**

cuboid $F\{BCD\}$ has been created.

Finally, the cuboid $F\{BCD\}$ fits in memory (in this example). Therefore, no further partition is needed. All the rest of the cuboids are computed from $F\{BCD\}$ in this stage.

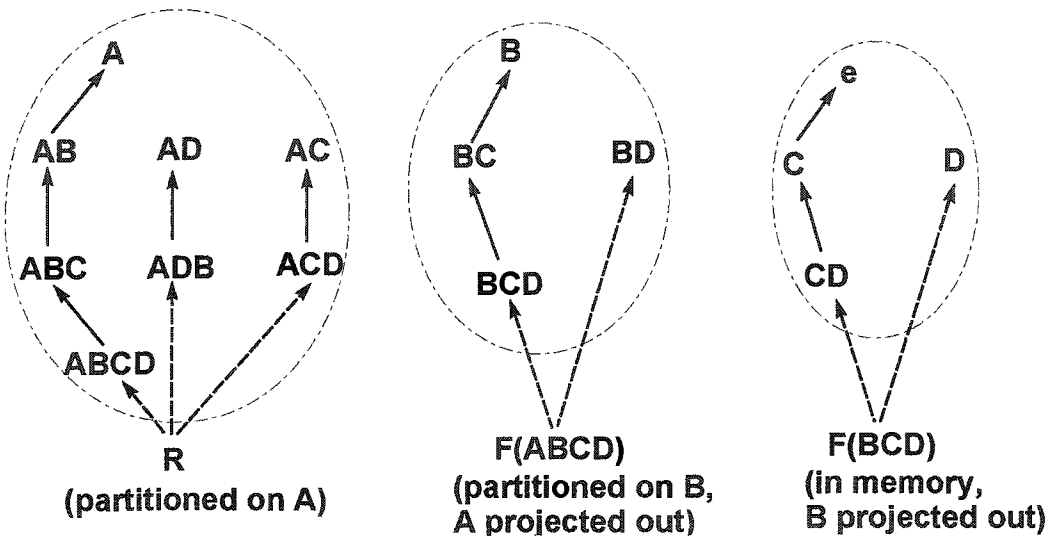


Figure 3.4: An illustrative example of Partitioned-Cube

As previously mentioned, the algorithm Memory-Cube is designed to work with Partitioned-Cube efficiently. We will describe the algorithm Memory-Cube in the following section.

Memory-Cube Algorithm

In order to compute a data cube, the only requirement of algorithm Memory-Cube is that the input relation fits in memory. The algorithm is described in Algorithm 4. The Memory-Cube algorithm takes advantages of the pipelining technique that is used in the PipeSort algorithm. Unlike the PipeSort, which tries to minimize

the total cost of computing all the cuboids, the Memory-Cube tries to minimize the number of pipelines, while at the same time, it tries to share as much sorting as possible by adjusting the sort order of pipelines. Hence the time spent on sorting for computing a data cube is significantly reduced. The algorithm that generates the minimum number of pipelines and the technique that is used to share sorts are keys to computing a date cube efficiently for the Memory-Cube algorithm. We will describe the path generation algorithm in the next section.

Paths Algorithm

We know that there are 2^d cuboids in a search lattice, and each path requires a sorting of the input relation at the root node of the path. Hence, minimizing the number of paths reduces the sorting cost for computing a data cube. Since there are $\binom{d}{d/2}$ group-bys with $d/2$ attributes (i.e., in the middle level of the lattice), and no path in the search lattice can pass through two nodes in the same level, the minimum number of paths is $\binom{d}{d/2}$ in order to cover all the cuboids. The algorithm Paths described in Algirithm 5 generates the minimum number of paths.

We now show how the Paths algorithm works by an example. The algorithm is recursive in structure and produces a set of paths in each round. Consider a data cube with four dimensions. Once the algorithm is executed, the paths generated in each step are shown as follows:

$$G(1) = D \rightarrow \epsilon$$

$$G(2) = C.D \rightarrow C \rightarrow \epsilon$$
$$\qquad\quad D$$

$$G(3) = B.C.D \rightarrow B.C \rightarrow B \rightarrow \epsilon$$
$$\qquad\quad B.D \rightarrow D$$

Algorithm 4 Memory-Cube(R,$\{B_1, ..., B_m\}$, A, G)

Input: A set of tuples R that fits in memory; CUBE BY attributes$\{B_1, ..., B_m\}$; attribute A to be aggregated; aggregate function $G()$.

Output: The data cube result for R over $\{B_1, ..., B_m\}$ in two horizontal fragments F and D on disk. F contains the finest granularity data cube tuples(i.e., grouping by all of $\{B_1, ..., B_m\}$, and D contains the remaining tuples.

Method:

```
1: sort R and combine all tuples that share all values of {B_1, ..., B_m};
2: /* Assume that tuples are sorted according to the first sort order */;
3: for each sort order do
4:   initialize accumulators for computing aggregates at each granularity;
5:   combine first tuple into finest granularity accumulator;
6:   for each subsequent tuple t do
7:     compare t with previous tuple, to find the position j of the first sort order
       attribute at which they differ;
8:     if (j is greater than the number of common attribute between this sort order
       and the next) then
9:       re-sort the segment from the previous tuple t' at which this condition was
         satisfied up to the tuple prior to t according to the next sort order;
10:    end if
11:    if (grouping attributes of t differ from those in finest granularity accumulator)
       then
12:      output and then combine each accumulator into coarser granularity accu-
         mulator, until the grouping attributes of accumulator match with those of
         t;
13:      /* the number of combining depends on the sort order length and on j */;
14:    end if
15:    combine current tuple with the finest granularity accumulator;
16:  end for
17: end for
```

Algorithm 5 Paths($\{B_1, ..., B_m\}$)

Input: CUBE BY attributes$\{B_1, ..., B_m\}$;
Output: A set $G(j)$ of $\binom{j}{j/2}$ paths in the search lattice that cover all the nodes;
Method:

```
1: if (j = 0) then
2:    return a single node with an empty attribute list, ε;
3: else
4:    { let G(j − 1) = paths{B_1, ..., B_{j−1}};
5:    let G_l(j − 1) and G_r(j − 1) denote two replicas of G(j − 1);
6:    prefix the attribute list of each node of G_l(j − 1) with B_j;
7:    for each path N_1 → ... → N_p in G_r(j − 1) do
8:       remove node N_p and the edge into N_p (if any) from G_r(j − 1);
9:       add node N_p to G_l(j − 1);
10:      add an edge from node B_j.N_p to node N_p in G_l(j − 1);
11:   end for
12:   return the union of the resulting G_l(j − 1) and G_r(j − 1)};
13: end if
```

$$C.D \rightarrow C$$

$$G(4) = A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$$

$$A.B.D \rightarrow A.D \rightarrow D$$

$$A.C.D \rightarrow A.C \rightarrow C$$

$$B.C.D \rightarrow B.C \rightarrow B$$

$$B.D$$

$$C.D$$

The paths shown in $G(4)$ are the final paths generated by the algorithm Paths. In order to compute all the views in a pipeline with a single sort and scan, the attribute order of the paths will be adjusted to follow the prefix property. $G(4)$ will be re-ordered as follows:

$$G(4) = A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$$

$$D.A.B \rightarrow D.A \rightarrow D$$

$C.A.D \rightarrow C.A \rightarrow C$

$B.C.D \rightarrow B.C \rightarrow B$

$B.D$

$C.D$

Finally, the order of the paths will be reordered alphabetically according to the first node of each path, so that it can take advantage of common sort orders:

$G(4) = A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

$B.C.D \rightarrow B.C \rightarrow B$

$B.D$

$C.A.D \rightarrow C.A \rightarrow C$

$C.D$

$D.A.B \rightarrow D.A \rightarrow D$

We note that the input relation has to be sorted several times in order to compute a data cube. Actually, the input relation has to be sorted in the order determined by the root/parent node of each path when processing that path. The primary benefit of the Memory-Cube algorithm is the sharing of sort costs. Consider the above example. The relation will be sorted 6 times in the following order:$A.B.C.D$, $B.D$, $C.A.D$, $C.D$, $D.A.B$. Here, we can see that some sorts can be shared between two successive paths. For example, the relation has been sorted in the order $B.C.D$. When processing the next pipeline $B.D$, the entire relation does not need to be resorted. Only each block of tuples that shares B values needs to be independently sorted in the $B.D$ order. Once all blocks are sorted, the entire relation will be in the desired $B.D$ order. This technique saves a significant amount of time on sorting.

3.3.2 Bottom-Up Algorithms

Unlike Top-Down algorithms, Bottom-Up algorithms process from the bottom of the lattice, and work their way up towards the cuboids with finer granularity. Figure 3.5 shows an example of a Bottom-Up data cube lattice with 4 dimensions. Note that as the dimensions increase, the high-dimension cuboids become increasingly sparse (i.e., relatively few of the possible records actually exist). Since top-down algorithms tend to utilize views in the upper portion of the lattice as pipeline input sets, sort costs can grow significantly in large sparse spaces. In order to reduce the sorting cost for sparse relations, bottom-up method has been proposed.

3.3.2.1 BUC

In this section, we will introduce one of the most well-known bottom-up algorithms called Bottom-Up Computation (BUC) [13].

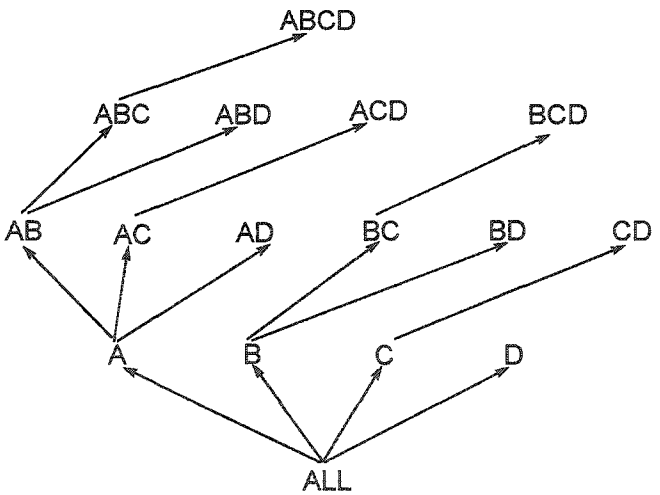


Figure 3.5: A Bottom-Up data cube lattice with 4 dimensions

Algorithm 6 ButtomUpCube(input,dim)

Input: The partition to be aggregated, and the current dimension dim;
Output: A single record that represents the aggregated input;
Recursively, output CUBE(dim, ...,number of dimension);
Method:

```
1: Aggregate(input); {/*place result in outputRec)*/}
2: if (input.count == 1) then
3:    WriteAncestor(); {/*write ancestor records*/}
4:    return;
5: end if
6: write outputRec.{/*write the result from step 1*/}
7: for (d = dim; d < the total number of dimensions ; d++) do
8:    let C = the cardinality of dimension d;
9:    Partition(input,d); {/*partition input on dimension d according to its unique
      values*/}
10:   for (i = 0; i < C; i++) do {/*for each partition*/}
11:     BottomUpCube(partition, d+1); {/*recursive call BottomUpCube*/}
12:   end for
13: end for
```

BUC is a recursive algorithm. It takes a relation and the current dimension (the first attribute of the current relation) as input. In the first step, it aggregates the entire input and writes the result to a single record. In the main loop of the algorithm, for each dimension d between dim and the total number of dimensions, the input is partitioned on dimension d into C partitions, where C (cardinality) represents the number of distinct values for the current dimension to be partitioned. When the partitions have been defined, BUC then iterates through the partitions, recursively calling the BUC algorithm, this time using the current partition as input, along with the incremented current dimension variable. Once the current set of partitions have been computed, the algorithm will backtrack and process the next partition at the previous level of recursion.

Figure 3.6 provides an illustration of the BUC partition procedure for computing

a four dimensional data cube. In this case, BUC first computes the "all" group-by. Next, BUC will partition the input on attribute A, producing partitions $a1, a2, a3$, and then recurses on partition $a1$. It will then aggregate on $< a1 >$ to produce a single record for group-by A before partitioning $< a1 >$ into its $< a1b1 >$ and $< a1b2 >$ components. After aggregation on $a1$ is done, it will partition $a1$ on dimension B. It recurses on the $< a1b1 >$ partition and writes a single $< a1b1 >$ tuple for the group-by AB. This recursive partitioning will continue until the last dimension has been reached. Eventually, the backtracking will return the algorithm to the $< a2 >$ partition, at which point the whole process is repeated.

a1	b1	c1	d1
			d2
		c2	d1
			d2
	b2	c1	d1
			d2
		c2	d1
			d2
a2	b1	c1	d1
			d2
		c2	d1
			d2
	b2	c1	d1
			d2
		c2	d1
			d2
a3	b1	c1	d1
			d2
		c2	d1
			d2
	b2	c1	d1
			d2
		c2	d1
			d2
	b3	c1	d1
			d2

Figure 3.6: BUC Partitioning

The main reasons that the BUC algorithm is well-suited to sparse, high dimensional data cube problems are:

1. It builds the CUBE from the most aggregated group-by to the least aggregated group-bys, which allows BUC to share partition costs and to prune the computation.

In Step 2 of the algorithm, it checks to see if the size of the current partition is equal to one. If it is, then we know that there is no value in continuing the recursion since no further partitioning can be performed. We therefore write out the aggregates for all ancestors and return immediately. For example, when we encounter the tuple $< a1b1c2 >$, we know that we can write the aggregate value for $< a1b1c2d >$ without further processing. Because many partitions will in fact have a size of one in sparse spaces, this short circuiting can significantly improve performance.

2. The BUC algorithm also utilizes the partition idea of the Partition-Cube algorithm. As it recursively partitions the input, BUC divides the data into smaller and smaller fragments. Consequently, it is increasingly likely that these partitions fit entirely into main memory, thus possibly reducing the reliance on more expensive external memory sorting.

3.3.3 Array-Based Algorithms

While the algorithms (PipeSort, Partition-Cube, BUC) we described above work specifically with relational tables, another approach – corresponding to the MOLAP (multidimensional online analytical processing) model – has been proposed. Unlike ROLAP systems, MOLAP systems store data in multidimensional arrays. Each cell of the cube is represented by an element of the array. Moreover, only the measure attributes are stored and the position of the cell determines the values of the dimensional attributes (In ROLAP systems, both the measure attributes and the feature attributes are stored).

3.3.3.1 Array-Cube

To date, relatively few algorithms for MOLAP computation have been presented in the academic literature (though the MOLAP approach has been used commercially). One of best known array-based methods is the Array-Cube algorithm proposed by Zhao et al.[39]. With the Array-Cube algorithm, 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. This algorithm performs well because it makes use of an array representation which allows direct access to the needed cells. Having said that, it is strongly dependent on the availability of main memory and cannot be used – at least in its standard form – for large sparse data sets. Since our research focuses on ROLAP algorithms, we are not going to discuss Array-Cube any further. Details can be found in [39].

3.3.4 Conclusion

In this chapter, we introduced several algorithms for the efficient computation of the data cube. The PipeSort algorithm executes a minimum weight matching algorithm on a bipartite graph to reduce the cost of the search lattice. However, it is not suitable for large and sparse data. The Partitioned-Cube algorithm has been proposed to address this problem. Partition-Cube is a divide-and-conquer algorithm which provides fast computation of sparse data sets. This technique minimizes the disk I/O for inputting the relation and outputting the data cube and demonstrates the advantage of sharing the sort orders in the data cube computation. Finally, we presented the BUC algorithm, another well-known solution used to efficiently compute the data cube for sparse data sets. In contrast to top-down algorithms, BUC computes

the data cube from the coarsest granularity group-bys to the finest granularity group-bys.

In the "real world", data sets are often large and sparse. The Partition-Cube, one of the most well-known sequential cube construction methods, forms the basis of our research, and we will discuss it in greater detail in the next chapter.

Chapter 4

Improving the Performance of the Partitioned-Cube and the Memory-Cube

In the last chapter, we described the PipeSort algorithm, one of the primary methods for the efficient construction of the data cube. The chief limitation of PipeSort, however, is that it does not scale well with respect to the number of CUBE BY attributes in the data cube query. When the underlying relation is sparse and much larger than available memory, many of the cuboids are also larger than the available memory. Sorting these cuboids requires external memory techniques, the result being a considerable increase in I/O costs. In contrast to the PipeSort algorithm, the Partitioned-Cube algorithm utilizes a divide-and-conquer technique to compute data cubes. Specifically, it partitions the relation on a chosen attribute, where the number of partitions is bounded by both the cardinality of the current partitioning attribute and the available memory. At that point, the Memory-Cube algorithm is run on each memory-sized partition. The final result is a dramatic reduction in I/O costs.

However, the original algorithm and implementation suffer from two significant problems which prevent it from achieving optimal performance. In this chapter, we

will discuss the problems that the Partitioned-Cube algorithm presents. We will then propose solutions that address these concerns.

4.1 Weaknesses in the Partitioned-Cube and the Memory-Cube

Recall that the Partitioned-Cube algorithm generates a set of sort paths that are used to drive the cube computation. Specifically, the Paths algorithm produces $\binom{d}{d/2}$ *pipelines* that dictate the sharing of sort costs. However, an examination of the pathways generated by the Paths algorithms reveals that the sort paths may conflict with the requirements of the primary Partitioned-Cube algorithm. The following example, based upon Figure 4.1, illustrates the potential problem. To begin, note that there are four groups of pipelines, with each group relative to a given partitioning attribute.

- A partition:

  $A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

  $A.D.B \rightarrow A.D$

  $A.C.D \rightarrow A.C$

- B partition:

  $B.C.D \rightarrow B.C \rightarrow B$

  $B.D$

- C partition:

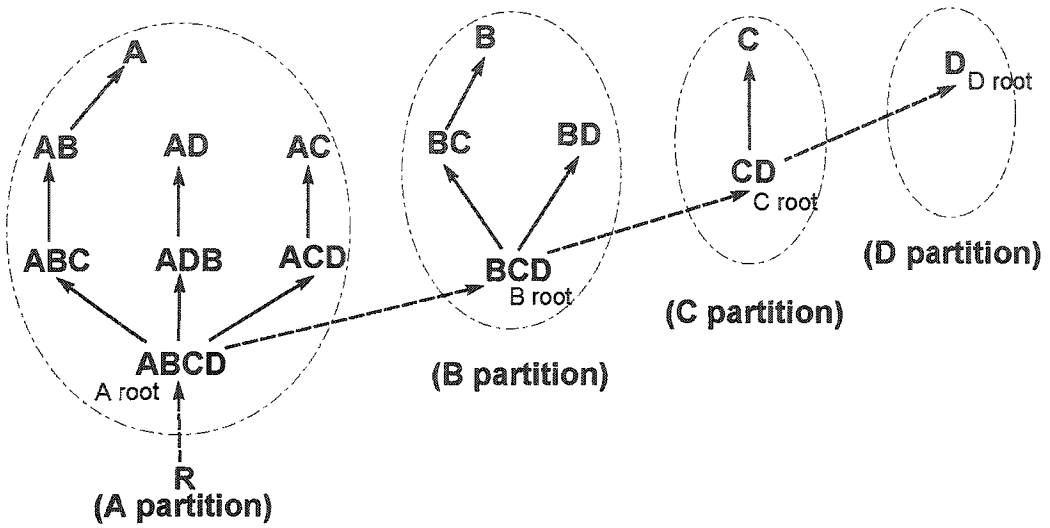


Figure 4.1: Partitions of a data cube with 4 dimensions

$C.D \rightarrow C$

- D partition:

  D

Now, let's look at the output of the Paths algorithm. Given the same example, the *complete* set of generated pipelines can be listed as:

$G(4) = A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

$A.B.D \rightarrow A.D \rightarrow D$

$A.C.D \rightarrow A.C \rightarrow C$

$B.C.D \rightarrow B.C \rightarrow B$

$B.D$

$C.D$

We can divide the paths into four groups in terms of the partition attribute:

- A partition:

  $A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

  $A.B.D \rightarrow A.D \rightarrow D$

  $A.C.D \rightarrow A.C \rightarrow C$

- B partition:

  $B.C.D \rightarrow B.C \rightarrow B$

  $B.D$

- C partition:

  $C.D$

- D partition:

  Note: No paths in this group.

In order to compute all the views in a pipeline, we have to reorder the attributes of each node in a pipeline. In other words, we must ensure that the attributes of the views in a specific pipeline are arranged in *prefix* order. After adjusting the attribute order and re-ordering the order of paths for each group, the paths will be:

- A partition:

  $A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

  $C.A.D \rightarrow C.A \rightarrow C$

  $D.A.B \rightarrow D.A \rightarrow D$

- B partition:

  $B.C.D \rightarrow B.C \rightarrow B$

  $B.D$

- C partition:

  $C.D$

- D partition:

  Note: No paths are in this group.

It should be clear that the paths required by the Partitioned-Cube algorithm and those actually generated by the Paths algorithm do not match. Although not immediately obvious, this creates a serious problem when the Partitioned-Cube algorithm attempts to merge the aggregated partitions. Recall that a key feature of Partitioned-Cube is that it partitions the relation on a given attribute in terms of the distinct values in that attribute's domain. For example, the attribute A is composed of partitions on A_1, A_2, A_3, etc. The objective is to allow a merge of distinct partitions such that the final combined result contains no duplicate records. Consider our running example. When the relation is partitioned on A then, as per the Partitioned-Cube algorithm, all cuboids that have A as an attribute will be computed (i.e., $A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A$, $A.D.B \rightarrow A.D$, $A.C.D \rightarrow A.C$). However, the paths actually generated by the Paths algorithm on the A attribute contain the views C and D. Since the partition attribute is A, we can be guaranteed that the value of A in each partition is distinct. On the other hand, the values of C and D are not necessarily distinct in each partition. As a result it is entirely possible that

when Partitioned-Cube combines the aggregated partitions into a single set, the final result will contain duplicate records.

One solution to the record duplication problem might be to re-order the attributes prior to arranging them into groups. In this case, the paths would be:

$G(4) = A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

$B.C.D \rightarrow B.C \rightarrow B$

$B.D$

$C.A.D \rightarrow C.A \rightarrow C$

$C.D$

$D.A.B \rightarrow D.A \rightarrow D$

We would then divide the paths into four groups according to the partitioning attribute:

- A partition:

  $A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

- B partition:

  $B.C.D \rightarrow B.C \rightarrow B$

  $B.D$

- C partition:

  $C.A.D \rightarrow C.A \rightarrow C$

  $C.D$

- D partition:

  $D.A.B \rightarrow D.A \rightarrow D$

This solution does, in fact, solve the merge problem. However, its use would generate entirely new problems. Consider that when Partitioned-Cube is employed, the relation is first partitioned and aggregated on A. We then take the finest granularity cuboid $ABCD$ as input, project out the attribute A, partition on B, and finally compute the views that have B as an attribute. This process of reducing, partitioning, aggregating, and combining is repeated for each attribute in turn. I/O costs are in theory reduced during execution as we continue to select the smallest parent from which to compute its children. However, there is one significant problem with this new approach. Specifically, during iteration i it is not possible to project out the partitioning attribute from iteration $i-1$ since it may be required during the current round. Referring to our current example, we see for instance that we cannot drop the A attribute since it is still required during the computation of the C partition. As a consequence, intermediate views remain quite large, thereby reducing or even eliminating the benefit of the proposed technique.

From the above discussion, it should be clear that we need a new solution to solve the merge problem so that the Partitioned-Cube and Memory-Cube can work together efficiently. In the following section, we present a method for path generation that works well with the Partitioned-Cube algorithm.

4.2 Augmentation of the Paths Algorithm

In this section, we generate a list of paths for the Partitioned-Cube algorithm by modifying the paths generation method presented in [34]. The New-Paths algorithm is described in Algorithm 7. It should be noted that the original Paths algorithm generates path listings such that attributes are presented in alphabetical order. Our

new method also provides this guarantee.

Algorithm 7 New-Paths

Input: A list of Paths generated by original Paths algorithm;
Output: A list of Paths in the search lattice that cover all the nodes;
Method:

```
1: for (each path Path[i] in the list) do
2:    compare the first attribute Attr between the first node Node_first and the last
      node Node_last of current path.
3:    if (different) then
4:       remove Node_last from current path Path[i]
5:       create a new path and add Node_last to the new path.
6:    end if
7: end for
8: sort the list of paths according to the root node of each path by alphabetical
   order.
9: for (each path in the list) do
10:   if (there is only one node in current path) then
11:      compare this node in current path with the last node of each path, find the
         best parent,which is that they have longest prefix attribute order.
12:      remove the current path and append the node to proper path
13:   end if
14: end for
15: adjust the attribute order for each path.
16: sort the list of paths according to the root node of each path by alphabetical
    order.
```

We now demonstrate how the algorithm works by a concrete example. Consider the paths generated by the Paths algorithm for a 4-dimensional data cube:

$A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

$A.B.D \rightarrow A.D \rightarrow D$

$A.C.D \rightarrow A.C \rightarrow C$

$B.C.D \rightarrow B.C \rightarrow B$

$B.D$

$C.D$

Recall that the original Paths algorithm generates path listings such that attributes are presented in alphabetical order. In order to compute views in terms of the partitioning attribute, the leading attribute of each view in a pipeline must be the same. However, the last node in a path generated by the original Paths algorithm might not contain the partitioning attribute. The mechanism of the original Paths algorithm guarantees that this difference only happens in the last node. Therefore, we need to move the last node to a proper path if the leading attribute of the last node and first node in a pipeline are different.

The New-Paths algorithm proceeds as follows. We begin by comparing the leading attribute of both the last and first view in a pipeline. If it is different, we create a new, initially empty, path and then transfer the last view to this new pipeline. Following this step, the paths will be as follows:

$A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

$A.B.D \rightarrow A.D$

$A.C.D \rightarrow A.C$

$B.C.D \rightarrow B.C \rightarrow B$

$B.D$

$C.D$

D

C

We now sort the path lists in alphabetical order as dictated by the first view in each pipeline.

$A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

$A.B.D \rightarrow A.D$

$A.C.D \rightarrow A.C$

$B.C.D \rightarrow B.C \rightarrow B$

$B.D$

C

$C.D$

D

Next, we identify those paths that contain just a single node. For the view $Node_{current}$ in such paths, we compare it to the trailing view $Node_{last}$ in the remaining paths. If $Node_{current}$ is a prefix of $Node_{last}$, we append it to associated path. At the same time, we delete the now empty path that originally contained $Node_{current}$. We now have the following set of pathways:

$A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

$A.B.D \rightarrow A.D$

$A.C.D \rightarrow A.C$

$B.C.D \rightarrow B.C \rightarrow B$

$B.D$

$C.D \rightarrow C$

D

In the final phase, we adjust the attribute order of views in the paths to ensure the prefix ordering property. In addition, we sort the path lists themselves to permit a full sharing of sort work between paths that share common leading attributes. With our running example, the final paths will be:

$A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

$A.C.D \rightarrow A.C$

$A.D.B \rightarrow A.D$

$B.C.D \rightarrow B.C \rightarrow B$

$B.D$

$C.D \rightarrow C$

D

We now briefly talk about the complexity of our new path algorithm. In the first step, we need to compare the leading attribute of the last node with first node for each pipeline. The cost of this step is $O(n)$, where n is the number of paths, $n=\binom{d}{d/2}$, d is the number of dimensions. The second step, we sort the list of paths. Additional paths were added to the list in last step. The number of paths in the worst case is $2n$. This means that every last node in each pipeline has to be removed to a new path. The cost of sorting the list of paths is bounded on $O(n^2)$ using bubble sort. The third step, we need to put the path that has only one node to a proper path. The cost of this step is bounded on $O(n^2)$. The last step, we need to sort the final list of paths, the cost is $O(m^2)$, where m is the final number of paths. We will present the calculation in the next section. Therefore, the complexity of our new path algorithm is bounded on $O(n^2 + m^2)$.

The original Paths algorithm guarantees the minimum number of paths, that is, $\binom{d}{d/2}$. In our new method, the number of paths is not guaranteed to be minimal as there may be extra paths added to the original list. (We note that this is also true of the PipeSort paths described in [35]). However, **our new method improves upon the original by reducing the size of intermediate parents and better coordinating the sharing or sort partitions.** We will provide a performance evaluation in Section 4.5.

We conclude this section by providing a practical path calculation measure. Let's look at Figure 4.2. When attribute A is projected out from A partition, we can get a list of paths that do not contain attribute A. This list of paths covers all the views of the remaining attributes(i.e. B,C,D in this case), and the number of paths is the minimal, which is $\binom{(d-1)}{(d-1)/2}$. In B partition, attribute B is projected out, we will have a list of paths, which has the minimal number of paths $\binom{(d-2)}{(d-2)/2}$ for the remaining attributes C, D. The same calculation on C partition, the number of paths in C partition is $\binom{(d-3)}{(d-3)/2}$. We also note that the list of paths in the last dimension has only one path. In this example, the total number of paths can be calculated as follows:

The number of paths of group A: $N_A = \binom{(d-k)}{(d-k)/2} = 3, (k = 1)$.

The number of paths of group B: $N_B = \binom{(d-k)}{(d-k)/2} = 2, (k = 2)$.

The number of paths of group C: $N_C = \binom{(d-k)}{(d-k)/2} = 1, (k = 3)$.

The number of paths of group D: $N_D = 1$.

So the total number of paths: $N = N_A + N_B + N_C + N_D = 3 + 2 + 1 + 1 = 7$.

We extend our path number calculation to a data cube with d dimensions. The number of paths generated by the new algorithm is:

$$N = \sum_{k=1}^{d-1} \binom{(d-k)}{(d-k)/2} + 1$$

Fox example, the number of paths of a 10-dimensional data cube calculated by the above formula is 274, while the minimal number of paths is $\binom{10}{5} = 252$. There are 22 extra paths added to the list of paths. Consider another data set with 12 dimensions, there are 988 paths in our new path algorithm, while the minimal number of paths is $\binom{12}{6} = 924$. Thus, there are 64 additional paths in this case.

So far, we have made changes on the Paths algorithms, thereby allowing the

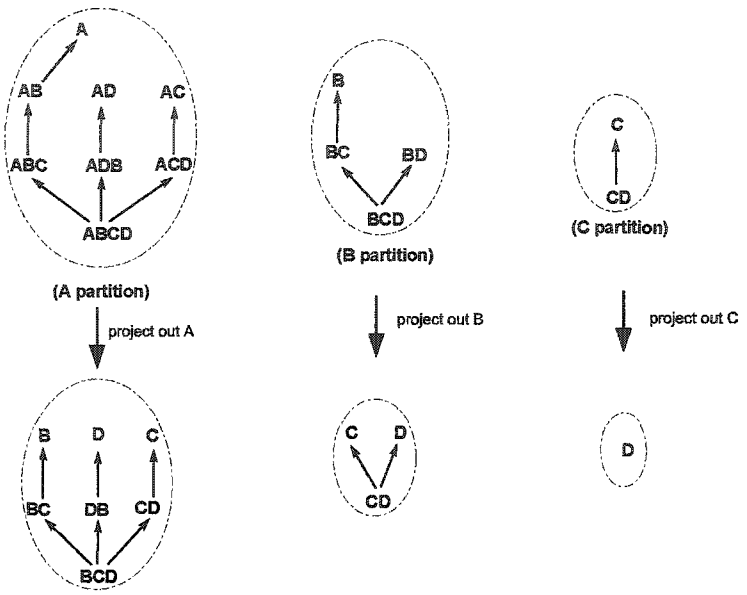


Figure 4.2: Calculating the number of paths for a 4-dimensional data cube

Partitioned-Cube algorithm to work with the Memory-Cube algorithm efficiently. In the next section, we will improve the performance of the Memory-Cube algorithm by adding the proper root node for each path.

4.3 Change on the Sort Parent

Recall that in the Memory-Cube algorithm, the raw data is loaded just once, and then the algorithm sorts the raw data according to the sort order of each pipeline. Consider an example with four dimensions and assume that the raw data fits in memory. After reordering the paths and adding a parent node to each pipeline, the paths are shown as follow:

$Raw \dashrightarrow ABCD \rightarrow ABC \rightarrow AB \rightarrow A \rightarrow all$

$ABCD \dashrightarrow BCD \rightarrow BC \rightarrow B$

$ABCD \dashrightarrow BD$

$ABCD \dashrightarrow CAD \rightarrow CA \rightarrow C$

$ABCD \dashrightarrow CD$

$ABCD \dashrightarrow DAB \rightarrow DA \rightarrow D$

Note that the first node in each pipeline represents the *input view* for that pipeline. In other words, the first node is sorted according to the prefix ordering established for the associated pipeline and is then scanned in order to compute (i.e., aggregate) the remaining views. By selecting $ABCD$ as the input view in the previous example, we directly benefit from any compression – by virtue of aggregation – that may have taken place on the raw data set. If the raw data set is large and sparse, the view $ABCD$ may be considerably smaller.

Now, look at our new paths in Figure 4.1. We build the paths according to the partition attribute so that we can always take the finest granularity view to compute the next group of views. For example, in group A, the partition root node is $ABCD$; therefore, all the views in partition A can be computed from this partition root node. Because the views in group B do not contain the attribute A, we can now use view BCD as the parent. The same is true for the C partition and D partition. Note that it is clearly faster to compute views in the B partition from view BCD than to compute them from view $ABCD$ because view BCD is likely to be much smaller than view $ABCD$. Therefore, after reordering the sort paths and adding a more cost-effective parent to each pipeline, the paths are:

$Raw \dashrightarrow ABCD \rightarrow ABC \rightarrow AB \rightarrow A \rightarrow all$

$ABCD \dashrightarrow ADB \rightarrow AD$

$ABCD \dashrightarrow ACD \rightarrow AC$

$ABCD \dashrightarrow BCD \rightarrow BC \rightarrow B$

$BCD \dashrightarrow BD$

$BCD \dashrightarrow CD \rightarrow C$

$CD \dashrightarrow D$

This is the final list of paths that will be used in the Memory-Cube algorithm. We will discuss our implementation in the following section.

4.4 Implementation

In this section we briefly discuss the fundamental design and implementation issues.

The PipeSort algorithm was first presented in [35] and a fully optimized implementation – for both sequential and parallel architectures — was described in [19, 15]. In this thesis, we implemented the Partitioned-Cube algorithm and Memory-Cube algorithm, as well as their adapted versions, in C++.

The input data is assumed to consist of 4-byte integer values on all grouping and aggregated attributes, and we use summation as the aggregation function. Moreover, in our implementation, sorting is performed in-place (on pointers to tuples) using quicksort [27]. The complexity of quicksort in the average case is $O(nlogn)$.

In addition, a number of third-party software libraries were also utilized. Thread functionality is provided by the Pthreads (POSIX Threads) libraries [8]. The use of threads improves the performance of I/O by allowing I/O threads and computational threads to execute concurrently. We have also incorporated the LEDA graph libraries into our data cube code base [29]. We selected LEDA because of its rich collection of fundamental data structures (including linked lists, hash tables, arrays, and graphs),

the extensive implementation of supporting algorithms, and the C++ code base. Though there is a slight learning curve associated with LEDA, the package has proven to be both efficient and reliable.

4.5 Experimental Evaluation

In this section, we discuss the performance of our implementation under a variety of test scenarios.

4.5.1 Default Parameters

We utilize a set of base parameters in each of the tests. These base parameters are (with defaults listed in parenthesis):

1. Fact Table Size (1 million rows)
2. Dimension Count (8)
3. Cardinality (10)
4. Skew (uniform distribution)

When default values are not, or cannot, be used for a particular test, this fact will be clearly noted.

4.5.2 Performance Study of Memory-Cube

The tests were performed on a 1.7 GHz Xeon Processor system with 1 GB RAM, and running Linux (kernel version 2.4.9-13). Since the algorithm Memory-Cube is

proposed for sparse and high dimension data sets, we test the data sets with different cardinalities and/or different dimensions in our experiments. Moreover, the most important feature of Memory-Cube is its ability to share its sorting costs. Thus, in our tests, we measure the sort time and the computation time (CPU cost).

In order to compare the performance of PipeSort and Memory-Cube, as well as our adapted version of Memory-Cube, we suppress the I/O cost for data cube output so that we could get an accurate measure of the CPU cost of computing data cubes. Actually, the I/O cost of both PipeSort and Memory-Cube is the same in that they output identical views when computing data cubes in memory. By comparing the in-memory processing time, we can measure the real difference among these three approaches.

We generate two groups of data sets. Sets in both groups contain 1,000,000 tuples. One group fixes the number of dimensions at 8, but allows for different cardinalities ranging from 10 to 1000. Another group has the same cardinality of 10, but different dimension counts varying from 4 to 11.

4.5.2.1 Memory-Cube: Shared Vs Nonshared

In this section, we measure the benefits of sharing the sorting workload compared with full resorting for algorithm Memory-Cube. Performance figures are given in Figure 4.3, 4.4, 4.5. The algorithm is run twice, once normally and the second time with sharing turned off. The Memory-Cube (nonshared) is executed without sharing the sort work. The Memory-Cube (shared) is the version of Memory-Cube with sharing.

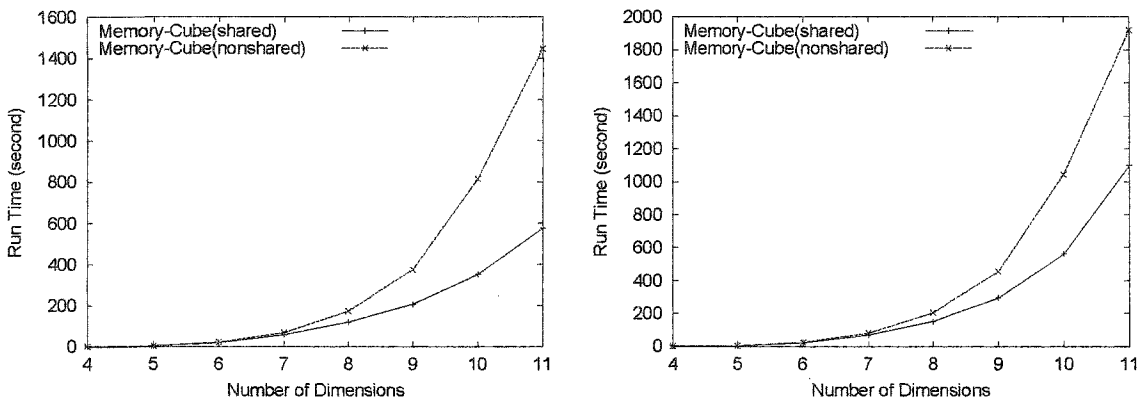


Figure 4.3: (a) Sort time in seconds(records = 10^6, cardinality=10)
(b) Computation time in seconds(records = 10^6, cardinality=10)

Figure 4.3 and Figure 4.4 confirm that the benefits of sharing sort work are significant. From Figure 4.3, as the number of dimensions increases, we can clearly see that the growth rate of both sorting time and CPU time for Memory-Cube with sort sharing is considerably lower than for the version of Memory-Cube without sharing. In Figure 4.4 (a), it is interesting to note that the sorting time of Memory-Cube with sharing is even reduced with increasing cardinality. The CPU time with sort sharing shown in Figure 4.4 (b) almost stays at the same level.

Figure 4.5 demonstrates the benefits of sharing the sorting work. The ratio of sort time and computation time are reported as a function of the number of cube by attributes and a function of the cardinalities respectively. The graph shows that the sorting time can be reduced significantly (over 60 percent of the original time). The computation time can be reduced over 50 percent of the original time.

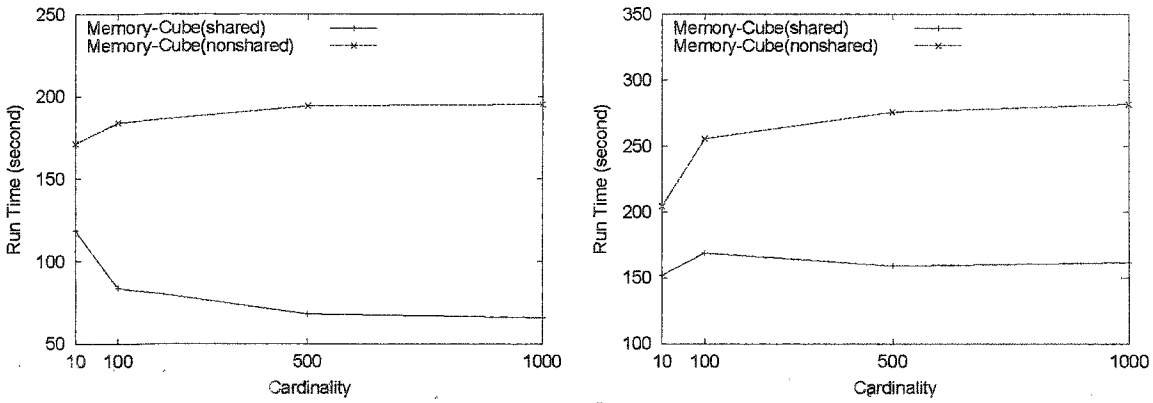


Figure 4.4: (a) Sort time in seconds(records = 10^6, dimensions = 8)
(b) Computation time in seconds(records = 10^6, dimensions = 8)

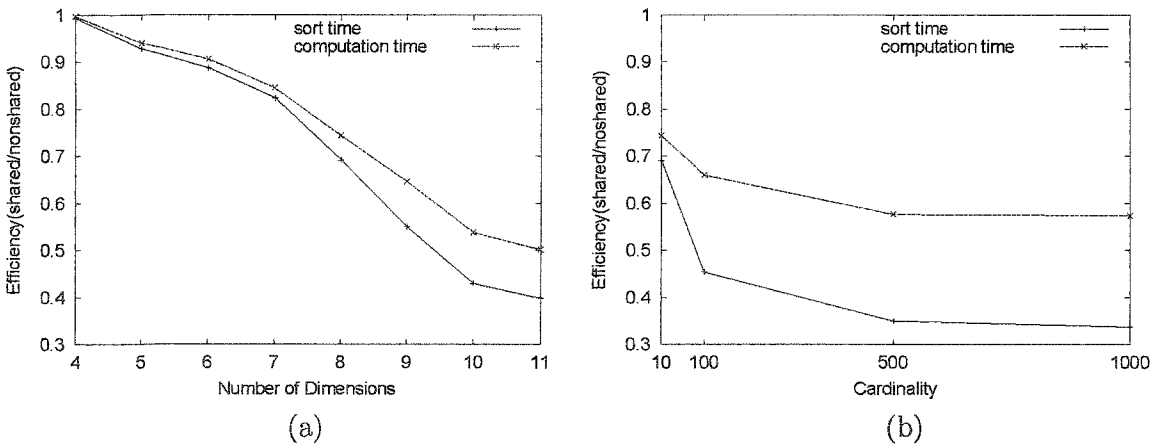


Figure 4.5: (a) Efficiency(records = 10^6, cardinality=10)
(b) Efficiency(records = 10^6, dimensions = 8)

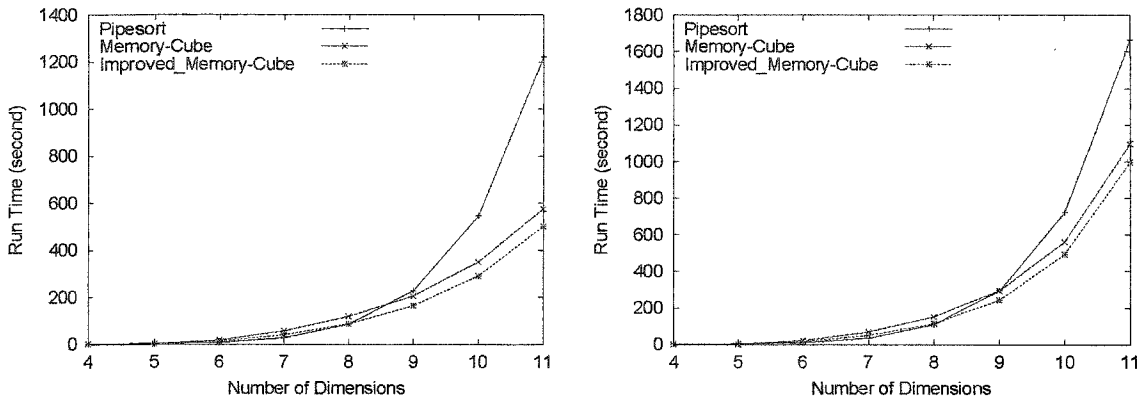


Figure 4.6: (a) Sort time in seconds(records = 10^6, cardinality=10)
(b) Computation time in seconds(records = 10^6, cardinality=10)

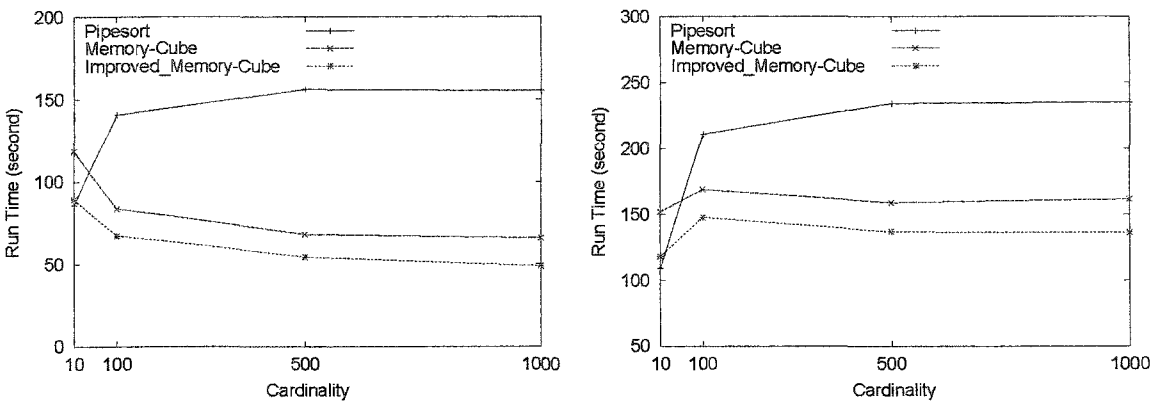
4.5.2.2 PipeSort Vs Memory-Cube (Improved-Memory-Cube)

In this section, we give a detailed comparison of two fast data cube algorithms, PipeSort and Memory-Cube. Also, we compare our improved version of Memory-Cube to the original Memory-Cube algorithm.

From Figure 4.6, we can clearly see that there is a slower growth rate for the Memory-Cube algorithm as the number of dimensions increases. When the number of dimensions is greater than 9, the performance of PipeSort becomes much slower than the Memory-Cube due to more significant benefits of sharing the sort workload for Memory-Cube. Our improved method remains consistently beneath the other two performance curves (about 15% faster than original Memory-Cube) by virtue of the fact that it offers the greatest reduction in sort costs.

Figure 4.7(a) shows that the sorting time of PipeSort goes up with increasing number of cardinalities. However, the sorting time of Memory-Cube is reduced. The impact on CPU time, illustrated in Figure 4.7(b) is similar.

Figure 4.6 and Figure 4.7 also show that a more cost effective sorting workload



(a) (b)

Figure 4.7: (a) Sort time in seconds(records = 10^6, dimensions = 8)
(b) Computation time in seconds(records = 10^6, dimensions = 8)

can be obtained in our approach relative to the original Memory-Cube algorithm. As a result, our approach steadily run 15% faster than the original Memory-Cube.

4.5.3 Performance Study of Partitioned-Cube

We implemented the Partitioned-Cube algorithm, working with the paths generated by our new method and our adapted version of Memory-Cube. The Partitioned-Cube runs on the data sets with 1,000,000 tuples, a domain cardinality of 10, and various dimension counts varied from 4 to 10. In order to test the Partitioned-Cube algorithm, we need to restrict the size of memory (otherwise, we would require massive data sets for even simple testing). We assume that the size of memory can hold 100,000 tuples in our experiments – so that the data sets will be partitioned into memory-sized fragments when they are larger than this size. We then apply the Memory-Cube algorithm on each partition.

The performance is shown in Figure 4.8. In our experiments, we measure the total running time. In order to evaluate the performance of Partitioned-Cube, we run the

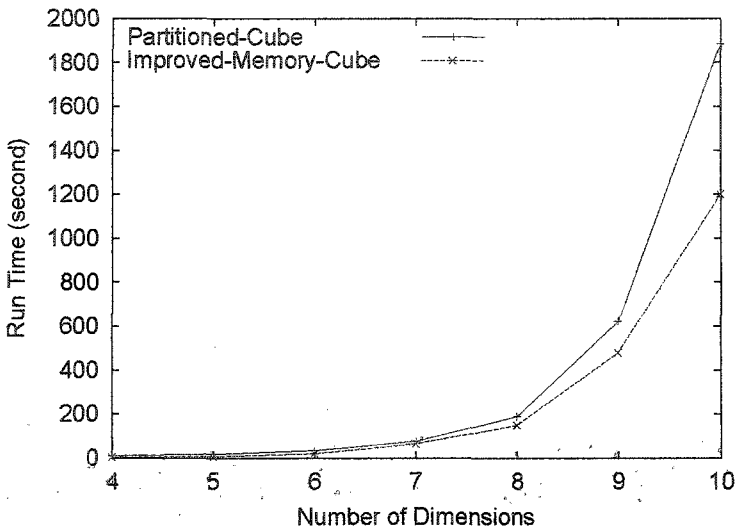


Figure 4.8: Total Run Time in seconds of Partition-Cube with memory size of 10^5 records, Memory-Cube is in-memory process. (records = 10^6, cardinality=10,)

Partitioned-Cube algorithm twice for each data set. In the first run we restrict the size of memory. In the second run we do not restrict the size of memory. Consequently, the input relation fits in memory and no partitioning is needed (i.e. Memory-Cube).

From Figure 4.8, we can clearly see that the difference between Partitioned-Cube (need partition) and Memory-Cube (no partition) is getting more significant as the number of dimensions grows. The reason is that fewer partitions are needed for the data sets with smaller numbers of dimensions. For example, a 4-dimensional data set is partitioned in dimension A, at which point we compute the views that have A as an attribute. After this step is done, the finest view $ABCD$ fits in memory, so no further partitioning is needed. We compute the rest of the views from view $ABCD$.

4.6 Conclusion

In this chapter, we proposed two new methods to address the problems associated with the original Partitioned-Cube and Memory-Cube that prevent them from achieving optimal performance. We do so by making changes to the structure of the sort paths and by adding proper sort parents for each path. In the original Partitioned-Cube algorithm, the paths used did not match the paths used in the Memory-Cube algorithm. Our techniques enable the Partitioned-Cube and Memory-Cube algorithms to work together efficiently, an accomplishment that has been validated through experimental evaluation.

From the above experimental analysis, we can clearly see that the Memory-Cube algorithm is more suitable for sparse and high dimension data sets. Moreover, we have shown that our new techniques achieve better performance than the original Memory-Cube algorithm. In addition, we have demonstrated by a performance study of Partitioned-Cube that this algorithm is a feasible solution for data sets that are larger than memory.

Chapter 5

Computing Full Data Cubes in Parallel

5.1 Introduction

We have discussed a number of the most important recent algorithms [35, 34, 9, 13, 39] for computing the data cube in Chapter 3. All of these algorithms are designed for implementation on sequential machines. However, computing a data cube can be an expensive task. The application of parallel processing has the potential to speed up this process. Despite the popularity and importance of data cubes, only a small of papers have been written in the area of parallel computation [22, 30, 31, 32, 15, 19]. Of these, the work described in [15] represents the largest and most sustained single project. Specifically, the authors present a general methodology for the efficient parallelization of existing data cube construction algorithms. Moreover, in [19], a detailed description of the design and implementation of a load-balanced and communication efficient parallel algorithm for full data cube construction in a shared disk environment is presented.

In our current research, we have opted for a shared-nothing parallel machine consisting of a number of processing nodes, each with its own local memory and local disk, and connected via a network or switch. For certain classes of algorithms (i.e., loosely coupled), this type of machine can often approximate the performance of more traditional MPPs (Massively Parallel Processors) at a much lower cost. The widespread availability of these inexpensive machines makes it possible to use parallel processing for the computation of data cubes, even in small size firms. We note that this type of system includes the increasingly popular Beowulf style clusters consisting of standard PCs connected via a simple Ethernet switch [1]. In [16], the authors describe how such a cluster architecture might be utilized for parallel data warehousing.

In this chapter, we first introduce the approach to computation of the full data cube described in [15, 19]. We then propose two new methods to parallelize the Partitioned-Cube algorithm and Memory-Cube algorithm respectively. We also present experimental results that demonstrate the viability of our approach.

The chapter is organized as follows. Section 5.2 describes the parallel PipeSort algorithm proposed in [15], while Section 5.3 discusses a parallelization of the BUC algorithm presented in the same paper. We present our approaches to parallel Partitioned-Cube and Memory-Cube in Section 5.4. In Section 5.5, we present our experimental results. The final conclusion is provided in Section 5.6.

5.2 Parallel PipeSort

In this section, we describe the parallel PipeSort algorithm proposed in [19]. As discussed in chapter 3, the PipeSort algorithm generates a minimum cost spanning tree (MCST) from the cuboid lattice. Since the PipeSort manipulates a weighted tree,

the primary objective when considering a parallel version of the original algorithm is to partition the tree into view subsets, and then to distribute view subsets to individual nodes, where efficient sequential algorithms can be used to independently calculate their assigned workload.

The parallel PipeSort algorithm opts for a partitioning strategy in which a master processor generates p independent subproblems from sub-trees of the MCST, each of which can be solved by an independent compute processor using a sequential pipeline algorithm. This approach makes use of a related partitioning problem on trees for which efficient algorithms exist, namely the *k-min-max partitioning problem.* Definition 5.2.1 provides a formal description of the fundamental problem.

Definition 5.2.1. The min-max problem can be defined as follows: Given a tree T with n vertices and a positive weight w assigned to each vertex, delete k edges in the tree such that the total weight $\sum_{i=0}^{l} w_i$ of the l nodes in the largest resulting subtree is minimized.

The k-min-max partitioning problem has been studied in [12, 20, 11]. The Min-Max shifting algorithm is presented in Algorithm 8, which is based the Min-Max Shifting Algorithm described in [12]. This algorithm is based on a pebble shifting scheme where k pebbles or cuts are shifted down the tree, from the root towards the leaves. The objective is to reduce the size of the largest current partition during each pass. Whenever possible, this partition is minimized by down-shifting the cut above its root node to the child edge with the heaviest down-component — defined as the weight of a partition beneath a given node or edge. Once done, it walks back up the tree, checking to see if the previous downshift has created an unnecessarily large partition above the original root. If so, it reduces its size by side-shifting cuts. The

algorithm continues until the largest partition can no longer be reduced in size [19].

Algorithm 8 k-min-max Algorithm: Becker, Schach and Perl

Input: A weighted tree T with positive weights assigned to its n vertices.
Output: A set of k sub-trees in which the weight of the largest sub-tree is as small as possible.
Method:

```
1: Place k − 1 cuts on the edge incident with the root vertex.
2: while (rootPartition is lightest partition) do
3:    if (largest partition has one or more vacant edges beneath its root node q)
      then
4:       Downshift cut at q to child edge with largest down-component
5:    else
6:       Terminate
7:    end if
8:    repeat
9:       Get the next node w along the path from q towards the root of T
10:      if (w has a cut on an incident edge e_i whose down-component < down-
         component of the current vacant edge e_v ) then
11:         side-shift the cut on e_i to e_v
12:      end if
13:   until (a cut is encountered on the path towards the root of T)
14: end while
```

In order to achieve a better distribution of the load, the Parallel PipeSort algorithm applies an over sampling mechanism: instead of partitioning the tree T into p subtrees, the algorithm partitions it into $s \times p$ subtrees, where s is an integer, $s \geqslant 1$. Then, we use a packing heuristic to determine which subtrees belong to a given processor. Essentially, this packing approach considers the weights of the subtrees and uses those weights to combine distinct trees so as to balance the cost of computation across nodes. It consists of $s - 1$ matching phases in which the p largest subtrees (or groups of subtrees) and the p smallest subtrees (or groups of subtrees) are paired up. In the end, s subtrees are assigned to every processor. Details are described in Algorithm 9.

Algorithm 9 Tree-Partition: Dehne et. al

Input: A spanning tree T of the lattice with positive weights assigned to the nodes (representing the cost to build each node from it's ancestor in T). Integer parameters s (over-sampling ratio) and p (number of processors).

Output: A partitioning of T into p subsets $\sum_1, ..., \sum_p$ of s subtrees each.

Method:

1: Use min-max algorithm to compute an $s \times p-$*partitioning* of T into $s \times p$ subtrees $T_1, ..., T_{s \times p}$.
2: Distribute subtrees $T_1, ..., T_{s \times p}$ among the p subsets $\sum_1, ..., \sum_p$, s subtrees per subset, as follows:
3: Create $s \times p$ sets of trees named Υ_i, $1 \leq i \leq sp$, where initially $\Upsilon_i = T_i$. The weight of Υ_i is defined as the total weight of the trees in Υ_i.
4: **for** $(j = 1$ to $s - 1)$ **do**
5: Sort the Υ_sets by weight, in increasing order. W.l.o.g., let $\Upsilon_i, ..., \Upsilon_{sp-(j-1)p}$ be the resulting sequence.
6: **for** $(i = 1$ to $p)$ **do**
7: Set $\Upsilon_i = \Upsilon_i \cup \Upsilon_{sp-(j-1)p-i+1}$
8: Remove $\Upsilon_{sp-(j-1)p-i+1}$
9: **end for**
10: **end for**
11: **for** $(i = 1$ to $p)$ **do**
12: Set $\sum_i = \Upsilon_i$
13: **end for**

The tree partitioning algorithm is embedded into the parallel data cube construction algorithm. Algorithm 10 presents the basic model. Like the sequential algorithm, the lattice must be augmented with appropriate costing values. First, the final size of the 2^d proposed cuboids must be estimated. Then, using these figures, a complete pipeline costing framework can be developed — including sorting, scanning and input/output estimates — that will eventually allow the bipartite matching mechanism of the PipeSort to determine the most cost effective means by which to move from $level_i$ to $level_{i-1}$ in the lattice. Once the minimum cost spanning tree has been extracted in Step 3, we use modified k-min-max with over-sampling to determine the appropriate sub-tree distribution across the p processors. Finally, when the local processors have received their task lists, a local pipeline computation algorithm generates the assigned output views to complete the distributed data cube computation [19].

5.3 Parallel BUC

In this section, we describe the Parallel Bottom-Up Cube Construction algorithm proposed by Dehne et. al. in [15]. The algorithm partitions the cuboid computations in to p independent subproblems, which can be processed efficiently by Bottom-Up sequential cube methods [13].

Let $A_1, ..., A_d$ be the attributes of relation R and assume $|A_1| \geq |A_2|, ..., \geq |A_d|$, where $|A_i|$ is the cardinality of A_i and d is the number of attributes. The cuboids can be partitioned into those that contain A_1 and those that do not contain A_1. The same scheme can be applied for the remaining attributes.

Let x,y,z be sequences of attributes representing sort orders and let A be a arbitrary single attribute. The definition of sets of attribute sequences is shown as

Algorithm 10 Parallel PipeSort: Dehne et al.

Input: A data set R, with each of its n records composed of d feature attributes and one measure attribute.

Output: A collection of 2^d cuboids, each presenting a summarized view of a unique subset of the d feature attributes.

Method:

1: On the master node M, apply a storage estimation method in to determine the approximate sizes of all 2^d cuboids in the data cube lattice L.
2: Augment L with the estimated costs of sorting and scanning each of its 2^d nodes.
3: Using the sequential PipeSort algorithm, extract a minimum cost spanning tree T from L.
4: Execute Algorithm Tree-partition, creating p sets $\sum_i, ..., \sum_p$. Each set $\sum_i$ contains s subtrees of T.
5: **for** ($i = 1$ to p) **do**
6: Distribute $\sum_i$ to p_i
7: **end for**
8: Each processor $p_i, 1 \leq i \leq p$, performs the following step independently and in parallel:
9: Compute all group-bys in subset $\sum_i$ using the sequential pipeline computation algorithm.

<table>
<tr><td rowspan="8">$S_4(\phi,ABCD,\phi)$</td><td rowspan="4">$S_3(\phi,BCD,A)$</td><td rowspan="2">$S_2(\phi,CD,BA)$</td><td>$S_1(\phi,D,CBA) = \{\phi,D\}$</td></tr>
<tr><td>$S_1(C,D,BA) = \{C,CD\}$</td></tr>
<tr><td rowspan="2">$S_2(B,CD,A)$</td><td>$S_1(B,D,CA) = \{B,BD\}$</td></tr>
<tr><td>$S_1(BC,D,A) = \{BC,BCD\}$</td></tr>
<tr><td rowspan="4">$S_3(A,BCD,\phi)$</td><td rowspan="2">$S_2(A,CD,B)$</td><td>$S_1(A,D,CB) = \{A,AD\}$</td></tr>
<tr><td>$S_1(AC,D,B) = \{AC,ACD\}$</td></tr>
<tr><td rowspan="2">$S_2(AB,CD,\phi)$</td><td>$S_1(AB,D,C) = \{AB,ABD\}$</td></tr>
<tr><td>$S_1(ABC,D,\phi) = \{ABC,ABCD\}$</td></tr>
</table>

Figure 5.1: Partitioning for a data cube with 4 dimensions, The 8 S_1 sets correspond to the 16 cuboids.

follows:

$$S_1(x, A, z) = \{x, xA\} \tag{5.3.1}$$

$$S_i(x, Ay, z) = S_{i-1}(xA, y, z) \cup S_{i-1}(x, y, Az), 2 \leq i \leq \log p + 1 \tag{5.3.2}$$

The entire data cube construction corresponds to the set $S_d(\phi, A_1...A_d, \phi)$ of sort orders and respective cuboids. i is the rank of S_i. The set is the union of two subsets of rank $d-1$: $S_{d-1}(A_1, A_2...A_d, \phi)$ and $S_{d-1}(\phi, A_2...A_d, A_1)$. $S_{d-1}(A_1, A_2...A_d, \phi)$ refers to the cuboids that contains A_1, $S_{d-1}(\phi, A_2...A_d, A_1)$ refers to the coboids that do not contain A_1. These, in turn, are the union of four subsets of rank $d-2$: $S_{d-2}(A_1A_2, A_3...A_d, \phi)$, $S_{d-2}(A_1, A_3...A_d, A_2)$, $S_{d-2}(A_2, A_3...A_d, A_1)$ and $S_{d-2}(\phi, A_3...A_d, A_2A_1)$. A complete example for 4-dimensional data cube with attributes A, B, C, D is shown in Figure 5.1.

For the sake of simplifying the discussion, assume that p is a power of 2, $p = 2^k$, where p is the number of processors. Consider the $2p$ S-sets of rank $d-k-1$. Let

$\beta = (\beta^1, \beta^2, ..., \beta^{2p})$ be these $2p$ sets in the order defined by Equation 5.3.2.

$$\text{Shuffle}(\beta) =< \beta^1 \cup \beta^2 p, \beta^2 \cup \beta^{2p-1}, \beta^3 \cup \beta^{2p-2}, ..., \beta^p \cup \beta^{p+1}$$

$$=< \Gamma_1, ..., \Gamma_p >$$

The algorithm then assigns set $\Gamma_i = \beta^i \cup \beta^{2p-i+1}$ to processor P_i. The parallel BUC algorithm is described in Algorithm 11.

Algorithm 11 Parallel BUC: Dehne et al.

Input: A data set R, with each of its n records composed of d feature attributes and one measure attribute.

Output: A collection of 2^d cuboids, each presenting a summarized view of a unique subset of the d feature attributes.

Method:

1: On the master node M, determine the two sets Γ_i.
2: Distribute the two sets Γ_i to P_i
3: Each processor $P_i, 1 \leq i \leq p$, compute all group-bys in subset Γ_i using the sequential Bottom-Up Cube computation algorithm independently and in parallel.

We illustrate the partitioning strategy by a concrete example with 10 dimensional data set and 8 processors. There are 16 S-sets of rank 6.

$$\beta = (ABCD, ABC, ABD, AB, ADC, AC, AD, A, BCD, BC, BD, B, CD, C, D, \phi)$$

Each processor is assigned to compute 2^7 group-bys as shown in Table 5.1.

5.4 Parallel Partitioned-Cube and Memory-Cube

We turn now to the Partitioned-Cube and Memory-Cube algorithms discussed in the previous chapter. We propose two approaches to parallelizing the Partitioned-Cube algorithm and Memory-Cube algorithm in this section.

Processor	$\Gamma - sets$	
1	$ABCD******$	$\bullet\bullet\bullet\bullet******$
2	$ABC\bullet******$	$\bullet\bullet\bullet D******$
3	$ABD\bullet******$	$\bullet\bullet C\bullet******$
4	$AB\bullet\bullet*****$	$\bullet\bullet CD*****$
5	$ACD\bullet******$	$\bullet B\bullet\bullet*****$
6	$AC\bullet\bullet*****$	$\bullet B\bullet D*****$
7	$AD\bullet\bullet*****$	$\bullet BC\bullet******$
8	$A\bullet\bullet\bullet******$	$\bullet BCD******$

Table 5.1: Γ-sets assigned to 8 processors for a 10-dimensional data set. • represents a project out attribute and * represents an existing attribute

5.4.1 Parallel Partitioned-Cube

As described in chapter 3, the Partitioned-Cube algorithm utilizes a divide-and-conquer technique to compute the data cube. Our approach exploits this same design theme. The details are presented in Algorithm 12.

In our approach, partitioning is performed by globally sorting the data set on a given dimension. Consider a 4-dimensional data set for a p processor parallel machine. On the master node, the original data set is first partitioned on A and split into n partitions $(A_1, A_2, ..., A_n)$, where n is cardinality of the current partition dimension, and then distributed to each processor. To do so, we use a simple round-robin mechanism to assign the n partitions into p groups, each processor is assigned n/p partitions. Then, for each group of partitions and in parallel, we compute the cubiods that have A as an attribute using the Memory-Cube algorithm. We note that the cubiods containing A as their first attribute can be computed independently, since the value of A is distinct within each partition.

Next, the master node receives the partial results and merges them into a single

Algorithm 12 Parallel Partitioned-Cube

Input: A data set R, with each of its n records composed of d feature attributes and one measure attribute.

Output: A collection of 2^d cuboids, each presenting a summarized view of a unique subset of the d feature attributes.

Method:

1: **for** ($j = 1$ to d) **do**
2: On the master node M, sort and partition on the underlying relation into C partitions according to the distinct value of the current partition dimension. {/*where the underlying relation is the finest view computed from last round(i.e, take raw data when partitioning on A, take $ABCD$ when partition on B, take BCD when partition on C and so on.) and C is the cardinality of current partition dimension */}.
3: Distribute partitions to each processor using round-robin mechanism.
4: Each processor $p_i, 1 \leq i \leq p$, compute the group-bys that have the current partition attribute as an attribute in subset using the sequential Memory-Cube algorithm and in parallel.
5: Master node collect the partial results from every processor and merges the results.
6: **end for**

view. This aggregated result contains no duplicate records. At this point, we sort and partition the finest granularity view computed from the last round (i.e. $ABCD$) on B. We then distribute each group of partitions to the associated processor. At each processor, we compute the cuboids that contain B as an attribute. The master node collects the partial results and merges them. The same procedures will be applied on the remaining attributes C and D respectively.

Once all attributes have been processed, the full data cube has been computed. Finally, we will have the full computed data cube on the master node.

5.4.2 Parallel Memory-Cube

Recall that the Memory-Cube is based on the idea of a *pipeline*. Each pipeline can be processed independently. The primary objective of our new parallel Memory-Cube is to distribute the pipelines to each processor evenly, and then have each processor compute its assigned pipelines using the sequential Memory-Cube algorithm independently and in parallel. The Parallel Memory-Cube algorithm is described in Algorithm 13 and the AssignPaths method is presented in Algorithm 14.

The algorithm proceeds as follows. First, on the master node, the Parallel Memory-Cube algorithm executes the standard Paths algorithm [34] to generate a list of paths. As described in Chapter 3, we adjust and reorder the list of paths using our modified Paths method in order to achieve superior performance. After this step is done, we will have the final list of paths that will be used in the Parallel Memory-Cube algorithm. Consider a 4-dimensional data set. the list of paths are shown as follow:

$p\_list[1] = A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow \epsilon$

$p\_list[2] = A.C.D \rightarrow A.C$

Algorithm 13 Parallel Memory-Cube

Input: A data set R, with each of its n records composed of d feature attributes and one measure attribute.

Output: A collection of 2^d cuboids, each presenting a summarized view of a unique subset of the d feature attributes.

Method:

1: On the master node M, execute the Paths algorithm to generate a list of paths.
2: On the Master node M, run the modified Paths algorithm to adjust and reorder the list of paths generated by the original Paths algorithm.
3: Divide the paths to p groups ($Path\_lists[1], Path\_list[2], ..., Path\_lists[p]$) using the Assign Paths function, where p is the number of processors.
4: **for** ($i = 1$ to p) **do**
5: Distribute $Path\_lists[i]$ to p_i
6: **end for**
7: Each processor $p_i, 1 \leq i \leq p$, process the assigned paths using the sequential Memory-Cube algorithm and in parallel.

Algorithm 14 Assign Paths

Input: A list of paths (p_list).

Output: p subsets of p_list ($path\_lists[1], path\_lists[1], ..., path\_lists[p]$), where p is the number of processors.

Method:

1: **for** ($i = 1$ to n) **do**
2: /*n is the number of paths */.
3: $path\_lists[i \ mod \ p] = path\_lists[i \ mod \ p] \cup p\_list[i]$.
4: **end for**

$p\_list[3] = A.D.B \rightarrow A.D$

$p\_list[4] = B.C.D \rightarrow B.C \rightarrow B$

$p\_list[5] = B.D$

$p\_list[6] = C.D \rightarrow C$

$p\_list[7] = D$

Next, we call the AssignPaths function to assign paths to each processor. This algorithm divides the paths into p groups ($Path\_lists[1], Path\_list[2], ..., Path\_lists[p]$), where p is the number of processors. To do so, it uses a simple round-robin mechanism to assign the $T = \sum_{k=1}^{d-1} \binom{(d-k)}{(d-k)/2} + 1$ total pipelines into p groups of T/p pipelines each. As a concrete example, assume we have four processors. Once the AssignPaths function is executed, the p groups of paths can be listed as follows:

$Path\_lists[1] = p\_list[1] \cup p\_list[5]$

$Path\_lists[2] = p\_list[2] \cup p\_list[6]$

$Path\_lists[3] = p\_list[3] \cup p\_list[7]$

$Path\_lists[4] = p\_list[4]$

We now distribute each group of paths ($i.e. Path\_lists[i]$) to the associated processor p_i. Finally, at each processor, we process the assigned paths using the sequential Memory-Cube algorithm independently and in parallel.

5.5 Experimental Evaluation

In this section, we discuss the performance of three parallel algorithm implementations including PipeSort, Memory-Cube and BUC. We note that given the complexity of parallel data cube algorithms, a concrete implementation of the Parallel

Partitioned-Cube has not been undertaken as part of this thesis program. Evaluation was conducted on a shared nothing Beowulf cluster architecture [4].

Beowulf Cluster Configuration:

- 32 nodes with dual 1.7 GHz Xeon Processors with 1 GB RAM per node 32 nodes with dual 2.0 GHz Xeon Processors with 1.5 GB RAM per node.
- Total of 128 processors.
- 60 GB of disk storage per node.
- The 1.7 GHz nodes use Intel Pro 1000 XT NICs.
- The 2.0 GHz nodes use on-board GigE interfaces.
- All nodes are interconnected via a Cisco 6509 switch using Gigabit ethernet.
- 8 nodes are equipped with high end Graphics cards.

We have implemented both the Parallel Memory-Cube and Parallel BUC algorithms in C++. Node-to-node communication is supported by LAM's Message Passing Interface (MPI) [5]. The Parallel PipeSort implementation has been provided by the authors of [7].

The data set used for our tests consists of 1,000,000 rows with a dimension count of 8 and a cardinality of 100 on each dimension.

5.5.1 Performance Study of Parallel PipeSort

Figure 5.2 depicts the performance curve that shows the Speedup of the Algorithm Parallel PipeSort on the Linux cluster as processor count increases from 1 to 16. Also

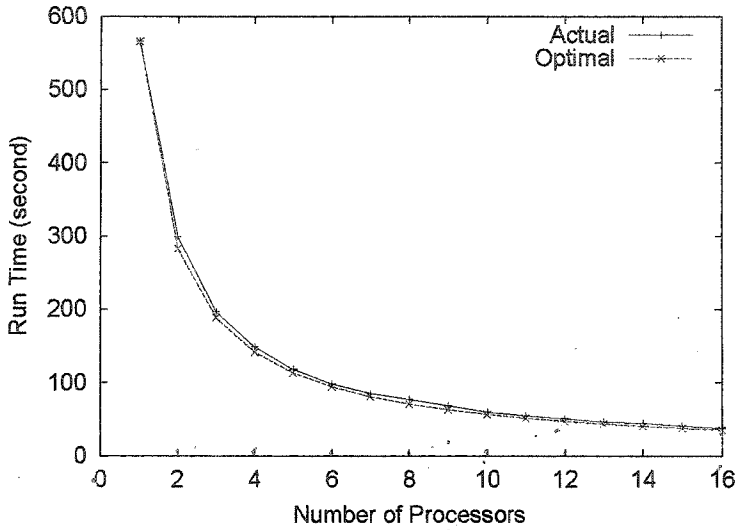


Figure 5.2: Speedup Test for Parallel PipeSort. (fixed parameter: Data size = 1,000,000, Number of Dimensions = 8, Cardinality = 100)

shown is the optimal curve, calculated as $T_{opt} = T_{sequential}/p$. Note that the actual performance tracks the optimal curve closely, indicating a near optimal utilization of parallel resources. These results are similar to those reported in [19].

5.5.2 Performance Study of Parallel Memory-Cube

Figure 5.3 shows the Speedup of our Parallel Memory-Cube approach on the Linux cluster as processor count increases from 1 to 16. While generally tracking the optimal curve, the actual performance curve does not do so as tightly as was the case for the Parallel PipeSort. The primary reason for this is that the current prototype for the Parallel Memory-Cube lacks the sophistication of the Parallel PipeSort in terms of its partitioning logic. In that case, a min-max k-partitioning algorithm was used to divide the global computation as evenly as possible. That implementation

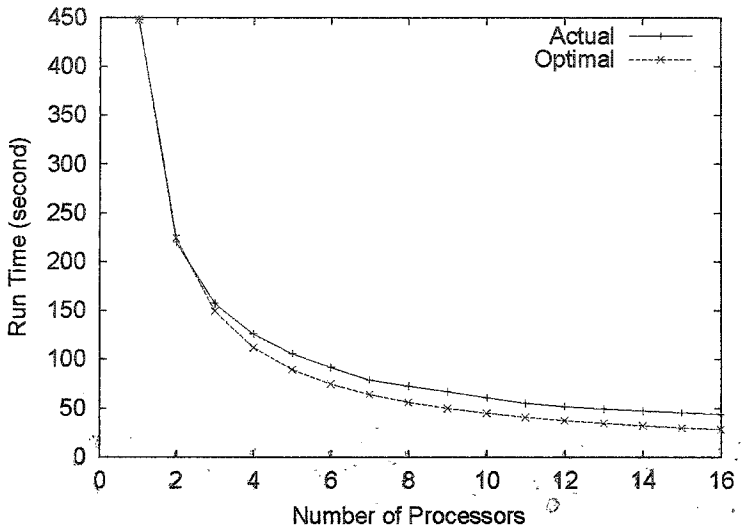


Figure 5.3: Speedup Test for Parallel Memory-Cube. (fixed parameter: Data size = 1,000,000, Number of Dimensions = 8, Cardinality = 100)

however required extensive analysis, revision, and modification in order to achieve its impressive load balancing characteristics. With the new Parallel Memory-Cube algorithm, our initial approach is somewhat simpler in that we distribute the workload via a round robin grouping of pipelines. No "global" costing perspective has yet been developed. We expect that a more sophisticated approach to load partitioning would result in improved parallel efficiency on higher processor counts. Given the prohibitive time requirements for such a design, however, it has not been implemented at this time. Instead, we hope to introduce a more powerful partitioning algorithm for the Parallel Memory-Cube in a future project.

5.5.3 Comparison of Parallel Memory-Cube and Parallel PipeSort

Figure 5.4 gives a "head-to-head" speedup comparison of Parallel PipeSort and Parallel Memory-Cube. The graph shows that the Parallel Memory-Cube is faster than the Parallel PipeSort at low processor counts. In other words, it achieves superior performance at low processor counts (i.e., from 1 to 8). Beyond this point, however, the Parallel PipeSort demonstrates a small but clear superiority. There are two reasons for this. First, because of the more simplistic partitioning logic, utilization of parallel resources is less efficient for the Parallel Memory-Cube, a problem that is exaggerated as the number of processors increases. Simply put, the new algorithm becomes *relatively* less efficient, while the parallel efficiency of the Parallel PipeSort is quite consistent.

The second reason is perhaps a little less obvious. One of the primary advantages of the Memory-Cube is that it can exploit previously sorted partitions when computing the current view. With the simple round-robin pipeline distribution scheme that we are using with the Parallel Memory-Cube, it is not always possible to optimally coordinate the sharing of sort costs between related pipelines. As a result, a great deal of efficiency may be lost as the pipelines are distributed more and more thinly across higher processor counts.

It is worth pointing out that a more powerful workload partitioning mechanism may help to address both of these problems.

5.5.4 Performance Study of Parallel BUC

The data set used for testing the Parallel BUC algorithm consists of 100,000 records with a dimension count of 8, and a cardinality for each dimension of 100, 80, 60, 50,

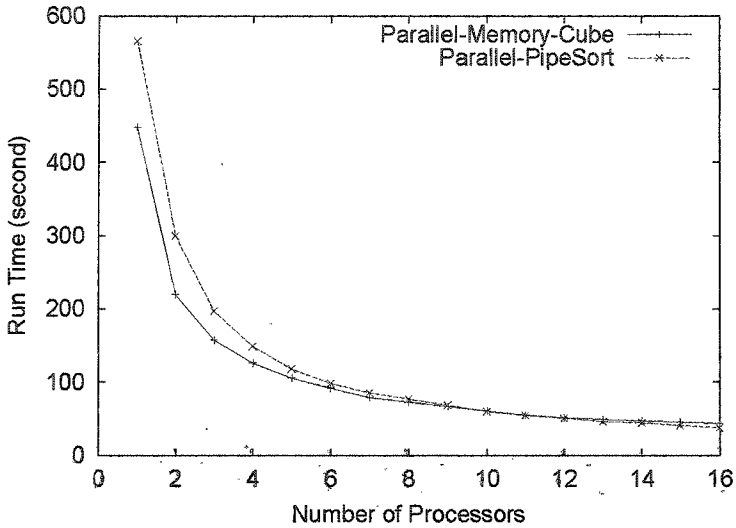


Figure 5.4: Running Time in Second for Comparison of Parallel Memory-Cube and Parallel PipeSort. (fixed parameter: Data size = 1,000,000, Number of Dimensions = 8, Cardinality = 100)

40, 30, 20, and 10 respectively. Figure 5.5 provides the speedup curve for the Parallel BUC algorithm.

We note that we do not provide direct comparisons of BUC with either Parallel PipeSort of Parallel Memory-Cube since the core implementation of the PipeSort algorithm (and the Memory-Cube that builds upon it) has been heavily optimized over a period of several years. The parallel BUC algorithm was implemented for the first time by the author of this thesis (it was never implemented by the original authors and was provided in [15] as an algorithm "sketch"). As such, any direct performance comparisons would be meaningless.

It is still useful, however, to assess the speedup curve in isolation since this provides a measure of the (potential) effectiveness of the load distribution mechanism. As can be seen from the graph, parallel efficiency is quite poor at a processor count of 16.

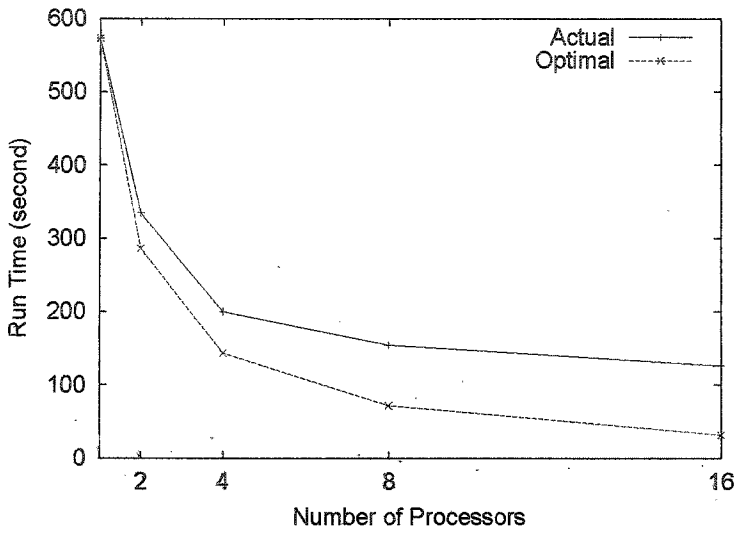


Figure 5.5: Speedup Test for Parallel BUC. (fixed parameter: Data size = 100,000, Number of Dimensions = 8, Cardinality = 100,80,60,50,40,30,20,10)

Specifically, parallel efficiency is roughly 25%. Another way of looking at this is that a 16-node parallel system running the current BUC algorithm would require as much time as a 4-node system running an *optimally balanced* BUC algorithm. While it is difficult to make definitive statements about the parallel BUC algorithm described in [15] from this single evaluation (it's possible that minor changes might produce major improvements), it appears likely at this point that the Parallel PipeSort/Parallel Memory-Cube may have far greater long term potential.

5.6 Conclusion

In this chapter, we have described two existing methods (Parallel PipeSort and Parallel BUC) for the parallelization of the data cube, a fundamental component of contemporary OLAP systems. Also, we have presented two algorithms – Parallel

Partitioned-Cube and Parallel Memory-Cube – that can also be used for high performance data cube computation. Our Parallel Partitioned-Cube method exploits the primary feature of the original sequential Partitioned-Cube Algorithm, which utilities a divide-and-conquer technique. The parallel version partitions the underlying relation on a given dimension, and distributes partitions to each processor. Locally, a view generation algorithm efficiently computes its portion of the workload.

The primary objective of our Parallel Memory-Cube method is to evenly distribute the sorting pipelines to each processor, where the sequential Memory-Cube algorithm computes its assigned workload efficiently. A concrete "prototype" implementation of the Parallel Memory-Cube has been developed. Comparative analysis shows that while the Parallel Memory-Cube algorithm is effective on low processor counts, its performance degrades more quickly than the PipeSort due to a more naive partitioning logic.

Chapter 6

Conclusions

6.1 Summary

The motivation of this research is to explore external memory algorithms. Specifically, our focus has been upon Ross's Partitioned-Cube algorithm [34]. We pointed out that most of the current research in data cube construction assumes the existence of an initial input set that fits entirely into main memory. In practice, with the extremely large data sets often found in contemporary decision support environments, this assumption may not be true. To properly support such large initial data sets, it will be necessary to extend the current algorithms into external memory.

The PANDA project, a joint effort between Dalhousie University, Concordia University, and Carleton University, focuses on parallel algorithms and data structures within the context of On-line Analytical Processing. In the PANDA project, new algorithms for parallelizing the data cube have been implemented. However, the Pandas systems are currently in-memory implementations only. It is necessary to extend the core algorithms into external memory. This objective motivates the research described in this thesis.

We have presented an approach to improve the performance of the original Memory-Cube algorithm. Also, we have proposed a method to address the problem that the original Partitioned-Cube algorithm does not work well in combination with the Memory-Cube algorithm. Moreover, we have presented two approaches to parallelizing the Partitioned-Cube and the Memory-Cube respectively.

The performance evaluations showed that our adapted Memory-Cube algorithm is quite efficient and that the Partitioned-Cube algorithm is a feasible solution for computing the data cube on input sets that are larger than main memory. The comparison of the performance of PipeSort and Memory-Cube supported the theory that the Memory-Cube algorithm is more suitable for sparse and high dimensions data set than the PipeSort algorithm. Finally, our evaluation of our initial prototype for the Parallel Memory-Cube demonstrated improved performance for small to moderate processor counts, with the potential for further improvement given a more powerful load distribution mechanism.

6.2 Future Work

It is an "unfortunate" fact that parallel algorithm implementations are quite time consuming, particularly when message passing and recursive algorithms like the Partitioned-Cube are involved. Due to prohibitive time requirements, in this thesis we have only implemented our Parallel Memory-Cube algorithm. Our Parallel Partitioned-Cube remains an algorithm description at this time. We expect to eventually implement the Parallel Partitioned-Cube idea and combine it with our Parallel Memory-Cube algorithm. Furthermore, we also know that the load balancing mechanism of our Parallel Memory-Cube algorithm is not as powerful as that of the Parallel PipeSort algorithm.

In the future, we would like to remedy this weakness, perhaps by identifying more favourable groupings of pipelines for each processor.

In addition, even though we implemented the BUC algorithm in both sequential and parallel form, neither implementation is well optimized. At some point in the future, we would like to evaluate an optimal BUC algorithm against both the Parallel PipeSort and Parallel Memory-Cube. This would be an interesting comparison given that the literature suggests that the bottom-up methods are better suited for high dimensional data.

In the PANDA project, both the sequential and parallel versions of the PipeSort algorithm support the construction of both full and partial data cubes (i.e., when only a subset of the group-bys are actually required). However, in this thesis, the Partitioned-Cube and the Memory-Cube (as well as the BUC) only support full data cube construction. In the future, we would like to extend our code to support partial data cube construction.

Finally, since the cube construction methods described in this thesis rely heavily upon sorting algorithms, it is clear that efficient external memory sorting algorithms are required. We have not explored this issue in this thesis. However, in our future work we hope to further investigate the design and implementation of OLAP-specific external memory sorting algorithms, perhaps based upon the fundamental research described in papers such as [37, 38]. Exploiting algorithms designed specifically to reduce the cost of I/O (for intermediate swap files in the case of sorting) would be a significant benefit to any practical data cube solution.

Bibliography

[1] *Beowulf project*, http://www.beowulf.org/.

[2] *Date warehouse*, http://www.webopedia.com/TERM/D/data-warehouse.html.

[3] *Decision support system*, http://www.webopedia.com/TERM/D/decision-support-system.html.

[4] *High performance computing virtual laboratory*, http://www.hpcvl.org/.

[5] *Lam mpi*, http://www.lam-mpi.org/.

[6] *On-line analytical processing*, http://www.webopedia.com/TERM/O/OLAP.html.

[7] *The panda project*, http://torch.cs.dal.ca/~panda/.

[8] *Programming posix threads*, http://www.humanfactor.com/pthreads.

[9] S. Agarwal, R. Agarwal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Srawagi, *On the computation of multi-dimensional aggregates*, Proc. 22nd VLDB Conf., 1996, pp. 506–521.

[10] G. Hofling B. Dinter, C. Sapia and M. Blaschka, *The olap market: State of the art and research issues*, ACM First International Workshop on Data Warehousing and OLAP (1998), 22–27.

[11] Ronald Becker, Bruno Simeone, and Yen-I Chiang, *A shifting algorithm for continuous tree partitioning*, Theor. Comput. Sci. **282** (2002), no. 2, 353–380.

[12] Ronald I. Becker, Stephen R. Schach, and Yehoshua Perl, *A shifting algorithm for min-max tree partitioning*, J. ACM **29** (1982), no. 1, 58–67.

[13] Kevin Beyer and Raghu Ramakrishnan, *Bottom-up computation of sparse and Iceberg CUBE*, Proceedings of the 1999 ACM SIGMOD Conference, 1999, pp. 359–370.

[14] Surajit Chaudhuri and Umeshwar Dayal, *An overview of data warehousing and olap technology*, SIGMOD Record **26** (1997), no. 1, 65–74.

[15] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin, *Parallelizing the data cube*, Distributed and Parallel Databases **11** (2002), no. 2, 181–201.

[16] F. Dehne, T. Eavis, and Andrew Rau-Chaplin, *A cluster architecture for parallel data warehousing*, Proc IEEE International Conference on Cluster Computing and the Grid (CCGrid 2001) (Brisbane, Australia), 2001.

[17] P.M. Deshpande, S. Agarwal, J.F. Naughton, and R. Ramakrishnan, *Computation of multi-dimensional aggregates*, Technical Report 10026, IBM Almaden Research Center, 1996.

[18] S. Codd E. Codd and C. Salley, *Providing olap (on-line analytical processing) to user-analysts: An it mandate*, Technical report, 1992.

[19] Todd Eavis, *Parallel relational olap*, 2002.

[20] Greg N. Frederickson, *Optimal algorithms for tree partitioning*, Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 1991, pp. 168–177.

[21] S. Goil and A. Choudhary, *A parallel scalable infrastructure for OLAP and data mining*, Proc. International Data Engineering and Applications Symposium (IDEAS'99) (Montreal), 1999.

[22] Sanjay Goil and Alok N. Choudhary, *High performance multidimensional analysis of large datasets*, International Workshop on Data Warehousing and OLAP, 1998, pp. 34–39.

[23] Goetz Graefe, *Query evaluation techniques for large databases*, ACM Comput. Surv. **25** (1993), no. 2, 73–170.

[24] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh, *Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals*, J. Data Mining and Knowledge Discovery 1 (1997), no. 1, 29–53.

[25] J. Han and M. Kamber, *Data mining: Concepts and techniques*, Morgan Kaufmann Publishers, 2000.

[26] V. Harinarayan, A. Rajaraman, and J.D. Ullman, *Implementing data cubes efficiently*, ACM SIGMOD Record **25** (1996), no. 2, 205–216.

[27] C. A. R. Hoare, *Quicksort*, Computer Journal **5** (1962), no. 1, 10–15.

[28] W. Inmon, *Building the data warehouse*, John Wiley, 1992.

[29] Max Planck Institute, *Leda*, http://www.algorithmic-solutions.info/leda-manual/MANUAL.html.

[30] H. Lu, X. Huang, and Z. Li, *Computing data cubes using massively parallel processors*, Proc. 7th Parallel Computing Workshop (PCW'97) (Canberra, Australia), 1997.

[31] Seigo Muto and Masaru Kitsuregawa, *A dynamic load balancing strategy for parallel datacube computation*, Proceedings of the second ACM international workshop on Data warehousing and OLAP, ACM Press, 1999, pp. 67–72.

[32] R. Ng, A. Wagner, and Y. Yin, *Iceberg-cube computation with pc clusters*, Proceedings of 2001 ACM SIGMOD Conference on Management of Data, 2001, pp. 25–36.

[33] C.H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: Algorithms and complexity*, ch. 11, pp. 247–254, 1982.

[34] Kenneth A. Ross and Divesh Srivastava, *Fast computation of sparse datacubes*, Proc. 23rd Int. Conf. Very Large Data Bases, VLDB (Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, eds.), Morgan Kaufmann, 25–27 1997, pp. 116–125.

[35] S. Sarawari, R. Agrawal, and A. Gupta, *On computing the data cube*, Technical Report 10026, IBM Almaden Research Center, 1998.

[36] J. P. Shima, Merrill Warkentina, James F. Courtney, Daniel J. Power, Ramesh Sharda, and Christer Carlsson, *Past, present, and future of decision support technology*, Decision Support Systems **33** (2002), no. 2, 111–126.

[37] J. S. Vitter, *External memory algorithms and data structures: Dealing with massive data*, ACM Computing Surveys **33** (2001), no. 2, 209–271.

[38] J. S. Vitter and E. A. M. Shriver, *Algorithms for parallel memory I: Two-level memories*, Algorithmica **12** (1994), no. 2-3, 110–147.

[39] Y. Zhao, P.M. Deshpande, and J.F.Naughton, *An array-based algorithm for simultaneous multidimensional aggregates*, Proc. ACM SIGMOD Conf., 1997, pp. 159–170.