Parallel Computation of Data Cubes

by

Soroush Momen-Pour


A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia
September 1999
© Soroush Momen-Pour, 1999
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Sep 30, 1999
Abstract

Since its proposal, data cube has attracted a great deal of attention in both academic and industry research communities. Many research papers have been published about different issues related to data cubes and many commercial OLAP (On-Line Analytical Processing) systems have been released to market with data cube operations as their core functions.

Several algorithms have been proposed to compute data cubes more efficiently. PIPESORT and PIPEHASH algorithms proposed by Agrawal et. al., OVERLAP algorithm proposed by Naughton et. al., Partitioned-Cube algorithm proposed by Ross et. al. and the Multi-way Array algorithm proposed by Naughton et. al. are the most significant ones.

All of these algorithms are designed for implementation on sequential machines, however computing a data cube can be an expensive task. For some organizations it may take a very powerful computer working around the clock for a week to compute all the data cubes they may want to use. 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 the parallel computation of them. The only parallel algorithm for computation of data cubes, which I am aware of, is the algorithm proposed by Goil et. al.. Their algorithm works for cases where the data
set fits in main memory, however, real world data sets rarely fit in main memory.

The widespread availability of inexpensive cluster machines makes it possible to use parallel processing for computation of data cubes, even in small size firms and as a result there could be a real demand for efficient parallel data cube construction algorithms. I have designed and implemented two parallel data cube computation algorithms (Parallel Partitioned-Cube algorithm and Parallel Single-pass Multi-way Array algorithm) based on sequential algorithms proposed in literature. The former algorithm is classified as a ROLAP (Relational OLAP) algorithm and the second one is considered as a MOLAP (Multi-dimensional OLAP) algorithm.
Contents

Abstract ii

Contents iv

List of Tables vi

List of Figures viii

Acknowledgements x

Dedication xi

1 Introduction 1

2 Background 5

2.1 Problems with GROUP BY operator 6

2.2 The Data Cube Operator 11

2.3 Different type of aggregate functions 13

3 Sequential Algorithms 15

3.1 Hierarchies 17

3.2 Selecting queries to precompute 18
List of Tables

2.1 Sales of a department store ........................................... 6
2.2 Sales of a department store grouped by Item and Date .......... 7
2.3 Sales of a department store at different granularities .......... 7
2.4 Sales of a department store at different granularities in a relational format ......................................................... 9
2.5 Sales of a department store at a granularity level not included in Table 2.4 ................................................................. 10
2.6 Sales of a department store in cross-tabulation format ....... 11
2.7 Sales of a department store in data cube format ................. 12

3.1 Benefits of materializing different views at each round .......... 23
3.2 Computing a 4-dimensional (30*24*2*158) Cube ............... 45
3.3 Computing a 5-dimensional (30*24*2*158*360) Cube ......... 46

4.1 Computing a 4-dimensional (30*24*2*158) Cube on 2 nodes using method 3 ........................................................... 55
4.2 Computing a 5-dimensional (30*24*2*158*360) Cube on 2 nodes using method 3 ........................................................... 55
4.3 Computing a 4-dimensional (30*24*2*158) Cube on 2 nodes using method 4 ........................................ 62

4.4 Computing a 5-dimensional (30*24*2*158*360) Cube on 2 nodes using method 4 ........................................ 62
## List of Figures

2.1 The syntax proposed for data cube operator ....................... 11  
2.2 The data cube expression for the data cube shown in Table 2.7 ... 12  
3.1 Data Cube Lattice Structure ...................................... 17  
3.2 Hierarchical *GROUP BY* attributes .............................. 19  
3.3 Hierarchical lattice structure ..................................... 20  
3.4 The greedy algorithm for selecting a set of views to materialize ... 21  
3.5 A data cube lattice structure with associated costs ............... 22  
3.6 Pipesort Algorithm ................................................. 28  
3.7 A layer of the search lattice after transformation .................. 29  
3.8 The layer of the search lattice after running weighted bipartite matching algorithm .................................................. 29  
3.9 Search lattice with associated costs ................................ 30  
3.10 The minimum cost sort plan ....................................... 30  
3.11 The pipelines that are executed ................................... 30  
3.12 Algorithm Partitioned-Cube ....................................... 32  
3.13 An example of running Partitioned-Cube Algorithm .............. 33  
3.14 Algorithm Memory-Cube ............................................ 35
3.15 Algorithm Paths ................................................. 36
3.16 A chunked array .................................................. 40
3.17 Minimum Memory Spanning Tree ............................... 43
3.18 Computing a 4-dimensional (30*24*2*158) Cube .......... 46
3.19 Computing a 4-dimensional (30*24*2*158*360) Cube .... 46

4.1 Computing a 5-dimensional (30*24*2*158*360) Cube on two nodes using three different methods ............................................. 53
4.2 Computing a 5-dimensional (30*24*2*158*360) Cube on four nodes using three different methods ............................................. 53
4.3 Computing a 4-dimensional (30*24*2*158) Cube on different number of nodes using method 3 ............................................. 54
4.4 Computing a 5-dimensional (30*24*2*158*360) Cube on different number of nodes using method 3 ............................................. 54
4.5 Hypercube structure ................................................ 58
4.6 Computing a 5-dimensional (30*24*2*158*360) Cube on two nodes using four different methods ............................................. 60
4.7 Computing a 5-dimensional (30*24*2*158*360) Cube on four nodes using four different methods ............................................. 60
4.8 Computing a 4-dimensional (30*24*2*158) Cube on different number of nodes using method 4 ............................................. 61
4.9 Computing a 5-dimensional (30*24*2*158*360) Cube on different number of nodes using method 4 ............................................. 61
Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Alan Wagner, for helping me in carrying out this research project and for reading the manuscript of my thesis and offering his valuable comments. I also would like to thank Dr. George Tsiknis for reading the manuscript of my thesis and providing me with his helpful comments.

Soroush Momen-Pour

The University of British Columbia

September 1999
To my dear wife Nooshin who always supported me.
Chapter 1

Introduction

Unlike an operational database that maintains the daily used information about activities of an enterprise a data warehouse contains historical data about its past. Data warehouses are normally huge because they accumulate information over a long period of time. 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 to assist managers of an enterprise in business related decision making process. OLAP [1] enables data analysts to ask numerous speculative what if and why questions within the context of some specific historical perspective. One operation that has been recently proposed is data cube operator [2] which computes $N$-dimensional aggregates that can be used in OLAP applications. Since its proposal, data cube has attracted a great deal of attention in both academic and industry research communities. Many research papers have been published about different issues related to data cubes and many commercial OLAP (On-Line Analytical Processing) systems have been released to market with data cube operations as their core functions.
Several algorithms have been proposed to compute data cubes more efficiently. PIPESORT and PIPEHASH algorithms proposed by Agrawal et. al. [3], OVERLAP algorithm proposed by Naughton et. al. [4], Partitioned-Cube algorithm proposed by Ross et. al. [5] and the Multi-way Array algorithm proposed by Naughton et. al. [6] are the most significant ones.

All of these algorithms proposed for computing data cubes are designed for implementation on sequential machines, however, computing a data cube can be an expensive task. For some organizations it may take a very powerful computer working around the clock for a week to compute all the data cubes they may want to use. Application of parallel processing can speed up this process and makes it possible to precompute more data cubes more often. Despite the popularity and importance of data cubes, very little research has been carried out on the parallel computation of them. The only parallel algorithm for computation of data cubes, which I am aware of, is the algorithm proposed by Goil et. al. [7]. Their algorithm works for cases where the data set fits in main memory, however, real world data sets rarely fit in main memory. The goals of this research are to design and implement efficient parallel algorithms for computation of data cubes when the input data does not fit in main memory and experimentally evaluate the use of cluster machines for computing data cubes.

Cluster machines are workstations connected via high speed interconnection networks such as Myrinet and are widely used to implement parallel algorithms. These machines can reach the performance of massive parallel machines with a much lower cost. The wide spread availability of these inexpensive machines makes it possible to use parallel processing for computation of data cubes, even in small size firms and as a result there could be a real demand for efficient parallel data cube
I have designed and implemented two parallel data cube computation algorithms (Parallel Partitioned-Cube algorithm and Parallel Single-pass Multi-way Array algorithm) based on sequential algorithms proposed in literature. The former algorithm is classified as a ROLAP (Relational OLAP) algorithm and the second one is considered as a MOLAP (Multi-dimensional OLAP) algorithm. Neither the Parallel Partitioned-Cube algorithm nor the Parallel Single-pass Multi-way Array algorithm requires storing of the entire input data in main memory.

Other than the problem of computing data cubes there are many other interesting data cube related issues that have appeared in the literature. Ho et. al. studied range queries in OLAP data cubes [8] and partial-sum queries in OLAP data cubes [9]. A range query applies an aggregation operation over all selected cells of an OLAP data cube where the selection is specified by providing ranges of values for numeric dimensions. A partial-sum query obtains the summation over a set of specified cells of a data cube. Ross et. al. studied complex aggregation at multiple granularities [10]. By complex aggregation they refer to queries involving multiple dependent aggregates at multiple granularities. Sarawagi et. al. studied discovery-driven exploration of OLAP data cubes [11]. They proposed a new discovery-driven exploration paradigm that could mine the data for exceptions. Han studied the integration of data mining and OLAP technologies [12]. Roussopoulos et. al. proposed cubetree as a storage abstraction of the data cube [13].

Structure of this thesis is as follows. Chapter 2 presents some background material that is useful to understand other chapters. Chapter 3 describes in detail 3 different sequential data cube computation algorithms and presents the results of some experiments performed with theses algorithms. Chapter 4 explains two parallel data cube computation algorithms that I have designed and implemented.
Chapter 5 is the conclusion.
Chapter 2

Background

In business and scientific applications, it is common to summarize data values using different aggregation functions. For example, consider the sales database of a department store. This store has different departments such as electronics, home appliances and others where each department carries different items and the sale transactions are all stored in a relational database. Table 2.1 shows partial sales of the department store on Boxing week.

We can extract different summaries from this table. One possible summary is the sale of each item on each day. This summary can be generated using the following SQL query:

```
SELECT item, date, Sum(sale)
FROM sales
GROUP BY item, date;
```

The GROUP BY operator divides tuples into different groups based on their values of GROUP BY attributes. Tuples with the same value on all GROUP BY attributes fall into the same group. Table 2.2 shows the result of the above query.
Other than *Sum* there are three other aggregate functions in standard SQL: *Avg*, *Max* and *Min*. Some database management systems support other domain specific functions, such as statistical functions (*Median, Standard Deviation, Variance*, etc.), physical functions (*Center of Mass, Angular Momentum*, etc.) and financial analysis functions(*Volatility, Alpha, Beta*, etc.) [2].

### 2.1 Problems with GROUP BY operator

There are some aggregate queries that are difficult to represent in standard SQL: histograms, roll-up totals and sub-totals for drill-downs and cross-tabulations [2].

Histograms are generated by aggregating over computed categories. For example if we want to extract the sales summary from the department store database based on the monthly sales of the store we need to categorize dates into months and group tuples based on this computed attribute and compute the aggregate function.
Table 2.2: Sales of a department store grouped by Item and Date

<table>
<thead>
<tr>
<th>Item</th>
<th>Date</th>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sony 25&quot; TV</td>
<td>98/12/26</td>
<td>1400</td>
</tr>
<tr>
<td>Sony 25&quot; TV</td>
<td>98/12/27</td>
<td>1400</td>
</tr>
<tr>
<td>JVC 21&quot; TV</td>
<td>98/12/26</td>
<td>800</td>
</tr>
<tr>
<td>JVC 21&quot; TV</td>
<td>98/12/27</td>
<td>400</td>
</tr>
<tr>
<td>Panasonic Hi-Fi VCR</td>
<td>98/12/26</td>
<td>250</td>
</tr>
<tr>
<td>Panasonic Hi-Fi VCR</td>
<td>98/12/27</td>
<td>250</td>
</tr>
<tr>
<td>MayTag 29 cubic foot refrigerator</td>
<td>98/12/26</td>
<td>1400</td>
</tr>
<tr>
<td>MayTag 29 cubic foot refrigerator</td>
<td>98/12/27</td>
<td>1400</td>
</tr>
<tr>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>98/12/26</td>
<td>1200</td>
</tr>
<tr>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>98/12/27</td>
<td>600</td>
</tr>
</tbody>
</table>

Table 2.3: Sales of a department store at different granularities

<table>
<thead>
<tr>
<th>Date</th>
<th>Item</th>
<th>Sale by Date and Item</th>
<th>Sale by Date</th>
<th>Total Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>98/12/26</td>
<td>Sony 25&quot; TV</td>
<td>1400</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JVC 21&quot; TV</td>
<td>800</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Panasonic Hi-Fi VCR</td>
<td>250</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MayTag 29 cubic foot refrigerator</td>
<td>1400</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>1200</td>
<td></td>
<td>5050</td>
</tr>
<tr>
<td>98/12/27</td>
<td>Sony 25&quot; TV</td>
<td>1400</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JVC 21&quot; TV</td>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Panasonic Hi-Fi VCR</td>
<td>250</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MayTag 29 cubic foot refrigerator</td>
<td>1400</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>600</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4050</td>
</tr>
</tbody>
</table>
for each group. If function values were allowed in the GROUP BY list the above query could be written as follows:

```
SELECT item, Month(date), Sum(sale)
FROM sales
GROUP BY item, Month(date);
```

The above SELECT query is not allowed in standard SQL but it can be translated into another standard SQL query:

```
SELECT item, month, Sum(sale)
FROM
(SELECT item, Month(date) AS month, sale
 FROM sales)
GROUP BY item, month;
```

Sometimes we want to generate a report containing total and sub-totals at different granularities. For example, we may want to know the sales by date and item, sales by date, and total sale of the store. Table 2.3 shows the required information. The sale figures in 3rd column represent total sales at finest granularity, the 4th column contains totals at next coarser granularity while the 5th column includes total sales at the coarsest granularity. Moving from a granularity level to a finer granularity level is called **drilling-down** and moving from one granularity level to a coarser granularity level is called **rolling-up**. Table 2.3 is not relational, because it contains null values in its primary key attributes. Moreover, the number of columns is large since it is the power set of the aggregation attributes. This table can be represented in a relational format where the number of columns grows linearly, rather
Table 2.4: Sales of a department store at different granularities in a relational format

<table>
<thead>
<tr>
<th>Item</th>
<th>Date</th>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sony 25&quot; TV</td>
<td>98/12/26</td>
<td>1400</td>
</tr>
<tr>
<td>JVC 21&quot; TV</td>
<td>98/12/26</td>
<td>800</td>
</tr>
<tr>
<td>Panasonic Hi-Fi VCR</td>
<td>98/12/26</td>
<td>250</td>
</tr>
<tr>
<td>MayTag 29 cubic foot refrigerator</td>
<td>98/12/26</td>
<td>1400</td>
</tr>
<tr>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>98/12/26</td>
<td>1200</td>
</tr>
<tr>
<td>ALL</td>
<td>98/12/26</td>
<td>5050</td>
</tr>
<tr>
<td>Sony 25&quot; TV</td>
<td>98/12/27</td>
<td>1400</td>
</tr>
<tr>
<td>JVC 21&quot; TV</td>
<td>98/12/27</td>
<td>400</td>
</tr>
<tr>
<td>Panasonic Hi-Fi VCR</td>
<td>98/12/27</td>
<td>250</td>
</tr>
<tr>
<td>MayTag 29 cubic foot refrigerator</td>
<td>98/12/27</td>
<td>1400</td>
</tr>
<tr>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>98/12/27</td>
<td>600</td>
</tr>
<tr>
<td>ALL</td>
<td>98/12/27</td>
<td>4050</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>9100</td>
</tr>
</tbody>
</table>

than exponentially, with the number of aggregation attributes. Table 2.4 shows the same data in a relational format. This table can be generated by the following SQL statement:

```
SELECT item, date, Sum(sale)
FROM sales
GROUP BY item, date
UNION
SELECT ALL, date, Sum(sale)
FROM sales
GROUP BY date
UNION
SELECT ALL, ALL, Sum(sale)
FROM sales;
```
Table 2.5: Sales of a department store at a granularity level not included in Table 2.4

<table>
<thead>
<tr>
<th>Item</th>
<th>Date</th>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sony 25&quot; TV</td>
<td>ALL</td>
<td>2800</td>
</tr>
<tr>
<td>JVC 21&quot; TV</td>
<td>ALL</td>
<td>1200</td>
</tr>
<tr>
<td>Panasonic Hi-Fi VCR</td>
<td>ALL</td>
<td>500</td>
</tr>
<tr>
<td>MayTag 29 cubic foot refrigerator</td>
<td>ALL</td>
<td>2800</td>
</tr>
<tr>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>ALL</td>
<td>1800</td>
</tr>
</tbody>
</table>

Table 2.4 is not symmetric. It does not contain the rows shown in Table 2.5. These rows can be added by appending the following clause to the above SQL statement:

```
UNION
SELECT item, ALL, Sum(sale)
FROM sales
GROUP BY item;
```

The symmetric aggregation table is called a cross-tabulation and can be represented as Table 2.6.

Histogram, roll-up, drill-down and cross-tab queries all can be formulated in standard SQL but their representation is inconvenient. Writing a 5 dimensional cross-tab query in standard SQL requires us to union 32 different GROUP BY queries where each GROUP BY query scans the table once and performs either a sort or a hash operation on it. Therefore a multi-dimensional cross-tab query is computationally intractable for large number of dimensions.
Table 2.6: Sales of a department store in cross-tabulation format

<table>
<thead>
<tr>
<th>Department Store Sales</th>
<th>98/12/26</th>
<th>98/12/27</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sony 25&quot; TV</td>
<td>1400</td>
<td>1400</td>
<td>2800</td>
</tr>
<tr>
<td>JVC 21&quot; TV</td>
<td>800</td>
<td>400</td>
<td>1200</td>
</tr>
<tr>
<td>Panasonic Hi-Fi VCR</td>
<td>250</td>
<td>250</td>
<td>500</td>
</tr>
<tr>
<td>MayTag 29 cubic foot refrigerator</td>
<td>1400</td>
<td>1400</td>
<td>2800</td>
</tr>
<tr>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>1200</td>
<td>600</td>
<td>1800</td>
</tr>
<tr>
<td>total(ALL)</td>
<td>5050</td>
<td>4050</td>
<td>9100</td>
</tr>
</tbody>
</table>

GROUP BY (  
{ ( <column name> | <expression>)  
[AS <correlation name>]  
[<collate clause>]  
,...}  
[WITH (CUBE | ROLLUP)]  
)

Figure 2.1: The syntax proposed for data cube operator

2.2 The Data Cube Operator

Gray et. al. [2] have proposed particular solutions for GROUP BY operator problems explained in section 2.1. They introduced Data Cube operator for generating cross-tabulations. This operator is a generalization of GROUP BY operator. An N-dimensional data cube consists of $2^N$ cuboids where each cuboid is the result of one of the $2^N$ possible GROUP BY queries that can be written with $N$ attributes. Table 2.7 shows the result of Data Cube operator when it is applied to the department store database.

The syntax proposed for data cube operator in [2] is shown in Figure 2.1. Figure 2.2 shows the data cube expression that can be written for Table 2.7 using this syntax.
Table 2.7: Sales of a department store in data cube format

<table>
<thead>
<tr>
<th>Item</th>
<th>Date</th>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sony 25” TV</td>
<td>98/12/26</td>
<td>1400</td>
</tr>
<tr>
<td>JVC 21” TV</td>
<td>98/12/26</td>
<td>800</td>
</tr>
<tr>
<td>Panasonic Hi-Fi VCR</td>
<td>98/12/26</td>
<td>250</td>
</tr>
<tr>
<td>MayTag 29 cubic foot refrigerator</td>
<td>98/12/26</td>
<td>1400</td>
</tr>
<tr>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>98/12/26</td>
<td>1200</td>
</tr>
<tr>
<td>ALL</td>
<td>98/12/26</td>
<td>5050</td>
</tr>
<tr>
<td>Sony 25” TV</td>
<td>98/12/27</td>
<td>1400</td>
</tr>
<tr>
<td>JVC 21” TV</td>
<td>98/12/27</td>
<td>400</td>
</tr>
<tr>
<td>Panasonic Hi-Fi VCR</td>
<td>98/12/27</td>
<td>250</td>
</tr>
<tr>
<td>MayTag 29 cubic foot refrigerator</td>
<td>98/12/27</td>
<td>1400</td>
</tr>
<tr>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>98/12/27</td>
<td>600</td>
</tr>
<tr>
<td>ALL</td>
<td>98/12/27</td>
<td>4050</td>
</tr>
<tr>
<td>Sony 25” TV</td>
<td>ALL</td>
<td>2800</td>
</tr>
<tr>
<td>JVC 21” TV</td>
<td>ALL</td>
<td>1200</td>
</tr>
<tr>
<td>Panasonic Hi-Fi VCR</td>
<td>ALL</td>
<td>500</td>
</tr>
<tr>
<td>MayTag 29 cubic foot refrigerator</td>
<td>ALL</td>
<td>2800</td>
</tr>
<tr>
<td>Whirlpool 22 cubic foot refrigerator</td>
<td>ALL</td>
<td>1800</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>9100</td>
</tr>
</tbody>
</table>

GROUP BY Item, Date
WITH CUBE

Figure 2.2: The data cube expression for the data cube shown in Table 2.7
If the WITH clause is used with CUBE then all the cuboids of the data cube will be computed but if it is used with ROLLUP then only the following cuboids are generated:

\((f_1, f_2, ..., f_{n-1}, ALL),\)
\((f_1, f_2, ..., f_{n-2}, ALL, ALL),\)
\((f_1, f_2, ..., f_{n-3}, ALL, ALL, ALL),\)
\((f_1, ALL, ..., ALL),\)
\((ALL, ALL, ..., ALL).\)

The AS clause is used to support histograms.

2.3 Different type of aggregate functions

Gray et. al. classified aggregate functions into three categories: Distributive, Algebraic and Holistic [2]. In order to define each category consider aggregating a two dimensional set of values \(\{X_{ij} | i = 1, ..., I; j = 1, ..., J\}.\)

**Distributive:** Aggregate function \(F()\) is distributive if there is a function \(G()\) such that \(F(\{X_{ij}\}) = G(\{F(\{X_{ij} | i = 1, ..., I\}) | j = 1, ..., J\}).\)

**Algebraic:** Aggregate function \(F()\) is algebraic if there is an \(M\)-tuple valued function \(G()\) and a function \(H()\) such that \(F(\{X_{ij}\}) = H(\{G(\{X_{ij} | i = 1, ..., I\}) | j = 1, ..., J\}).\)

**Holistic:** Aggregate function \(F()\) is holistic if there is no constant \(M\), such that an \(M\)-tuple characterizes the computation of \(F(\{X_{ij} | i = 1, ..., I\}).\)

\(COUNT(), MIN(), MAX()\) and \(SUM()\) are all distributive functions. For all of these functions except the \(COUNT()\), \(F\) equals \(G\). For the \(COUNT()\) function, \(G\) equals \(SUM()\). Average and Standard Deviation are examples of algebraic...
functions, where for Average, the function $G()$ returns the sum and count of the subset and $H()$ adds the two quantities returned by $G$ and then divides to compute the global average. *Median()* and *Mode()* are two examples of holistic functions where it is not possible to compute them without scanning the entire data.

There are many algorithms for efficient computation of a data cube with a distributive or algebraic function as aggregating function. Some of these algorithms are described in chapter 3. I am not aware of any algorithm for efficient computation of data cubes with holistic aggregating functions, although we can always compute these data cubes by computing each of their underlying GROUP BYs independently.
Chapter 3

Sequential Algorithms

OLAP queries are used to extract useful summary information from data warehouses. These queries usually include aggregation over several attributes of a table and are more complicated than OLTP (On-Line Transaction Processing) queries. Since data warehouses are larger than operational databases and OLAP queries are more complex than OLTP queries, it is important to use efficient algorithms to answer these queries.

One way to achieve satisfactory performance is to precompute common queries. Common queries can be precomputed when the system is not being used and updated on a regular basis. Due to disk storage limitations it may not be possible to precompute all of the common queries. In this case we can precompute only a subset of them and compute the remaining queries upon request. Computing these queries can be speeded up by taking advantage of precomputed queries.

For example, consider the sales database given in the previous chapter. In order to precompute all the \textit{GROUP BY} queries resulted from a three dimensional cube operation it is necessary to compute the following \textit{GROUP BY} queries:
1. \textit{GROUP BY Store, Item, Date}

2. \textit{GROUP BY Store, Item}

3. \textit{GROUP BY Store, Date}

4. \textit{GROUP BY Store}

5. \textit{GROUP BY Item, Date}

6. \textit{GROUP BY Item}

7. \textit{GROUP BY Date}

8. None

There is a lattice structure described first in [14] among these queries. Figure 3.1 shows this structure where each node represents a \textit{GROUP BY} query or in other words a view of the data cube. An edge from a node to another node with higher number of attributes implies the result of the former \textit{GROUP BY} query can be computed from the latter one. For example, \textit{GROUP BY Item} can be computed from both \textit{GROUP BY Store, Item} and \textit{GROUP BY Item, Date}. We may decide to materialize only a subset of all the views. For example, we may decide to precompute only \textit{GROUP BY Store, Item, Date} (the core \textit{GROUP BY} or the top view) and compute all the other \textit{GROUP BY} queries by using the precomputed one. In this case, if the user asks for \textit{GROUP BY Store, Item}, it is necessary to aggregate all the rows of the \textit{GROUP BY Store, Item, Date} with the same value for Store and item attributes over Date attribute.
3.1 Hierarchies

In many cases a GROUP BY attribute has some hierarchical structure. For example, the date attribute has the hierarchy:

Day --> Month --> Year

We can also consider another hierarchy for date:

Day --> Week

Existence of such hierarchies is an important issue that should be taken into account when computing the data cube. As previously mentioned, two common operations in OLAP are drill-down and roll-up. We have already seen an application of these operations when we compute aggregated data at different granularities by choosing a different number of GROUP BY attributes. In this case, an example of
a drill-down operation is to start from \textit{GROUP BY store} and move to a finer granularity level, e.g. \textit{GROUP BY store, date}, and finally move to the finest granularity level, i.e. \textit{GROUP BY store, date, item}:

\texttt{GROUP BY store ---+ GROUP BY store, date ---+ GROUP BY store, date, item}

Another application of these operations is conceivable when we compute aggregated data at different granularities by choosing a different level of hierarchies for each of the \textit{GROUP BY} attributes. In this case, an example of a drill-down operation is to start from \textit{GROUP BY date} at the yearly level and move to a finer granularity level, e.g. \textit{GROUP BY date} at the monthly level and finally moving to the finest granularity level, i.e. \textit{GROUP BY date} at the daily level:

\texttt{GROUP BY year ---+ GROUP BY month ---+ GROUP BY day}

The lattice structure is more complex in the presence of hierarchies. For example, consider attribute hierarchies shown in Figure 3.2. Figure 3.3 shows the data cube lattice structure in the presence of these hierarchies.

## 3.2 Selecting queries to precompute

As previously mentioned, one way to achieve satisfactory performance is to precompute common queries or in other words to materialize the views of the data cube associated with them, but, due to disk storage limitations, it may not be possible to precompute all the common queries. In this case, we should decide which queries to precompute or, in other words, which views to materialize. If we do not want to refer to the raw data we always have to materialize the top view because it can not be constructed by using the other materialized views.
Harinarayan et. al. proposed a greedy algorithm in [14] for this problem. They assumed that

1. We wish to minimize the average time to materialize a view.

2. We can only materialize a fixed number of views.

They claimed that this problem was NP-complete and therefore suggested a heuristic solution for it. The input to their algorithm is a data cube lattice with space costs associated with each view. Furthermore they assume that the cost in terms of space is the number of the rows in the view.

Let $C(v)$ denote the cost of view $v$, and $k$ represent the number of views in addition to the top view that we want to materialize. The benefit of view $v$ relative to some set $S$ of views that has already been selected, is denoted by $B(v, S)$ and is defined as follows:
Figure 3.3: Hierarchical lattice structure
S = \{top view\};
for i = 1 to k do begin
    select a view v not in S such that B(v, S) is maximized;
    S = S union \{v\};
end;
return S;

Figure 3.4: The greedy algorithm for selecting a set of views to materialize

1. For each view w which can be constructed from materialized view v(w ≤ v), define $B_w$ as:
   
   (a) Let u denote the view with least cost in S such that w ≤ u. Since the top view is in S, there must be at least one such view in S.
   
   (b) If $C(v) < C(u)$, then $B_w = C(u) - C(v)$, otherwise $B_w = 0$.

2. Define $B(v, S) = \sum_{w \leq v} B_w$.

In other words the benefit of selecting a view v, is equal to the sum of all the benefits that selecting v makes for each of the views w that can be constructed from v. This individual benefit, in turn, is measured by comparing the cost of v with the cost of the cheapest view u, in the set of all so far selected views S that w can be constructed from. If cost of v is less than cost of u then selecting v makes some benefit with respect to w, otherwise it makes no benefit.

The greedy algorithm proposed in [14] is shown in Figure 3.4. The algorithm starts with the top view of the lattice and then it iterates a loop k times (k is number of the views, other than the top view, to materialize). In each iteration of the loop the algorithm selects a view v which has not been added to S before and materializing it would cause the maximum benefit to the views that can be
constructed from \( v \), including \( v \) itself. Table 3.1 shows a trace of running this algorithm on the lattice of Figure 3.5 with \( k = 2 \).

In the first round \( SD \) is selected to be materialized, because the benefit of materializing it is the highest among all the candidates. In the second and third rounds, \( SI \) and \( I \) are selected respectively, so among all the views we only materialize \( SID \) (the top view), \( SD \), \( SI \) and \( I \).

Harinarayan et. al. showed in [14] that the benefit of the greedy algorithm was at least 63% of the benefit of the optimal algorithm where the benefit of each algorithm was defined as the total benefit of selecting the views that were selected by the algorithm. The precise fraction is \((e - 1)/e\) where \( e \) is the base of the natural logarithms.

### 3.3 Computing data cubes

There are many algorithms proposed for computing the data cubes. These algorithms fall into two broad categories: algorithms suitable for ROLAP (Relational On-Line Analytical Processing) systems and algorithms suitable for MOLAP (Mul-
Table 3.1: Benefits of materializing different views at each round

<table>
<thead>
<tr>
<th></th>
<th>First Round</th>
<th>Second Round</th>
<th>Third Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td>(4 \times (6M - 2M)) (= 16.00M)</td>
<td>(2 \times (6M - 2M)) (= 8.00M)</td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>(4 \times (6M - 36500)) (\approx 23.85M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>(4 \times (6M - 5M)) (= 4.00M)</td>
<td>(2 \times (6M - 5M)) (= 2.00M)</td>
<td>(1 \times (6M - 5M)) (= 1.00M)</td>
</tr>
<tr>
<td>S</td>
<td>(2 \times (6M - 100)) (\approx 11.99M)</td>
<td>(2 \times (36500 - 100)) (\approx 0.07M)</td>
<td>(2 \times (36500 - 100)) (\approx 0.07M)</td>
</tr>
<tr>
<td>D</td>
<td>(2 \times (6M - 365)) (\approx 11.99M)</td>
<td>(2 \times (36500 - 365)) (\approx 0.07M)</td>
<td>(2 \times (36500 - 365)) (\approx 0.07M)</td>
</tr>
<tr>
<td>I</td>
<td>(2 \times (6M - 0.05M)) (= 11.90M)</td>
<td>(1 \times (6M - 0.05M)) (= 5.95M)</td>
<td>(1 \times (2M - 0.05M)) (= 1.95M)</td>
</tr>
<tr>
<td>None*</td>
<td>(1 \times (6M - 1)) (\approx 5.99M)</td>
<td>(1 \times (36500 - 1)) (\approx 0.03M)</td>
<td>(1 \times (36500 - 1)) (\approx 0.03)</td>
</tr>
</tbody>
</table>

tidimensional On-Line Analytical Processing) systems. ROLAP systems use relational tables as their data structure. In these systems each view of the data cube is represented by a table and each cell is stored as a tuple. Some of the attributes of these tuples identify its position in the cube and the others contain data values associated with it. MOLAP systems store data in sparse arrays. Each cell of the cube is represented by an element of the array. Unlike ROLAP systems only the data values are stored. The position of a cell determines the values of its dimensional attributes. In the remaining sections of this chapter two ROLAP algorithms and one MOLAP algorithm will be studied.

Sarawagi et. al. summarized in [3] some of the optimization techniques that ROLAP algorithms try to take advantage of:

1. **Smallest-parent**: This optimization technique was first proposed in [2]. According to this technique we should compute a \(GROUP BY\) from its smallest
parent. For example $\text{GROUP BY Store}$ can be computed from any the following $\text{GROUP BY}$ queries:

(a) $\text{GROUP BY Store, Item, Date}$

(b) $\text{GROUP BY Store, Item}$

(c) $\text{GROUP BY Store, Date}$

Obviously the first $\text{GROUP BY}$ is larger than the other two, therefore according to smallest-parent rule it should not be used. The size of the second and the third $\text{GROUP BY}$ queries might be significantly different so the smaller of them should be used to compute $\text{GROUP BY Store}$ query.

2. **Cache-results**: This optimization technique requires us to compute a $\text{GROUP BY}$ query while its parent is still in memory. For example, if $\text{GROUP BY Store, Item}$ has been computed we should compute $\text{GROUP BY Store}$ from it while it is still in memory.

3. **Amortize scans**: This technique tries to amortize disk scans by computing as many $\text{GROUP BY}$ queries as possible, together in memory. For example, if $\text{GROUP BY Store, Item}$ has been computed and stored on disk, we should compute $\text{GROUP BY Store}$ and $\text{GROUP BY Item}$ together in one scan of the $\text{GROUP BY Store, Item}$.

4. **Share Sorts**: This technique is used in sort-based algorithms and is about sharing sorting costs across multiple $\text{GROUP BY}$ queries.

5. **Share Partitions**: This technique is used in hash-based algorithms. When the hash table does not fit in memory data is partitioned into partitions that fit
in memory and aggregation is performed for each partition separately. This technique attempts to share the partitioning cost across multiple GROUP BY queries.

It is not always possible to use all of these optimization techniques simultaneously because they might be contradictory. For example, if GROUP BY Store, Date is smaller than GROUP BY Store, Item, according to Smallest-parent rule we should compute GROUP BY Store from the first GROUP BY, but according to Cache-results rule we should compute it from the second GROUP BY if the second GROUP BY is still in memory.

3.4 PipeSort Algorithm

PipeSort algorithm was proposed by Sarawagi et. al. in [3]. This algorithm uses share-sorts technique by using data sorted in a particular order to compute all the GROUP BY queries that are prefixes of that order. For example if the raw data is sorted in attribute order Store, Item, Date then all the following GROUP BY queries will be computed from it without an additional sort:

1. GROUP BY Store, Item, Date
2. GROUP BY Store, Item
3. GROUP BY Store

This optimization can conflict with smallest-parent technique. For example, if the smallest parent of GROUP BY Store is GROUP BY Store, Date, according to smallest-parent technique we should compute it from this GROUP BY but according to share-sorts technique we should compute it from GROUP BY Store, Item if the
raw data is already sorted in Store, Item, Date attribute order. PipeSort uses a combination of share Sorts and smallest-parent optimizations.

PipeSort also incorporates cache-results and amortize-scan optimizations by computing different GROUP BY queries in a pipelined fashion. For example, if the raw data is already sorted in attribute order Store, Item, Date then all the GROUP BY queries that are prefixes of this order are computed in a pipeline:

Raw Data --> GROUP BY Store, Item, Date --> GROUP BY Store, Item
--> GROUP BY Store

By scanning the raw data and aggregating rows with the same value for all the dimension attributes, rows of GROUP BY Store, Item, Date are computed. When a row of GROUP BY Store, Item, Date is ready then it is propagated up to compute GROUP BY Store, Item and when a row of this GROUP BY is ready it is propagated up to compute GROUP BY Store.

PipeSort algorithm needs an estimate of the number of unique values of each attribute. The input to the PipeSort is the search lattice structure proposed in [14]. Each node of the lattice represents a GROUP BY query. A directed edge connects node $i$ to node $j$ if the GROUP BY $j$ can be computed from GROUP BY $i$ and GROUP BY $j$ has exactly one attribute less than GROUP BY $i$. GROUP BY $i$ is called the parent of GROUP BY $j$. Each node may have several parents. The node at level 0 (the topmost level) of the lattice corresponds to the GROUP BY with no attribute and each node at level $k$ represents a GROUP BY with $k$ attributes. There are two costs associated with each edge $e_{ij}$. The first cost $S(e_{ij})$ is the cost of computing GROUP BY $j$ from GROUP BY $i$ when $i$ is not already sorted and the second cost $A(e_{ij})$ is the cost of computing GROUP BY $j$ from GROUP BY $i$ when $i$ is already sorted.
The output of PipeSort algorithm is a subgraph of the search lattice where each node other than the root node has exactly one parent. The parent of each \textit{GROUP BY} determines the \textit{GROUP BY} that it is computed from. The root \textit{GROUP BY} (\textit{GROUP BY All Attributes}) is computed from the raw data. There is an attribute order associated with each node of the output subgraph. This attribute order determines the sort order in which the \textit{GROUP BY} result will be sorted. If the attribute order of \textit{GROUP BY} \( j \) is a prefix of the attribute order of its parent \textit{GROUP BY} \( i \) then \textit{GROUP BY} \( j \) can be computed without having to sort \textit{GROUP BY} \( i \), but if it is not a prefix of the attribute order of \textit{GROUP BY} \( i \) then in order to compute \textit{GROUP BY} \( j \), \textit{GROUP BY} \( i \) has to be sorted first. The edge that connects \textit{GROUP BY} \( i \) to \textit{GROUP BY} \( j \) where the sort order of \( j \) is a prefix of the sort order of \( i \), is labeled as \( A \) and incurs cost \( A(e_{ij}) \). The edge that connects \textit{GROUP BY} \( i \) to \textit{GROUP BY} \( j \) where the sort order of \( j \) is not a prefix of the sort order of \( i \), is labeled \( S \) and incurs cost \( S(e_{ij}) \). Obviously there is at most one edge labeled \( A \) originating from node \( i \), because only one of the children of node \( i \) can be associated with a prefix order of node \( i \), but there might be several edges labeled as \( S \) originating from node \( i \). The goal of the algorithm is to find a subgraph of the input lattice structure that has the minimum total cost of edges.

### 3.4.1 Algorithm

PipeSort algorithm is shown in Figure 3.6. The algorithm prunes the search lattice level by level starting from level \( k = 0 \) to level \( k = N - 1 \). Where \( N \) is total number of attributes. At each level \( k \), it finds the optimum way of computing nodes at level \( k \) from nodes at level \( k + 1 \). In order to do that PipeSort reduces the problem to a weighted bipartite matching problem. In order to solve the weighted bipartite
PipeSort: (Input search lattice with A and S edges costs)
For level $k = 0$ to $N - 1$ /* $N$ is the total number of attributes */
Generate-Plan($k + 1$ $\rightarrow$ $k$);
For each group-by $g$ in level $k + 1$
  Fix the sort order of $g$ as the order of the level $k$ group-by
  that is connected to $g$ by an A edge;
Generate-Plan($k + 1$ $\rightarrow$ $k$)
  Create $k$ additional copies of each level $k + 1$ node;
  Connect each copy vertex to the same set of level $k$ vertices
    as the original vertex;
  Assign costs $A()$ to edges from the original vertex
    and costs $S()$ to edges from the copy vertex;
  Find the minimum cost matching on the transformed level $k + 1$
    with level $k$;

Figure 3.6: Pipesort Algorithm

matching problem, first, $k$ additional copies of each node at level $k + 1$ are created,
therefore each GROUP BY at level $k + 1$ which has $k + 1$ attributes will have the
same number of nodes. From each replicated node the same set of edges originate as
from the original node. The edge emanating from the original node $i$ at level $k + 1$
to node $j$ at level $k$ is labeled $A$ and incurs cost $A(e_{ij})$ and the edge originating
from each replicated node of $i$ at level $k + 1$ to node $j$ at level $k$ is labeled $S$ and
incurs cost $S(e_{ij})$. We then apply weighted-bipartite matching algorithm to the
transformed subgraph. After pruning some of the edges that connect nodes at level
$k + 1$ to nodes at level $k$, each node at level $k$ is only connected to one node at level
$k + 1$ through either an $A$ edge or an $S$ edge. In the first case (an $A$ edge), the
attribute order of the node at level $k$ determines the order of the attributes of the
node at level $k + 1$ and no sort is necessary. In the second case (an $S$ edge), the
node at level $k + 1$ is resorted in order to compute the node at level $k$.  

28
Figure 3.7 shows a layer of the search lattice after transformation and Figure 3.8 shows the same layer of the search lattice after execution of the weighted bipartite matching algorithm.

Figures 3.9, 3.10 and 3.11 respectively show the original search lattice, the minimum cost sort plan produced by PipeSort algorithm and the final set of pipes used to compute the data cube.
Figure 3.9: Search lattice with associated costs

Figure 3.10: The minimum cost sort plan

Figure 3.11: The pipelines that are executed
3.5 Partitioned-Cube Algorithm

Partitioned-Cube algorithm was proposed by Ross et. al. in [10]. This algorithm is based on two fundamental ideas that have already been used for performing operations such as sorting and joining very large tables:

1. Partition the large table into fragments that fit in memory.

2. Perform the operation over each fragment independently.

Algorithm Partitioned-Cube is described in Figure 3.12. This algorithm depends on another algorithm called Memory-Cube to compute the data cube when the input relation fits in memory. Memory-Cube algorithm will be explained in section 3.5.1. The input of Partitioned-Cube algorithm is a set of tuples $R$, which might be stored in horizontal fragments, the dimension attributes $\{B_1, ..., B_m\}$, aggregation attribute $A$ and the aggregate function $G(.)$. The output of the algorithm is the data cube result for $R$ over $\{B_1, ..., B_m\}$. The output is returned in two fragments $F$ and $D$. Fragment $F$ contains the $GROUP BY$ tuples at finest granularity level and $D$ contains the remaining tuples.

Partitioned-Cube algorithm chooses an attribute $B_j$ among dimension attributes $\{B_1, ..., B_m\}$ of the data cube. This attribute is used to partition input relation $R$ into sets of tuples $\{R_1, ..., R_n\}$. The number of these fragments, $n$, is bounded by both number of available buffers in memory and the domain cardinality of the attribute $B_j$. After the data is partitioned the algorithm Partitioned-Cube is applied on each fragment separately. The union of $F_i$'s returned from applying algorithm on each individual fragment results in $F$, the finest granularity $GROUP BY$ tuples of relation $R$. Algorithm Partitioned-Cube is then applied on $F$ using all the dimension attributes except $B_j$, the attribute that was used to partition relation
Algorithm Partitioned-Cube(R, {B1, ... , Bm}, A, G)

INPUTS: A set of tuples R, possibly stored in horizontal fragments;
        CUBE BY attributes {B1, ... , Bm};
        attribute A to be aggregated; aggregate function G(.)

OUTPUTS: The data cube result for R over {B1, ... , Bm} in two horizontal
         fragments F and D on disk. F contains the finest granularity
         data cube tuples (i.e., grouping by all of {B1, ... , Bm}),
         and D contains the remaining tuples. (F and D may themselves be
         further horizontally partitioned.)

METHOD:
1) if (R fits in memory)
2) then return Memory-Cube(R, {B1, ... , Bm}, A, G);
3) else {choose an attribute Bj among {B1, ... , Bm};
4) scan R and partition on Bj into sets of tuples R1, ... , Rn;
5) /* n \leq \text{card}(Bj) and n \leq \text{number of buffers in memory} */
6) for i = 1 ... n
7)     let (Fi, Di) = Partitioned-Cube(Ri, {B1, ... , Bm}, A, G);
8)     let F = the union of the Fi's;
9)     let (F', D') = Partitioned-Cube(F, {B1, ... , Bj-1, Bj+1,
10)     ... , Bm}, A, G);
11)    let D = the union of F', D' and the Di's;
12)    return (F, D); }

Figure 3.12: Algorithm Partitioned-Cube
Figure 3.13: An example of running Partitioned-Cube Algorithm

$R$, where the result will be stored in $F'$ and $D'$. The union of $F'$, $D'$ and $D_i$'s results in $D$, the remaining tuples of the final output.

There is a problem with line 11 of the description of the algorithm in [10] which is shown in Figure 3.12. We should exclude part of the $D_i$'s from the union of $F'$, $D'$ and $D_i$'s before we assign it to $D$. This line should be:

\[
\text{let } D = \text{the union of } F', D' \text{ and ((union of the } D_i\text{'s) minus tuples that do not include attribute } B_j\text{)};
\]

The reason for this modification is that those tuples of $D_i$'s that do not include $B_j$ are being incorporated in the final result through $F'$ and $D'$. We should not incorporate them into the final result twice.

Figure 3.13 shows an example of running Partitioned-Cube algorithm. First the input relation, which does not fit in memory, is partitioned based on attribute $S$, and all the $\text{GROUP BY}$'s that include attribute $S$ are computed. Then the result of $\text{GROUP BY } S, I, D$ is projected on attributes $I$ and $D$ and then it is partitioned based on attribute $I$ and all the $\text{GROUP BY}$'s that contain $I$ but do not contain $S$ are computed. Finally the result of $\text{GROUP BY } I, D$, which fits in memory, is
projected on attribute $D$ and all the remaining $GROUP BY$s are computed. Dashed lines indicate that sorting is required before computing the $GROUP BY$ and solid lines indicate that sorting is not required because the target $GROUP BY$ is a prefix of the source $GROUP BY$.

### 3.5.1 Algorithm Memory-Cube

The Memory-Cube algorithm was proposed by Ross et. al. in [10]. The only requirement for applicability of this algorithm is that the input relation fits in memory. This algorithm takes advantage of the Pipelining technique that is used in PipeSort algorithm. Unlike the PipeSort which tries to minimize the total cost of computing all the $GROUP BY$s, Memory-Cube tries to minimize the number of pipelines and hence the number of the sort operations that need to be performed in order to compute all the $GROUP BY$s. Algorithm Memory-Cube is described in Figure 3.14.

The sort orders mentioned in algorithm Memory-Cube are sort orders of the pipeline heads that need to be executed. These pipelines are generated by another algorithm called Paths shown in Figure 3.15.

The sort orders mentioned in algorithm Memory-Cube are sort orders of the pipeline heads that need to be executed. These pipelines are generated by another algorithm called Paths shown in Figure 3.15.

The number of paths for $k$ attributes that Paths algorithm generates is $inom{k}{[k/2]}$ and this is the minimum number of paths that cover all the nodes in the search lattice. The following is the result of running algorithm Paths on attributes $\{A, B, C, D\}$:

- $G(1) = D \rightarrow \text{NULL}$
- $G(2) = C.D \rightarrow C \rightarrow \text{NULL}$
- $D$
Algorithm Memory-Cube\( (R, \{B_1, \ldots, B_m\}, A, G) \)

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

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

**METHOD:** Sort \( R \) and combine all tuples that share all values of \( \{B_1, \ldots, B_m\} \);
/* Assume that tuples are sorted according to first sort order */
for each sort order {
    initialize accumulators for computing aggregates at each granularity;
    combine first tuple into finest granularity accumulator;
    for each subsequent tuple \( t \) {
        compare \( t \) with previous tuple, to find the position \( j \) of the first sort order attribute at which they differ;
        if (\( j \) is greater than the number of common attributes between this sort order and the next) then {
            resort the segment from previous tuple \( t' \) at which this condition was satisfied up to the tuple prior to \( t \) according to the next sort order;
        }
        if (grouping attributes of \( t \) differ from those in finest granularity accumulator) then {
            output and then combine each accumulator into coarser granularity accumulator, until the grouping attributes of accumulator match with those of \( t \);
            /* the number of combinings depends on the sort order length and on \( j \) */
        }
    }
    combine current tuple with the finest granularity accumulator;
}

Figure 3.14: Algorithm Memory-Cube
Algorithm Paths\{\{B_1 \ldots B_j\}\}

**INPUTS:** CUBE BY attributes \{B_1 \ldots B_j\};

**OUTPUTS:** A minimal set \(G(j)\) of paths in the search lattice that covers all the nodes.

**METHOD:**

if \((j = 0)\) then return a single node with an empty attribute list;

else \{ let \(G(j - 1) = \text{Paths}\{\{B_1 \ldots B_{j-1}\}\};\)

let \(G_l(j - 1)\) and \(G_r(j - 1)\) denote two replicas of \(G(j - 1)\);

prefix the attribute list of each node of \(G_l(j - 1)\) with \(B_j\);

for each path \(N_1 \rightarrow \ldots \rightarrow N_p\)
in \(G_r(j - 1)\) \{ remove node \(N_p\) and the edge into \(N_p\) (if any) from \(G_r(j - 1)\);

add node \(N_p\) to \(G_l(j - 1)\);

add an edge from node \(B_j.N_p\) to node \(N_p\) in \(G_l(j - 1)\);\}

return the union of the resulting \(G_l(j - 1)\) and \(G_r(j - 1)\);\}

Figure 3.15: Algorithm Paths
\[ G(3) = B.C.D \rightarrow B.C \rightarrow B \rightarrow NULL \]
\[ B.D \rightarrow D \]
\[ C.D \rightarrow C \]

\[ G(4) = A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow NULL \]
\[ A.B.D \rightarrow A.D \rightarrow D \]
\[ A.C.D \rightarrow A.C \rightarrow C \]
\[ B.C.D \rightarrow B.C \rightarrow B \]
\[ B.D \]
\[ C.D \]

The attributes of the paths in \( G(4) \) will be reordered to follow the prefix property:

\[ G(4) = A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow NULL \]
\[ D.A.B \rightarrow D.A \rightarrow D \]
\[ C.A.D \rightarrow C.A \rightarrow C \]
\[ B.C.D \rightarrow B.C \rightarrow B \]
\[ B.D \]
\[ C.D \]

After reordering the paths, they finally will be sorted to be able to take advantage of common sort orders:

\[ G(4) = A.B.C.D \rightarrow A.B.C \rightarrow A.B \rightarrow A \rightarrow NULL \]
\[ B.C.D \rightarrow B.C \rightarrow B \]
\[ B.D \]
\[ C.A.D \rightarrow C.A \rightarrow C \]
It is now possible to use partial sorts that already exists. For example $B.C.D$ and $B.D$ have a prefix in common, i.e. $B$. When tuples that are already sorted based on $B.C.D$, are being scanned, each time a tuple with a new $B$ value is encountered, all the tuples with the previous value of $B$ will be sorted based on $D$ attribute. In this way rather than re-sorting all the tuples based on $B$ and $D$, groups of tuples that share the same value of $B$ will be sorted based on $D$.

### 3.6 Array-Based Algorithm

Unlike ROLAP systems MOLAP systems store data in multidimensional arrays. Each cell of the cube is represented by an element of the array. Unlike ROLAP systems where both the measure attributes and the dimensional attributes are stored, in MOLAP systems only the measure attributes are stored and the position of the cell determines the values of the dimensional attributes. There are not as many algorithms for computing data cube in MOLAP systems. One of them is the array-based algorithm proposed by Naughton et. al. in [6]. This algorithm uses the chunked array data structure to store the data cube. An $n$-dimensional chunked array is an array divided into small pieces called chunks. Each chunk is an $n$-dimensional array by itself and is stored on a single block of the disk. The array-based algorithm compresses each chunk if less than 40% of its cells are valid. The compression method that the array-based algorithm uses is called chunk-offset compression. In this compression method each valid element of a chunk is stored as a pair, $(offset \ in \ chunk, \ data)$. $Offset \ in \ chunk$ is an integer representing the offset of the element from the...
beginning of the chunk in some standard order, such as row major order.

To see how chunking can affect the way we compute GROUP BYs, let's consider a three dimensional array with dimensions $A$, $B$, and $C$. In order to compute GROUP BY $A$, $B$ we project all the cells of the array onto the $AB$ plane. In the absence of chunks, if the array is stored in $A$, $B$, $C$ order on disk, we have to keep the entire $AB$ plane in memory, because we traverse the array in the same order and therefore the value of each $AB$ element is determined only when we finish traversing the array. On the other hand, with chunks we only need to keep a sub-plane of $AB$ corresponding to a single chunk, because assuming that we have direct access to each chunk we can traverse the array in such a way that we aggregate all the $ABC$ chunks that project on the same $AB$ chunk together.

If we want to compute all the GROUP BYs rather than only one, obviously we can compute all of them separately from GROUP BY $A$, $B$, $C$, but this is not necessary. Having computed GROUP BY $A$, $B$, we can compute GROUP BY $A$ or GROUP BY $B$ far more efficiently from GROUP BY $A$, $B$ than from GROUP BY $A$, $B$, $C$. We can use the same lattice structure proposed by [14] and embed a tree in this lattice and compute each GROUP BY from its parent in the tree. Now the question is which tree we should choose. In order to minimize memory requirement of computing the data cube, a spanning tree of the lattice structure can be used that is called minimum size spanning tree. In this tree the parent of each node $n$ is the minimum size node $n'$ from which $n$ can be computed. We can define minimum size spanning tree because unlike in ROLAP systems where the size of each node of the lattice structure is not known in advance, in MOLAP systems the dimension sizes of the array and its chunk sizes are known. Therefore the exact size of each lattice node can be calculated.
Before explaining the array-based algorithm, let's review a basic version of it first. In the beginning we construct the minimum size spanning tree for the GROUP BYs of the cube. Each GROUP BY \( D_{i_1}D_{i_2}...D_{i_k} \) is computed from its parent GROUP BY \( D_{i_1}D_{i_2}...D_{i_{k+1}} \) which has the minimum size. Chunks of GROUP BY \( D_{i_1}D_{i_2}...D_{i_{k+1}} \) are read along the dimension \( D_{i_{k+1}} \) and aggregated to a chunk of \( D_{i_1}D_{i_2}...D_{i_k} \). When each chunk of \( D_{i_1}D_{i_2}...D_{i_k} \) is complete, it is written on disk and its memory is used for the next chunk of \( D_{i_1}D_{i_2}...D_{i_k} \). Only one chunk of \( D_{i_1}D_{i_2}...D_{i_k} \) is kept in memory at any time.

For example consider the array structure of Figure 3.16. This array is a \( 9 \times 9 \times 9 \) array with \( 3 \times 3 \times 3 \) array chunks stored in the dimension order \( ABC \). Chunk numbers shown in Figure 3.16 indicate the order of the layout of array chunks on disk. In order to compute GROUP BY \( B, C \) the chunks are read in order from 1 to 27, and each 3 of them are aggregated to a \( BC \) chunk. The memory that is being used to compute a \( BC \) chunk will be released and reused after this chunk is
written on disk.

The basic algorithm computes each \textit{GROUP BY} independently. For example it scans \textit{ABC} three times to compute \textit{AB}, \textit{AC}, and \textit{BC}. Then it scans the smaller of \textit{AB} and \textit{AC} to compute \textit{A} and the smaller of \textit{AB} and \textit{BC} to compute \textit{B}, etc.. The algorithm is intelligent in choosing the smallest parent to compute each \textit{GROUP BY}, and reusing the memory, but it is naive in the sense that it computes \textit{GROUP BY}s independently. The actual array-based algorithm computes all the children of a parent in a single pass of it.

3.6.1 Single-pass Multi-way Array Algorithm

When we compute multiple \textit{GROUP BY}s at the same time, we need to allocate some memory to each of them. The amount of memory that is required, depends on the order in which the input array is scanned. The array-based algorithm uses a special logical order called \textit{dimension order} to minimize this total amount of memory.

To see how the order in which the input array is scanned determines the total memory required, consider the array structure of Figure 3.16. Assume we scan the array in \textit{ABC} row major order. First we read chunk \textit{a0b0c0} of \textit{ABC}. This chunk is aggregated along dimension \textit{A} to chunk \textit{b0c0} of \textit{BC}, along dimension \textit{B} to chunk \textit{a0c0} of \textit{AC} and along dimension \textit{C} to chunk \textit{a0b0} of \textit{AB}. Then chunk \textit{a1b0c0} of \textit{ABC} is read and it is aggregated to chunks \textit{b0c0}, \textit{a1c0} and \textit{a1b0}. The third chunk of \textit{ABC}, chunk \textit{a2b0c0}, is read in and aggregated to chunks \textit{b0c0}, \textit{a2c0} and \textit{a2b0}. Now chunk \textit{b0c0} is complete and it is written on disk and its memory is reused for chunk \textit{b1c0}. Chunk \textit{b1c0} is written on disk when chunks 4, 5 and 6 are aggregated and then its memory is reused for chunk \textit{b2c0}, which is written on disk after chunks 7, 8 and 9 are aggregated. Chunk \textit{a0c0} is complete when chunks 1, 4 and 7 are
aggregated along dimension B. Chunk a1c0 is written after chunks 2, 5 and 8 are aggregated and chunk a2c0 is ready when chunks 3, 6 and 9 are all aggregated along dimension B. Chunks of AB are complete only when all the chunks from 1 to 27 are aggregated along dimension C.

In this example, to compute BC we need to keep 1 chunk of BC in memory, to compute AC we need to keep 3 chunks of AC in memory and for AB we need memory for $3 \times 3 = 9$ chunks of AB. In general we need $|B_c||C_c|u$ memory for BC, $|A_d||C_c|u$ memory for AC and $|A_d||B_d|u$ memory for AB, where $|X_d|$ indicates the size of dimension $X$, $|X_c|$ stands for the chunk size of dimension $X$ and $u$ is the size of each chunk element. The following rule states generalizes the idea:

**Rule 1:** For a GROUP BY $(D_{j1}, D_{j2}, \ldots, D_{jn-1})$ of the array $(D_1D_2\ldots D_{n-1})$ read in the dimension order $O = (D_1D_2\ldots D_n)$, if $(D_{j1}, D_{j2}, \ldots, D_{jn-1})$ contains a prefix of $(D_1D_2\ldots D_n)$ with length $p$, $0 < p < n - 1$, we allocate $\prod_{i=1}^{p} |D_i| \times \prod_{i=p+1}^{n-1} |C_i|$ units of array element to GROUP BY $(D_{j1}, D_{j2}, \ldots, D_{jn-1})$, where $|D_i|$ is the size of dimension $i$ and $|C_i|$ is the chunk size of dimension $i$.

Based on the above rule, for many of the GROUP BYs the allocated memory is less than the actual size, since $|C_i|$ is much smaller than $|D_i|$ in most of the dimensions. The memory that is being saved in this way can be used to overlap computation of more GROUP BYs. Some kind of structure is needed to coordinate the overlapped computation. A spanning tree of the GROUP BYs’ lattice can serve this purpose. For a given dimension order, different spanning trees require different amounts of memory. A minimum memory spanning tree will be defined next.

A Minimum Memory Spanning Tree (MMST) for a cube $(D_1D_2\ldots D_n)$ with respect to a dimension order $O = (D_1D_2\ldots D_n)$ has $n + 1$ levels with the root
Figure 3.17: Minimum Memory Spanning Tree

$(D_1, D_2, \ldots, D_n)$ at level $n$. Any node $N$ at level $i$, $0 \leq i < n$, can be computed from all the nodes at level $i + 1$ that contain its dimensions. The parent of node $N$ is the node at level $i$ that minimizes the required memory to compute $N$ according to rule 1. In other words, the parent of node $N$ has the minimum common prefix with $N$ among all the nodes at level $i + 1$ that contain node $N$'s dimensions. If there are more than one node that minimize the required memory, the one with minimum size is selected. Figure 3.17 shows the MMST for the cube $ABC$ in the dimension order $(A, B, C)$.

The total memory required by all the nodes at each level of the MMST can be calculated using the following rule. It is assumed that the chunk size is the same for all dimensions, i.e., for all $i$, $|C_i| = c$.

Rule 2: The total memory requirement for level $n - j$ of the MMST for a dimension order $O = (D_1, D_2, \ldots, D_n)$, is given by:

$$\prod_{i=1}^{n-j} |D_i| + C(j, 1)(\prod_{i=1}^{n-j-1} |D_i|)c + C(j + 1, 2)(\prod_{i=1}^{n-j-2} |D_i|)c^2 + \ldots + C(n - 1, n - j)c^{n-j}.$$
Different dimension orders of an array \((D_1D_2\ldots D_n)\) may have different \(MMSTs\). Each of these \(MMSTs\) may have different memory requirements. The optimal dimension order is the dimension order whose \(MMST\) requires the minimum amount of memory. It is proved by Zhao et. al. in [6] that the optimal dimension order \(O\) is \((D_1D_2\ldots D_n)\), where \(|D_1| \leq |D_2| \leq \ldots \leq |D_n|\). \(|D_i|\) denotes size of the dimension \(D_i\).

If the available memory is less than the memory required for the \(MMST\) for the optimal dimension ordering \(O\), we can not allocate sufficient memory for some of the subtrees of the \(MMST\). These subtrees are called \textit{incomplete subtrees}. Naughton et. al. proposed a heuristic algorithm called \textit{Multi-pass, Multi-way, Array Algorithm} in [6] for computing the cube in this situation. They used the heuristic of allocating memory to subtrees of the root from the right to the left order.

### 3.7 Performance comparison

I have implemented \textit{PipeSort}, \textit{Partitioned-Cube} and \textit{Single-pass Multi-way Array} algorithms using C++ programming language in a UNIX environment. Each of these algorithms was run on a PC with 266 MH Pentium processor with 128 MB of RAM. The time was measured from the point the input relation was read to the point the data cube was written on disk. In order to compare performance of these three algorithms I used real-world data on the cloud coverage of the globe presented in [15]. The data used corresponds to the measurement of the cloud coverage of the globe over a one month period, September 1985. The reason that this month was chosen is that the same data was used in [5], and I wanted to compare my experiment results with those in that paper. For each month there are two data sets available: one contains measurements made over the ocean, and another contains measurements
made over land. I chose the land data set. This data set consists of 1,015,367 rows which I divided into five nearly equal partitions. In each experiment I used one, two, three, four or all of the five partitions. There are about 20 different fields, among which the following were chosen as data cube dimensions, with cardinalities listed in parentheses: day(30), hour(24), sky brightness(2), latitude(180) and longitude(360). In each experiment I chose either the first four, or all the five attributes as cube dimensions. The aggregation attribute is a measure (between 0 and 8) of cloud coverage. Each row of the input data set was originally 56 character long, but for an implementation reason each row was expanded to 61 characters, so the expanded data file is about 62 MB.

Tables 3.2 and 3.3 show the results of the experiments. Figures 3.18 and 3.19 show the same results in pictorial format.

As tables 3.2, 3.3 show, Partitioned-Cube outperforms PipeSort on computing 4-dimensional cube, but they both perform almost the same on 5-dimensional cube computation. The reason may be the preprocessing that PipeSort performs on the input data to determine cardinalities of cube dimensions. PipeSort needs these cardinalities to estimate each *GROUP BY*’s size and hence to assign a pair of costs to each edge of the search lattice. Partitioned Cube does not need to do that so
Table 3.3: Computing a 5-dimensional (30*24*2*158*360) Cube

<table>
<thead>
<tr>
<th>Number of Rows</th>
<th>Pipe Sort Algorithm</th>
<th>Partitioned Cube Algorithm</th>
<th>Multi-way Array Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>203074</td>
<td>68.79</td>
<td>76.73</td>
<td>89.37</td>
</tr>
<tr>
<td>406148</td>
<td>139.52</td>
<td>155.06</td>
<td>169.05</td>
</tr>
<tr>
<td>609222</td>
<td>212.56</td>
<td>236.03</td>
<td>247.01</td>
</tr>
<tr>
<td>812296</td>
<td>288.34</td>
<td>317.36</td>
<td>328.57</td>
</tr>
<tr>
<td>1015367</td>
<td>352.85</td>
<td>398.95</td>
<td>418.11</td>
</tr>
</tbody>
</table>

Figure 3.18: Computing a 4-dimensional (30*24*2*158) Cube

Figure 3.19: Computing a 4-dimensional (30*24*2*158*360) Cube
it does not perform any preprocessing on the input data. As the number of cube dimensions increases, the time it takes to perform the preprocessing becomes less significant when compared to the total execution time of the PipeSort algorithm. The reason is that the time it takes to perform the preprocessing is $O(n)$ but the time it takes to compute $GROUP BY$s is $O(2^n)$.

Although Partitioned-Cube minimizes the number of sort operations and it takes advantage of partial sort order similarities among different $GROUP BY$s, it does not seem that it has a big advantage over PipeSort when the input data fits in memory and number of dimension attributes is not less than 5. PipeSort and Partitioned-Cube were not run on a data set that does not fit in memory, but in this case PipeSort has the advantage that it can handle the problem regardless of the distribution of data, but Partitioned-Cube may not be able to do so because it partitions the large data sets into smaller ones based on one of their dimension attributes. Partitioned-Cube recursively repartitions each partition until all of them fit in memory. Now the question is what happens if the distribution of values in the partitioning attribute does not allow us to partition the data set into partitions that fit into memory. For example consider $gender$ attribute where there are only two possible values, e.g. male and female. One solution to this problem is to choose another attribute as partitioning attribute that has a finer distribution. But what if all the dimension attributes have the same coarse distribution as gender attribute? Even if we can find an attribute with desired distribution in the first partitioning step, we may finally run into this problem when we want to choose a new attribute for future partitioning operations. PipeSort uses external sort algorithms if the input data set does not fit into memory. It does not rely on the distribution of data and so does not suffer from this problem. That means there are some cases that
Partitioned-Cube can not be used while PipeSort can always be used.

The Multi-way Array algorithm is the only MOLAP algorithm that was implemented. This algorithm reads the input data in the same format as the two other ROLAP algorithms, but it writes its output in a different format than the other two algorithms. The output of Multi-way Array algorithm consists of a row for each valid cell of the data cube. For each cell its offset from the beginning of the chunk along with its value are stored, so the output of this MOLAP algorithm is more compressed than ROLAP algorithm outputs where both dimension attribute values and aggregation operation result are stored. Despite the advantage of producing smaller sized outputs, Multi-way Array algorithm did not perform well in any of the experiments, specially when the number of dimension attributes was higher. The reason may be that the Multi-way Array algorithm does not take advantage of pipelining, as a result increasing the number of dimension attributes has a more dramatic effect on its performance than it has on PipeSort or Partitioned-Cube algorithms.
Parallel Algorithms

Several parallel versions of the Partitioned-Cube and Single-pass Multi-way Array algorithms were designed and implemented. The reason that these two algorithms were chosen, was that the former had a good potential for parallelism and the latter was a good representative of MOLAP algorithms. Each of these algorithms was implemented in C++ using MPI as message passing library. They were run on a cluster of 16 Pentium 266 MH machines each with 128 MB Ram and 2 GB hard disk.

4.1 Parallel Partitioned-Cube Algorithm

In this algorithm one of the nodes of the network is chosen as coordinator node. The input to and output of the algorithm are stored at this node. Usually data warehouses are centralized and so must be the result of any cube calculations. That is why having a coordinator node is a reasonable assumption. The input to Parallel Partitioned-Cube algorithm is a set of tuples $R$, which may be partitioned in horizontal fragments, the dimension attributes $\{B_1, ..., B_m\}$, aggregation attribute
A and the aggregate function $G(.)$. The output of the algorithm is the data cube result for $R$ over \{$B_1, ..., B_m$\}. The output is returned in two fragments $F$ and $D$. Fragment $F$ contains the $GROUP BY$ tuples at finest granularity level and $D$ contains the remaining tuples.

Like the sequential version, Parallel Partitioned-Cube algorithm chooses an attribute $B_j$ among dimension attributes $\{B_1, ..., B_m\}$ of the data cube. This attribute is used to partition input relation $R$ into sets of tuples $\{R_1, ..., R_n\}$. The number of these fragments, $n$, is bounded by both number of available network nodes and the domain cardinality of the attribute $B_j$. Unlike the sequential version each fragment is sent to one of the nodes by coordinator node and then algorithm Memory-Cube is applied on all of the fragments concurrently. The result of Memory-Cube algorithm at each node is sent back to the coordinator node. The union of $F_i$'s obtained by applying algorithm Memory-Cube on each individual fragment, results in $F$, the finest granularity $GROUP BY$ tuples of relation $R$. Then algorithm Parallel Partitioned-Cube is applied on $F$ using all the dimension attributes except $B_j$, the attribute that was used to partition relation $R$, and the result will be stored in $F'$ and $D'$. Like the sequential version the union of $F'$, $D'$ and tuples of $D_i$'s that include attribute $B_j$ results in $D$, the remaining tuples of the final output.

It is assumed that we can always find an attribute $B_j$ that can be used to partition the input relation into fragments that fit into memory. The same assumptions were made in the sequential version of the algorithm by Ross et. al. in [10]. Three different versions of the Parallel Partitioned-Cube algorithm were implemented using the following three different methods:

**Method 1**: The coordinator node only partitions the input relation if it does not fit into memory, otherwise it applies the Memory-Cube algorithm and returns
the results. The coordinator node stores fragments locally on disk while performing partitioning process. Then it reads fragments back from the disk and sends each of them to one of the network nodes.

Method 2: The coordinator node partitions the input relation even if it does fit into memory. The coordinator node stores fragments locally on disk while performing partitioning process. Then it reads fragments back from the disk and sends each of them to one of the network nodes.

Method 3: The coordinator node only partitions the input relation if it does not fit into memory, otherwise it applies the Memory-Cube algorithm and returns the results. The coordinator node does not store fragments locally on disk. It maintains a buffer in memory for each fragment and sends the contents of the buffer to corresponding node when it becomes full and reuses the buffer for the rest of the partitioning process.

The difference between methods 1 and 2 is in the criteria for partitioning the input relation. In method 1, the input relation is partitioned only if it does not fit into memory. In method 2, whether the input relation fits into memory or doesn't, it is partitioned and its fragments are sent to different nodes. The rationales for these two methods are as follows. After algorithm Parallel Partitioned-Cube is applied on the input relation $R$ once, the finest granularity $GROUP BY, F$, is used as the input relation to a recursive call of the algorithm. $F$ is likely to be significantly smaller than $R$, therefore it may fit in memory. Method 1 does not partition $F$ if it fits in memory. This method is designed based on the assumption that the overhead of partitioning a fairly small relation and sending out its fragments to different nodes and then collecting back the results from these nodes dominates the
benefit of parallel processing of the relation. Method 2 is designed based on the opposite assumption.

Method 3 uses the same criteria as method 1 for partitioning the input relation, but it behaves differently while it is partitioning the relation. In method 1 fragments of the input relation are stored locally during the partitioning phase and only after this phase is complete they are sent to other nodes of the network. Method 3 does not store fragments locally. It keeps a piece of each fragment in a buffer. When this buffer becomes full, its content is sent to the corresponding node of the network. The rational for this method is the fact that storing fragments on disk and then reading them back into memory can cause a significant overhead. This overhead can be avoided if we use method 3.

In order to compare the relative performance of these algorithms experiments were performed and the total execution time of each algorithm was measured. To measure the execution time of an algorithm a timer class was used that worked like a stop watch. The timer class has methods to reset, start, stop and resume the timer. This timer was started at the very beginning of the program and stopped at the very end. Since the tested algorithms were all parallel, the maximum reported value among all the nodes was considered as the total execution time. Separate timers were used for CPU, I/O, communication and total execution time when it was necessary to break down the total execution time into its components.

Figures 4.1 through 4.2 compare the execution time of each of these methods under different circumstances.

There is not a significant difference between the performance of Methods 1 and 2. This suggests that it does not matter whether we partition and distribute small $F$ relations that are resulted from the union of $F_i$’s obtained by applying
algorithm Memory-Cube on each individual fragment, or process these small $F$ relations in their entirety locally.

Method 3 outperformed both methods 1 and 2 in all the experiments. This is due to the savings in I/O time that is made in this method. Writing fragments on the disk and reading them back into memory is a time consuming task. It may take up to 50% of the total execution time. The only disadvantage of this method over the other methods is its need to have a buffer in main memory for each fragment. But this memory overhead is well justified by the better performance of this method. This method was chosen to perform more experiments.

Figures 4.3 and 4.4 show how this method scales up with respect to number of the nodes of the network. Method 3 shows a speed up close to linear from 1 to 2 and 2 to 4 nodes, but speed up decreases as number of nodes increase. The primary reason for this problem is the way the input relation $R$ is partitioned. This relation
Figure 4.3: Computing a 4-dimensional (30*24*2*158) Cube on different number of nodes using method 3

is partitioned into almost equal fragments among 2 and 4 nodes, but in 8 and 16 node cases it is not partitioned evenly among all of the nodes and that is because of the limitation of the hash function that was used and the domain cardinality of the partitioning attribute. This suggests that using a good hash function and choosing an attribute with large enough cardinality is critical in achieving a good speed up.

Tables 4.1 and 4.2 show how the total execution time breaks down into CPU, I/O and Communication times when method 3 is used on two nodes.

In both 4-dimensional and 5-dimensional cases CPU takes between 55% to
Table 4.1: Computing a 4-dimensional \((30 \times 24 \times 2 \times 158)\) Cube on 2 nodes using method 3

<table>
<thead>
<tr>
<th>Number of Rows</th>
<th>CPU Time Percentage</th>
<th>I/O Time Percentage</th>
<th>Communication Time Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>203074</td>
<td>59.78</td>
<td>38.70</td>
<td>1.52</td>
</tr>
<tr>
<td>406148</td>
<td>59.08</td>
<td>39.34</td>
<td>1.58</td>
</tr>
<tr>
<td>609222</td>
<td>60.32</td>
<td>38.00</td>
<td>1.68</td>
</tr>
<tr>
<td>812296</td>
<td>60.45</td>
<td>38.11</td>
<td>1.44</td>
</tr>
<tr>
<td>1015367</td>
<td>59.39</td>
<td>39.00</td>
<td>1.61</td>
</tr>
</tbody>
</table>

Table 4.2: Computing a 5-dimensional \((30 \times 24 \times 2 \times 158 \times 360)\) Cube on 2 nodes using method 3

<table>
<thead>
<tr>
<th>Number of Rows</th>
<th>CPU Time Percentage</th>
<th>I/O Time Percentage</th>
<th>Communication Time Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>203074</td>
<td>59.48</td>
<td>39.59</td>
<td>0.93</td>
</tr>
<tr>
<td>406148</td>
<td>55.67</td>
<td>42.95</td>
<td>1.38</td>
</tr>
<tr>
<td>609222</td>
<td>58.46</td>
<td>40.45</td>
<td>1.09</td>
</tr>
<tr>
<td>812296</td>
<td>57.58</td>
<td>41.06</td>
<td>1.36</td>
</tr>
<tr>
<td>1015367</td>
<td>56.93</td>
<td>41.60</td>
<td>1.47</td>
</tr>
</tbody>
</table>
61%, I/O between 38% to 43% and communication less than 2% of the total execution time. This suggests that method 3 is more CPU bound than I/O bound and its communication overhead is minimal.

4.2 Parallel Single-pass Multi-way Array Algorithm

In this algorithm one of the network nodes works as coordinator node. The input and output of the algorithm are stored at this node. In the beginning of the algorithm this coordinator node reads the input data into memory and distributes it evenly among all the nodes of the network. This phase could be eliminated if the input relation was already partitioned and distributed among the nodes.

For ease of implementation it is assumed that each node has enough free memory to receive and hold one fragment of the input data in main memory. Unlike parallel Partitioned-Cube algorithm, parallel Multi-way Array algorithm does not use any of the dimension attributes for partitioning purpose. Instead it just sends a group of rows to a node when it reads as many rows as the number of rows per partition. This makes the partitioning process faster and creates more equal partitions than in the parallel Partitioned-Cube algorithm.

When the partitioning process is complete then each node extracts unique values of each dimension attribute of the raw data fragment it has in memory and sends out the set of unique values to the coordinator node. The coordinator node places all the unique value sets for an attribute together and sorts them. Then it sends back the sorted global unique value set for that attribute to each node and each node finds the position of each of the unique values it has extracted in the global sorted list of unique values. This process is repeated for all of the dimension attributes. After this phase is completed each node computes the core cuboid
(cuboid resulted from computing GROUP BY all of the dimension attributes) for
the fragment of the input data it has in memory. The core cuboids are stored on
disk for further processing. Once core cuboids are computed, input data fragments
are not needed any more, so the memory that was allocated to them can be released.

The next phase of the algorithm is to read the core cuboid and compute the
other cuboids of the data cube. This is done at each node separately, therefore the
resulting cuboids at different nodes need to be combined. After all the cuboids other
than the core cuboid are combined and stored, chunks of the core cuboid are sent
directly to the coordinator node and combined and stored there. The computation
and combination of different cuboids of the data cube are implemented using four
different methods.

**Method 1**: Cuboids are computed at different nodes independently and are com­
bined at the end by sending them directly to the coordinator node and merging
them at this node.

**Method 2**: Cuboids are computed at different nodes independently and are com­
bined at the end by sending them indirectly to the coordinator node and
merging them at this node.

**Method 3**: Cuboids are computed at different nodes and combined at coordina­
tor node at the same time by sending each computed chunk directly to the
coordinator node and merging these chunks at this node.

**Method 4**: Cuboids are computed at different nodes and combined at coordinator
node at the same time by sending each computed chunk indirectly to the
coordinator node and merging these chunks at this node.
In methods 1 and 2 there is no interaction among different nodes while they are reading their own core cuboid and computing the other cuboids. In these two methods each node stores chunks of different cuboids on disk locally and sends these chunks to coordinator node once all the chunks of all the cuboids are computed. The difference between these two methods is in the way the computed chunks are sent to the coordinator node. In method 1 these computed chunks are sent directly from each node to the coordinator node, while in method 2 chunks are sent to the coordinator node in several steps. The number of these steps is equal to \( d = \log(N) \) where \( N \) is number of the nodes of the network.

In method 2 nodes of the network form a \textit{d-dimensional hypercube} structure (see Figure 4.5). A \textit{d-dimensional hypercube} consists of \( N = 2^d \) nodes where each node is assigned a label \( 0, \ldots, 2^n - 1 \). Two nodes communicate with each other if and only if their corresponding binary representation of their labels differs in exactly one bit. Therefore each node communicates with \( d = \log(N) \) other nodes.

In the first step of sending chunks to the coordinator node, each node with 1 in the first bit of its binary number sends its computed chunks to the neighbor node with 0 in the first bit of its binary number. The receiver nodes combine received chunks with their own computed chunks. In the second step of sending chunks to
the coordinator node, each receiver node of the previous step with 1 in the second bit of its binary number sends its computed chunks to another neighbor phase 1 receiver node with 0 in the second bit of its binary number. The receiver nodes combine received chunks with the chunk results of the previous step. This process will be repeated $d = \log(N)$ times where $N$ is number of nodes of the network. After the final step node 0, the coordinator node, will have the combined chunks of all the other nodes and can store them as final results on disk.

The reason that hypercube structure is used explicitly rather than implicitly by using standard gather operation is the fact that in the gather operation all of the nodes send their data to a single node based on a hypercube structure and it is up to this node to combine the received data to form the desired result where as here each node that receives some data combines it with its own data before it sends it to another node, therefore the combining process is distributed among all of the receiver nodes and is performed faster,

In methods 3 and 4 each node sends a chunk either directly or indirectly to coordinator node as soon as that chunk is computed. Therefore no nodes other than coordinator node stores chunks locally. Like the previous 2 methods the difference between these two methods is in the way the computed chunks are sent to the coordinator node. In method 3 these computed chunks are sent directly from each node to the coordinator node, while in method 4 chunks are sent to the coordinator node in the same fashion as in method 2.

The rationales for these four methods are as follows. In methods 1 and 2 cuboid chunks are sent in bulk to the coordinator node but in methods 3 and 4 they are sent one at a time, therefore communication overhead is lower in methods 1 and 2 than methods 3 and 4. On the other hand methods 1 and 2 store computed chunks
Figure 4.6: Computing a 5-dimensional (30*24*2*158*360) Cube on two nodes using four different methods

Figure 4.7: Computing a 5-dimensional (30*24*2*158*360) Cube on four nodes using four different methods

locally on disk while methods 3 and 4 send them to the coordinator node while computing them, therefore methods 3 and 4 do not spend I/O time for writing and reading back computed chunks at each node. Methods 1 and 3 send chunks directly to the coordinator node while methods 2 and 4 send them indirectly based on the hypercube structure. The advantage of sending chunks indirectly to the coordinator node over sending them directly is to distribute the combining task among different nodes and therefore saving in the time it takes to combine all the chunks. On the other hand sending chunks indirectly to the coordinator node causes higher communication overhead.

Figures 4.6 and 4.7 compare the execution time of each of these methods under different circumstances.

These four methods do not show significant differences in computation of data cube on 2 nodes. In computation of data cube on 4 nodes, method 4 outperforms the
other three methods in most experiments. This is mostly due to the saving in I/O time that is achieved in this method. Writing chunks on the disk and reading them back in memory can take about 10% of the total execution time. The other reason is the saving made in combination time in method 4. This method was chosen to perform more experiments.

Figures 4.8 and 4.9 show how method 4 scales up with respect to number of nodes in the network. Method 4 shows some speed up from 1 to 2 and 2 to 4 nodes, but speed up decreases dramatically and even turns into slow down as the number of the nodes increases. This may partially be due to the communication overhead that is associated with this method.

Tables 4.3 and 4.4 show how the total execution time breaks down into CPU,
Table 4.3: Computing a 4-dimensional (30*24*2*158) Cube on 2 nodes using method 4

<table>
<thead>
<tr>
<th>Number of Rows</th>
<th>CPU Time Percentage</th>
<th>I/O Time Percentage</th>
<th>Communication Time Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>203074</td>
<td>66.30</td>
<td>29.39</td>
<td>4.31</td>
</tr>
<tr>
<td>406148</td>
<td>65.22</td>
<td>30.65</td>
<td>4.13</td>
</tr>
<tr>
<td>609222</td>
<td>63.31</td>
<td>32.39</td>
<td>4.30</td>
</tr>
<tr>
<td>812296</td>
<td>63.75</td>
<td>31.96</td>
<td>4.29</td>
</tr>
<tr>
<td>1015367</td>
<td>63.59</td>
<td>32.23</td>
<td>4.18</td>
</tr>
</tbody>
</table>

Table 4.4: Computing a 5-dimensional (30*24*2*158*360) Cube on 2 nodes using method 4

<table>
<thead>
<tr>
<th>Number of Rows</th>
<th>CPU Time Percentage</th>
<th>I/O Time Percentage</th>
<th>Communication Time Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>203074</td>
<td>59.74</td>
<td>27.81</td>
<td>12.45</td>
</tr>
<tr>
<td>406148</td>
<td>57.49</td>
<td>29.25</td>
<td>13.26</td>
</tr>
<tr>
<td>609222</td>
<td>56.26</td>
<td>30.35</td>
<td>13.39</td>
</tr>
<tr>
<td>812296</td>
<td>56.47</td>
<td>30.04</td>
<td>13.49</td>
</tr>
<tr>
<td>1015367</td>
<td>56.77</td>
<td>29.46</td>
<td>13.77</td>
</tr>
</tbody>
</table>
I/O and Communication times when method 4 is used on two nodes.

In 4-dimensional case CPU takes between 63\% to 67\%, I/O between 29\% to 33\% and communication about 5\% of the total execution time. In 5-dimensional case CPU takes between 56\% to 60\%, I/O between 27\% to 31\% and communication about 13\% of the total execution time. This suggests that the role of the communication time becomes more important as number of the cube dimensions increases. This is consistent with the poorer speed up of method 4 when it was used to compute a 5-dimensional cube than the case it was used to compute a 4-dimensional cube (see Figures 4.8 and 4.9).
Chapter 5

Conclusion

In the previous chapters a detailed description of three different sequential algorithms for computing data cubes along with performance comparison of these algorithms were presented. Two of these algorithms, PipeSort and Partitioned-Cube, were ROLAP and the third one, Multi-way Array, was a MOLAP algorithm. Experiments split in showing the best between the two ROLAP algorithms but the MOLAP algorithm was always outperformed by the two ROLAP algorithms. Partitioned-Cube and Multi-way Array algorithms were chosen for parallel implementation. Several parallel versions of each of these two algorithms were implemented. For each algorithm the best implementation was chosen to perform more experiments.

Experiments showed that Parallel Partitioned-Cube algorithm performed better than Parallel Multi-way Array algorithm and it scaled up better with the number of the nodes of the network. This suggests that Partitioned-Cube algorithm is a more suitable candidate for incorporating parallel processing although in order to achieve the best results of this algorithm, it should be made sure that in the distribution phase the input relation is evenly distributed among the nodes of the
network. This can be done by choosing an appropriate partitioning attribute and applying a hash function with enough discriminating capability.
Bibliography


[12] Jiawei Han: Towards On-Line Analytical Mining in Large Databases. newblock SIGMOD Record 27(1): 97-107 (1998)


67


