# INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

# Parallel Generation of ROLAP Data Cubes

by
Ying Chen

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
June, 2005

DALHOUSIE UNIVERSITY

To comply with the Canadian Privacy Act the National Library of Canada has requested that the following pages be removed from this copy of the thesis:

Preliminary Pages

Examiners Signature Page (pii)

Dalhousie Library Copyright Agreement (piii)

Appendices

Copyright Releases (if applicable)

*To all who love me.*

# Table of Contents

# List of Tables

# List of Figures

## Abstract

More and more organizations, such as business, health care providers and scientific enterprises, rely on *Online Analytical Processing* (OLAP) to analyze massive data sets at a variety of summary levels and in a multidimensional way. In OLAP systems, one of the most computationally intensive tasks is to execute the *Cube* query, which was proposed by Gray et al. in 1997 as an extension of the Structured Query Language (SQL). A cube query generates a set of group-bys/views over all combinations of a set of attributes/dimensions from a table. The result of the query is a collection of multidimensional data, called a *Data Cube*. Pre-computing of data cubes can dramatically reduce the response time of other queries. Recently many sequential algorithms have been proposed to generate data cubes efficiently, however as the size of data sets grows, there is a need for even more scalable algorithms. Currently, for large data sets, the cube queries may require hours or even days to run on standard sequential machines. *Parallel Computing* can provide two key ingredients for dealing with large data size: 1) increased computational power through multiple processors and 2) increased I/O bandwidth through multiple parallel disks.

The work presented in this thesis combines 1) the design of efficient parallel cube generation algorithms for the three basic types of data cubes: full cubes, partial cubes and iceberg cubes, with 2) careful system work associated with parallelism and external memory issues, and 3) extensive experiments and evaluation. The proposal algorithms are both external memory and parallel. They are designed for shared-nothing clusters, and use explicitly represented cost models which aid in performance tuning and portability. Our experiments show that the relative speedup of the algorithms is close to optimal/linear speedup for a wide range of input parameters, and the scalability is almost linear on large data sets. The proposed algorithms have been carefully implemented in our cgmOLAP prototype, which is to our knowledge the first fully functional parallel OLAP system able to build data cubes at a rate of more than half terabyte per hour.

## List of Abbreviations

| | |
|---|---|
| BESS | Bit-encoded Spare Storage |
| BI | Business Intelligence |
| BUC | Bottom-Up Cube |
| DSS | Decision Support Systems |
| ER | Entity-Relationship |
| ETL | Extract, Transform and Load |
| HOLAP | Hybrid OLAP |
| MDDB | Multidimensional Databases |
| MMST | Minimum Memory Spanning Tree |
| MOLAP | Multidimensional OLAP |
| MPI | Message Passing Interface |
| MSEMPP | Multiple Scan External Memory Pipeline Processing |
| MST | Minimum Spanning Tree |
| ODBC | Open Database Connectivity |
| OLAP | Online Analytical Processing |
| OLTP | Online Transaction Processing |
| PnP | Pipe 'n Prune |
| RDBMS | Relational Database Management System |
| ROLAP | Relational OLAP |
| SMP | Symmetric Multiprocessors |
| SQL | Structured Query Language |
| SSEMPP | Single Scan External Memory Pipeline Processing |

## Acknowledgements

Thanks to Dr. Bipin C. Desai, Dr. Qigang Gao, Dr. Michael Shepherd and Dr. Milios for your time and your patience.

Thanks to Frank and Todd for your help with my research.

Thanks to Mechelle for correcting grammar errors in this thesis.

And great thanks to Andrew for your patience and your commitment. Without your help, this thesis would never be finished.

# Chapter 1

## Introduction

The management and extraction of knowledge from data is central to the modern enterprise. More and more organizations rely on Decision Support Systems (DSS) to analyze their data and make better and faster decisions. For example, a retail store may analyze transaction records to decide promotion strategy. A health care organization may analyze patient treatment records to understand the effectiveness of treatment. While scientific institutions and enterprises may analyze huge amount of raw data, such as geographical data or biology data to discovery new knowledge. To analyze data efficiently, we need appropriate technologies, such as data warehousing. *Data Warehousing* is "a collection of decision support technologies, aimed at enabling the knowledge worker to make better and faster decisions [22]". In data warehousing, data from multiple operational data sources, such as operational databases, flat files and web logs, are transformed into "an object-oriented, integrated, time-varying, non-volatile collection of data that is used primarily in organizational decision making [46]". This collection of data is called a *Data Warehouse.*

| Date | Store | Product | Quantity |
|---|---|---|---|
| 2003 | Montreal | Computers | 2 |
| 2003 | Montreal | Sofeware | 2 |
| 2003 | Toronto | Computers | 2 |
| 2003 | Toronto | Software | 2 |
| 2004 | Montreal | Computers | 2 |
| 2004 | Montreal | Software | 2 |
| 2004 | Toronto | Computers | 2 |
| 2004 | Toronto | Software | 2 |

Figure 1.1: An Example Table with Eight Rows and Three Dimensions.

One of the core decision support tools in data warehousing is *Online Analytical Processing (OLAP).* It provides decision makers the ability to analyze data in a multi-dimensional way. For a given table, $R$, with $n$ rows and $d$ dimensions, a view/group-by

| Date | Store | Product | Quantity |
|---|---|---|---|
| 2003 | Montreal | Computers | 2 |
| 2003 | Montreal | Sofeware | 2 |
| 2003 | Toronto | Computers | 2 |
| 2003 | Toronto | Software | 2 |
| 2004 | Montreal | Computers | 2 |
| 2004 | Montreal | Software | 2 |
| 2004 | Toronto | Computers | 2 |
| 2004 | Toronto | Software | 2 |

| Date | Store | Quantity |
|---|---|---|
| 2003 | Montreal | 4 |
| 2003 | Toronto | 4 |
| 2004 | Montreal | 4 |
| 2004 | Toronto | 4 |

| Date | Product | Quantity |
|---|---|---|
| 2003 | Computers | 4 |
| 2003 | Software | 4 |
| 2004 | Computers | 4 |
| 2004 | Software | 4 |

| Store | Product | Quantity |
|---|---|---|
| Montreal | Computers | 4 |
| Toronto | Software | 4 |
| Montreal | Computers | 4 |
| Toronto | Software | 4 |

| Date | Quantity |
|---|---|
| 2003 | 8 |
| 2004 | 8 |

| Store | Quantity |
|---|---|
| Montreal | 8 |
| Toronto | 8 |

| Product | Quantity |
|---|---|
| Computers | 8 |
| Software | 8 |

| Quantity |
|---|
| 16 |

Figure 1.2: Eight Possible Views.

is generated by aggregating $R$ along a subset of dimensions. For example, a table with eight rows, three dimensions and one measure is showed in Figure 1.1. In this table, the three dimensions are "date", "store" and "product", and the last attribute "quantity" is a measure to be aggregated in the query. In OLAP systems, one of the most popular queries is to generate the set of views over all combinations of a set of dimensions from a table. This query allows the original data to be viewed from different perspectives. Figure 1.2 shows the eight possible views generated from the table in Figure 1.1. In 1997, Gray et al. proposed the *Cube* operator as an extension to the Structured Query Language (SQL) in order to support this query directly in relational databases [42]. The result of a cube query is a collection of multidimensional data or views, called a *Data Cube*. The *Full Data Cube* consists of all $2^d$ possible views, while a *Partial Data Cube* consists of a subset of the $2^d$ possible views. Another type of

cube is called an *Iceberg Cube*. In an iceberg cube each view consists of those rows, whose measures are greater than a minimal support parameter, which is predefined by the users. For example, this cube query,

```
SELECT date, store, product, SUM(quantity)
FROM salefact
GROUP BY CUBE(date, store, product),
```

generates a full data cube, and this one,

```
SELECT date, store, product, SUM(quantity)
FROM salefact
GROUP BY date, CUBE(store, product),
```

generates a partial data cube, which consists of the only four views like "date, store, product", "date, store", "date, product" and "date". The next cube query,

```
SELECT date, store, product, SUM(quantity)
FROM salefact
GROUP BY CUBE(date, store, product)
HAVING SUM(quantity) ≥ 5,
```

generates a partial data cube, where 5 is the minimal support value. The pre-computation of full/partial data cubes is often used to support the fast execution of subsequent queries [42] in which subsets of dimensions have been selected for aggregation. In this case, the queries are not answered from the original tables, but from the pre-computed views directly. The computation of iceberg data cubes can speed iceberg cube queries and some data mining tasks, such as association rules mining [56].

The efficient computation of data cubes is a challenging task, which has attracted a great deal of research [59, 17, 57, 65, 38, 20, 64]. A large number of sequential algorithms have been proposed for efficiently computing full data cubes [59, 17, 57, 65], partial data cubes [59, 38] and iceberg data cubes [20, 64]. However in the face of ever increasing data sizes, more efficient and scalable techniques are required.

## 1.1 Motivation - The Need for High Performance Computing

For large input tables, current sequential techniques often require days to generate large data cubes. Therefore a critical challenge in the OLAP area is how to meet

the scalability needs of large scale data warehouses, in which the amount of data is constantly growing. In the Winter Corporation's report [63], the largest three data warehouses exceed 20 terabyte in size. More importantly, it is expected that as of 2005, the storage requirements of more than 40% of production data warehouses will exceed one terabyte [31]. Inevitably, new solutions and new algorithms have to be proposed to meet the scalability needs of OLAP systems.

An OLAP system is considered scalable if it can maintain consistent response time by adding hardware, such as processors, proportionally to the workload added. For a single OLAP query, the increased workload comes from increased table sizes or increased number of dimensions in tables. Most of the response time for answering a query is composed of two parts: computation time and disk I/O time. One approach to meeting the scalability requirements is parallel computing. *Parallel Computing* is the use of multiple computer resources to solve a computational problem. The computer resources can be processors in a computer or computers in a network. Therefore, parallel computing can provide two key ingredients for dealing with the large data size: 1) increased computational power through multiple processors and 2) increased I/O bandwidth through multiple parallel disks.

In recent years, there has been a trend in parallel computing to move away from specialized super computing platforms to cheaper general purpose clusters made from single or multiple computing PCs or workstations. A *cluster* is a collection of interconnected nodes, working as an integrated computing resource. The nodes in a cluster can be single-processor machines or Symmetric Multiprocessors (SMPs). If neither memory nor disks are shared among the nodes in a cluster, it is called a shared-nothing cluster. A popular, low cost shared-nothing cluster is the *Beowulf Cluster* [1], which consists of standard PCs connected via a data switch. To exchange data among nodes in a shared-nothing cluster, we may use message passing libraries, such as the *Message Passing Interface (MPI)* [15], a standard interface for message passing implementations among processors in a cluster.

In this thesis, we design and implement parallel data cube generation algorithms for shared-nothing clusters. We analyze the efficiency of our algorithms using the CGM model of computation which has been shown to be a good predictor of parallel performance [37]. Our experimental platform is a Linux cluster, which consists of a

number of Linux PCs, connected via a high speed network. However, our implementations are platform independent and can be ported without any substantial change to other cluster platforms, such as IBM SP2 clusters or Windows NT clusters.

## 1.2 Contributions

This thesis makes contributions in the following areas: 1) the design of efficient parallel cube generation algorithms for the three basic types of data cubes: full cubes, partial cubes and iceberg cubes, 2) careful system work associated with parallelism and external memory issues in OLAP systems, and 3) extensive experiments and evaluation.

Specifically, the algorithms presented in this thesis are as follows:

1. **Parallel Generation of Full/Partial Data Cube:** We present a novel parallel full and partial data cube generation algorithm for shared-nothing clusters, described in Section 2.4. The algorithm accepts as input a data set with $n$ rows and $d$ dimensions, which is distributed evenly over the $p$ nodes in a cluster, and a list of views to be selected from the $2^d$ possible views. It computes and outputs the selected views to a set of $p$ disks, one per processor. The output data of each view is distributed evenly across the nodes/disks of a cluster. Our experiments show that the speedup of the algorithm is close to optimal/linear speedup. For example, the speedup for a data set with 8 million rows and 8 dimensions on 16 nodes is 14.3, 15.5 and 13.6 for the full cube, the partial cube with 75% selected views and the partial cube with 50% selected views, respectively. We have described these methods previously in [23, 25].

2. **External Memory and Parallel Generation of Full/Partial Data Cube:** We present an external memory adaptation of our basic parallel cube generation methods and a set of algorithmic enhancements that address the I/O challenges that arise with such increases in data size. We solve two critical problems: 1) how to compute data cubes efficiently in external memory and 2) how to reduce the high cost of disk I/O and network I/O. For the first problem, we introduce a shared prefix pipeline processing technique that speeds up pipeline processing by reducing the size of blocks that must be sorted. We also compare two approaches

to external memory pipeline processing. For the second problem, we design an adaptive data partitioning scheme, which uses a cost model to estimate the cost of computation, disk I/O and network I/O based on a given parallel machine and computes a "best data partitioning" to reduce the global cost. Our experiments show that the adaptive data partitioning method reduces the parallel running time by up to 40% and increases the speedup by up to 50%. For example, in experiments with a data set of 8 million rows and 8 dimensions on 16 nodes, linear speedup was achieved for both the full cube and the partial cubes. More importantly, with these enhancements our external memory method can handle huge data sets, such as a data set with 250 million rows and 8 dimensions. Our method generates the full cube from this data set in 70 minutes and outputs 7 billion rows or 200 gigabyte of data on 16 nodes. The time for reading input data from disks is about one second and the time for writing output data to disks is about 853 seconds or 14 minutes. Therefore the disk I/O time is about 20% of the total time for generating the full cube. For the partial cube with 25% selected views, it generates 752 million rows or 20 gigabyte of data in just 35 minutes. We have described these methods previously in [24].

3. **Parallel Generation of Iceberg Data Cube:** We present a novel PnP operator and "Pipe 'n Prune" (PnP) algorithm for the computation of iceberg cube queries. The novelty of our method is that it completely interleaves a top-down piping approach for data aggregation with bottom-up *Apriori* data pruning. A particular strength of PnP is that it is very efficient for sequential iceberg-cube queries, external memory iceberg-cube queries and parallel iceberg-cube queries on shared-nothing clusters. This makes PnP an interesting new alternative method, especially in applications where performance stability over a wide range of input parameters is important. For parallel iceberg cube generation, parallel PnP is the first parallel algorithm that shows good speedup on 16 or 32 node shared-nothing clusters. For example, the average speedup observed for a data set of 8 million rows and 11 dimensions, is 15.2 on 16 nodes. Since our parallel algorithm is based on the external memory PnP, it can handle large data sets easily. For example, for a data set with 250 million rows and 8 dimensions, it generates the iceberg cube in 25 minutes and outputs 28 million rows or 832

megabyte of data on 16 nodes. We have described these methods previously in [26].

Besides the design of parallel cube generation algorithms, our contributions also include significant system work. We have developed a software architecture that integrates all three basic cubing tasks and have realized this architecture in our cgmOLAP prototype. The task of generating full, partial and iceberg data cubes is not separate. In practice, the cube queries are handled by a single Structured Query Language (SQL) command with different options [12]. In the default case, it generates a full cube. If there is a required subset of views in the command, it generates a partial cube. Or if a "having" clause specifies a minimal support, it generates an iceberg cube. Our approach has been to integrate these three algorithms into a single system to fully support data cube queries in OLAP systems.

In this thesis, we also report on extensive experiments. Parallel algorithms are always more complicated than sequential algorithms because there are multiple processors which work on different partitions of data sets. The performance of parallel algorithms is effected by many factors, such as the size of data sets, the number of dimensions, the data skew, the number of processors, the speed of disk I/O and the speed of the network. Therefore, the performance of a systematic set of experiments is key to evaluating the performance of parallel algorithms. We provide extensive experiments with a variety of both synthetic and real data sets. These experiments show that in most cases our algorithms exhibit good speedup and scalability on a wide variety of data sets using up to 16 or 32 processors.

## 1.3 Organization

This thesis is organized as follows: Chapter 2 introduces basic concepts in data warehousing, OLAP and data cubes, as well as principal sequential data cube generation algorithms that have been described in the literature. Chapter 3 describes our in-memory parallel generation algorithm for full and partial data cube on shared-nothing clusters. In Chapter 4 we present the external memory and parallel data cube generation algorithms and an adaptive load-balancing technique. Chapter 5 describes our PnP algorithm for generating iceberg cubes. Chapter 6 introduces our cgmOLAP

system. Lastly, Chapter 7 presents a summary of contributions and suggesting future work.

# Chapter 2

# An Introduction to Data Warehousing

In this chapter, we introduce basic concepts of data warehousing, OLAP and data cubes. We also review a wide range of sequential data cube algorithms. Some algorithms which are keys to understanding our parallel methods are discussed in more detail. Finally, we introduce the shared-nothing cluster architecture, which is the experimental platform used in this thesis.

## 2.1 Data Warehousing

Today, more and more organizations rely on Decision Support Systems (DSS) to analyze their data to help make better and faster decisions. For example, a retail store may analyze transaction records to decide promotion strategy, a health care organization may analyze patient treatment records to understand the effect of treatment, and scientific institutions may analyze huge amount of raw data, such as geographical data or biology data to discovery new knowledge. To analyze data efficiently, we need appropriate technologies, such as data warehousing. *Data Warehousing* is "a collection of decision support technologies, aimed at enabling the knowledge worker to make better and faster decisions" [22]. A key difference between databases and data warehouses is that data warehouses are primarily used as data sources for analysis, instead of operational needs. *A data warehouse* is "an object-oriented, integrated, time-varying, non-volatile collection of data that is used primarily in organizational decision making" [46].

Recently a new term, Business Intelligence (BI), has become widely used in industry [3, 53, 16]. *Business Intelligence (BI)* is "a broad category of applications and technologies for gathering, storing, analyzing, and providing access to data to help enterprise users make better business decisions" [2]. BI has three advantages for business data analysis [53]. First, BI focuses on both technologies and applications while data warehousing focuses on technologies only. Second, BI is involved in all phases of

the data processing chain, from gathering and storing to analyzing and access, while data warehousing focuses on building data warehouses mainly. Third, BI can access all possible business information sources, not just data in data warehousing.

However data warehousing can be used in many more areas for data analysis, such as health care and science. It is not only for enterprise users to analyze business data like BI. Moreover, most of the decision support technologies used in BI are from data warehousing. Therefore, in this thesis we focus on data warehousing, but the algorithms discussed in this thesis are applicable for both data warehousing and BI.

### 2.1.1 The Architecture of Data Warehousing

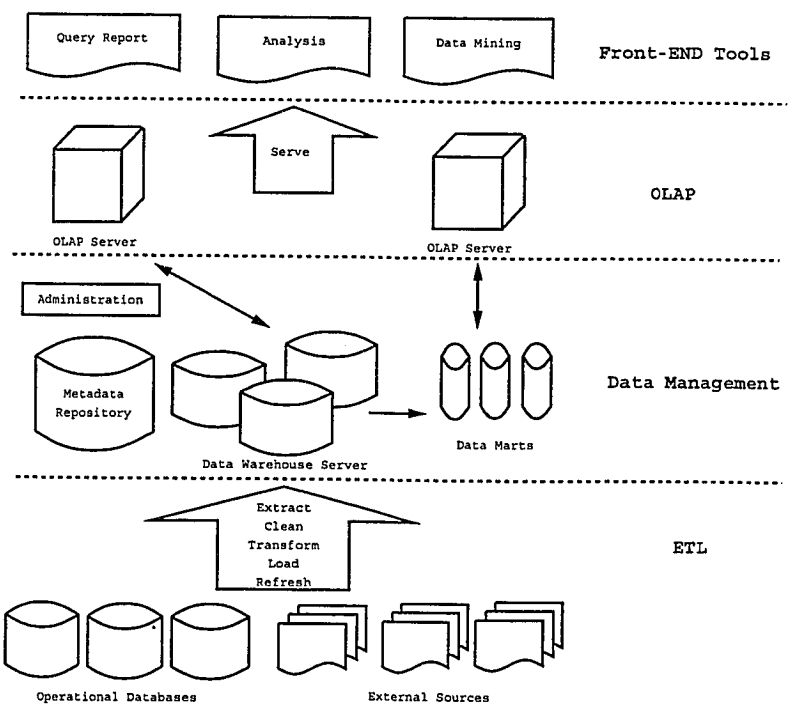



Figure 2.1: Data Warehousing Architecture. [22]

Figure 2.1 illustrates the architecture of a typical data warehousing system. In this architecture, there are four layers: Extract/Transform/Load tools, data management, decision support tools, and front-end tools. In the following, we will give more details for these layers.

### 2.1.2 ETL

The data in a data warehousing system can come from almost all possible information sources, such as operational databases, flat files, web pages and even the output of applications. Useful data is extracted from these data sources, transformed into target formats and loaded into the data warehouse. These processes are called *ETL*, the abbreviation for *E*xtract, *T*ransform and *L*oad.

The first step in ETL is to extract data from all kinds of sources. We may use some standard interfaces, such as Open Database Connectivity (ODBC) [11], or tools provided by database vendors, to extract online data from operational databases. For file sources, we may use FTP or HTTP tools to download files from remote hosts. For particular application sources we may use special interfaces.

The second step is to clean and transform inconsistent source data into consistent data formats. These include field lengths, field descriptions, value assignments, missing entries and integrity constrains [22]. We may also need to normalize data and convert continuous or categorical data into discrete data. For example, we may define a set of age arranges: 1-12, 13-17, 18-22, 23-35, ... , and then use integers to represent the arranges: 1, 2, 3, 4 and so on. Another example is using "1" to represent "male" and using "2" to represent "female", instead of using text strings. For some complicate fields, we need to parse them to get more delicate sub-fields. For example, an address may consist of the street number, the street name, the apartment number, the city name, the province or state name, the post code and the country name. An IP address may consist of the network address and the network mask.

The last step in the ETL process is to load the processed data into data warehouses. Before loading, we may need to sort, aggregate the data, and pre-compute some materialized views in order to improve the performance of data warehouses [22]. It may take a long time to load massive data sets into a large scale data warehouse. To decrease the load time, we may use parallel loading, multiple storage disks and data partitioning.

ETL can be done only once or be executed periodically. If we need to execute ETL more than once, we may refresh data warehouses instead of re-loading all the data again. The updated data could include the basic tables, and materialized views computed from the basic tables. Replication servers can be used to refresh data

warehouses. There are two basic replication technologies [22]. One is *data shipping* [22]. It uses the snapshots of tables in the remote source databases. The other is *transaction shipping* [22]. It uses the transaction logs in the remote source databases.

### 2.1.3 Data Management

The data management in Figure 2.1 consists of the data warehouse server, data marts, metadata repository and data administration. The core of the data management is the data warehouse, which stores all the data for analysis in a data warehousing system. The data marts are small data warehouses, which store the departmental data only. The data marts copy data from the data warehouse.

#### RDBMS and MDDB

Most of data in a data warehousing system is organized in a multidimensional way. Two basic types of database systems can be used to store multidimensional data sets. They are Relational Database Management System (RDBMS) and Multidimensional Databases (MDDB) [54]. RDBMS uses rational tables to store multidimensional data sets. The advantage of RDBMS is that it can handle huge amount of data sets, and is supported by most database vendors. The disadvantage of RDBMS is that its performance is not as good as MDDB.

MDDB uses multidimensional arrays to store multidimensional data sets directly and takes less time to access arrays than tables in RDBMS. The disadvantage of MDDB is that it could need huge memory to store the multidimensional arrays even for small data sets due to the empty cells in arrays. When the memory is not enough, MDDB has to swap the data between memory and disks, which degrades the performance of MDDB. Therefore MDDB is not suitable for large scale data sets. Table 2.1 illustrates the tradeoffs of RDBMS and MDDB.

Considering the tradeoffs, we may use both RDBMS and MDDB in a data warehousing system at the same time. For example, we can use RDBMS to store the data warehouse and big data marts, while use MDDB to store small data marts, which support OLAP applications.

| | RDBMS | MDDB |
|---|---|---|
| **Size** | Up to terabyte | Up to 100 gigabyte |
| **Source Data** | Handle volatile source data well | Take long time to update |
| **Aggregate** | Compute aggregate slow | Compute aggregate fast on small data |
| **Investment** | Reuse the previous tools and skills | Need new tools and skills |
| **Management** | Easy to manage complex systems | Hard to manage complex systems |

Table 2.1: The Tradeoffs between RDBMS and MDDB [54]

### Data Warehouse Modelling

Data warehouse modelling is an important step before ETL. In this step, a data warehouse architect determines what data is to be stored in the data warehouse, how the data is to be stored in the data warehouse and which schema is to be used in the data warehouse. This information is called *metadata*. Metadata is the data about the data, and typically is shared by all other components in a data warehousing system.

A popular type of data warehouse modelling is called dimensional modelling [50]. In the dimensional modelling, there are two basic objects: facts and dimensions. *Facts* are numeric measures and can be aggregated by a function, such as SUM. An example of facts is the sale quantity of products or the sale dollar amount of products. *Dimensions* are used to determine the granularity of facts. In other words, the intersection of a set of dimensions determines a measurement of facts. For example, the dimensions "Date", "Product" and "Store" can determine the sale quantity of a certain product on a certain day at a certain store. A dimension may consists of some *levels*, which can determine the different granularity of facts. For example, the levels in a "Date" dimension can be "Year", "Month", "Day" and "Weekday". The relationship of a set of levels is called a *hierarchy*. The common relationship between two levels in a hierarchy is one-to-many, which means the granularity becomes finer. Two possible hierarchies in a "Date" dimension can be "Year-Month-Day" and "Weekday-Day".

MDDB Data warehouses use multidimensional arrays to store facts directly. In a multidimensional array, each dimension consists of a set of unique values as the indexes for the dimension. The intersection of the indexes from all the dimensions

determines a storage cell in the array. The cell stores the measures responding to the dimension indexes. The key challenge for MDDBs is how to store such multidimensional arrays efficiently, since the most proportion of cells may be empty, and it adds too much overhead to retrieval.

### Schemas in RDBMS Data Warehouses

The most common approach of modelling used in traditional RDBMS is Entity-Relationship (ER) modelling [19]. In ER modelling, there are two basic objects: entities and relationships between entities. To store multidimensional data in a RDBMS, we need to map the dimensional model to a ER model. The approach is to treat dimensions as entities and to treat facts as many-to-many relationships.

To convert entities and relationships into tables in RDBMS, we need to determine what schema is used. *A schema* is "a collection of database objects, including tables, views, indexes and synonyms" [4]. Two fundamental schemas in RDBMS data warehouses are *Star Schema* and *Snow-flake Schema*. Their names are derived from the shapes of their ER diagrams.

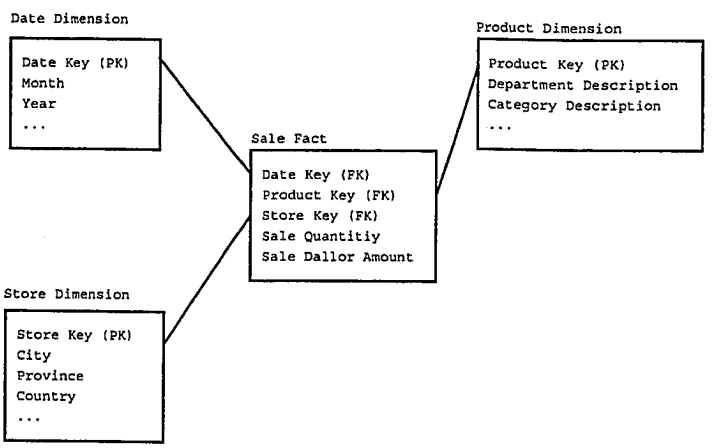



Figure 2.2: A Star Schema.

Figure 2.2 illustrates an example of a star schema. In this example, there are three dimension tables and a fact table, where "PK" denotes primary keys and "FK" denotes foreign keys. In a star schema, each dimension has only one table and dimension tables are connected by exactly one fact table. Since there is only one table

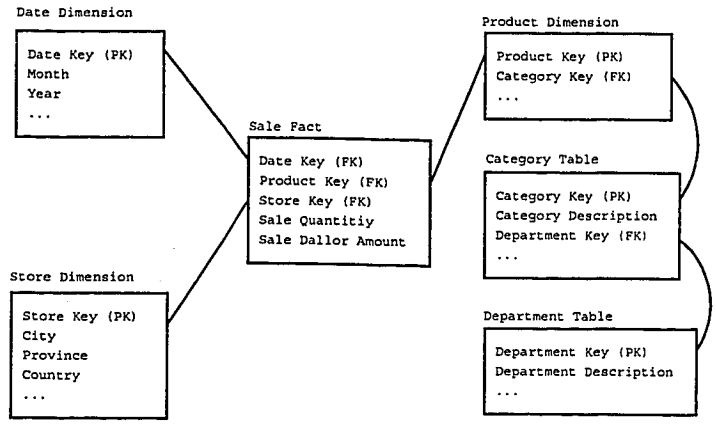



Figure 2.3: A Snow-Flake Schema.

| | The Star Schema | The Snow-flake Schema |
|---|---|---|
| **Complexity** | Simple structure | Complex structure |
| **Joins** | Less joins in quiries | More joins in quiries |
| **Disk Space** | More disk space | Less disk space |
| **Browsing** | Easy to browse in a dimension | Hard to browse in a dimension |
| **Bitmap Indexes** | Easy to use bitmap indexes | Prohibit the use of bitmap indexes |

Table 2.2: The Star Schema V.S. The Snow-flake Schema [50]

for each dimension, the star schema may consist of some redundant data. For example, in the product dimension table, many rows may include the same descriptions of departments and categories.

We may normalize the dimension tables by using more than one table to represent a dimension to extend a star schema. In this case, it becomes a snow-flake schema, as illustrated in Figure 2.3. In a snow-flake schema, some hierarchies in dimension tables are normalized using several tables. For example, in Figure 2.3 the hierarchy "department-category-product" is normalized using three tables.

A comparison of star schema and snow-flake schema is given in Table 2.2. The main advantage of the snow-flake schema is that it can save disk space. However the space saved in dimension tables may be very small relative to the size of the fact table. Therefore, some experts [50] and database inventors [4] recommend the star schema for use in RDBMS data warehouses.

In practice, there may be several facts tables sharing one or more dimension tables in a data warehouse. We may organize these tables as the *galaxy schema* or the *fact constellation*, since the shape of the ER diagram is like a galaxy. We may view a galaxy schema as a combination of multiple star schemas or snow schemas. An example of a galaxy schema is illustrated in Figure 2.4. In this figure, there are two fact tables: the sale fact table and the repository fact table. They share the date dimension and the product dimension.

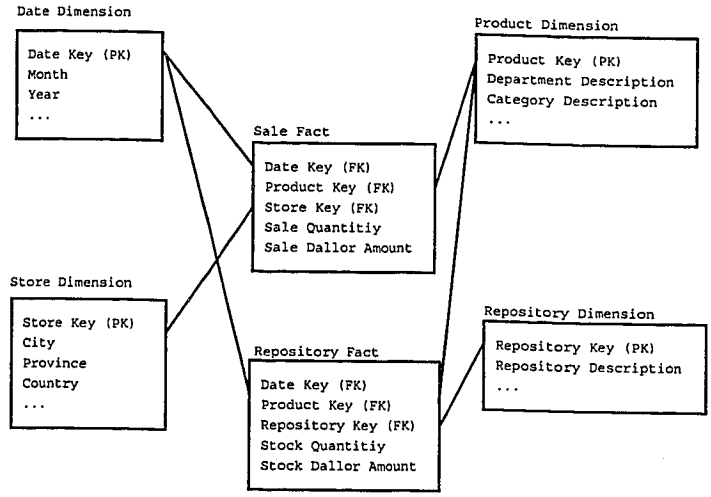



Figure 2.4: The Galaxy Schema.

### 2.1.4 Decision Support Tools

The three main types of decision support tools used in data warehousing systems are as follows:

**Query and Reporting** Query and reporting use Structured Query Language (SQL) to execute ad-hoc queries in relational data warehouses, and generate formatted reports from query results. They are easy to use and suitable for almost all kinds of users.

**OLAP** *OLAP* is "category of software technology that enables analysts, managers and executives to gain insight into data through fast, consistent, interactive access to a wide variety of possible views of information that has been transformed

from raw data to reflect the real dimensionality of the enterprize as understood by the user" [10]. It is suitable for high level users in organizations.

**Statistical Analysis and Data Mining** Statistical analysis and data mining use mathematical, financial and statistical methods to analyze data. Data mining focuses on using data analysis and discovery algorithms to find out patterns from data [35]. They are tools and techniques suitable for experts only, such as data analysts.

Based on the above three decision support tools, a data warehousing system can provide front-end tools, which are the top layer of Figure 2.1. These tools can present the results of queries in user friendly format, such as reports and forms, or visualize the multidimensional data of OLAP. They also provide necessary interfaces between users and data warehousing systems.

Among the three decision support tools, query and reporting are the easiest ones to use, but they can not present data in a multidimensional way. Statistical analysis and data mining require users to have professional knowledge. Only OLAP can provide powerful multidimensional analysis on data sets for most decision makers. This makes OLAP an essential component in a data warehousing system.

## 2.2 OLAP and Data Cubes

One of the most powerful and prominent technologies in a data warehousing system is OLAP. By exploiting multidimensional views of the underlying data warehouses, the OLAP server allows users to *drill-down* or *roll-up* on hierarchies, *slice* and *dice* on particular dimensions, or perform various statistical operations such as ranking and forecasting. To support this functionality, OLAP relies heavily upon the data cube. In the remainder of this section, we introduce basic concepts of OLAP and data cubes.

### 2.2.1 OLTP and OLAP

*Online Transaction Processing (OLTP)* systems [60] record and constantly update day-to-day operations, such as store transactions in a superstore, account transactions in a bank or calling information in a telecom company. The goal of OLTP is to process

as many transactions as possible in a unit time. The data in OLTP is the source data for a data warehousing system. After we execute ETL processing on this online data, it is stored in data warehouses for analysis by OLAP in a multidimensional way.

Table 2.3 lists the differences between OLTP and OLAP systems.

| | OLTP | OLAP |
|---|---|---|
| **Typical User** | Regular employees | Managers and analysts |
| **Usage of System** | Day-to-day operation | Business analysis |
| **User Interface** | Pre-determined | Ad-hoc |
| **Data** | Current data | Historical data |
| **Data Characteristics** | Atomic | Summarized |
| **Work Characteristics** | Read/Write | Read (except off-line updates) |
| **Unit of work** | Transaction | Query |
| **Processing** | Process-oriented | Subject-oriented |
| **Updates** | One record at a time | Several records at a time |

Table 2.3: A Comparison of OLTP and OLAP Systems [62]

### 2.2.2 OLAP Categories

The data in OLAP systems is organized in multidimensional data sets, called *Data Cubes*. An example of a data cube with three dimensions is illustrated in Figure 2.5.

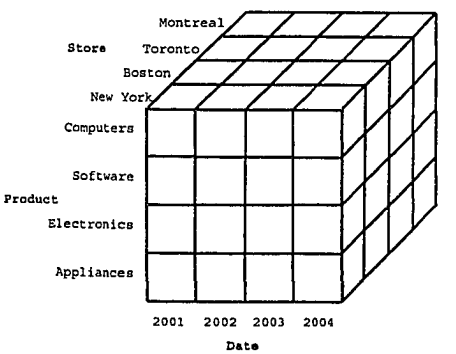



Figure 2.5: A Data Cube with Three Dimensions.

Data cubes can be stored as relational tables, such as in RDBMS. The OLAP based on relational tables is called a *Relational OLAP (ROLAP)*. Data cubes can

also be stored as multidimensional arrays, such in MDDB. The OLAP based on multidimensional arrays is called a *Multidimensional OLAP (MOLAP)*. If data cubes in a OLAP system are stored as both relational tables and multidimensional arrays, this kind of OLAP is called a *Hybrid OLAP (HOLAP)*.

ROLAP stores data cubes in relational tables. In most cases, ROLAP accesses fact tables to answer queries. This makes the query response of ROLAP slower than MOLAP or HOLAP. The pre-computed data cubes can be used to dramatically speed up the query response of ROLAP. In this case, queries are not answered from fact tables but from the pre-computed views directly. The advantage of ROLAP is the ability to handle large data sets that are infrequently queried, such as terabyte historical data. The ROLAP products include ROLAP Informix's MetaCube, ROLAP Option for the Informix Dynamic Server [7] and MicroStrategy 7i OLAP [9].

MOLAP stores data cubes in multidimensional arrays. The main problem in MOLAP is how to operate on sparse data cubes, where many cells are empty. Some special technologies can be used to compress data cubes, such as Bit-encoded Spare Storage (BESS) [58]. Generally MOLAP can provide the most rapid query response in three types of OLAP on small data sets. The MOLAP products include Essbase OLAP Server [6] and Oracle 9i OLAP [12].

Table 2.4 [62] lists the main differences between ROLAP and MOLAP .

| | **ROLAP** | **MOLAP** |
|---|---|---|
| **Storage and Access** | Tables/tuples | Proprietary arrays |
| | SQL access language | Lack of a standard language |
| | Third party tools | Sparse data compression |
| **Usage** | Variable performance | Good performance |
| | Relational engine | Multidimensional engine |
| **Database Size** | Up to terabyte | Up to gigabyte |
| | Large space for indexes | 2% index space |
| | Easy updating | Difficult updating |

Table 2.4: A Comparison of ROLAP and MOLAP

HOLAP combines attributes of both MOLAP and ROLAP. It stores aggregate data in multidimensional arrays and basic fact tables in RDBMS. For queries only related to aggregate data, the response time is almost equal to that in MOLAP. Queries that access basic fact tables from RDBMS will be slow. The problem of

HOLAP is that it is more complicated to implement and administer than ROLAP and MOLAP. The HOLAP products include Microsoft SQL Server OLAP Services [8] and Pilot Decision Support Suite [14].

### 2.2.3 The Cube Operator

In OLAP systems, one of popular queries is to generate a set of group-bys over all combinations of a set of dimensions from a fact table. The query allows us to view the original data from different perspectives. In order to execute the query in one Structured Query Language (SQL) command, Gray et al. proposed the *Cube* operator to extend the SQL standard in the paper [42]. A cube operator query is similar to $2^d$ group-by queries, where $d$ is the number of dimensions in the query. For example, a cube query based on the fact table in Figure 2.5 is: "SELECT date, store, product, SUM(salequantity) FROM salefact GROUP BY CUBE(date, store, product)." The result of this cube query is equal to those of the eight group-by queries listed in Table 2.5.

| | | | |
|---|---|---|---|
| SELECT | date, store, product, | SUM(salequantity) FROM salefact GROUP BY | date, store, product |
| | date, store, | | date, store |
| | date, product, | | date, product |
| | store, product, | | store, product |
| | date, | | date |
| | store, | | store |
| | product, | | product |
| | NULL | | NULL |

Table 2.5: The Eight Group-by Queries

Each of the group-bys in a cube query is called a *view* or a *cuboid*. The view including all the dimensions is called *core/base view* or *core/base cuboid*. For example, the view of "date, store, product" in the above example is the core/base view. The set of the views in a cube query is called a *Data Cube*. Therefore, to answer a cube query is to generate a data cube from the input table.

We may use a special value "ALL" to represent any value in a dimension. Then the data cube from a cube query can be stored in only one table in ROLAP. For example, for the fact table "salefact" shown in Table 2.6, the resulting data cube is

listed in Table 2.7. It can also be stored as a multidimensional array in MOLAP, illustrated in Figure 2.6.

The results of cube queries can be stored in data warehouses. These data sets are called *pre-computed data cubes*. The views in the pre-computed data cube are called *pre-computed views*, *materialized views* or *aggregate tables*. Using pre-computed data cubes, data warehouses and OLAP systems can answer queries much faster than computing them from fact tables.

| Date | Store | Product | Sale Quantity |
|---|---|---|---|
| 2003 | Montreal | Computers | 2 |
| 2003 | Montreal | Software | 2 |
| 2003 | Toronto | Computers | 2 |
| 2003 | Toronto | Software | 2 |
| 2004 | Montreal | Computers | 2 |
| 2004 | Montreal | Software | 2 |
| 2004 | Toronto | Computers | 2 |
| 2004 | Toronto | Software | 2 |

Table 2.6: The Fact Table "salefact"

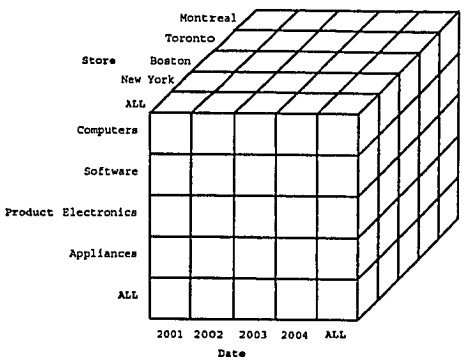



Figure 2.6: The Result of the Cube Query in MOLAP.

### 2.2.4 OLAP Operations on Data Cubes

Most OLAP systems provide five major operations on data cubes to analyze data in a multidimensional way [48]. We use a set of examples to illustrate these operations. In these examples, the original cube, in Figure 2.7, consists of three dimensions:

| Date | Store | Product | Sale Quantity |
|---|---|---|---|
| 2003 | Montreal | Computers | 2 |
| 2003 | Montreal | Software | 2 |
| 2003 | Toronto | Computers | 2 |
| 2003 | Toronto | Software | 2 |
| 2004 | Montreal | Computers | 2 |
| 2004 | Montreal | Software | 2 |
| 2004 | Toronto | Computers | 2 |
| 2004 | Toronto | Software | 2 |
| 2003 | Montreal | ALL | 4 |
| 2003 | Toronto | ALL | 4 |
| 2004 | Montreal | ALL | 4 |
| 2004 | Toronto | ALL | 4 |
| 2003 | ALL | Computers | 4 |
| 2003 | ALL | Software | 4 |
| 2004 | ALL | Computers | 4 |
| 2004 | ALL | Software | 4 |
| ALL | Montreal | Computers | 4 |
| ALL | Montreal | Software | 4 |
| ALL | Toronto | Computers | 4 |
| ALL | Toronto | Software | 4 |
| 2003 | ALL | ALL | 8 |
| 2004 | ALL | ALL | 8 |
| ALL | Montreal | ALL | 8 |
| ALL | Toronto | ALL | 8 |
| ALL | ALL | Computers | 8 |
| ALL | ALL | Software | 8 |
| ALL | ALL | ALL | 16 |

Table 2.7: The Result of the Cube Query in ROLAP

"store", "product" and "date". In the "store" dimension, the hierarchy is "country-city", where "country" is the high level or coarser level, and "city" is the low level or finer level. In the "product" dimension, the hierarchy is "department-category", where "department" is the high level and "category" is the low level.

The five operations are as follows:

**Roll-up:** It merges values along a particular dimension based on hierarchical relation to a coarser level of granularity. For example, in Figure 2.8, the level of "stores" is changed from "city" to "country".

**Drill-down:** It splits values along a particular dimension based on hierarchical relation to a finer level of granularity. For example, in Figure 2.9, the level of "products" is changed from "department" to "category". Drill-down is the reverse operator for roll-up.

**Slice:** It extracts a single value of a dimension from the original data cube. For example, in Figure 2.10, the value "Electronics" is extracted from the dimension of "Product".

**Dice:** It extracts a subset of values of some dimensions from the original data cube to generate a small data cube. In Figure 2.11, we extract "Montreal" and "Toronto" from the dimension of "stores" and extract "2003" and "2004" from the dimension of "dates" to generate a small data cube.

**Pivot** It rotates the data cube to change the dimensional orientation in order to visualize cubes in more natural or intuitive ways. Figure 2.12 is an example of the pivot operation.

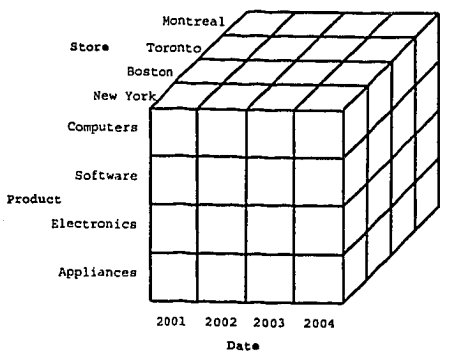



Figure 2.7: The Original Cube.

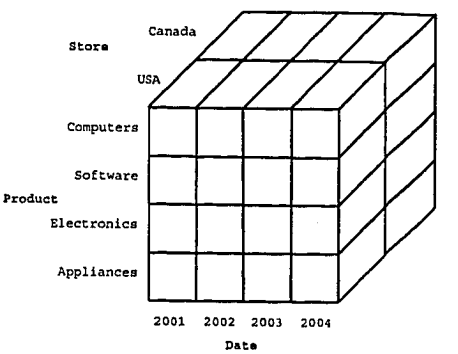



Figure 2.8: OLAP Roll-up

### 2.2.5 Aggregate Functions

In the previous examples, we use SUM as the aggregate function. However, in the data cube generation there are many other aggregate functions, such as COUNT, MIN, MAX, AVG and customized functions. We may classify aggregate functions into three categories [42]: distributive functions, algebraic functions and holistic functions.

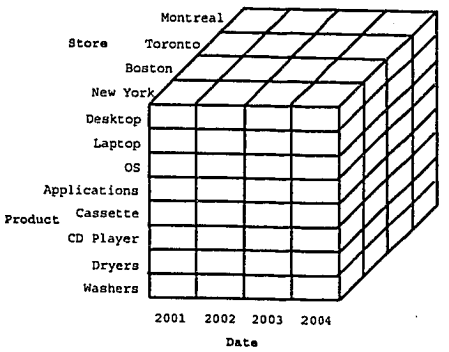



Figure 2.9: OLAP Drill-down.

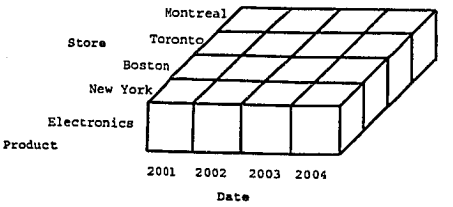



Figure 2.10: OLAP Slice.

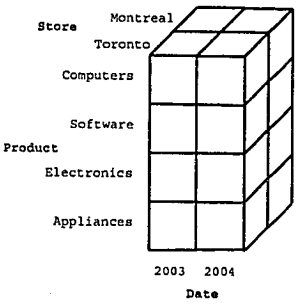



Figure 2.11: OLAP Dice.

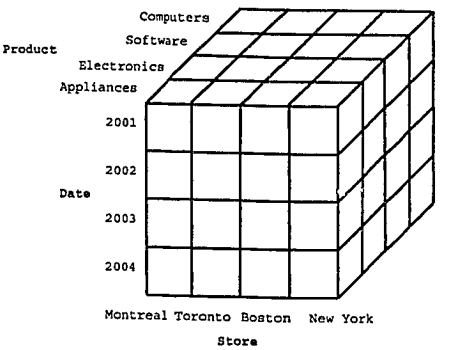



Figure 2.12: OLAP Pivot.

An aggregate function, $agg()$, is *distributive*, if a function $f()$ exists and $agg(S \bigcup S') = f(agg(S), agg(S'))$, where $S$ and $S'$ are data sets. For example, SUM is distributive because $SUM(S \bigcup S') = SUM(SUM(S), SUM(S'))$. COUNT is distributive because $COUNT(S \bigcup S') = SUM(COUNT(S), COUNT(S'))$. MIN is distributive because $MIN(S \bigcup S') = MIN(MIN(S), MIN(S'))$. And MAX is distributive because $MAX(S \bigcup S') = MAX(MAX(S), MAX(S'))$.

An aggregate function, $agg()$, is *algebraic*, if a M-tuple valued function $f()$ and a function $g()$ exist and $agg(S \bigcup S') = g(f(S), f(S'))$, where $S$ and $S'$ are data sets. "The key to algebraic functions is that a fixed size result (a M-tuple) can summarize the sub-aggregate." [42] For example, AVG is algebraic because $f()$ calculate the sum and count of $S$ and $S'$ and the $g()$ adds the sum and divides by the total count to get the average of two data sets. Here, $f()$ is a two-tuple function: the sum and the count.

An aggregate function, $agg()$, is *holistic*, if there is no constant bound on the size of the storage needed to describe a sub-aggregate. MEDIAN and RANK are common examples of holistic functions.

The data cubes using distributive and algebraic aggregate functions are easy to compute by using other views instead of raw data. For the data cubes using holistic aggregate functions, there are no more efficient algorithms than regular group-by queries [42]. In this thesis, we focus on the distributive and algebraic aggregate functions, and always use SUM in our examples and experiments.

## 2.3 Sequential Data Cube Algorithms

The computation of data cubes is critical to improve the response time in OLAP systems and can be instrumental in accelerating other applications in a large scale data warehousing system, such as ad-hoc queries and data mining [45]. Even in the processing od *Extract, Transform and Load (ETL)*, we also need to pre-compute some data cubes (materialized views) before we load them into data warehouses [50]. For most of data cube computation problems, we may design efficient algorithms to compute a set of views together, rather than compute individual views using separate group-by queries. In this section, we will introduce the most popular data cube algorithms.

### 2.3.1 Three Types of Data Cubes

For a given raw data set, $R$, with $n$ rows and $c$ columns. The $c$ columns include $d$ dimensions and $m$ measures, a view is computed by an aggregate of $R$ on $m$ measures along a set of dimensions. The total number of possible views is $2^d$. For each dimension $D_i$, $1 \leq i \leq d$, there are $|D_i|$ unique values. $|D_i|$ is called the *cardinality* of $D_i$.

We may use English letters as the identifiers of dimensions, such as $A$ for the first dimension $D_1$, $B$ for the second dimension $D_2$ and so on. Then we may use letter strings as the identifiers of views, such as $AB$ for the view grouped by $D_1$ and $D_2$. The order of letters in the string represents the order of group-by dimensions in the view. We use *raw* to represent the raw data set. The dimension order in *raw* can be in any order, and we use *all* to represent the view without group-by dimensions.

We may put all $2^d$ view identifiers in a graph and use edges to connect them. This graph is called the *Lattice*, where an edge between two view identifiers indicates that the short view can be computed from the other by aggregating along one dimension. Figure 2.13 shows a lattice with four dimensions $A$, $B$, $C$, and $D$. In a lattice, we put the same length views on the same level, put the long views on the upper level and put the short views on the lower level. The top level is Level $d$, and the bottom level is Level 0. Therefore a lattice consists of $d + 1$ levels, and the length of each view on Level $k$ is $k$.

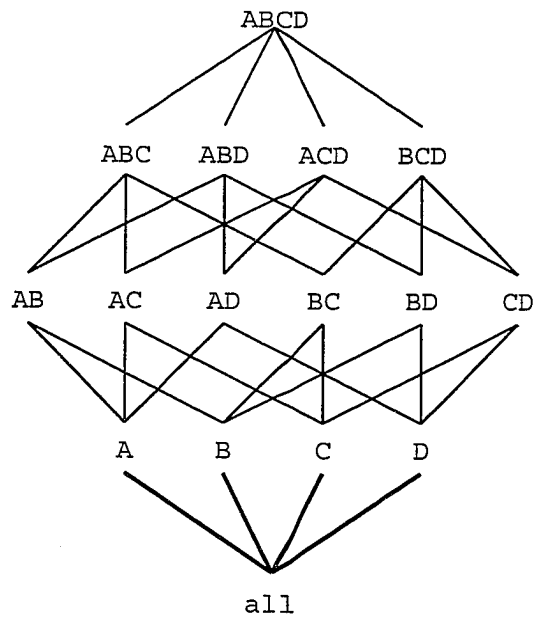



Figure 2.13: A Lattice with Four Dimensions.

We may compute the entire data cube, which is the set of all $2^d$ views. This data cube is called a *full data cube*. A full data cube may be too huge to be stored on disks, or OLAP users may require to compute a part of a full cube. Therefore we may select a subset of $2^d$ views to compute. The subset of $2^d$ views is called a *partial data cube*. Another type of data cube is called an *iceberg data cube*. In an iceberg data cube, we still compute all the $2^d$ views, but for each view, we only output the rows whose measures are above specific thresholds. In a iceberg cube query, there is a "having" clause to indicate the thresholds. The following query is an example of

an iceberg cube query: "SELECT A, B, C, D, SUM(measure) FROM R GROUP BY CUBE(A, B, C, D) HAVING SUM(measure) $\geq$ 10."

There are a number of algorithms for computing three types of data cubes: the full data cube [59, 17, 57, 65], the partial data cube [59, 38] and the iceberg data cube [20]. These algorithms can be categorized into ROLAP algorithms [59, 17, 57, 38, 20] and MOLAP algorithms [65] according to OLAP implementation, or be categorized into sort based algorithms [59, 17, 57, 38, 20], hash based algorithms [59] and array based algorithms [65] according to methods used.

Most algorithms begin with a lattice, and then tries to search for the best path in the lattice to compute all the views. If the search direction is from the upper level to the lower level, this method is called a *Top-down* method, such as PipeSort [59]. If the search direction is from the lower level to the upper level, this method is called a *Bottom-up* method, such as Bottom-up CUBE [20].

In the rest of this section, we introduce the most popular algorithms for each category of data cube algorithms, and give more details on PipeSort and PipeSort for partial data cubes [38], since they are the sequential base for our parallel algorithms.

### 2.3.2 PipeSort

PipeSort is proposed in the paper [59] to compute a full data cube. The PipeSort algorithm includes two steps: generating a plan and executing the plan. In the first step, PipeSort searches the lattice for a spanning tree with the minimal cost using a top-down method. In the second step, PipeSort converts the spanning tree into a set of pipelines, and execute them to generate views.

#### Generate a Plan

In PipeSort, each view is computed by sorting or scanning one of its parent views. If one view shares a prefix with one of its parent views, we may scan the sorted parent view to compute this view without sorting. This method is called *share-sort*. If there is no shared prefix between a view and any of its parent views, we have to sort and scan a parent view to compute this view. For example, we may compute $ABC$ by scanning $ABCD$, but we have to compute $BCD$ by sorting $ABCD$. Since sorting spends much more time than scanning, Pipesort tries to use share-sort as much as

possible.

On the other hand, a view always has several parent views, which have different sizes. The smallest parent view is always the better choice than other views for computing a view, because we may sort or scan less data. This method is called *smallest-parent*. For example, we may use $ABCD$ or $ABD$ to compute $AB$. Since $ABD$ is smaller than $ABCD$ in size, we should choose $ABD$.

In practice the methods of share-sort and smallest-parent often conflict with each other. For example, if we use the method of share-sort, we may compute $AB$ from $ABC$, but if we use the method of smallest-parent, we may choose $BDA$ because $BDA$ might be smaller than $ABC$. Therefore, PipeSort tries to combine both methods in a cost model and searches for a spanning tree with the minimal global cost.

**Algorithm 1** Generate-Plan

**Input:** A $d$ dimension lattice with sort costs and scan costs on each edge.

**Output:** A PipeSort spanning tree with the minimal global cost.

1: **for** $k = 0$ to $d - 1$ **do**
2: Duplicate nodes on the $k + 1$ level and add new edges between duplicates and the nodes on the $k$ level.
3: Assign sort costs to the original edges, and scan costs to the new edges.
4: Use the weighted bipartite matching algorithm to find out the best paths between the two levels.
5: For each node on the $k + 1$ level, fix the order as the node connected by the scan edge on the $k$ level.
6: **end for**

Algorithm 1 outlines steps to generate a spanning tree in PipeSort. The input of Algorithm 1 is a $d$ dimension lattice with sort costs and scan costs on each edge. We will introduce the cost estimation in the later part of this section. PipeSort searches for the spanning tree level by level. In order to find out the best paths between two levels, PipeSort duplicates the nodes on the upper level, and adds new edges between duplicates and the nodes on the lower level. Afterwards, PipeSort assigns the sort costs to the original edges and scan costs to the new edges. Figure 2.14 illustrates the two levels with new nodes and new edges. The solid edges represent sorts, and the dash edges represent scans. The costs of sort and scan are shown above the nodes of

the upper level. Then Pipesort uses the weighted bipartite matching algorithm [59] to find out the best paths between the two levels. The best paths have the minimal total costs among all possible paths. PipeSort also fixes the orders of the nodes of the upper level based on the nodes of the lower level. Figure 2.15 illustrate the best paths generated from Figure 2.14. After the best paths for all levels are found, the lattice are transformed to a spanning tree, illustrated in Figure 2.16.

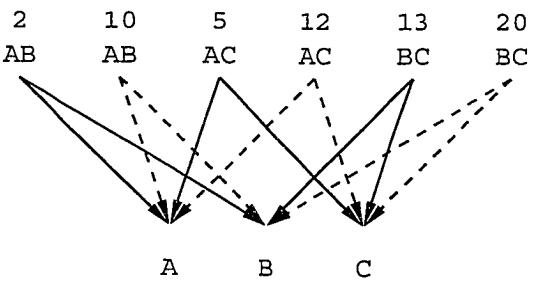



Figure 2.14: Transformed Search Lattice. [59]

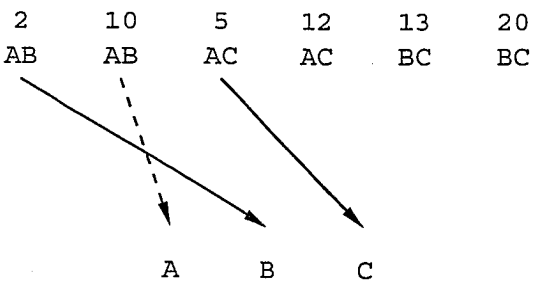



Figure 2.15: The Best Paths with the Minimum Total Cost. [59]

### Cost Estimation

In the plan generation step, we need to estimate scan costs and sort costs on each edge. Suppose the number of rows is $n$ and the number of dimensions is $m$ for a view $V$. The cost of generating $V$ by scanning its parent view is $n(m+1)$ [32], and the cost of generating $V$ by sorting its parent view is $n \log n + n(k/2)$ [32]. In the scan cost, $n(m+1)$, $nm$ is the cost for checking a row of $V$, and $n$ is the cost for aggregating and

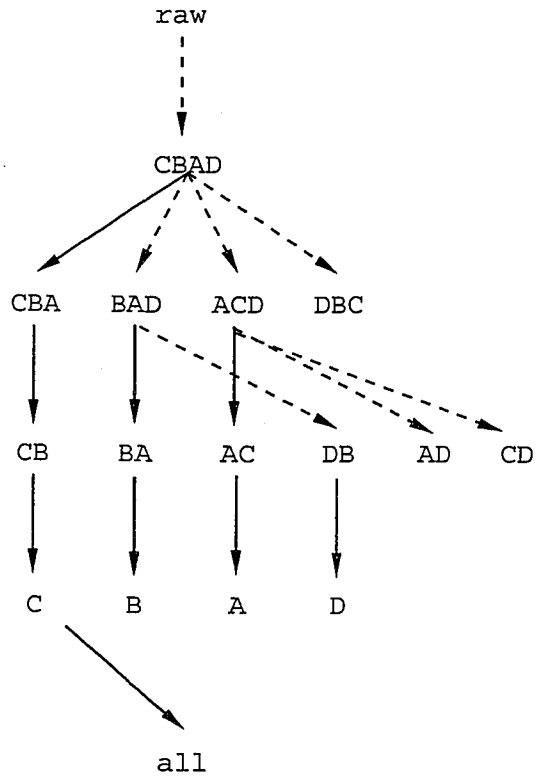



Figure 2.16: A PipeSort Spanning Tree [59]

moving a row. In the sort cost, $n \log n + n(k/2)$, $n \log n$ is the cost of the quicksort algorithm. And $n(k/2)$ is the cost of finding the position of the first dimension that differs in the sorted parent view. The position can be any value between 1 and $k$, so we choose the average value $k/2$ for simplicity. It is easy to get the number of dimensions, $m$, whose value is the level of the view. However, we cannot find the number of the rows, $n$, before we compute the view. Therefore we estimate the number of rows by using a number of algorithms, such as cardinality-based estimation [59], sample scaling [44] and probabilistic methods [49].

In the PipeSort paper [59], the authors used a cardinality-based method to estimate the sizes of views. For a view $V$, the initial estimate size is MIN($\prod_{D_i \subset V} |D_i|$, $n_{raw}$), where $\prod_{D_i \subset V} |D_i|$ is the product of the cardinalities of dimensions in $V$, and $n_{raw}$ is the size of raw data. The result was refined in two methods. The first way is to compute a view $V_p$, then fix all the estimate size of views derived from $V_p$ to be

smaller than $V_p$. The other way is to fix the estimate size using the last two levels. For example, once we compute $ABCD$, $ABC$, $ABD$, then we refine the size of $AB$ by

$$\frac{|ABCD|}{|ABC|} \leq \frac{|ABD|}{|AB|} \quad or \quad |AB| \leq \frac{|ABD||ABC|}{|ABCD|}.$$

By this way, we may update the estimate cost on the fly. However it is still not accurate for meaningful pipeline computation [32].

Sample scaling is proposed in the paper [44]. This method randomly selects some sample data from the raw data, and builds a small data cube from the sample data. After that, the size of sample cube may be scaled up by the ratio of the size of the raw data to the size of the sample data. Sample scaling gives a more accurate estimation than the cardinality-based method. However, it could underestimate the duplicates in large raw data [32].

A probabilistic method is proposed in the paper [49]. It is based on the probabilistic counting algorithm [36]. The counting algorithm scans the raw data once, concatenates the dimensions into bit-vectors of length $L$, and hash the bit-vectors into a range of $[0, 2^L - 1]$. After that, it counts the number of distinct rows that are likely to exist in each view. The probabilistic method in the paper [49] improves the estimate accuracy by using a universal hashing function [61], to guarantee a bound error for skewed data sets. The experiments in the thesis [32] shows the probabilistic method works well for small problems, but takes a long time for high dimensional data sets. Even though we optimize the implementation of the algorithm and the new codes are a factor of 30 times faster, the running time for estimating the sizes of views is longer than that for computing data cubes itself [32].

Todd Eavis proposed our own probabilistic approach in his thesis [32]. The approach is based on the theorem [32]:

> "For an input set of size n, and a view $V$ with a potential space of size $S_v$, we may estimate the number of records $r$ in $V$ by performing the summation
>
> $$x = \sum_{i=0} \frac{S_v}{S_v - i}$$
>
> and terminating in one of two possible cases:
>
> 1. $i >= S_v$, in which case r = $S_v$.

2. $x >= n$, in which case $r = i$."

In this theorem, the potential space $S_v = \prod_{D_i \subset V} |D_i|$, which is the product of the cardinalities of dimensions in $V$. In the first terminating case, $i >= S_v$, it means $S_v$ has been fully saturated. Therefore, we can estimate the size of $V$ as $S_v$. In the second case, $x >= n$, we have used all $n$ records without saturating $S_v$, so we estimate the size of $V$ as $i$.

In this approach, we assume that the data distribution is approximately uniform. Even though it is not always true, its accuracy and efficiency are suited to the costing model in PipeSort used in our parallel algorithms [32].

### Execute the Plan

In the second step, PipeSort converts the spanning tree into a set of pipelines, and executes them. Two optimizations are implemented in this step: *cache-results* and *amortize-scans*. The optimization of cache-results tries to cache the views in memory, from which some other views can be computed to reduce disk I/O. The optimization of amortize-scans tries to compute as many as possible views from the views already in memory to reduce disk reads.

A pipeline consists of a number of views, connected by one sort edge and several pipeline edges. The sort edge connects the first two views, and the pipeline edges connect the rest of the views. Except for the first view, the other views in a pipeline share the prefix with each other. For example, in a pipeline $raw \Rightarrow ABCD \rightarrow ABC \rightarrow AB \rightarrow A$, the sort edge "$\Rightarrow$" connects $raw$ and $ABCD$, and the pipeline edges "$\rightarrow$" connect the remaining views. Figure 2.17 illustrates the pipelines from the spanning tree in Figure 2.16.

Algorithm 2 outlines steps to execute the plan in PipeSort. In Algorithm 2, the spanning tree is converted into a set of pipelines, and then the pipelines are executed one by one in a *pipelined* fashion [59]. Using a pipelined fashion, we first sort the first view into the order of the second view. Then we scan the sorted data and check the row for the second view. Once a row of the second view is found, we aggregate it and check the row for the third view. This continues until the last view in the pipeline. Table 2.8 shows how to execute a pipeline in a pipelined fashion, $raw \Rightarrow ABCD \rightarrow ABC \rightarrow AB \rightarrow A$. In the table, the numbers before the commas

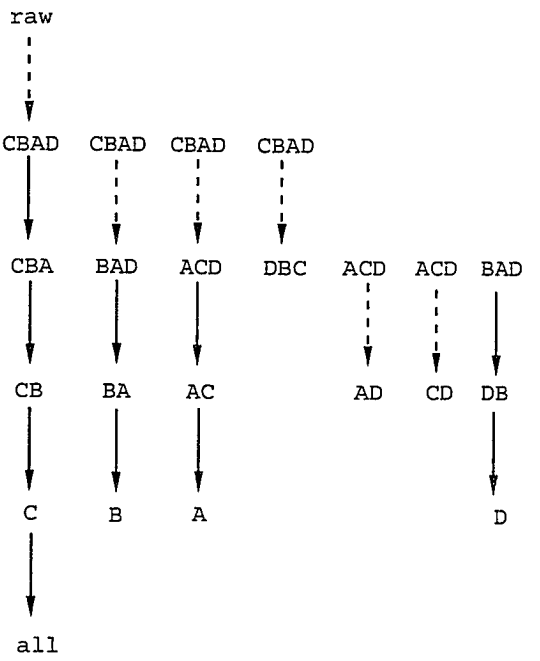



Figure 2.17: The Pipelines. [59]

are the values of dimensions and the numbers after the commas are the values of the measure. In this example, the aggregate function is "SUM". The blanks in the table represents no operation, and "Check" represents checking whether a row can be generated by aggregating.

To compute a full data cube with $d$ dimensions, PipeSort needs $\binom{d}{\lceil d/2 \rceil}$ sorts at least [41]. It is exponential with $d$, so PipeSort does not scale well with very high dimensional cubes. However in most OLAP applications, the number of dimensions is below ten [50]. The advantage of PipeSort is that it can handle huge data sets up to terabyte if external memory sorts are used. Therefore PipeSort is a suitable sequential data cube generation algorithm for designing a parallel data cube generation algorithm based on it.

### 2.3.3 PipeHash

PipeHash is also proposed in the paper [59] to compute a full data cube. PipeHash is a hash based ROLAP algorithm. To compute a view, PipeHash scans one of its

**Algorithm 2** Execute-Plan

**Input:** A PipeSort spanning tree.

**Output:** The views in the spanning tree.

1: Convert the spanning tree into a set of $j$ pipelines.
2: **for** $i = 1$ to $j$ **do**
3: Sort the first view into the order of the second view in the $i$th pipeline.
4: Scan the sorted data to generate the rest of the views in the $i$th pipeline.
5: **end for**

| Sorted Raw | ABCD | ABC | AB | A |
|---|---|---|---|---|
| 1 1 1 1,1 | Check | | | |
| 1 1 1 1,1 | 1 1 1 1,2 | Check | | |
| 1 1 1 2,1 | 1 1 1 2,1 | 1 1 1,3 | 1 1,3 | Check |
| 1 2 1 2,1 | Check | | | |
| 1 2 1 2,1 | 1 2 1 2,2 | 1 2 1,2 | Check | |
| 1 2 2 1,1 | Check | | | |
| 1 2 2 1,1 | 1 2 2 1,2 | Check | | |
| 1 2 2 2,1 | 1 2 2 2,1 | 1 2 2,3 | 1 2,5 | 1,8 |
| 2 1 1 1,1 | Check | | | |
| 2 1 1 1,1 | Check | | | |
| 2 1 1 1,1 | 2 1 1 1,3 | Check | | |
| 2 1 1 2,1 | 2 1 1 2,1 | 2 1 1,4 | 2 1,4 | Check |
| 2 2 1 2,1 | Check | | | |
| 2 2 1 2,1 | 2 2 1 2,2 | 2 2 1,2 | Check | |
| 2 2 2 2,1 | Check | | | |
| 2 2 2 2,1 | 2 2 2 2,2 | 2 2 2,2 | 2 2,4 | 2,8 |

Table 2.8: An Example of the Pipelined Fashion

parent views to build a hash table of the view in memory to order to implement aggregates. PipeHash generates a minimum spanning tree (MST) by choosing the parent view with the smallest estimated size for each view in the lattice, illustrated in Figure 2.18. Since memory is not always enough to hold all the hash tables in one MST, PipeHash partitions a big MST into small subtrees on one or more dimensions. In each subtree, the target views can be computed by scanning the root view once from the disk, and the hash tables of target views can fit in memory. PipeHash uses a heuristic method to find the best partitioning so that the scanning cost of root views is minimized. Figure 2.19 illustrates the subtrees generated from Figure 2.18, and the first subtree is partitioned on $A$. To compute a subtree, PipeHash scans the root

view only once to generate all the hash tables in memory for its children views, and then saves the view to disks and releases the memory. Next it computes the next subtree until all the subtrees are finished.

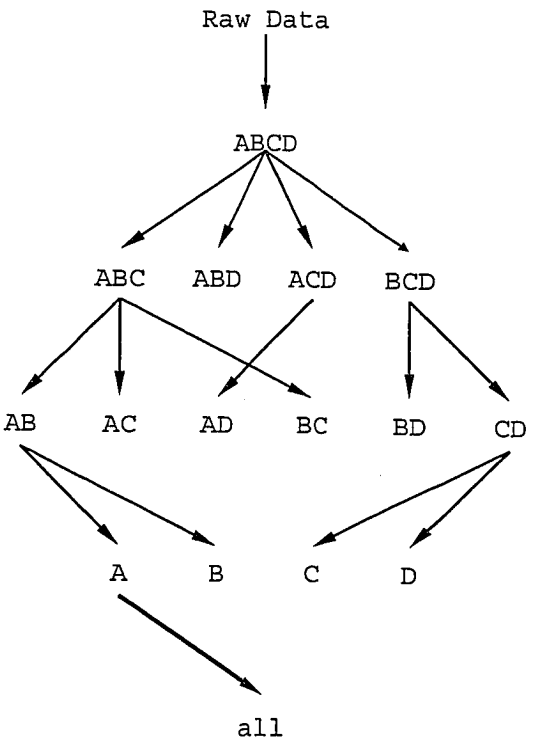



Figure 2.18: A Minimal Spanning Tree of PipeHash. [59]

### 2.3.4 OVERLAP

OVERLAP is proposed in the paper [17] to compute a full data cube. It is a sort based algorithm. OVERLAP tries to minimize the number of disk accesses by overlapping the computations of different views and reduces the number of sort steps by using partially matching sort orders. Suppose a view, $V$, and its parent view, $V_p$, share the same prefix $D_1D_2...D_i$. OVERLAP partitions $V_p$ on $D_1D_2...D_i$, and computes $V$ from these independent partitions. Therefore, the memory required to compute $V$ depends on the biggest partition of $V_p$ instead of $V_p$. OVERLAP begins with a sorted raw data on the specific dimensional order, and keeps the order when it computes all the views

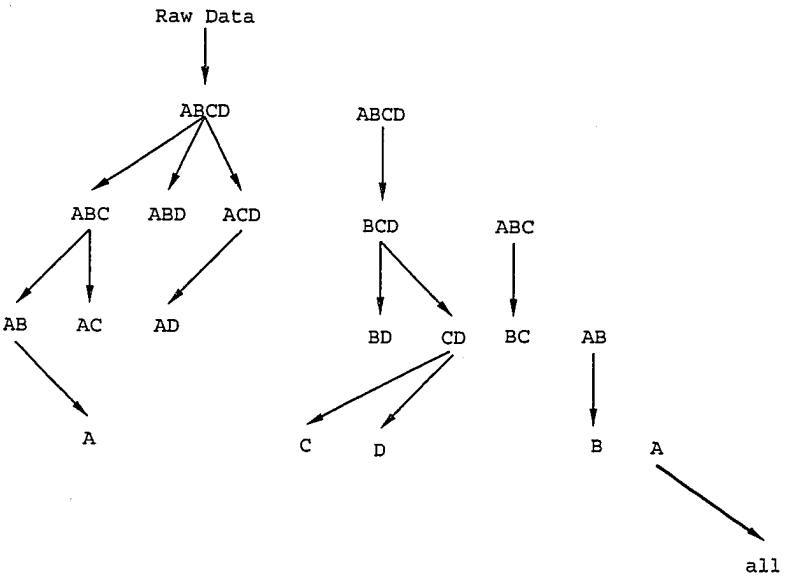



Figure 2.19: Subtrees of PipeHash. [59]

in the lattice. First, it generates a spanning tree from the lattice to minimize the size of the partitions of views. Figure 2.20 illustrates a possible spanning tree for the four dimension lattice. In this step, OVERLAP estimate the sizes of views and partitions, which uses the estimation technologies mentioned in the previous section. If the partition of a view can be allocated in memory, the view is marked as *Partition* state, or it is marked as *SortRun* state. Overlap scans the parent view once to compute a *Partition* state view, and uses pipelines to compute more views if possible. For *SortRun* state views, OVERLAP reads data into memory, sorts them and outputs to disks block by block, and then merges them externally. The paper [57] points out the I/O cost of OVERLAP is at least $O(d^2)$ for sparse data sets, where $d$ is the number of dimensions.

### 2.3.5 Partitioned-Cube and Memory-Cube

Partitioned-Cube and Memory-Cube are proposed to compute a full sparse data cube in the paper [57]. The main idea in this algorithm is divide-and-conquer. Partitioned-Cube partitions large views into small blocks which can fit in memory. Then Memory-Cube computes a set of views from the blocks in memory. Partitioned-Cube partitions

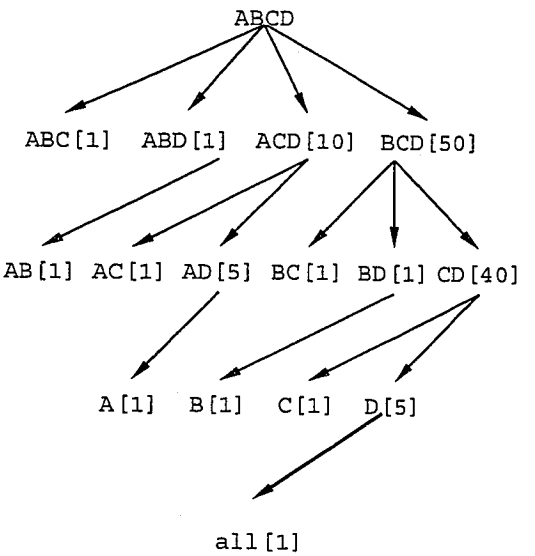



Figure 2.20: A OVERLAP Spanning Tree with Estimated Partition Size in Memory Pages. [17]

views on the first dimension, and recursively partitions views on the rest of dimensions one by one until all partitions can fit in memory. Figure 2.21 illustrates partitions for a four dimension data cube. In this example, Partitioned-Cube partitions the raw data on $A$ first, and computes the views from these partitions in memory. The dashed lines in Figure 2.21 represents in-memory sort, and the solid line represents in-memory scan. After all the views with the prefix $A$ are finished, Partitioned-Cube partitions $ABCD$ on $B$, while $A$ dimensions are projected out. Then after all the views with the $B$ prefix are finished, Partitioned-Cube stops partitioning, because $BCD$ can fit in memory. It projects out $B$ and computes the rest of views. Memory-Cube uses a similar pipeline fashion as PipeSort to compute data cube in memory, and needs to process $C^d_{\lceil d/2 \rceil}$ pipelines. The paper claims this algorithm is linear on I/O cost, while other algorithms, such as PipeSort or OVERLAP are quadratic at least on I/O cost.

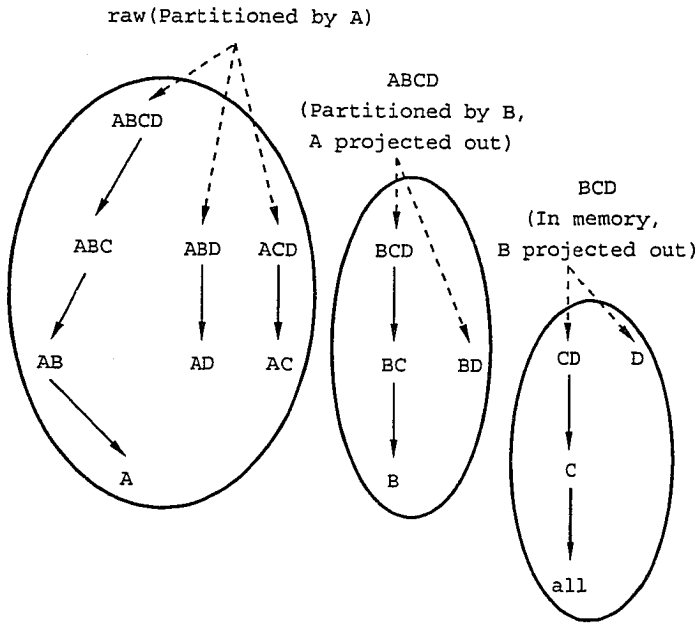



Figure 2.21: Partitions in Partitioned-Cube. [57]

### 2.3.6 Array-Based Cube

The paper [65] proposed an array-based algorithm to compute a full data cube for MOLAP. We call it the *array-based cube*. Instead of using tables in ROLAP, the array-based cube uses multidimensional arrays to store data cubes, and partitions cubes into chunks which can fit in memory. For sparse data cubes, the array-based cube uses "chunk-offset compression" to compress them. This method converts an address of a cell in an uncompressed multidimensional array into an integer offset in a one-dimensional array, where compressed data cubes are stored. And for a given dimension order, the array-based cube indexes the chunks using a sequence integer. Figure 2.22 illustrate a three dimension array with the order $ABC$. The number in the cells indicates the indexes of chunks.

In the array-based cube, each view is computed from one of its parent views by scanning chunks of the parent in the order of indexes. To store the aggregate results in memory, the minimal number of memory units for each view must be allocated for each view. The array-based cube generates a minimum memory spanning tree (MMST) to minimize the total memory for a given dimension order. Figure 2.23

illustrates a three dimension MMST in dimension order of $ABC$. In this figure, the size of a chunk is 4x4x4 memory units. We begin with $ABC$, which needs only one chunk memory, and reads and scans $ABC$ chunk by chunk. For $AB$, we need 4x4 chunks or 16x16 memory units for the aggregate buffers, because we have to scan all the 64 chunks of $ABC$ before we can output one chunk of $AB$ to disks. For $AC$, we need 4x1 chunks or 16x4 memory units, because we have to scan 16 chunks of $ABC$ before we can output one chunk of $AC$ to disks. And for $BC$, we need 1x1 chunk or 4x4 memory units, because we have to scan 4 chunks of $ABC$ before we can output one chunk of $BC$ to disks. For different dimension orders, the array-based cube generates different MMSTs, which need different sizes of memory. The optimal order is incremental in the cardinalities of dimensions, such as $|D_1| \leq |D_2| \leq ... \leq |D_d|$.

After the array-based cube generates a MMST, it computes child views from the root view first. As soon as a chunk of a child view finished, it recursively computes child views of this child view. After that, the chunk is written to disks and reused for further data. If the memory is insufficient for the whole MMST, some subtrees in MMST need to generate intermediate results and swap them between memory and disks.

The array-based cube is very efficient on low dimension dense cubes. It needs too much memory for sparse cube, and does not scale well in the number of high dimensions.

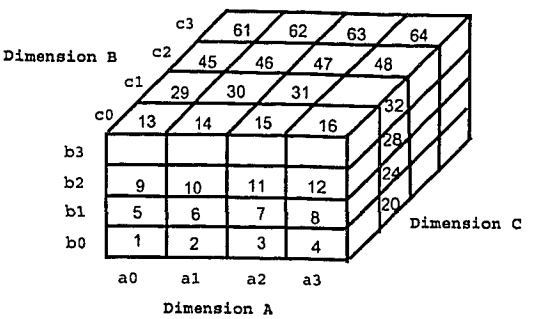



Figure 2.22: A Three Dimension Array. [65]

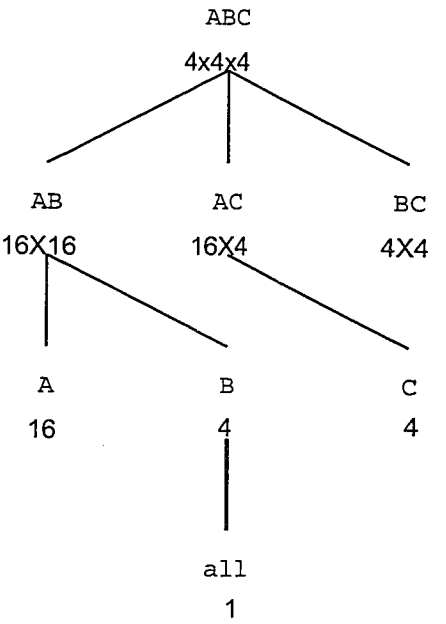



Figure 2.23: A Three Dimension MMST in Dimension Order of "ABC". [65]

### 2.3.7 PipeSort for Partial Data Cubes

A partial data cube includes a subset of $2^d$ views in a full cube. Even though both Top-down methods and Bottom-up methods can be designed for computing partial data cubes, Bottom-up methods cannot efficiently compute an arbitrary subset of views and they are not good for low dimensional views [38]. The popular Top-down method, PipeSort, is a good choice to develop partial data cube algorithms.

In the PipeSort paper [59], they proposed a PipeSort algorithm for partial data cubes. The main idea of their algorithm is to convert the lattice into an augmented lattice first, then create a spanning tree from the augmented lattice by using a minimum Steiner tree approximation algorithm. In an augmented lattice, there are some Steiner nodes and Steiner edges, which are not in the original lattice. An augmented lattice can be built by adding all possible permutations of each dimension's ordering and edges, which connect every permutation with every possible descendant. The problem of this algorithm is that there are too many Steiner nodes and Steiner edges. For example, for a ten dimension augmented lattice, there are almost ten million nodes and 40 trillion edges [32]. It takes forever to generate a spanning tree.

In the paper [38], we proposed two methods based on PipeSort to compute partial data cubes. They are *Tree Partial Cube* and *Lattice Partial Cube*. The difference between them is that the tree partial cube generates a spanning tree from the spanning tree for a full cube, and the lattice partial tree generates a spanning tree directly from the lattice of $2^d$ views. Algorithm 3 describes the main steps of the tree partial cube algorithm. Algorithm 4 describes the main steps of the tree partial cube algorithm.

**Algorithm 3** Tree Partial Cube

**Input:** A lattice of $d$ dimensions with sort costs and scan costs on each edge; A set of selected views, $S$.

**Output:** A PipeSort spanning tree with minimal global cost.

1: Compute the Pipesort spanning tree for a full data cube, $L$ , and prune it by deleting all nodes which have no descendent in $S$. Let $G$ denote the result.
2: Call PartialCubeSchedule($S$, $G$, $T$), where $T$ is an output spanning tree for the partial data cube.
3: Call FixPipelines($T$).
4: Build the partial data cube $PC$ according to the schedule tree $T$.

**Algorithm 4** Lattice Partial Cube

**Input:** A lattice of $d$ dimensions with sort costs and scan costs on each edge, $L$; A set of selected views, $S$.

**Output:** A PipeSort spanning tree with minimal global cost.

1: Prune all nodes in the lattice $L$, which have no descendent in $S$. Let $G$ denote the result.
2: Call PartialCubeSchedule($S$, $G$, $T$), where $T$ is an output spanning tree for the partial data cube.
3: Call EstablishAttributeOrderings($T$).
4: Build the partial data cube $PC$ according to the schedule tree $T$.

In both algorithms, the core procedure is *PartialCubeSchedule*, which is defined in [38]. This procedure includes two steps [38]: first it organizes the nodes of $S$ into a tree with the minimum total cost, and second, it tries to add intermediate nodes from $G - T$ to the tree to further minimize the total cost by using the "plan" variables for a given node $v$, listed in Table 2.9. In the first step, the procedure begins with

| Name | Meaning |
|---|---|
| Node | The node $v$ to be inserted into the spanning tree. |
| Parent | The parent node of $v$. |
| Parent Mode | Sort or scan between $v$ and its parent. |
| Scan Child | The child node of $v$, computed by scanning $v$. |
| Insertion Scan Child | The inserted node $v$, computed by scanning from a node in the spanning tree. |
| Sort Children | The child node of $v$, computed by sorting $v$. |
| Benefit | The improvement in the total cost by inserting $v$. |

Table 2.9: The "Plan" Variables

an empty $T$. Then from $S$ it selects the best node and the best plan, which generate the biggest benefit. Next it inserts the node into $T$ based on the plan, and removes it from $S$. This repeats until $S$ is empty. In the second step, the procedure uses the similar methods to add nodes into $T$ from $G - S$, instead of $S$.

After the the procedure *PartialCubeSchedule*, Algorithm 3 executes a post-processing procedure *FixPipelines*, which identifies nodes that have no scan child, creates a scan child for such nodes, and fixes the dimension orders. Algorithm 4 executes a post-processing procedure *EstablishAttributeOrderings*, which identifies pipes of possible scan orders, and fixes the dimension orders if possible. Both algorithms then compute the pipelines using the same methods as the full cube PipeSort algorithm.

The experiments in the paper [38] show that the tree partial cube and the lattice partial cube have the similar shapes of curves most of the time, while the lattice partial cube is a little faster than the tree partial cube. And the lattice partial cube is very close to the PipeSort algorithm when we generate a full data cube, even though they are based on fundamentally different schedule tree generation methods. The paper [38] suggests that the lattice partial cube is a general purpose replacement for PipeSort for full data cubes, with the ability of generating partial data cubes efficiently.

### 2.3.8 Bottom-Up Cube

For an iceberg data cube, we compute all the $2^d$ views, but for each view, we only keep the rows, whose measures are above specific thresholds. In a iceberg cube query, there is a "having" clause to indicate the thresholds. The thresholds are also called

*minimal support* in some of the literature. Bottom-Up Cube (BUC) is designed to compute sparse iceberg cubes in the paper [20]. We have our own ice-berg cube algorithm, which is more suitable for all kinds of iceberg cubes, where we will discuss it in the later chapter of this thesis.

BUC is a bottom-up algorithm, which computes the small views first, and then large views. In a lattice it processes views from lower(bottom) levels to higher(up) levels, so it is called Bottom-Up Cube. Algorithm 5 outlines the main steps of BUC. It combines I/O efficiency by using partitioning and pruning using the anti-monotonicity to recursively compute views in the lattice from the bottom to the top.

**Algorithm 5** BUC

**Input:** The input data $R$; the first dimension $D_f$ of $R$.

**Output:** The rows in $R$, whose measures are greater than *minsup*.

1: Aggregate $R$ along $D_f$, and output the tuples, whose measures are greater than *minsup*.
2: **for** $D = D_f$ to $D_d$ **do**
3: Partition $R$ into $|D|$ partitions on $D$.
4: **for** $i = 1$ to $|D|$ **do**
5: **if** The measure of the partition $R_i$ is greater *minsup* **then**
6: Call BUC($R_i$, $D_f + 1$).
7: **end if**
8: **end for**
9: **end for**

Figure 2.24 illustrates a four dimension BUC processing tree. BUC begins with the sorted raw data and computes "ALL" first, and then partitions the data on $A$. For each partition of $A$, BUC checks if the measure is greater than *minsup*. If so, BUC outputs the results and continues to partition it on $B$, and then calls the same procedure on these $AB$ partitions. If the measure is not greater than *minsup*, the partition is pruned. After $A$ dimension is processed, BUC partitions on $B$, $C$ and $D$. In this way, BUC walks through the processing tree in a depth-first fashion. Even though BUC can work in a breadth-first fashion like Apriori [18], the paper [20] points out that the breadth-first fashion needs more memory and shows poor performance on high skewed input. In order to pruning views as early as possible, BUC arranges the

dimensions in the descent order on their cardinalities, such as $|A| \geq |B| \geq |C| \geq |D|$ in the above example.

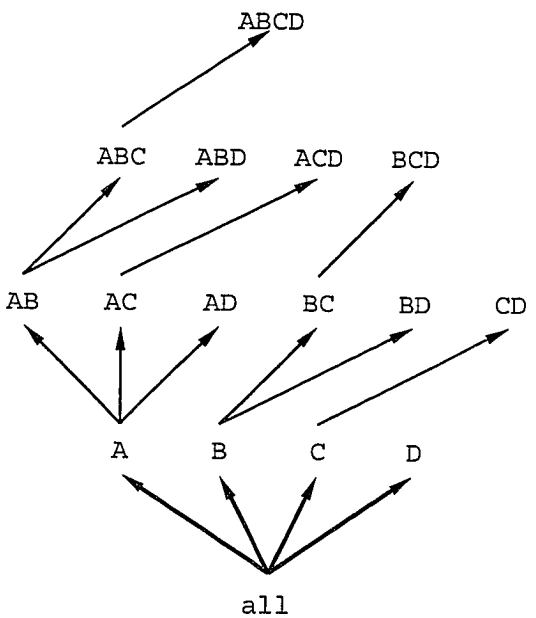



Figure 2.24: A Four Dimension BUC Processing Tree. [20]

BUC takes advantage of monotonic aggregate functions to prune views. For an aggregate function $agg()$, if $agg(S) \geq agg(T)$, where $S \supseteq T$, then $agg()$ is monotonically decreasing; if $agg(S) \leq agg(T)$, where $S \supseteq T$, then $agg()$ is monotonically increasing. COUNT, SUM , MIN and MAX are monotonic, but AVG is not monotonic. The above BUC algorithm works for monotonically decreasing aggregate functions. If the aggregate function is monotonically increasing, BUC will work for the queries, where measures are smaller than thresholds, like: "SELECT A, B, C, D, SUM(measure) FROM R GROUP BY CUBE(A, B, C, D) HAVING SUM(measure) $\leq$ 10."

The experiments in the paper [18] demonstrated that BUC was faster at computing full sparse data cubes that other full data cube algorithms. And for iceberg cube queries, BUC improves upon its own performance.

### 2.3.9 Generation of Data Cube with Dimensional Hierarchies

In Section 2.1.3, we described the concepts of levels and hierarchies in dimensional tables. Levels in a dimensional table can determine the granularity of facts and a hierarchy is a set of levels whose granularity becomes finer each step down the hierarchy. For example, in Figure 2.2, the hierarchy in the "date" dimension is "Year-Month-Day", the hierarchy in the "store" dimension is "Store-City-Province-Country" and the hierarchy in the "product" dimension is "Product-Department-Category".

In data cube generation, if we consider dimensional hierarchies, the total number of views to be generated will be much greater than the number of views without hierarchies considered. For examples, if we do not consider hierarchies for the dimensions of "date", "store" and "product", the total number of view in a full data cube is $2^3 = 8$. If we consider hierarchies and the number of levels are three, four and three for "date", "store" and "product" respectively, then the total number of views in a full data cube is $(3+1)(4+1)(3+1) = 80$. These are ten times more views than we would have if we did not consider hierarchies.

Suppose the total number of dimensions is $d$ and the number of levels in each dimensions are $L_0, L_1, ..., L_{d-1}$. Then the total number of views for a full cube is

$$\prod_{i=0}^{d-1}(L_i + 1),$$

where we consider "all" is a special level for every dimension, so that the total number of level for each dimension is $(L_i + 1)$. If $L = L_0 = L_1 = ... = L_{d-1}$, then the total number of views is

$$\prod_{i=0}^{d-1}(L + 1),$$

or

$$(L+1)^d.$$

If we do not consider hierarchies, then $L = 1$ and the number of views is $2^d$.

Most of the previous sequential data cube generation algorithms [59, 17, 57, 65, 38, 20, 64] do not consider the hierarchies on each dimension, so that they generate only $2^d$ views. However, these algorithms can be extended to compute all possible views with hierarchies defined on dimensions. For example, in the PipeSort algorithm, we may extend the lattice to include all possible views with different levels. Figure 2.25

shows a lattice of two dimensions with hierarchy, where $A$ has three levels $A1$, $A2$ and $A3$ and $B$ has two levels $B1$ and $B2$. In the lattice, we connect any two views where only one dimension has the next higher level in the hierarchy.

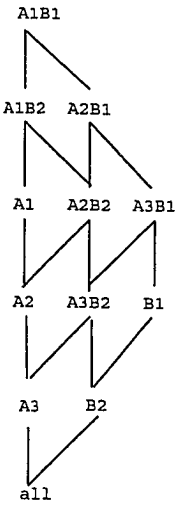



Figure 2.25: A Lattice with Hierarchy. [59]

In the PipeSort paper [59], they points out that we may apply the PipeSort algorithm on the extended lattice with only one modification, which is that when we sort data on $Ai$, we sort for all higher levels of the hierarchy, i.e., Ai, A(i+1), ..., all. In this case the complexity of PipeSort with dimensional hierarchies is $O(\binom{d}{\lceil d/2 \rceil} L^d)$, where $L$ is the number of levels in every dimension.

Since the total number of views with hierarchies is often too large to handle in practice, we may choose not store all the possible views in a data warehouse. For example, for a data warehouse with ten dimensions and three levels on each dimension, the total number of views is $(3+1)^{10} = 1048576$ or 1 million roughly. In practice, we may choose to only generate the data cubes on the base levels of each dimension, and then use roll-up operators to transform base level views to high level views at query time. This is the case that most of data cube generation algorithms are designed for generating base level data cubes. In this thesis, our parallel data cube algorithms are also designed for base level data cubes, however they can be easily extended to handle the data cubes with hierarchies by using simple parallel sorting to implement the required roll-up operators.

## 2.4 Shared-Nothing Clusters

*Parallel Computing* is using multiple computer resources to solve a computational problem simultaneously. The computer resources can be processors in a computer or computers in a network or networks connected with each other. A *cluster* [21] is a type of parallel or distributed processing system. In this system there is a collection of interconnected nodes working as an integrated computing resource. The nodes in a cluster can be PCs, workstations or Symmetric Multiprocessors (SMPs). An SMP is one computer with multiple processors which share memory, disks and one operating system.

In recent years, there has been a trend in parallel computing to move away from specialized super computing platforms to cheaper general purpose clusters made from single or multiple processor PCs or workstations. The advantages of clusters are: [21]

- Workstations are becoming increasingly powerful and the performance doubles every 18 to 24 months.
- The bandwidth of networks for clusters is increasing while the latency is decreasing.
- Workstation clusters are easier to integrate into existing networks than special parallel computers.
- Typical low user utilization of personal workstations.
- The development tools for workstations are more mature than special tools for parallel computers.
- Workstations are much cheaper than parallel computers.
- Clusters can be easily grown. The processors and memory can be easily doubled.

Figure 2.26 shows a cluster computer architecture. In this architecture, high speed networks can be Gigabit Ethernet. Nodes can be PCs, workstations or SMPs. Operation systems can be Linux, Solaris, Windows NT, AIX or HP-UNIX based on different hardware. However, high speed networks are expensive, so the commonly used networks for clusters are those networks with small bandwidth and high latency,

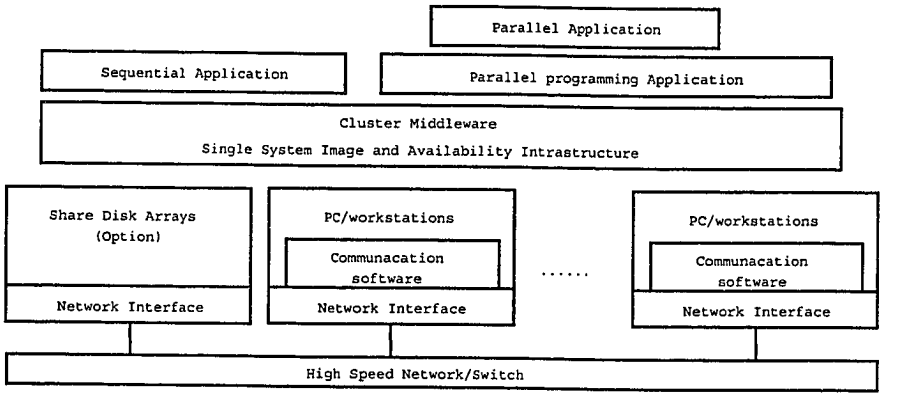



Figure 2.26: Cluster Computer Architecture. [21]

such 10M or 100M LANs. Clusters of workstations with slow networks are suitable for only those applications which need only light communications. To deal with huge data movement among nodes, we have to design specific parallel algorithms for clusters to decrease communications across the network.

The cluster middleware in Figure 2.26 are the tools or interfaces to hide the various hardware and networks from the applications and to provide a standard way to share data among nodes or processors. One popular piece of middleware is the Message Passing Interface (MPI) [15], which is a standard interface for message passing implementations among processors in a cluster. MPI supports both shared memory systems and distributed memory systems. In shared memory systems, MPI sends messages by using shared memory. In distributed memory systems, MPI sends messages by using standard network protocols, such as TCP/IP. MPI is the most widely used message passing model on various operation system platforms in industry.

The nodes in a cluster may share memory or disk arrays. If there is neither memory nor disks shared among nodes in clusters, they are called shared-nothing clusters, such as the popular, low cost, Beowulf style clusters [1], which consist of standard PCs connected via a data switch without any expensive shared disk array; see Figure 2.27.

In this thesis, we implement and test our algorithms on a Beowulf cluster [1], which consists of 16 nodes, with 1.8 GHz Intel Xeon processor, 512 MB RAM and two 40 GB 7200 RPM IDE disk drives on each node. Every node is running Linux Redhat

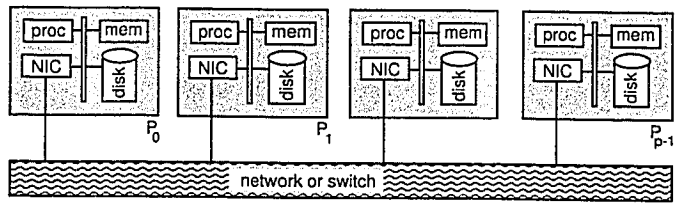



Figure 2.27: A Shared-Nothing Cluster.

7.3 with gcc 2.96 and MPI/LAM 6.5.6 as part of a ROCKS cluster distribution. All nodes were interconnected via an Intel 100 Megabyte Ethernet switch.

## 2.5 Summary

Data Warehousing provides a collection of decision support technologies to analyze data and make better and faster decisions. In data warehousing, data warehouses are used as data sources for analysis. OLAP is one of the essential decision support tools in data warehousing as it can analyze data in a multidimensional way. Data cubes are the core of OLAP systems. A lot of sequential algorithms have been proposed to compute the three types of data cubes: full cubes, partial cubes and iceberg cubes. However, new challenges are constantly arising in the OLAP area. One of these challenges is how to meet the scalability needs in large scale data warehouses. *Parallel Computing* can provide two key ingredients for dealing with the large data size: increased computational power through multiple processors and increased I/O bandwidth through multiple parallel disks. In recent years, it has been a trend in parallel computing to move away from specialized super computing platforms to cheaper general purpose shared-nothing clusters. In the following chapters, we will present our novel parallel algorithms for computing all the three types of data cubes for shares-nothing clusters.

# Chapter 3

# In-Memory Parallel Generation of Full and Partial ROLAP Data Cubes

In this chapter, we present parallel algorithms for generating both full and partial ROLAP data cubes. First we present algorithms and design issues, and then we describe a systematic evaluation of these algorithms.

## 3.1 Introduction

In recent years, the size of data warehouses has been increasing constantly. Many commercial data warehouses contain terabytes of data. This makes the size of the associated data cubes potentially very large. In order to meet the scalability needs on large data sets, parallel solutions for generating data cubes have become increasingly important. The current parallel approaches can be grouped into two broad categories: 1) *work partitioning* [28, 33, 52, 55, 56] and 2) *data partitioning* [39, 40].

Work partitioning methods assign different view computations to different processors. Consider, for example, the lattice for a four dimensional data cube as shown in Figure 2.13. From the raw data set $ABCD$, 15 views need to be computed. Given a parallel computer with $p$ processors, work partitioning schemes partition the set of views into $p$ groups and assign the computation of the views in each group to a different processor. The main challenges for these methods are load balancing and scalability, which are addressed in different ways by the different techniques studied in [28, 33, 52, 55, 56]. One distinguishing feature of work partitioning methods is that all processors need simultaneous access to the entire raw data set. This access is usually provided through the use of a shared disk system (available e.g. for SunFire 6800 and IBM SP systems). Todd Eavis presented extensive parallel algorithms based on work partitioning methods in his thesis [32]. Since in this thesis, we are interested in algorithms for shared-nothing parallel systems, we are forced to pursue the more challenging data partitioning approach.

Data partitioning methods divide the raw data set into $p$ partitions and store each partition locally on one processor. All views are computed on every processor but only with respect to the partition of data available at each processor. A subsequent "merge" procedure is required to aggregate data across processors. The advantage of data partitioning methods is that they do not require all processors to have access to the entire raw data set. Each processor only requires a local copy of a portion of the raw data which can be stored on its local disk. Therefore, data partitioning method is very suitable for a shared-nothing cluster. The main problem with data partitioning methods is that the "merge", which has to be performed for every view of the data cube, has the potential to create massive data movements between processors with serious consequences for performance and scalability of the entire system. A data partitioning method for MOLAP representations has been presented in [39, 40]. This method is based on a space partitioning of the multidimensional array and a spatial "merge" between different sub-cubes of the MOLAP cube. The spatial "merge" operation can be reduced to a parallel prefix which is a well studied operation for parallel computers.

In this chapter, we present our data partitioning method for computing ROLAP data cubes. The principal advantage of ROLAP is that it allows for tight integration with current relational database technologies. Another advantage of ROLAP is that it requires only linear space and is therefore particularly suitable for the construction of very large data cubes. Our algorithm is, to our knowledge, the first parallel ROLAP data cube generation method for shared-nothing clusters. Our method has the additional advantage that it can be extended to the *partial* cube case where not all views, but only a subset of views, selected by the user, are to be created. This case occurs frequently in practice because the user often knows that some views will not be required for OLAP queries on a given data set.

## 3.2 Full Data Cube Generation on Shared-Nothing Clusters

We consider a shared-nothing cluster consisting of $p$ processors $P_0$, $P_1$ ... $P_{p-1}$, each with its own local memory and local disk, connected via a network or switch; see Figure 2.27. There is no shared memory or shared disk available. As input, we assume a raw data set, $R$, with $n$ rows and $d$ dimensions $D_0$, $D_1$ ... $D_{d-1}$ distributed

evenly over the $p$ disks; see Figure 3.1.

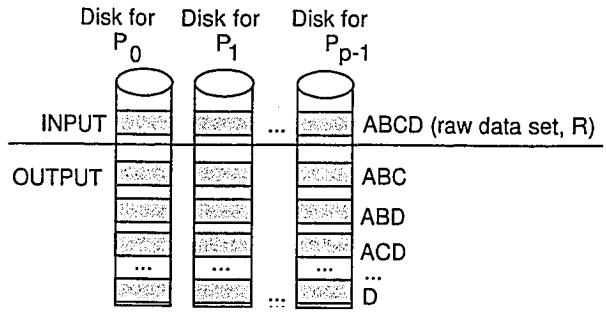



Figure 3.1: Distributed Data Sets.

Without loss of generality, let $|D_0| \geq |D_1| \geq ... \geq |D_{d-1}|$, where $|D_i|$ is the cardinality for dimension $D_i$, $0 \leq i \leq d-1$ (i.e. the number of distinct values for dimension $D_i$). Let $S$ be the set of all $2^d$ view identifiers. Each view identifier consists of a subset of $\{D_0, D_1 ... D_{d-1}\}$, ordered by the cardinalities of the selected dimensions (in decreasing order). The goal is to create a data cube $DC$ containing the views in $S$. We ensure that, when our algorithm terminates, every view is distributed evenly across the $p$ disks; see Figure 3.1. It is important to note that, for the subsequent use of the views by OLAP queries, each view needs to be evenly distributed in order to achieve maximum I/O bandwidth for subsequent parallel disk accesses.

The basic communication operation used by our data cube algorithm is the $h$-relation (method `MPI_ALL_TO_ALL_v` in MPI). The basic computation operations used in this chapter is in-memory scan, in-memory sort and in-memory PipeSort. In the next chapter, we will present an extension to the in-memory algorithm for the external memory case.

### 3.2.1 Algorithm Outline

Let $S_i \subset S$ be the subset of view identifiers in $S$ that start with $D_i$, and let $DC_i$ be the data cube for $S_i$. We call $DC_i$ the $D_i$-partition of the data cube $DC$. Furthermore, we refer to the view consisting of all dimensions contained in views of $S_i$ as the $D_i$-root; see Figure 3.2.

Algorithm 6 describes the global structure of our parallel data cube algorithm for

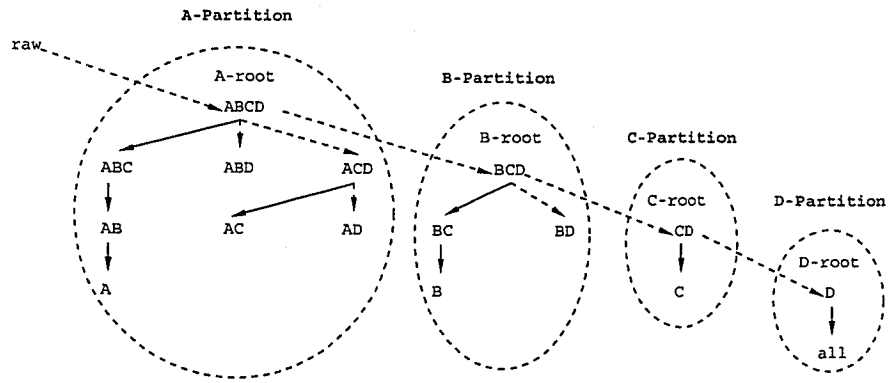



Figure 3.2: Partitions of a Four Dimension Data Cube.

shared-nothing clusters. The algorithm consists of three main phases: data partitioning, computation of local $D_i$-partitions, and merge of local $D_i$-partitions. Subsequent sections will discuss each phase in more detail.

### 3.2.2 Data Partitioning

Good data partitioning is a key factor in obtaining a good load balance and, consequently, good performance. Some researchers partition data on one or several dimensions [55, 58]. In order to achieve sufficient parallelism, they assume that the product of the cardinalities of these dimensions is much larger than the number of processors [58]. The advantage of their method is that they do not need to merge data of views among processors. For example, if we partition on $A$, then $ABC$ and $AC$ do not need to be merged, or if we partition on $A$ and $B$, then $ABC$ and $ABD$ do not need to be merged. However, in practice, this assumption is often not true. The cardinality of some dimensions may be small, such as gender, months and intervals for a numeric attribute. The number of processors in a cluster may be large, especially for low-cost clusters of PCs. Therefore, those methods are often not scalable. Our method avoids these problems by partitioning on all dimensions and then applying a merge procedure. As our experiments show, the cost for the additional merge is more than compensated for by better overall performance and scalability.

To partition data, we use parallel sample sort [51]. As discussed in [51], one global data movement via one single $h$-relation is often sufficient to obtain sorted and

**Algorithm 6** ParallelSharedNothingDataCube

**Input:** Raw data set $R$ with $n$ rows and $d$ dimensions, distributed arbitrarily over the $p$ processors, $n/p$ records per processor.

**Output:** Data cube, $DC$, distributed over the $p$ processors. Each views is evenly distributed over the $p$ processors' disks.

1: **for** i=0 TO d-1 **do**

2: (1) Data Partitioning:

3: (a) Each processor $P_j$ ($j = 0 \ldots p-1$) computes locally the $D_i$-root for its subset of data. (Essentially a sequential sort followed by a sequential scan.) Let $D_i\text{-root}|_j$ denote the $D_i$-root created by processor $P_j$.

4: (b) Call AdaptiveSampleSort($D_i\text{-root}|_0, \ldots, D_i\text{-root}|_{p-1}$; $D_i, \ldots, D_{d-1}$; $\gamma = 1\%$), to sort $\cup_{j=0,\ldots p-1} D_i\text{-root}|_j$ by $D_i, \ldots D_{d-1}$.

5: (c) Each processor $P_j$ ($j = 0, \ldots p-1$) computes locally the $D_i$-root for its subset of data received in the previous step. Let $D_i\text{-root}||_j$ denote the $D_i$-root created by processor $P_j$.

6: (2) Computation Of Local $D_i$-Partitions:

7: (a) Processor $P_0$ locally computes, by applying the first phase of a sequential top-down data cube method, the schedule tree $T_i$ for building the $D_i$-partition with respect to $D_i\text{-root}||_0$.

8: (b) Processor $P_0$ broadcasts $T_i$ to $P_1$ ... $P_{p-1}$.

9: (c) Each processor $P_j$ ($j = 0, \ldots p-1$) computes locally the $D_i$-partition with respect to $D_i\text{-root}||_j$ by applying the second phase of a sequential top-down data cube method to the schedule tree $T_i$ received in the previous step.

10: (3) Merge Of Local $D_i$-Partitions:

11: (a) Call MergePartitions($D_i$).

12: **end for**

well balanced data. The subsequent "global shift" operation, which needs another $h$-relation, is not always necessary. In our implementation of parallel sample sort we measure the imbalance of the sizes of the local data sets after the first $h$-relation and perform a second "global shift" $h$-relation only if necessary. Let $y_0, \ldots, y_{p-1}$ be the sizes of the sets $Y_0, \ldots, Y_{p-1}$ created on processors $P_0, \ldots, P_{p-1}$, respectively, after the first $h$-relation. We calculate the relative imbalance $\mathcal{I}(y_0, \ldots, y_{p-1}) = max\{(y_{max} - y_{avg})/y_{avg}, (y_{avg} - y_{min})/y_{avg}\}$, where $y_{max}, y_{min}$, and $y_{avg}$ are the maximum, minimum and average of $y_0, \ldots, y_{p-1}$, respectively. If $\mathcal{I}(y_0, \ldots, y_{p-1}) > \gamma$ for some threshold value $\gamma$, we apply a subsequent "global shift" operation. In our implementation we use a threshold value of $\gamma = 1\%$. As discussed in [51], the imbalance after the first $h$-relation is less if there are no duplicate keys. However, in most data, there are many duplicate values. Therefore, in Step 1a of Algorithm 6, we first compute locally on each processor $P_j$ $(j = 0, \ldots p-1)$ the $D_i$-root for its subset of data. This eliminates all duplicate keys $D_i \ldots D_{d-1}$ for the sort in the subsequent Step 1b.

We refer to our sample sort implementation as *AdaptiveSampleSort*. Since there are so many "folk" versions of parallel sample sort in the literature, we briefly review in Algorithm 7 the exact sequence of steps implemented in our system.

Following the above global sort, each processor $P_j$ $(j = 0, \ldots p-1)$ applies in Step 1c of Algorithm 6 a sequential scan to its data set in order to compute the $D_i$-root ($D_i$-root$||_j$) for its local data.

### 3.2.3 Computation Of Local $D_i$-Partitions

In this section, we discuss Step 2 of Algorithm 6. The goal of this step is to compute on each processor $P_j$ the $D_i$-partition with respect to $D_i$-root$||_j$. For this, we apply on each processor the sequential partial data cube construction method [38], introduced in Section 2.3.7. In the paper [38], two algorithms are proposed: *Tree Partial Cube* and *Lattice Partial Cube*. Since the lattice partial cube is a little faster than the tree partial cube and it is very close to PipeSort in the full cube running time [38], we choose the lattice partial cube algorithm, described in Algorithm 4 to compute $D_i$-partition from $D_i$-root$||_j$.

For shared-nothing parallel data cube construction, a problem that arises is that each processor $P_j$ has a different data set, namely $D_i$-root$||_j$, and that the schedule

**Algorithm 7** AdaptiveSampleSort($X_0, \ldots, X_{p-1}$; $D_{i_1}, \ldots, D_{i_k}$; $\gamma$)

**Input:** Sets $X_0, \ldots, X_{p-1}$ stored on processors $P_0, \ldots, P_{p-1}$, respectively;

**Output:** Sets $X_0, \ldots, X_{p-1}$ globally sorted by dimensions $D_{i_1}, \ldots, D_{i_k}$.

1: Each processor $P_j$ ($j = 0, \ldots p-1$) locally sorts $X_j$ by $D_{i_1}, \ldots, D_{i_k}$ and selects a set of $p$ *local pivots* consisting of the elements with rank 0, $(n/p^2), \ldots ((p-1)n/p^2)$. Each processor $P_j$ then sends its local pivots to processor $P_0$.

2: Processor $P_0$ sorts the $p^2$ local pivots received in the previous step. Processor $P_0$ then selects a set of $p-1$ *global pivots* consisting of the elements with rank $(p + \lfloor p/2 \rfloor), (2p + \lfloor p/2 \rfloor) \ldots ((p-1)p + \lfloor p/2 \rfloor)$ and broadcasts the $p$ *global pivots* to all other processors.

3: Using the $p-1$ *global pivots* received in the previous step, each processor $P_j$ ($j = 0, \ldots p-1$) locally partitions $X_j$ (sorted by $D_{i_1}, \ldots, D_{i_k}$ from Step 1) into $p-1$ subsequences $X_j^0 \ldots X_j^{p-1}$.

4: Using one global $h$-relation, every processor $P_j$, $j = 0 \ldots p-1$, sends each $X_j^k$, $k = 0 \ldots p-1$, to processor $P_k$.

5: Each processor $P_j$, $j = 0, \ldots p-1$, receiving $p$ sorted sequences $X_k^j$, $k = 0 \ldots p-1$, in the previous step, locally merges those sequences into a single sorted sequence $Y_j$ and sends the size $y_j$ of $Y_j$ to processor $P_0$.

6: **if** $\mathcal{I}(y_0, \ldots, y_{p-1}) > \gamma$, as determined by processor $P_0$ **then**

7: all processors $P_0, \ldots, P_{p-1}$ balance the sizes of $Y_0, \ldots, Y_{p-1}$ via a "global shift", implemented by one $h$-relation operation.

8: **end if**

trees can be different for these different sets. Indeed, the computation of the schedule tree is usually very much data driven. PipeSort and most other methods make statistical estimates of the view sizes, based on the data available, and schedule tree construction is based on those view sizes. In our case, we could allow each processor $P_j$ to build its own *local schedule tree* for its local data set $D_i$-root$||_j$ and build its $D_i$-partition accordingly. However, different local schedule trees for different processors imply that views of the $D_i$-partition created on different processors may be in different sort orders. This creates a problem during the subsequent merge phase in Step 3 of Algorithm 6. When views of the same partition but for different subsets of data (i.e. on different processors) need to be merged, they need to have the same sort order or some of them have to be re-sorted. That re-sort creates a large amount of additional computation. Another possibility is to let one processor, say $P_0$, build the schedule tree for its data set $D_i$-root$||_0$, broadcast that schedule tree, referred to as the *global schedule tree*, and then let all processors use the same global schedule tree for their local cube construction. The advantage of this method is that we do not need to change the sort order of views during the merge. A potential disadvantage is that the sequential, local, top-down data cube methods (e.g. PipeSort) may not be using the "optimal" schedule tree for their data set. Recall that, the schedule trees generated by PipeSort and other top-down sequential methods are based on size *estimates*. As discussed in Section 3.4, the experiments indicate that, among the above two approaches, the latter method is far superior. For the data sets that we tested, the additional work on some processors because of non-optimal global schedule trees was much less than the overhead created through the need to re-sort views during the merge in Step 3. Therefore, Steps 2a, 2b and 2c of Algorithm 6 implement the latter *global schedule tree* method. And in the next chapter, where we present our external memory parallel data cube generation algorithm, the *global schedule tree* method is also used instead of the *local schedule tree* method.

### 3.2.4 Merge Of Local $D_i$-Partitions

At the end of Step 2 of Algorithm 6, each processor $P_j$ has computed the $D_i$-partition for its local data set. For a view $v$ of the $D_i$-partition, let $v_j$ be the view created by processor $P_j$. In Step 3 of Algorithm 6 we need to merge, for each view $v$ in the

$D_i$-partition, the $p$ different views $v_j$ created on the $p$ different processors $P_j$. This merge is performed in Algorithm *MergePartitions($D_i$)* which will be discussed in the remainder of this section.

Consider Algorithm 6 for $i = 0$ and the $A$-partition shown in Figure 3.2. In Step 1 of Algorithm 6, the $A$-roots are globally sorted by $ABCD$. Then, in Step 2, each processor $P_j$ computes locally the A-partition for its data set. Consider the views $ABCD_j$, $ABC_j$, $AB_j$, and $A_j$ computed in Step 2. All these views are in the same sort order as the global sort order created in Step 1 because they are a prefix of $ABCD$. We shall refer to these views as the *prefix views*. The other views, $ABD_j$, $AC_j$, $ACD_j$ and $AD_j$, are not a prefix of $ABCD$ and are therefore in a sort order that is different from the global sort order. We shall refer to them as the *non-prefix views*.

Consider a prefix view $v$ and the problem of merging $v_0, \ldots, v_{p-1}$ stored on processors $P_0, \ldots, P_{p-1}$. For example, consider the view $v = AB$ in Figure 3.2 and the problem of merging $AB_0, \ldots, AB_{p-1}$. The goal is to obtain a global $AB$ order for $AB_0 \cup AB_1 \ldots \cup AB_{p-1}$ and then agglomerate those items that have the same values for dimensions $A$ and $B$. Since $AB$ is a prefix of the global sort order, $ABCD$, the first part is already done and the only items that, potentially, need to be agglomerated are the last item of $v_j$ and the first item if $v_{j+1}$ for each $0 \leq j < p-1$. Therefore, in Algorithm *MergePartitions($D_i$)*, for each prefix view $v$ every processor $P_{j+1}$ simply sends the first item of $v_{j+1}$ to processor $P_j$ which compares it with the last item of $v_j$. Nothing else needs to be done in order to merge all $v_j$. Figure 3.3 illustrates the case of a prefix view $v$ as "Case 1".

We now study the case of merging the views $v_0, \ldots, v_{p-1}$ stored on processors $P_0, \ldots, P_{p-1}$ for a non-prefix view $v$. For example, consider the view $v = AC$ in Figure 3.2 and the problem of merging $AC_0, \ldots, AC_{p-1}$. Again, the goal is to obtain a global $AC$ sort order for $AC_0 \cup AC_1 \ldots \cup AC_{p-1}$ and then agglomerate those items that have the same values for dimensions A and $C$. However, $AC$ is *not* a prefix of $ABCD$ and, therefore, the different $v_j$ can have considerable overlap with respect to $AC$ order. Figure 3.3 illustrates the case of a non-prefix view $v$ as "Case 2" and "Case 3". The rectangles represent the $v_j$ with respect to AC order. The shaded areas represent the overlap which, in contrast to Case 1 (prefix view), can now be

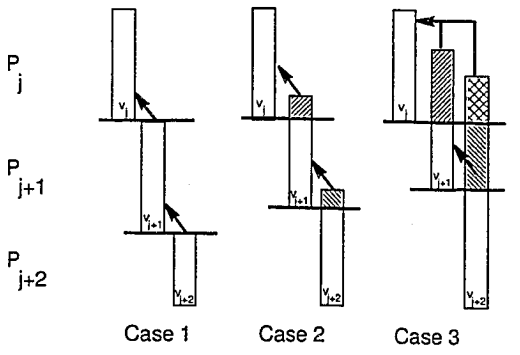



Figure 3.3: Illustration of Cases in Algorithm. *MergePartitions*

considerably more than just one element. In Algorithm $MergePartitions(D_i)$, for each non-prefix view $v$ every processor $P_j$ sends its last element to every other other processor. Each processor $P_k$ then determines its overlap with each $P_j$ and sends that overlap to $P_j$. For each processor $P_j$ let $v'_j$ be the view $v_j$ plus all the overlap received by processor $P_j$. We distinguish two cases which are both illustrated in Figure 3.3. "Case 2": IF $\mathcal{I}(|v'_0|, |v'_1|, \ldots |v'_{p-1}|) \leq \gamma$ for a non-prefix view $v$ THEN each $P_j$ locally sorts $v'_j$ and agglomerates the items with same values for dimensions in $v$. "Case 3": IF $\mathcal{I}(|v'_0|, |v'_0|, \ldots |v'_0|) > \gamma$ for a non-prefix view $v$ THEN the $v_j$ are merged by a global sort. The distinguishing criterion between Cases 2 and 3 is the balance between the $v'_j$. If the imbalance is smaller than $\gamma$ (Case 2) then we proceed similar to Case 1. If the imbalance is larger than $\gamma$ (Case 3) then we need to completely re-balance via a global sort. In fact, for Case 3 we do not wish to even route the overlap between processors. We rather re-sort immediately. Hence, in order to determine whether Case 2 or Case 3 applies, each processor $P_k$ first determines the *size* of its overlap with each $P_j$ and sends only the information about the size of that overlap to $P_j$. Algorithm 8 is an outline of Algorithm $MergePartitions(D_i)$.

## 3.3 Partial Data Cube Construction On Shared-Nothing Clusters

Algorithm 6 has the advantage that it is easily extended to the case where not the entire data cube but only a subset of selected views are to be computed. This case occurs frequently in practice because the user often knows that some views will not

**Algorithm 8** *MergePartitions*($D_i$)

1: For each view $v \in DC_i$, each processor $P_j$ broadcasts the last item of $v_j$ to every other processor $P_k$ and receives back the seizes of all overlaps.

2: For each view $v \in DC_i$, every processor $P_j$ determines $|v'_j|$ and sends all of its $|v'_j|$ values to processor $P_0$.

3: Processor $P_0$ determines for each view $v \in DC_i$ whether it is a "Case 1", "Case 2", or "Case 3".

4: Every processor $P_{j+1}$ sends for each "Case 1" view $v \in DC_i$ the first item of $v_{j+1}$ to processor $P_j$, and $P_j$ compares/agglomerates that item with the last item of $v_j$.

5: Every processor $P_k$ sends for each "Case 2" view $v \in DC_i$ its overlap with every $v_j$ to the respective processor $P_j$. Every processor $P_j$ merges/agglomerates all received overlaps with $v_j$.

6: All remaining "Case 3" views $v \in DC_i$ are merged via global sort, using Algorithm 7 with $\gamma = 3\%$.

be required for the subsequent OLAP queries that are executed on the data cube.

For the purpose of computing a partial data cube, we redefine $S$. Instead of being the set of all $2^d$ view identifiers, we define $S$ to be the view identifiers of the subset of selected views. The definitions of $S_i$, $DC_i$, etc. are then all with respect to the new set $S$ of *selected* views. For each $S_i$, if it is empty, we skip the $i$th loop in Algorithm 6. If it is not empty, we generate $D_i$-root from the last available $D_j$-root, where $0 \leq j < i$. Note that, for optimal performance, some "intermediate" views need to be constructed in addition to the selected views. These intermediate views are determined by a greedy view selection method [38], which was introduced in Section 2.3.7. The other steps in Algorithm 6 remain completely unchanged.

Figure 3.4 illustrates an example of partitioning a partial data cube. In this example, the selected views, $S$, are $ABCD$, $AB$, $A$, $AC$, $AD$ and $C$. In $A$-partition, two intermediate views, $ABC$ and $ACD$ are added into the partition in order to minimize the total cost. In $C$-partition, the intermediate view, $CD$, is added arbitrarily, because $CD$ is the $C$-root. There is no selected view for $B$-partition or $D$-partition, so they are skipped.

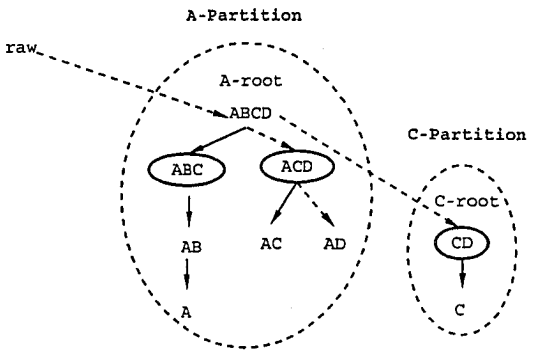



Figure 3.4: Partitions of a Four Dimension Partial Data Cube.

## 3.4 Performance Evaluation

We have implemented our parallel shared-nothing data cube generation method using C++ and the MPI communication library. This implementation evolved from the code base for a fast sequential Pipesort [28] and the sequential partial cube method described in [38]. Most of the required sequential graph algorithms, as well as data structures like hash tables and graph representations, were drawn from the LEDA library [47]. Our experimental platform is a Linux cluster, described in Section 2.4.

In the following experiments all sequential times were measured as wall clock times in seconds. All parallel times were measured as the wall clock time between the start of the first process and the termination of the last process. We will refer to the latter as *parallel wall clock time*. All times include the time taken to read the input from files and write the output into files. Furthermore, all wall clock times were measured with no other user except us on the Linux cluster.

In our experimentation we generated a large number of synthetic data sets which varied in terms of the following parameters: $n$ - number of records, $d$ - number of dimensions, $|D_0|, |D_1| \ldots |D_{d-1}|$ - cardinality in each dimension, and $\alpha$ - skew in the data. Experiments using "real" (non-synthetic) data are described in Section 6.4.

Our experiments explored the following five performance issues:

1. *Relative Speedup:* We investigated the effect of increasing the number of processors on the time required to solve data cube generation problems and measured

the relative speedup, i.e. the ratio between observed sequential time and observed parallel time. Sequential times for computing full cubes and partial cubes were measured on a single processor of our parallel machine using our sequential implementations of PipeSort [28] and partial cube [38], respectively.

2. *Local vs. global schedule trees:* We compared the effect on parallel wall clock time of using local vs. global schedule trees.

3. *Data skew:* We investigated the effect on parallel wall clock time of data sets with varying skew distributions. We used the standard ZIPF [66] distribution with $\alpha = 0$ (no skew) to $\alpha = 3$ (high skew) and explored the relationship between data skew and the amount of data that must be communicated.

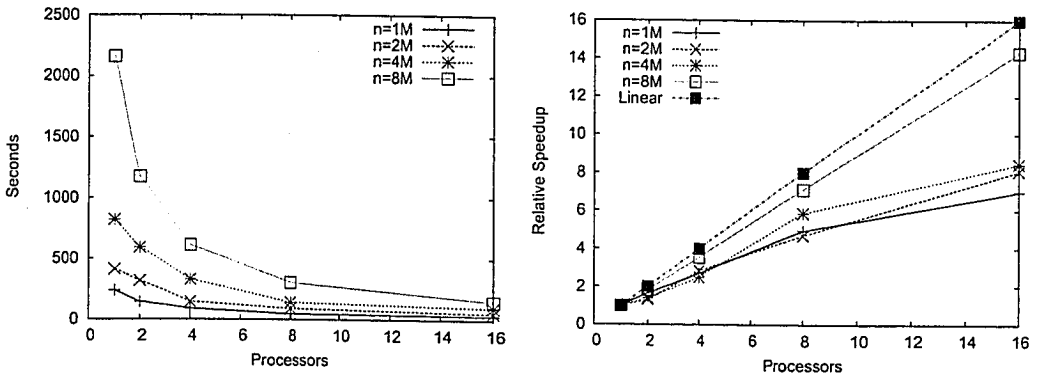
4. *Cardinality of dimensions:* We investigated the effect of varying dimension cardinalities on parallel wall clock time for both skewed and non-skewed data sets.

5. *Data dimensionality:* We investigated the effect of varying dimensionality, and therefore the effects of relative density or sparsity, on parallel wall clock time.

6. *Balance Tradeoffs:* Lastly, we investigated the effect of varying the balance threshold parameter $\gamma$. As $\gamma$ is decreased we improve the balance in the distribution of views across processors, but at the cost of more data movement.

### 3.4.1 Relative Speedup

Speedup experiments are at the heart of the experimental evaluation of our parallel shared-nothing data cube generation method. It is one of the key metrics for the evaluation of parallel database systems [30] as it indicates the degree to which adding processors decreases the running time. The relative speedup for $p$ processors is defined as $S_p = \frac{t_1}{t_p}$, where $t_1$ is the running time of the parallel program using one processor, all communication overhead having been removed from the program, and $t_p$ is the running time using $p$ processors. An ideal $S_p$ is equal to $p$, which implies that $p$ processors are $p$ times faster than one processor. That is, the curve of an ideal $S_p$ is a linear diagonal line.




(a) (b)

Figure 3.5: (a) Parallel wall clock time in seconds as a function of the number of processors for the data size $n$ = from 1 million to 8 million rows and (b) corresponding speedup. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The balance threshold parameter $\gamma = 5\%$)

Figure 3.5 shows for *full cube construction* the parallel wall clock time observed for data sets of varying sizes as a function of the number of processors used, and the corresponding relative speedup. We observe that the speedup is in general better when the input data is larger for the same number of processors, or when we use less processors for the same size of input data. This is not surprising because sequential computing time will increase when each processor gets more data. For example, we observe that the speedup is close to the linear line for 8M rows of data. In general, we observe reasonable speedup when we have at least 0.5M rows of data per processor. The speedup for smaller problems is lower as there is insufficient local computation over which to amortize the cost of communications. On the other hand, our method works well on very large data sets.

Figure 3.6 shows for *partial cube construction* the parallel wall clock time observed for a range of different percentages of selected views as a function of the number of processors, and the corresponding relative speedup. We observe that for up to 8 processors, the speedup of 75% and 50% selected views is better than the full cube. This is because the running time is not proportionally increased with the number of selected views, but at a faster rate. See Figure 3.6a. We also observe that when we use 16 processors, the speedup for 50% selected views drops below the speedup of full

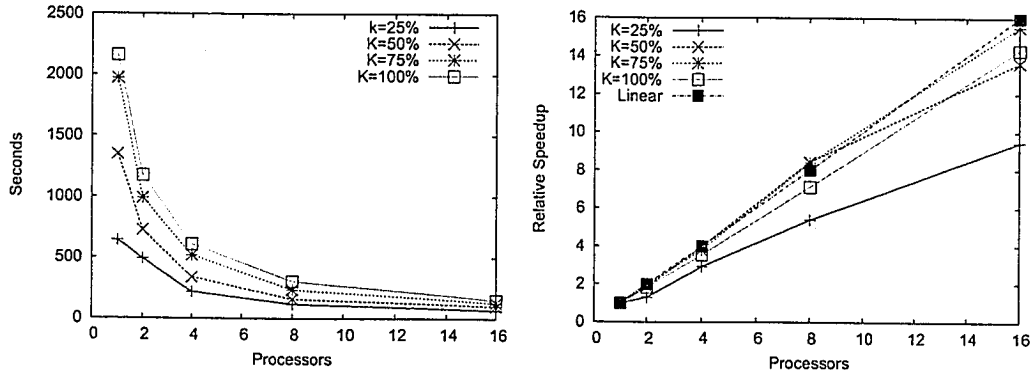


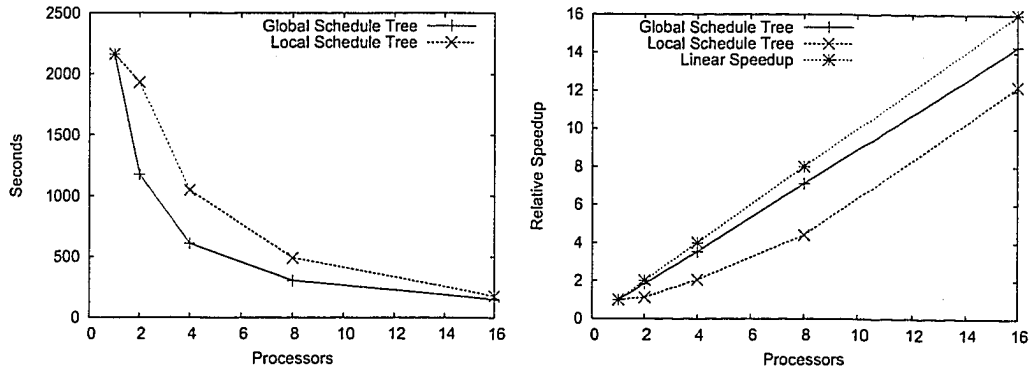

Figure 3.6: (a) Parallel wall clock time in seconds as a function of the processors for a range of different percentages of selected views and (b) corresponding speedup. (Fixed parameters: The data size $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. The balance threshold parameter $\gamma = 5\%$)

cube. This is because the computation for each processor is decreased faster for 50% selected views. However, for 50%, 75% selected views and full cubes, the speedup obtained is still close to the linear line. The reason for that is when the number of views selected gets very small, speedup falls off, as the local work within partitions is too small relative to the parallel overhead. In such cases, when there are only a handful of selected views, creating each view from an independent sort of the original data set may be preferable.

### 3.4.2 Local vs. global schedule trees

As described in Section 3.2.3, for shared-nothing parallel data cube construction it is possible for each processor to use either a local or a global schedule tree. Local schedule trees are built by each processor $P_j$ relative to their own data set $D_i$-root$||_j$, whereas a global schedule tree is built by a single processor, say $P_0$, relative to its data set $D_i$-root$||_0$, and then broadcast to all other processors.

The use of local schedule trees might appear at first preferable, since they are optimized relative to a processor's own data set. However, they have one serious drawback. When views of the same partition but for different subsets of data (i.e. on different processors) need to be merged, they need to have the same sort order or




(a) (b)

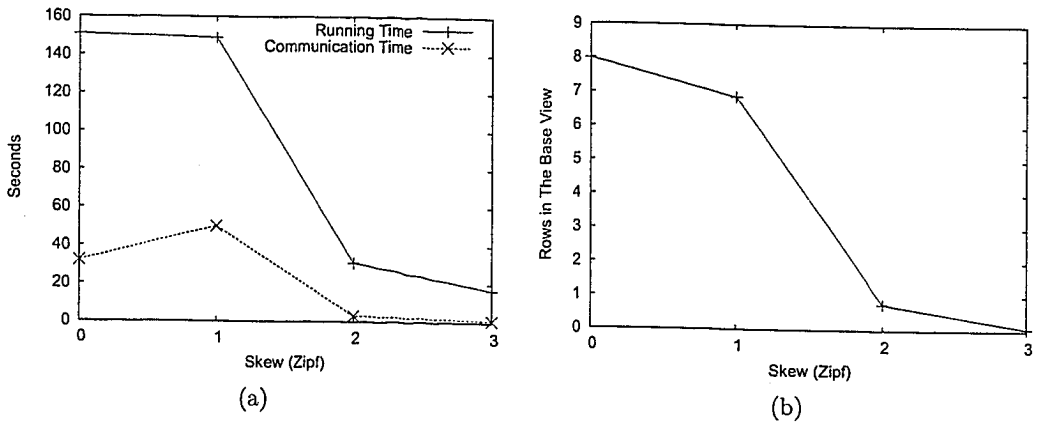
Figure 3.7: (a) Parallel wall clock time in seconds as a function of the number of processors for local schedule trees and global schedule trees and (b) corresponding speedup. (Fixed parameters: The data size $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. The balance threshold parameter $\gamma = 5\%$)

one of them has to be re-sorted. That re-sort creates a large amount of additional computation. As can be seen in Figure 3.7, our experiments indicate that global schedule trees offer better performance in practice. For the data sets that we tested, the additional work on some processors because of non-optimal schedule trees was significantly less than the overhead created through the need to re-sort views during Algorithm *MergePartitions($D_i$)*.

### 3.4.3 Data Skew

Data sets with skewed distributions can pose an interesting challenge to parallel data cube generation methods. As skew increases, data reduction tends also to increase, particularly in top-down generation methods [59, 17]. Data reduction is typically positive, as it reduces the total amount of work to be performed. However, if data reduction is not large and unevenly spread across the processors, it may unbalance the computation and cause the amount of data that has to be communicated to rise. To explore this issue we generated data sets using the standard ZIPF [66] distribution in each dimension with $\alpha = 0$ (no skew) to $\alpha = 3$ (high skew).

Figure 3.8a shows the impact of skew on parallel wall clock time and Figure 3.8(b) shows, for the same data sets, the rows of the base view, which indicates the data




(a) (b)

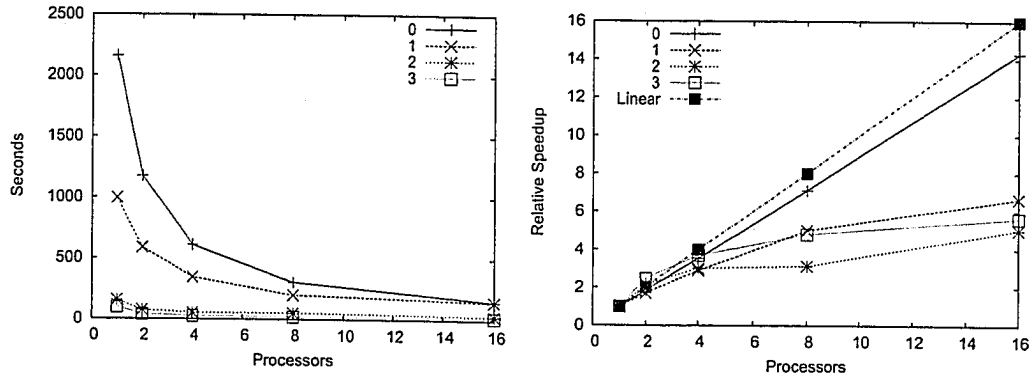
Figure 3.8: (a) Parallel wall clock time in seconds as a function of the skew of $\alpha = 0, 1, 2, 3$ for the running time and the communication time and (b) the corresponding size of the aggregated raw data. (Fixed parameters: The data size $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. The number of processors $p = 16$. The balance threshold parameter $\gamma = 5\%$)

reduction due to the skew data. We observe that, from $\alpha = 0$ to $\alpha = 1$ there is a rise in the communication time, which offsets gains from the reduced local computation time, so that the total running time of $\alpha = 1$ is almost equal to the time of $\alpha = 0$. For $\alpha > 1$ the data reduction is so significant that only very little data needs to be computed and communicated, so that the running time and the communication time drop significantly.

Figure 3.9 shows the running time of full cube generation with the skew for $\alpha = 0, 1, 2, 3$, and the corresponding relative speedup. We observe that the speedup for skewed data sets is close to linear on small numbers of processors, such as two or four and it drops sharply on 8 and 16 processors. The reason for this is both the data reduction for high skewed data sets and the insufficient local computation on each processor.

### 3.4.4 Cardinality of Dimensions

The cardinality of the dimensions in a data set can affect the performance of our method. As cardinalities increase so does the sparsity of the data set and this may adversely effect parallel time especially given that top-down methods [59, 17] are designed primarily for dense data cubes.

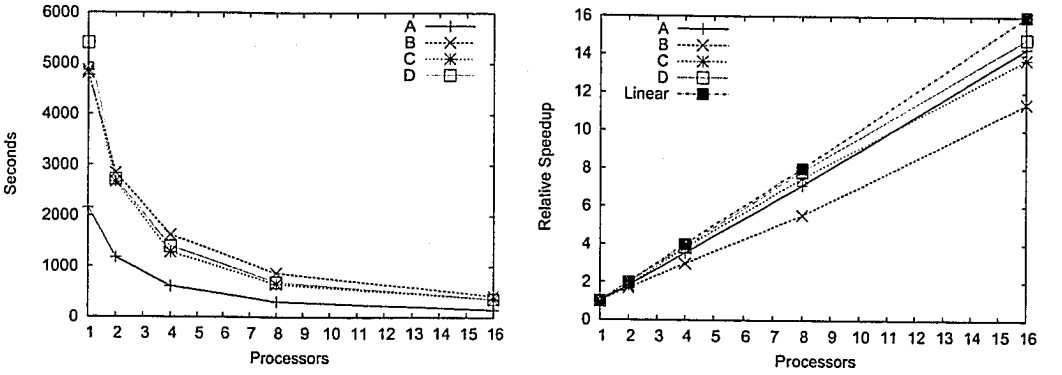



(a) (b)

Figure 3.9: (a) Parallel wall clock time in seconds as a function of the number of processors for the skew of $\alpha = 0, 1, 2, 3$ and (b) the corresponding relative speedup. (Fixed parameters: The data size $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. The number of processors $p = 16$. The balance threshold parameter $\gamma = 5\%$)

In Figure 3.10, we test our algorithm on four data sets: one dense data set ($|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$) and three sparse data sets ($|D_i| = 64, 128, 256$ for all dimensions respectively). In Figure 3.10(a), we observe the dense data set(Set A) requires less time than the sparse ones (Set B,C,D), because the sparse data sets generate much larger data cubes, and therefore cause larger I/O costs. We also observe that all three sparse data cubes require approximately the same amount of time. The reason for this is that each of these data sets includes only 8 million rows within a vastly large data space, there is little aggregation and therefore the resulting cubes are of approximately the same size.

In Figure 3.10(b), we observe that the speedup for Sets A, C, D is better than that observed for Set B, and close to linear. For Sets C, D, the carnalities are much larger than the number of processors, so that the workload associated with Sets C, D is better balanced, and less time is used in merging and shifting. For Set A, the carnalities of first two dimensions are much larger than the number of processors, and even though the carnalities of the rest dimensions are small, the views from the rest dimensions are also small, so that the workload of Set A is better balanced, and again less time is used in merging and shifting. For Set B, the carnalities are not much larger than the number of processors. In this case, it is hard to balance the



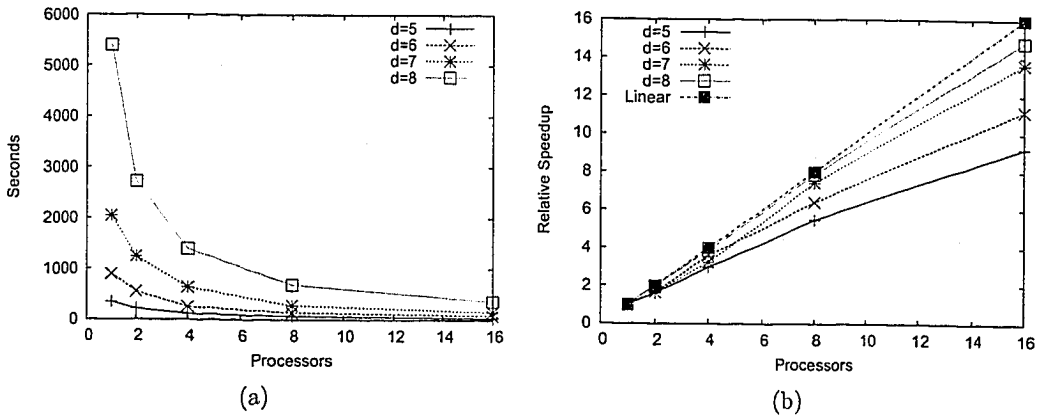

(a) (b)

Figure 3.10: (a) Parallel wall clock time in seconds as a function of the number of processors for data sets with different cardinality mixes, and (b) corresponding relative speedup. (Fixed parameters: Data size $n = 8$ million rows. Dimensions $d = 8$. Cardinalities and skews (A)$|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. (B)$|D_i| = 64$, $1 \leq i \leq d$. (C)$|D_i| = 128$, $1 \leq i \leq d$. (D)$|D_i| = 256$, $1 \leq i \leq d$. Percentage of views selected $k = 100\%$. Skew $\alpha = 0$. The balance threshold parameter $\gamma = 5\%$)

data across the processors, so the speedup is somewhat reduced. In general, as long as there is at least one dimension in which the cardinality is significantly greater than the number of processors, the algorithm appears to performance well.

### 3.4.5 Data Dimensionality

Figure 3.11(a) shows parallel wall clock time in seconds as a function of the dimensionality of the raw data set. Note that, the number of views that must be computed grows exponentially with respect to the dimensionality of the data set. In Figure 3.11a, we observe that the sequential running time grows faster than the linear speed. However, the parallel running time grows essentially linearly with respect to the increasing dimensions. The reason is that the data on each processor is smaller when we use more processors, and the memory size is the same as in the sequential version. So we benefit from more in-memory local computing. These factors cause the speedup to go up in Figure 3.11b as the dimensions increase. This experiment suggests our algorithm scales well on high dimensions.




(a) (b)

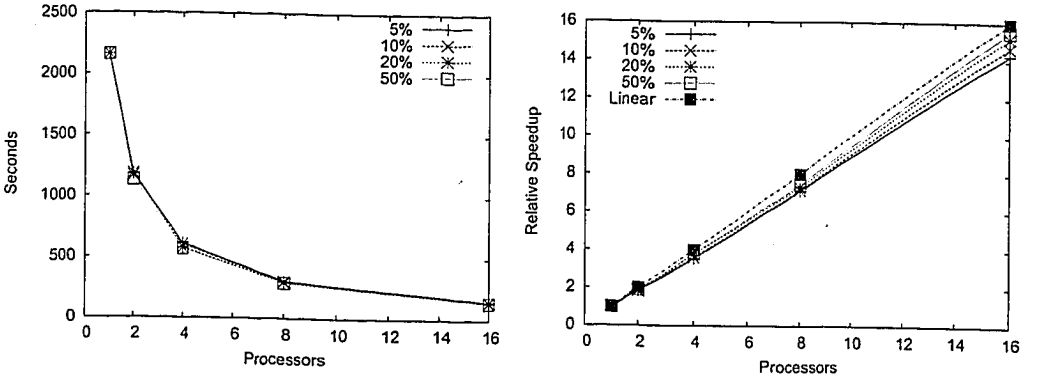
Figure 3.11: Parallel wall clock time in seconds as a function of the the number of dimensions. (Fixed parameters: Data size $n = 8$ million rows. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Percentage of views selected $k = 100\%$. Skew $\alpha = 0$. The balance threshold parameter $\gamma = 5\%$)

### 3.4.6 Balance Tradeoffs

One important feature of our shared-nothing data cube generation algorithm is that it balances *each* view in the generated data cube over the processors within a balance threshold $\gamma$. The more balanced each view is across the processors (i.e. the smaller $\gamma$) the more balanced any subsequent parallel computation on each view will be. However, the cost of selecting a small $\gamma$ is that it may cause more data movement and therefore increase the time required for data cube generation.

Figure 3.12 shows parallel wall clock time in seconds as a function of the number of processors for a range of different balance thresholds, as well as the corresponding speedup curves. We observe that increasing the balance threshold $\gamma$ has little effect on the parallel time. The reason for this is that our synthetic data sets are generated without skew ($\alpha = 0$), causing Algorithm 7 to partition the raw data sets evenly. Therefore the generated views are distributed across processors almost evenly, and the imbalance of most views is less than 5%, so that the larger $\gamma$ does not effect the running time greatly.

Figure 3.13 shows parallel wall clock time in seconds as a function of the number of processors for a range of different balance thresholds on a skewed data set, as well as the corresponding speedup curves. We observe that increasing the balance threshold $\gamma$ reduces the parallel time for this skewed data set. The reason for this is that data




(a) (b)

Figure 3.12: (a) Parallel wall clock time in seconds as a function of the number of processors for a range of different balance thresholds $\gamma$ and (b) corresponding speedup. (Fixed parameters: Data size $n = 8$ million rows. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Balance threshold $\gamma = 5\%$, 10%, 20%, and 50%.)

skew causes Algorithm 7 to partition the raw data sets unevenly, and then the output views are distributed across processors unevenly either, so that less views need to be shifted among processors when we choose larger $\gamma$.

From Figure 3.12 and Figure 3.13, a $\gamma$ of 5% appears to be a good threshold in practice. However, individual applications may want to tune this parameter according to their needs and the performance characteristics of their parallel machines.

## 3.5 Summary

In this chapter, we have presented the parallel algorithms for generating both full and partial data cubes that can be executed on shared-nothing clusters. We have implemented our parallel data cube algorithms and evaluated them on a Linux cluster, exploring relative speedup, local and global schedule trees, data skew, cardinality of dimensions, data dimensionality, and balance tradeoffs. These algorithms have achieved near linear speedup for a wide range of large input data sets. In the next chapter, we will extend our algorithms to handle much larger data data sets which are stored in extended memory.

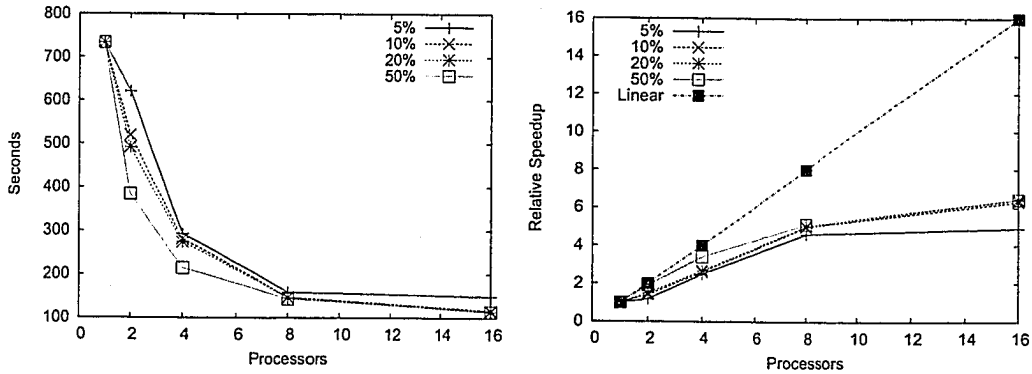



Figure 3.13: (a) Parallel wall clock time in seconds as a function of the number of processors for a range of different balance thresholds $\gamma$ and (b) corresponding speedup. (Fixed parameters: Data size $n = 8$ million rows. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Balance threshold $\gamma = 5\%, 10\%, 20\%$, and $50\%$.)

# Chapter 4

# External Memory and Parallel Generation of Full and Partial ROLAP Data Cubes

In the first half of this chapter, we present a sequential external memory adaptation of PipeSort. We introduce a shared prefix pipeline processing technique that speeds up pipeline processing by reducing the size of blocks that must be sorted. We also compare two approaches to external memory pipeline processing. In the second half of this chapter, we present a parallel external memory data cube generation method based on the sequential external memory PipeSort and parallel partitioning techniques described in Chapter 3. With the addition of external memory, the typical data sizes to be addressed increase very significantly. This makes the merge stage of the parallel algorithm even more critical. To improve the efficiency of the merge stage, we introduce an adaptive partitioning technique and evaluate the parallel algorithm extensively.

## 4.1 Introduction

Generating data cubes from large data sets entails reading and writing huge amounts of data, which cannot all be fit in memory. For example, for a fact table with 256 million rows and 8 dimensions the input data size is 9.2 gigabyte, and the output data size, assuming a full cube with $2^8 = 256$ views, is up to 200 gigabytes or 7 billion rows [24]. Available memory for generating data cubes is typical not large enough to hold complete data sets. For example, on our shared-nothing cluster, the memory on each node is only 512 megabyte. Therefore external memory parallel data cube generation algorithms are very important for OLAP systems in practice.

In the previous chapter, we presented our basic in-memory parallel data cube generation algorithm. In this chapter, we present an external memory adaptation of this method and a set of algorithmic enhancements. The enhancements address the I/O challenges that arise with the increases in data size associated with external

memory computations. We address two critical problems: 1) how to compute data cube efficiently in external memory and 2) how to reduce the associated high cost of disk I/O and network I/O.

In the sequential computation of data cubes, most of the time is spent executing a large number of pipelines. For example, to compute a $d$ dimension data cube, we have to execute at least $\binom{d}{\lceil d/2 \rceil}$ pipelines. Therefore the number of pipelines to be executed is exponential with $d$. For each pipeline, we need to read the input view from disk, sort and scan it to generate output views, and then output the results to disk. When available memory is not large enough, we have to read data block-by-block, and then output data block-by-block, plus perform time-consuming external merges. To compute the data cube efficiently in external memory, we introduce a shared prefix pipeline processing technique that speeds up pipeline processing by reducing the size of blocks that must be sorted. We also evaluate two approaches to external memory pipeline processing that differ in how to divide up the memory betweens views in pipelines and identify the approach that is more efficient in practice.

Partitioning is a key factor in achieving a good load balance in parallel algorithms. In our basic parallel data cube algorithm described in Chapter 3, we partition the input data evenly over processors. This leads to a merge stage in which a small amount of data must be communicated and merged using an in-memory merge procedure. In external memory data cube generation, the data to be communicated and merged may be much greater than the size of main memory. The merge stage has the potential to introduce high cost of disk I/O and network I/O due to the large amounts of data. For example, in internal memory data cube generation, to merge the data of a single view, we need to execute only one `MPI_ALL_TO_ALL_v` operation to exchange the data among processors and then execute only one in-memory merge. In external memory data cube generation, we need to read the data of a single view chunk-by-chunk, and for each chunk we have to execute one `MPI_ALL_TO_ALL_v` operation. After that, we shall need to execute external memory merge. Reducing the size of the data to be communicated and merged is clearly critical in the external memory case.

We observe that partitioning the input data evenly may actually increase the total time in situations where an even partitioning leads to more data than necessary being involved in the merge stage. To reduce the associated high cost of disk I/O and

network I/O, we may choose a slightly unbalanced partitioning in external memory parallel data cube generation. The unbalanced partitioning can reduce the merging time dramatically at the cost of a bit more sequential running time, and as a result the global running time may be reduced. In this chapter, we describe an adaptive data partitioning scheme, which uses a cost model to estimate the cost of computation, disk I/O and network I/O based on different parallel machines, and computes a "best partitioning" to reduce the the total running time.

## 4.2 External Memory Sequential Data Cube Generation

Recall from Chapter 3 that in the sequential computation of data cubes, most of the time is spent on executing pipelines. A pipeline consists of a number of views, connected by one sort edge and some pipeline edges. The sort edge connects the first two views, and the pipeline edges connect the rest of the views. In most cases, except perhaps for the first view, the other views in a pipeline share a prefix with each other. For example, in Figure 2.17 a pipeline $raw \Rightarrow CBAD \rightarrow CBA \rightarrow CB \rightarrow C$ is shown, where the sort edge "$\Rightarrow$" connects $raw$ and $CBAD$, and the pipeline edges "$\rightarrow$" connect the rest of views. Recall that, our general approach for processing a pipeline is to initially sort the first view to transform it into the second view, and then to scan the whole of second view to generate the other views in the remainder of the pipeline. As will be demonstrated in this section, this may not be the most efficient approach.

In the remainder of this section, we first introduce a shared prefix pipeline processing technique and give performance evaluation for this technique. Then we compare two approaches to external memory pipeline processing by both complexity analysis and experimental evaluation. Lastly, we combine the two techniques and give further experimental evaluation.

### 4.2.1 Shared Prefix Pipeline Processing

The first step in processing a pipeline is to sort the first view to the second view. In our implementation, we chose to use the quicksort algorithm. Its average complexity is $O(n \log(n))$, where $n$ is the number of rows to be sorted. The value of $n$ is typically very large in OLAP applications. Our goal here will be to avoid sorting the whole set

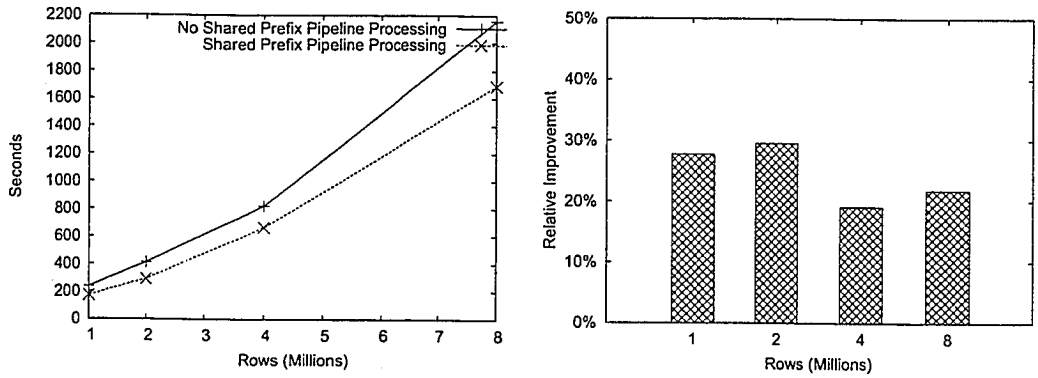
| ABDCE | ABCD | ABC | AB | A |
|---|---|---|---|---|
| 1 1 1 1 1,1 | Check | | | |
| 1 1 1 1 2,1 | 1 1 1 1,2 | Check | | |
| 1 1 1 2 2,1 | 1 1 1 2,1 | 1 1 1,3 | 1 1,3 | Check |
| 1 2 1 2 1,1 | Check | | | |
| 1 2 1 2 1,1 | 1 2 1 2,2 | 1 2 1,2 | Check | |
| 1 2 2 1 2,1 | Check | | | |
| 1 2 2 1 2,1 | 1 2 2 1,2 | Check | | |
| 1 2 2 2 1,1 | 1 2 2 2,1 | 1 2 2,3 | 1 2,5 | 1,8 |
| 2 1 1 1 1,1 | Check | | | |
| 2 1 1 1 1,1 | Check | | | |
| 2 1 1 1 2,1 | 2 1 1 1,3 | Check | | |
| 2 1 1 2 1,1 | 2 1 1 2,1 | 2 1 1,4 | 2 1,4 | Check |
| 2 2 1 2 1,1 | Check | | | |
| 2 2 1 2 1,1 | 2 2 1 2,2 | 2 2 1,2 | Check | |
| 2 2 2 2 2,1 | Check | | | |
| 2 2 2 2 2,1 | 2 2 2 2,2 | 2 2 2,2 | 2 2,4 | 2,8 |

Table 4.1: An Example of Shared Prefix Pipeline Processing

of $n$ rows at once. Instead, we will exploit the shared prefixes of view in a pipeline so as to only have to sort smaller blocks at any one time. Consider for example, the pipeline $ABDCE \Rightarrow ABCD \rightarrow ABC \rightarrow AB \rightarrow A$ illustrated in Table 4.1. Observe that the first two views share a common prefix namely $AB$. Instead of sorting all $n$=16 rows at one time we will partition the data set by $AB$ and sort each partition individually. In this case, instead of having to sort 16 rows at once, the maximum partition size we have to sort is five for the partition where $AB$=12. Clearly, this does not benefit us much on such a tiny input view, however when applied to a input view containing perhaps millions of rows we can have a very significant positive effect on running time. The other benefit here is that partitions will often fit in main memory where whole views will not. This approach is similar to the OVERLAP approach described in [17].

Suppose in a pipeline, the first two views, $A_1...A_lA_1...A_i$ and $A_1...A_lA_1...A_j$ have the same prefix, $A_1...A_l$, and $A_1...A_lA_1...A_i$ is sorted. We first partition $A_1...A_lA_1...A_i$ on $A_1...A_l$. Then for each partition, we sort $A_1...A_lA_1...A_i$ into $A_1...A_lA_1...A_j$ and scan the sorted partition to generate the other views at the same time. We call this approach *Shared Prefix Pipeline Processing.*

If the prefix shared by the two views is $A_1...A_l$, and then the size of a partition




(a) (b)

Figure 4.1: (a) Sequential running time in seconds as a function of the size of raw data, $n$ = from 1 million to 8 million rows and (b) corresponding relative improvement. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes.)

can be roughly estimated by $\frac{n}{\prod_{i=1}^{l}|A_i|}$ and the complexity of sorting a partition is approximately $O(\frac{n}{\prod_{i=1}^{l}|A_i|}\log(\frac{n}{\prod_{i=1}^{l}|A_i|})$. The complexity of finding partitions is $O(n)$ as we use a simple linear scan. So the complexity of shared prefix pipeline processing is

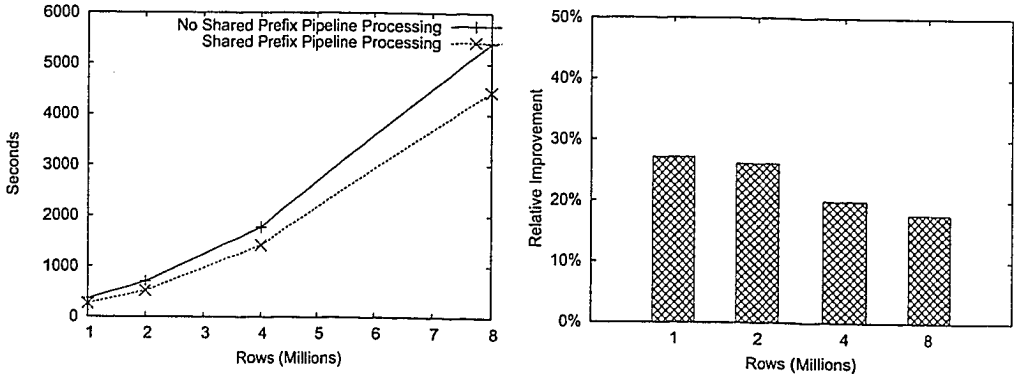
$$O(n\log(\frac{n}{\prod_{i=1}^{l}|A_i|})+n).$$

It is in practice smaller than $O(n\log(n))$.

Table 4.1 shows an example of executing a pipeline by shared prefix pipeline processing. In this example, the pipeline is $ABCDE \Rightarrow ABCD \rightarrow ABC \rightarrow AB \rightarrow A$. The first view $ABCDE$ and the second view $ABCD$ share the prefix of $AB$. We first partition $ABCDE$ on $AB$. Then for each partition, we sort $ABCDE$ into $ABCD$, and scan $ABCD$ to generate the other views.

Using the same experimental platform as in the Chapter 3, we implemented the shared prefix pipeline processing and compared it with the original method. We use a measure, called the *relative improvement*, to capture the effect of improvement. If the running time of the original method is $t$, and the running time of new method is $t_{new}$, then the relative improvement is defined as

$$\frac{t - t_{new}}{t}.$$



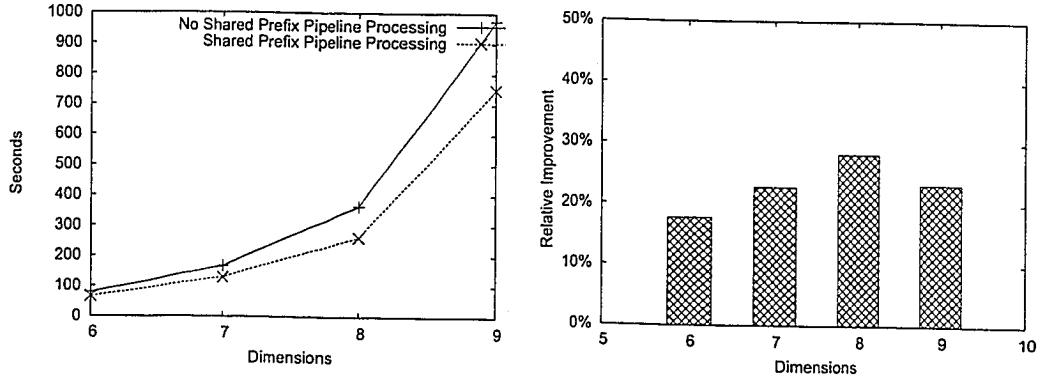

(a) (b)

Figure 4.2: (a) Sequential running time in seconds as a function of the size of raw data, $n$ = from 1 to 8 million rows and (b) corresponding relative improvement. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes.)

Figure 4.1(a) shows the running time of generating a full data cube using and not using shared prefix pipeline processing. Figure 4.1(b) shows the corresponding relative improvement. Overall we observe a relative improvement of between 20% and 30%. Note that the curve of relative improvement drops between 2M and 4M rows in Figure 4.1(b). This is likely because with a total memory of 100 megabytes, the maximal data which can fit in memory, is 100M/(8+1)/4=2.78M rows, assuming four bytes per dimension or measure. Therefore for 1M and 2M data sets, data fits in memory, while for 4M and 8M data sets, the sort must be done externally. The extra disk I/O reduces the improvement gained by shared prefix pipeline processing, although it still decreases the total running time.

Figure 4.2 shows the running time for using and not using shared prefix pipeline processing and the relative improvement for a full sparse data cube, when the cardinalities of all dimensions are 256. Again, we observe an overall average improvement of 20%-30%. Because of the same reason for Figure 4.1, we observe the curve of relative improvement drops between 2M and 4M rows.

Figure 4.3(a) shows the running time of generating a full data cube with different dimensions using and not using shared prefix pipeline processing. Figure 4.3(b) shows the corresponding relative improvement. We observe an overall average improvement for a range of number of dimensions. We observe the improvement for 9 dimensions

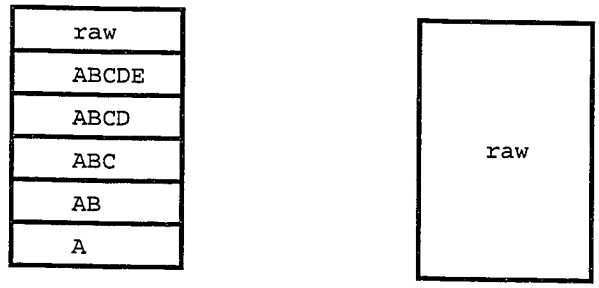



(a) (b)

Figure 4.3: (a) Sequential running time in seconds as a function of the dimensions of raw data, $d = 6, 7, 8, 9$ and (b) corresponding relative improvement. (Fixed parameters: The size of raw data $n = 1$ million rows. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes.)

drops. The reason for this is that because the running time increases so fast that reduce the improvement of shared prefix pipeline processing.

### 4.2.2 External Memory PipeSort




(a) (b)

Figure 4.4: (a) Buffer layout for SSEMPP and (b) Buffer layout for MSEMPP.

In data cube generation, the input data is often in practice too large to fit in memory. In such cases we have to read it from disk block-by-block, and then write the output views to disk block-by-block. After that, we also have to merge these blocks externally. These operations involve a lot of disk I/O, both for reading and

writing. Disk I/O is typically very slow compared with computation, so we must optimize our algorithms for their I/O efficiency as well as their computational efficiency. In the following, we compare two approaches to external memory PipeSort: Single Scan External Memory Pipeline Processing (SSEMPP) and Multiple Scan External Memory Pipeline Processing (MSEMPP).

Recall that our general approach for processing a pipeline of $l$ views is to sort the first/input view to transform it into the second view and then to scan the whole of the second view to simultaneously generate the other views in the remainder of the pipeline. This approach can be applied almost directly in external memory. The idea is to divide the available memory $M$ into a set of $l$ buffers $B_1 \ldots B_l$. Buffer $B_i$ will be used to store a portion of the $i$th view in the pipeline. The sizes of the buffers being chosen such that each buffer stores the same number of rows of its corresponding view. We call this approach *Single Scan External Memory Pipeline Processing (SSEMPP)*. See Figure 4.4(a) for the buffer layout of SSEMPP.

Let $c_i$ denote the number of bytes required to store a row of the $i$th view in a pipeline. Then the maximal number $n_{max}$ of rows for each view to be stored in memory is

$$n_{max} = \frac{M}{\sum_{i=1}^{l} c_i}.$$

If the number of rows of the input view $n \leq n_{max}$, all the data can fit in memory and then SSEMPP uses in-memory pipeline processing. If $n > n_{max}$, SSEMPP is extended to process pipelines externally. See Algorithm 9 for the steps of SSEMPP.

The advantage of SSEMPP is that it sorts and scans the input view only once to generate the rest of views in a pipeline simultaneously. The disadvantage is that each view in a pipeline is allocated a small part of the total available memory and $l$ files have to be open at the same time.

In *Multiple Scan External Memory Pipeline Processing (MSEMPP)*, the total available memory is allocated to the first/input view as its data buffer. After a block of data for the first view is read into memory, we sort and scan it to generate the second view, and then output the data of the second view to disk. Next we scan the second view to generate the third view and output the result to disk. It continues until all the views in a pipeline are generated. This approach needs one sort and $l-1$ scans to generate views one by one. Figure 4.4(b) for the buffer layout.

The maximal number $n_{max}$ of rows for the input view to be stored in memory is

$$n_{max} = \frac{M}{c_1}.$$

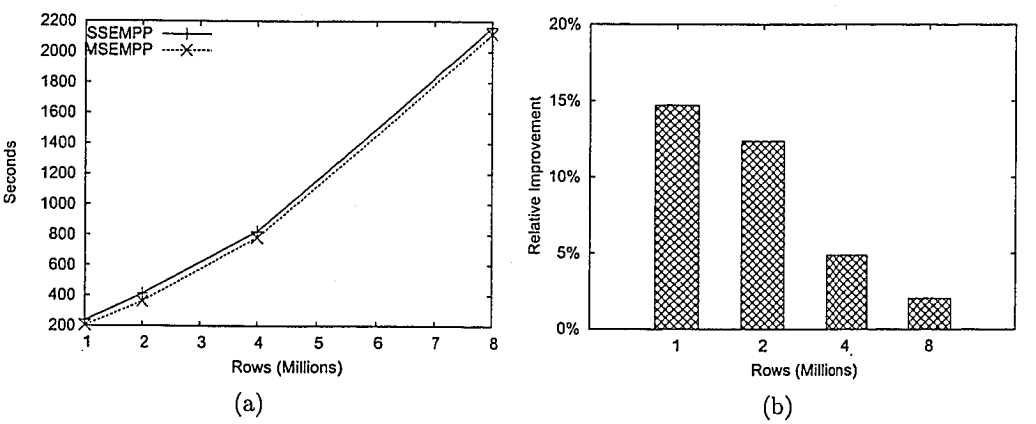
If the number of rows of the input view $n \leq n_{max}$, all the data can fit in memory and then MSEMPP uses in-memory pipeline processing. If $n > n_{max}$, MSEMPP is extended to process pipelines externally. See Algorithm 10 for the steps of MSEMPP.

The advantage of MSEMPP is that each view in a pipeline shares the total available memory and only two files are open at the same time. The disadvantage is that it needs to scan data multiple times.

Using the same experimental platform described in Section 2.4, we implemented SSEMPP and MSEMPP, and then compare them using the relative improvement defined as

$$\frac{t_s - t_m}{t_s},$$

where $t_s$ is the running time of SSEMPP and $t_m$ is the running time of MSEMPP.




(a) (b)

Figure 4.5: (a) Sequential running time in seconds as a function of the size of raw data, $n$ = from 1 to 8 million rows and (b) corresponding relative improvement. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes.)

Figure 4.5(a) shows the running time of generating a full data cube using SSEMPP and MSEMPP. Figure 4.5(b) shows the corresponding relative improvement and for all data sizes in this experiment the MSEMPP method is better than the SSEMPP

**Algorithm 9** Single Scan External Memory Pipeline Processing

**Input:** A set of pipelines, $PL$; the number of $PL$, $n_{pl}$; the maximal available memory $M$ in bytes.

**Output:** The views in the pipelines.

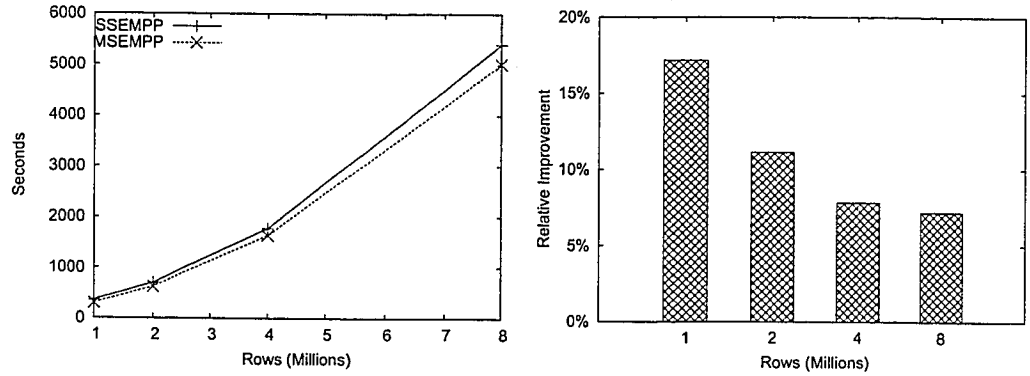
```
1: for i=1 TO n_pl do
2:   Calculate n_bmax for the ith pipeline, PL_i.
3:   if n_bmax ≥ ni, where ni is the number of rows of the input view in PL_i then
4:     Allocate ni row memory for each view in PL_i.
5:     Read all the data of the input view to memory.
6:     Sort and scan the data to generate the views in PL_i, and store the results in
       memory.
7:     Output the results to disks.
8:   else
9:     Allocate n_bmax row memory for each view in PL_i.
10:    while More data of the input view in PL_i do
11:      Read n_bmax rows of the input view into memory from disks.
12:      Sort and scan the data to generate the views in PL_i, and store the inter-
         mediate results in memory.
13:      Output the intermediate results to the separated files on disks.
14:    end while
15:    for Each output view V in the PL_i do
16:      Externally merge the separated files into one file using M memory.
17:    end for
18:  end if
19: end for
```

**Algorithm 10** Multiple Scan External Memory Pipeline Processing

**Input:** A set of pipelines, $PL$; the number of $PL$, $n_{pl}$; the maximal available memory $M$ in bytes.

**Output:** The views in the pipelines.

1: **for** i=1 TO $n_{pl}$ **do**
2: Let $ni_{max} = M/ci$.
3: **if** $ni \leq ni_{bmax}$, where $ni$ is the number of rows of the input view in $PL_i$ **then**
4: Allocate $ni$ row memory for the input view in $PL_i$.
5: Read all the data of the input view to memory.
6: Sort the data into the order of the second view in $PL_i$.
7: **for** $j = 1$ TO $l$ **do**
8: Scan and aggregate the $(j-1)$th view to generate the the $j$th view in $PL_i$, and output the result to disks.
9: **end for**
10: **else**
11: Allocate $ni_{max}$ row memory for each view in $PL_i$.
12: **while** More data of the input view in $PL_i$ **do**
13: Read $ni_{max}$ rows of the input view into memory from disks.
14: Sort the data into the order of the second view in $PL_i$.
15: **for** $j = 1$ TO $l$ **do**
16: Scan and aggregate the $(j-1)$th view to generate the the $j$th view in $PL_i$, and output the intermediate result to disks.
17: **end for**
18: **end while**
19: **for** Each output view in the $PL_i$ **do**
20: Externally merge the separated files into one file using $M$ memory.
21: **end for**
22: **end if**
23: **end for**




(a) (b)

Figure 4.6: (a) Sequential running time in seconds as a function of the size of raw data, $n$ = from 1 to 8 million rows and (b) corresponding relative improvement. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes.)

method. We observe the relative improvement decreases as the data size increases, likely because the total running time is increasing faster than the benefit from MSEMPP.

Figure 4.6 shows the running time and the relative improvement for a sparse data cube, as the cardinalities of all dimensions are 256. We also observe that the MSEMPP method is better than the SSEMPP method. Note that the values of improvement is larger than the values in Figure 4.5. The reason for this is that a sparse data cube includes more rows than a dense one, so that the both SSEMPP and MSEMPP need more disk I/O when the input raw data sets are same in size. Therefore MSEMPP can benefit more from larger buffers than SSEMPP.

Figure 4.7(a) shows the running time of generating a full data cube with different dimensions using SSEMPP and MSEMPP. Figure 4.7(b) shows the corresponding relative improvement. We observe that MSEMPP is significantly better than SSEMPP and the average improvement is around 15% for a range of numbers of dimensions.

From these experiments, MSEMPP always achieves better performance than SSEMPP. Therefore we choose the MSEMMP method in our external memory parallel algorithm.

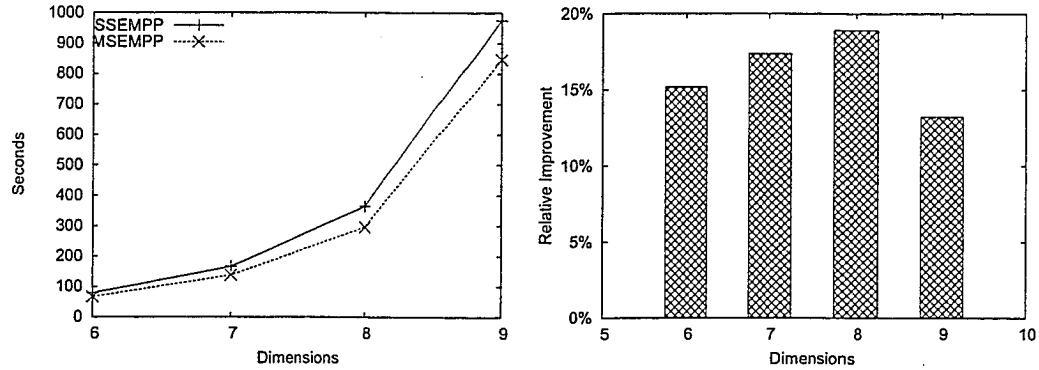



Figure 4.7: (a) Sequential running time in seconds as a function of the dimensions of raw data, $d = 6, 7, 8, 9$ and (b) corresponding relative improvement. (Fixed parameters: The size of raw data $n = 1$ million rows. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes.)

### 4.2.3 Sequential Relative Improvement

In the previous experiments, we evaluate the effects of shared prefix pipeline processing and the external memory PipeSort separately. From the experiments, we observe the improvement is affected by the combinations of some factors, such as the size of input data sets, the size of the total available memory, the sparsity of data cubes and the number of dimensions. Therefore we do not observe the constant improvement. Next we combine the two enhancements, shared prefix pipeline processing and MSEMMP, together in our sequential data cube generation algorithm, and compare them with the SSEMPP method without shared prefix pipeline processing to examine the relative improvement from the two enhancements together.

Figure 4.8(a) shows the running time of generating a full data cube using the basic approaches and the enhanced approaches. Figure 4.8(b) shows the corresponding relative improvement. We observe an improvement of up to 40% after enhancements for a range of the size of data although the relative improvement generally decreases as the data size increases.

Figure 4.9 shows the running time and the relative improvement for a sparse data cube, as the cardinalities of all dimensions are 256. We again observe an improvement of up to 40% after enhancements for a range of the data sizes.

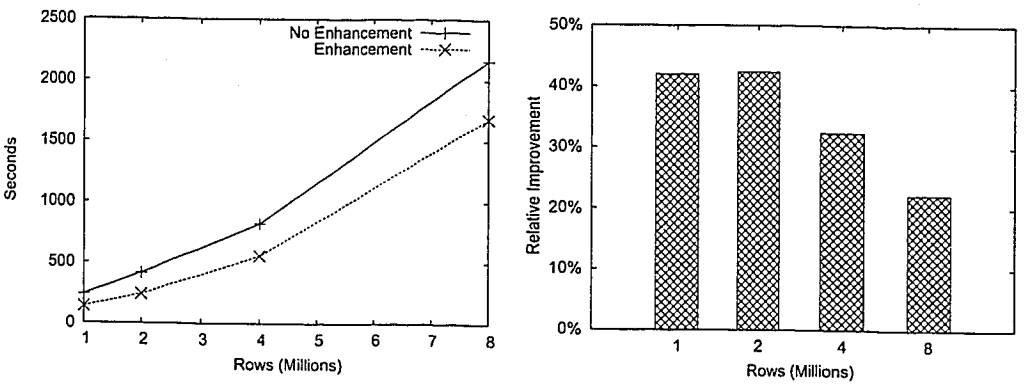



Figure 4.8: (a) Sequential running time in seconds as a function of the size of raw data, $n$ = from 1 to 8 million rows and (b) corresponding relative improvement. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes.)

Figure 4.10(a) shows the running time of generating a full data cube with different dimensions using the basic approaches and the enhanced approaches. Figure 4.10(b) shows the corresponding relative improvement. We observe an improvement of up to 45% for a range of number of dimensions.

On the whole, the above experiments shows that the two enhancements, shared prefix pipeline processing and MSEMMP, generates of up to 40% improvement in the running time of the sequential data cube generation for a range of values of the data size and the number of dimensions. In the next section, we will address the network I/O problem by introducing an adaptive data partitioning scheme.

## 4.3 External Memory Parallel Data Cube Generation

For parallel data cube generation, good data partitioning is a key factor in obtaining good performance on shared nothing clusters. In the last chapter, our basic parallel algorithm, Algorithm 6 partitions on all dimensions and then applys a parallel merge procedure. The challenge here is that for large data sets, external merge procedures based on fixed data partitioning schemes often lead to excess inter-processor communications which may greatly reduce the speedup achieved by the parallel system

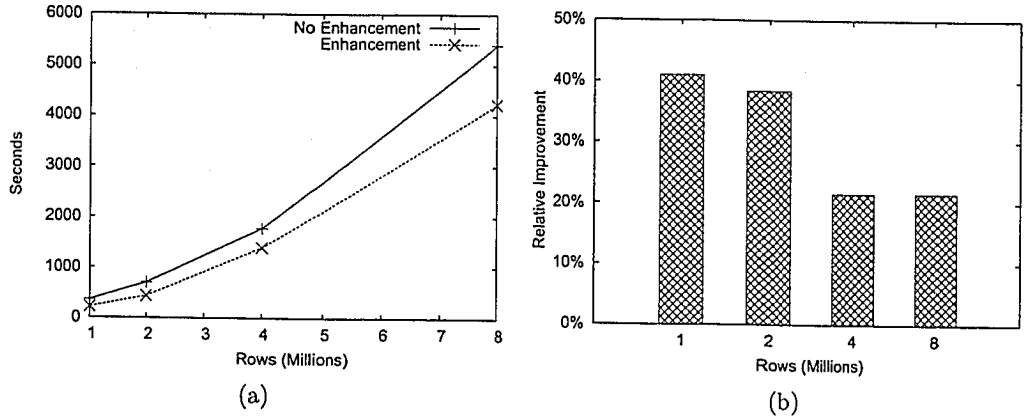



Figure 4.9: (a) Sequential running time in seconds as a function of the size of raw data, $n$ = from 1 to 8 million rows and (b) corresponding relative improvement. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes.)

and limit its effective scalability. In this section, we describe and evaluate an adaptive data partition scheme for parallel ROLAP data cube generation. This dynamic data partitioning scheme adapts to both, the current data set and the performance parameters of the parallel machine.

### 4.3.1 Algorithm Outline

Before discuss an adaptive data partitioning scheme, we first describe the external memory parallel data cube generation algorithm based on this partitioning scheme briefly; See Algorithm 11. Compared with Algorithm 6, Algorithm 11 has one more step, *adaptive data partitioning*, which uses the dynamic data partitioning scheme to shift partitions among processors to minimize the total cost. Another change in Algorithm 11 is that we do not need to balance the partitions using the $\gamma$ parameter in *AdaptiveSampleSort*, since adaptive data partitioning generates a better partitioning than using an arbitrary parameter $\gamma$. Next we discuss the adaptive data partitioning in more details.

**Algorithm 11** ExternalMemoryParallelSharedNothingDataCube

**Input:** Raw data set $R$ with $n$ rows and $d$ dimensions, distributed arbitrarily over the $p$ processors, $n/p$ records per processor.

**Output:** Data cube, $DC$, distributed over the $p$ processors. Each views is evenly distributed over the $p$ processors' disks.

1: **for** i=0 TO d-1 **do**

2: (1) Data Partitioning:

3: (a) Each processor $P_j$ ($j = 0 \ldots p-1$) computes locally the $D_i$-root for its subset of data. (Essentially a sequential sort followed by a sequential scan.) Let $D_i\text{-root}|_j$ denote the $D_i$-root created by processor $P_j$.

4: (b) Call AdaptiveSampleSort($D_i\text{-root}|_0, \ldots, D_i\text{-root}|_{p-1}$; $D_i, \ldots, D_{d-1}$), to sort $\cup_{j=0,\ldots p-1} D_i\text{-root}|_j$ by $D_i, \ldots D_{d-1}$.

5: (c) Each processor $P_j$ ($j = 0, \ldots p-1$) computes locally the $D_i$-root for its subset of data received in the previous step. Let $D_i\text{-root}||_j$ denote the $D_i$-root created by processor $P_j$.

6: (2) Adaptive Data partitioning:

7: (a) Processor $P_0$ locally computes, by applying the first phase of a sequential top-down data cube method, the schedule tree $T_i$ for building the $D_i$-partition with respect to $D_i\text{-root}||_0$.

8: (b) Processor $P_0$ broadcasts $T_i$ to $P_1$ ... $P_{p-1}$.

9: (c) Each processor, in parallel, execute AdaptivePartition($D_i\text{-root}||_j$) to obtain an optimized partitioning of $D_i\text{-root}||_j$ into $D_i\text{-root}|||_j$.

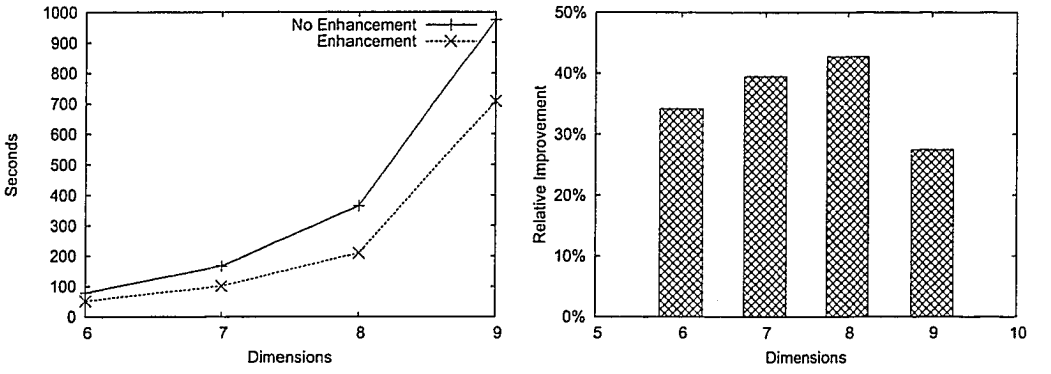
10: (3) Computation Of Local $D_i$-Partitions:

11: (a) Each processor $P_j$ ($j = 0, \ldots p-1$) computes locally the $D_i$-partition with respect to $D_i\text{-root}|||_j$ by applying the second phase of a sequential top-down data cube method to the schedule tree $T_i$ received in the previous step.

12: (4) Merge Of Local $D_i$-Partitions:

13: (a) Call MergePartitions($D_i$).

14: **end for**




(a) (b)

Figure 4.10: (a) Sequential running time in seconds as a function of the dimensions of raw data, $d = 6, 7, 8, 9$ and (b) corresponding relative improvement. (Fixed parameters: The size of raw data $n = 1$ million rows. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes.)

### 4.3.2 Approach of Adaptive Data Partitioning

The dynamic data partitioning scheme is based on a cost model, which is adaptive to parallel machines. For a given parallel machine, we introduce four performance parameters $t_{compute}$, $t_{read}$, $t_{write}$ and $t_{network}$ defined as follows: $t_{compute}$ is the average time in microseconds to fetch/compare/store a data item in main memory; $t_{read}$ is the average time in microseconds to read a data item from disk; $t_{write}$ is the average time in microseconds to write a data item to disk; $t_{network}$ is the average time in microseconds for communicating a data item between processors. For heterogeneous parallel machines (e.g. clusters with different generations of processors), the parameters $t_{compute}$, $t_{read}$ and $t_{write}$ can differ between processors. In this case, we choose the parameters for the slowest processor. The parameter $t_{network}$ depends on both, the network hardware and the number of processors used. Based on the above four parameters, we devise a cost model to estimate the time for communication and computation, and determine the best data partitioning for Algorithm 11. Before starting Algorithm 11, our software enters a test phase where it measures automatically the parameters $t_{compute}$, $t_{read}$, $t_{write}$ and $t_{network}$ for the given machine.

After the $i$-th iteration of Step 5 of Algorithm 11, the partitions of the $D_i$-root are well balanced over processors $P_j$ $(1 \leq j \leq p)$. However, as a result of the

global sort, subsequent items with the same sort key may end up on two different (subsequent) processors. This is especially the case when the cardinality of some dimensions is small, such as for attributes like gender, months and intervals for a numeric attribute. The situation is illustrated in Figure 4.11 for an attribute "A" with attribute values $a1, a2, \ldots, a10$. When the data is sorted by "A" in Step 2, each processor receives a range of data as indicated. Consider the range of items with value $a4$. Some items are on Processor 1 and some are on Processor 2. The problem is that during the merging of partitions in Step 12 of Algorithm 11, data movement occurs because Processor 2 has to send its items with value $a4$ to Processor 1. Instead, we could have made $a4$ the dividing line between the data between Processors 1 and 2 and moved all items with value $a4$ to Processor 2. We call this process "pivoting" and refer to $a4$ as the pivot. If we choose $a4$ as a pivot, then no data will have to be transferred between Processors 1 and 2 during the merging of partitions in Step 12 of Algorithm 11. However, on the negative side, choosing $a4$ as a pivot introduces an imbalance in data size between Processors 1 and 2, and other steps of Algorithm 11 may now have a longer computation time because of this imbalance, since the total computation time is always determined by the slowest processor.

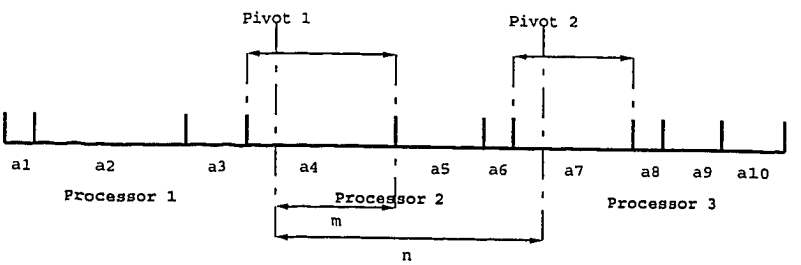



Figure 4.11: Data Partitioning and Pivots.

Our strategy is to choose pivots in such a way that we obtain the best tradeoff between lower communication due to less data movement and longer computation due to imbalance. We build a cost model to measure the impact of each possible pivot and choose the one with the lowest cost. We iterate this process until the total cost can be no further reduced.

### 4.3.3 The Cost Model

We now discuss our cost model for the performance of Algorithm 11 with respect to a chosen set of pivots. Note that the steps before optimizing partitioning are not impacted by pivots. Our model therefore measures only the performance of the three steps after optimization: shifting partitions (sub-step in optimizing partitioning), computing cubes, and merging cubes.

An important factor to be taken into consideration is the impact of external memory. For our implementation of Algorithm 11, views that are small enough to fit into main memory are created in memory for better speed, while larger views are built in external memory through disk scan and external memory sort; see Algorithm 10. In order to determine which version is used at run time, we calculate a maximal number of records, $n_{max}$ based on the total available memory $M$. If the number of records of a view is smaller than $n_{max}$, we calculate the cost according to a formula for internal memory computation. Otherwise, we calculate the cost according to a formula for external memory computation. For example, if $n_{max}$ is $1,000,000$, view $ABCD$ has $2,000,000$ records and view $BCD$ has $500,000$ records, then we process $BCD$ in main memory using the internal memory cost calculation and process $ABCD$ in external memory using the external memory cost calculation.

To calculate the cost of the three steps of Algorithm 11 for each view $v$, we use two basic numbers for each processor: $n$, the number of records stored at the processor and $m$, the number of moved records. Figure 4.11 illustrates $n$ and $m$ for Processor 2. The $n$ and $m$ values for $D_i$-root are obtained through a local linear scan. For every other view $v$ in $D_i$-partition, we estimate values $n_{est}$ and $m_{est}$ as follows: Set $n$ to the estimated view size calculated in Step 7 of Algorithm 11. Set $m = \frac{m_{D_i-root}}{n_{D_i-root}} n$ where $n_{D_i-root}$ and $m_{D_i-root}$ are the $n$ and $m$ values for $D_i$-root, respectively. Note that a record is composed of $d$ feature attributes and 1 measure attribute so that the size of a record is proportional to $d+1$.

We are now ready to analyze the three steps of Algorithm 11: shifting partitions (sub-step in Step 9), computing data cubes (Step 11), and merging data cubes (Step 13). For each step, we will give the cost for internal and external memory calculation and outline our rationale for the given formulae.

| Step | Internal Memory | External Memory |
|---|---|---|
| Scanning | $n(d/2) * t_{compute}$ | $n(d+1) * t_{read} + n(d/2) * t_{compute}$ |
| Exchanging | $m(d+1) * t_{network}$ | $m(d+1) * t_{network}$ |
| Merging | $n(d/2) * t_{compute}$ | $n(d+1) * t_{write} + n(d/2) * t_{compute}$ |

Table 4.2: The Costs of Shifting Partitions

| Step | Internal Memory | External Memory |
|---|---|---|
| Sorting | $n \log n * t_{compute} + n(d/2) * t_{compute}$ | $n(d+1) * t_{read} + n \log n * t_{compute} + n(d/2) * t_{compute}$ |
| Scanning | $n(d+1) * t_{compute}$ | $n(d+1) * t_{write} + n(d+1) * t_{compute}$ |

Table 4.3: The Costs of Computing Data Cubes

### Shifting Partitions

Table 4.2 lists the costs of shifting partitions. This step shifts partitions of root views among processors. It consists of three sub-steps: scanning data, exchanging data and merging data. Each processor scans the local data and compares each row with the pivots considered. To compare a row with a pivot, we compare dimension values one by one. In the best case, only one comparison is needed, and $d$ comparisons in the worst case, where $d$ is the number of dimensions. The average number of comparisons is $d/2$. In the external memory version, the cost for reading data from disk is $n(d+1) * t_{read}$, where $n(d+1)$ is the number of item in $D_i$-root since each row contains $d+1$ items. In both versions, the communication cost is $m(d+1) * t_{network}$, where $m(d+1)$ is the number of items moved across the network. The cost of the last sub-step is $n(d/2) * t_{compute}$. For the merging, the number of comparisons is a function of both, $n$ and $m$. However $m$ is much smaller than $n$ and we ignore $m$ in order to simplify calculations. In the external version, the cost for writing the data to disks is $n(d+1) * t_{write}$. Note that, this is also an approximation since data is exchanged between processors.

### Computing Data Cubes

Table 4.3 lists the costs of computing data cubes. Step 11 of Algorithm 11 calculates the schedule tree used to generate the views. As described in [59], we compute pipelines one by one. For each pipeline, the first view is sorted and the remaining

| Step | Internal Memory | External Memory |
|---|---|---|
| **Scanning** | $n(d/2) * t_{compute}$ | $n(d+1) * t_{read} + n(d/2) * t_{compute}$ |
| **Exchanging** | $m(d+1) * t_{network}$ | $m(d+1) * t_{network}$ |
| **Merging** | $n(d/2) * t_{compute}$ | $n(d+1) * t_{write} + n(d/2) * t_{compute}$ |

Table 4.4: The Costs of Merging Data Cubes

views are generated by scanning. For example, in Figure 3.2, the schedule tree for the 1-subcube consists of a pipeline, $ABCD \Rightarrow BCD \rightarrow BC \rightarrow B$. The cost of sorting is $n \log n * t_{compute} + n(d/2) * t_{compute}$ [32] for the internal memory version. The external version includes an additional cost for disk reading: $n(d+1) * t_{read}$. The cost for scanning is $n(d/2) * t_{compute}$ for the internal memory version, plus $n(d+1) * t_{write}$ for the external memory version.

### Merging Data Cubes

Step 13 of Algorithm 11 merges $D_i$-partition between processors. The cost calculation is similar to the calculation for *Shifting Partitions*. See Table 4.4.

### 4.3.4 Algorithm of Adaptive Data Partitioning

Based on the above cost model, we may evaluate possible partitioning and choose an optimal partition with minimum cost. Algorithm 12 shows our method to select a set of pivots and shift the partitions. The function $Cost()$ represents the cost function for a given set of pivots as discussed above. Algorithm 12 first calculates the cost of the partitioning generated by Steps 2 and 3 of Algorithm 12 without any pivots. We then select pivots, calculate the cost based on those pivots and update the partitioning if the new cost is smaller than the old one. This process will continue until the cost can not be reduced any further. Unfortunately, the number of possible pivot combinations is very high. For $p$ processors, the maximum number of possible pivots is $p-1$. Each pivot can either be not selected or selected for its left adjacent processor (all data move left) or its right adjacent processor (all data move right). Hence, the total number of possible data partitioning is $3^{p-1}$. If we have 32 processors in a cluster, the total number of partitioning is $3^{32-1} = 617,673,396,283,947$. In Algorithm 12, we choose a

greedy method to reduce the cost as much as possible. In each iteration of the repeat-until loop, we choose the pivot which generates the greatest cost reduction among all possible remaining pivots. We update the partitioning and the cost, and search again until we cannot reduce the cost further by adding another pivot. Algorithm 12 then re-partitions $D_i$-root, using the chosen set of pivots.

---

**Algorithm 12** AdaptivePartition($D_i$-root$||_j$)

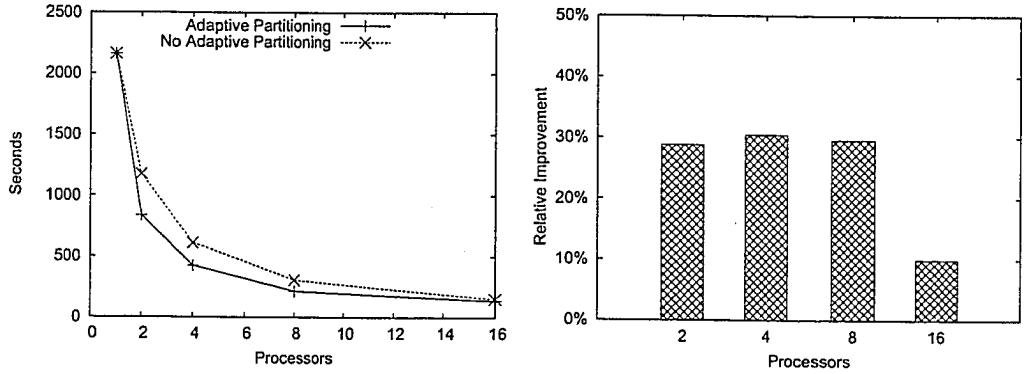
---

**Input:** $D_i$-root$||_j$, the globally sorted root view distributed on Processor $j$.

**Output:** $D_i$-root$|||_j$, the globally sorted root view distributed on Processor $j$. Its cost is the smallest among all possible partitioning.

1: Each processor $P_j$ collects locally, for its data set $D_i$-root$||_j$, the partitioning information (pivots and their $n$, $m$ values) required for the evaluation of the function $Cost()$. The partitioning information is broadcast to all processors.
2: Each processor $P_j$ computes $cost = Cost$(current partition without pivots).
3: done = FALSE.
4: **repeat**
5: **for** each processor $P_j$ **in parallel do**
6: Processor $P_j$ calculates the new cost $cost_j^{new}$ obtained by adding pivot $j$, (moving the respective data to the left or right processor, whichever is lower cost).
7: **end for**
8: Let $cost^{new} = Min(cost_1^{new}, cost_2^{new}, ..., cost_{p-1}^{new})$
9: **if** $cost^{new} < cost$ **then**
10: update partition by adding the chosen pivot.
11: $cost = cost^{new}$
12: **else**
13: done = TRUE
14: **end if**
15: **until** done
16: $D_i$-root$||_j$ is re-partitioned using the chosen set of pivots to generate $D_i$-root$|||_j$.

---



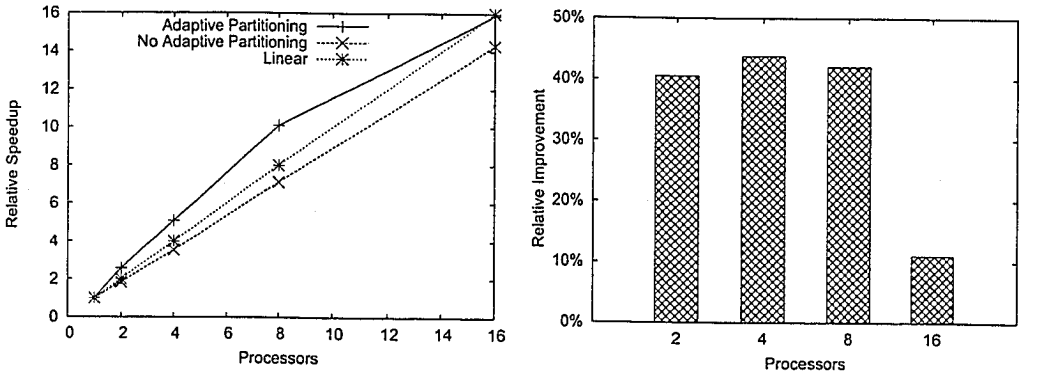

(a) (b)

Figure 4.12: (a) Parallel wall clock time in seconds as a function of the number of processors before and after the optimized partitioning and (b) corresponding relative improvement. (Fixed parameters: The size of raw data, $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes. Balance threshold $\gamma = 5\%$)

### 4.3.5 Experimental Evaluation of Adaptive Data Partitioning

Using the experimental platform described in Section 2.4, we implemented the adaptive partitioning described in the previous section and compared it with the fixed partitioning scheme described in Chapter 3. In order to isolate the improvement which is generated only from the adaptive partitioning scheme, we do not use shared prefix pipeline processing or MSEMPP in comparisons and use SSEMPP to handle external memory PipeSort in this evaluation. In the next section, we will evaluate all of the enhancements together.

Our implementation of adaptive partitioning initially runs a performance test to calculate the key machine specific cost parameters, $t_{compute}$, $t_{read}$, $t_{write}$ and $t_{network}$, that drive our dynamic data partitioning method. On our experimental platform these parameters were as follows: $t_{compute} = 0.0293$ microseconds, $t_{read} = 0.0072$ microseconds, $t_{write} = 0.2730$ microseconds. The network parameter, $t_{network}$, captures the performance characteristics of the MPI `MPI_ALL_TO_ALL_v` operation on a fixed amount of data relative to the number of processors involved in the communication. On our experimental platform, $t_{network} = 0.0551, 0.0873, 0.1592, 0.2553$ microseconds for $p = 2$, 4, 8 and 16, respectively.



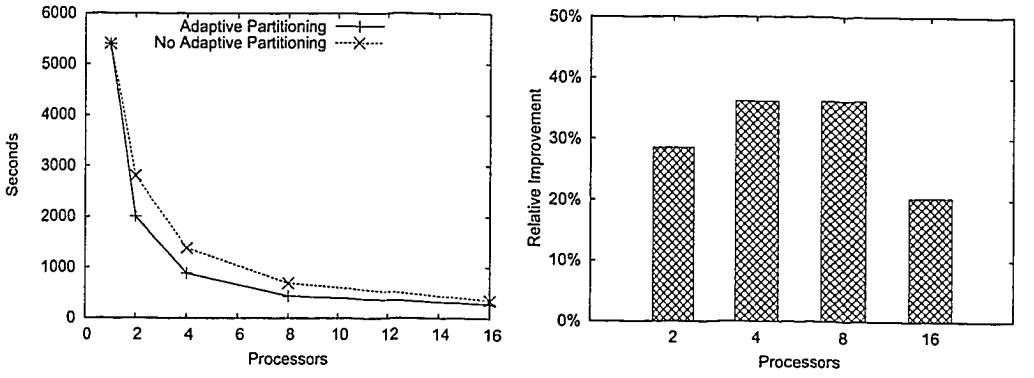

(a) (b)

Figure 4.13: (a) Relative speedup as a function of the number of processors before and after the optimized partitioning and (b) corresponding relative improvement. (Fixed parameters: The size of raw data, $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes. Balance threshold $\gamma = 5\%$)

The experiments include the two groups. In the first group, we use the cardinalities, $256, 128, 64, 32, 16, 8, 4, 2$ for 8 dimensions. In the second group, we use 256 for every dimensions. In both groups, we chose 8 million rows for the size of the raw data. Therefore, the data cube in the first group is a dense one, and it consists of fewer rows because of more aggregation. In the second group, the data cube is sparse, and consists of more rows because less aggregation takes place.

Figure 4.12 and Figure 4.13 are for the dense data cubes. Figure 4.12(a) shows parallel wall clock time in seconds as a function of the number of processors before and after using the adaptive partitioning scheme, and Figure 4.12(b) shows the corresponding relative improvement. We observe the relative improvement is approximately 30% up to eight processors and then falls to about 10% for 16 processors. The reason is that the adaptive partitioning scheme tries to reduce the time for merging, but when we use 16 processors, the data on each processor becomes so small that the time for merging is a small part of the total time. This decreases the benefit from the adaptive partitioning scheme.

Figure 4.12 focuses on the relative improvement in total time, however it is also interesting to consider the relative improvement in speedup for the same data sets. Figure 4.13(a) shows for the same data set relative speedup as a function of the



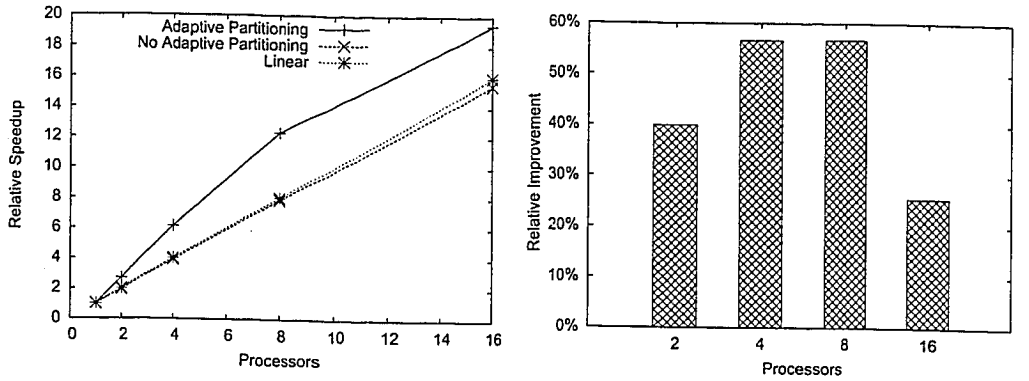

(a) (b)

Figure 4.14: (a) Parallel wall clock time in seconds as a function of the number of processors before and after the optimized partitioning and (b) corresponding relative improvement. (Fixed parameters: The size of raw data, $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes. Balance threshold $\gamma = 5\%$)

number of processors before and after using the adaptive partitioning scheme, and Figure 4.13(b) shows the corresponding relative improvement of Figure 4.13(a). We observe the same trend as in Figure 4.12. The values of the improvement are around 40% for 2-8 processors and the relative speedup *super-linear*, although the improvement decreases at 16 processors. The reason for super-linear is that we are increasing not only the number of processors but also the ratio of available memory to data sizes. The benefit from adding processors and memory is larger than the cost of the extra communication required.

Figure 4.14 and Figure 4.15 are for the sparse data cubes. Figure 4.14(a) shows parallel wall clock time in seconds as a function of the number of processors before and after using the adaptive partitioning scheme, and Figure 4.14(b) shows the corresponding relative improvement. Again we observe the same basic trends. The improvement increases first and then decreases with the processors increases. However, the values of the improvement is larger than those in Figure 4.12, for the dense cube cases. The data cube in this figure is sparse, so that it consists of more rows than a dense cube and it needs more time in merging among processors than a dense cube when no adaptive partitioning is used.

For the same data size as Figure 4.14, Figure 4.15(a) shows the relative speedup as



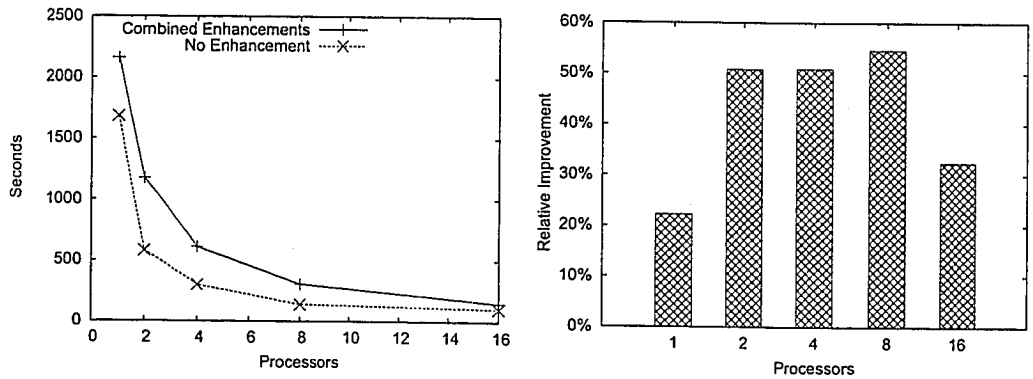

(a) (b)

Figure 4.15: (a) Relative speedup as a function of the number of processors before and after the optimized partitioning and (b) corresponding relative improvement. (Fixed parameters: The size of raw data, $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes. Balance threshold $\gamma = 5\%$)

a function of the number of processors before and after using the adaptive partitioning scheme, and Figure 4.15(b) shows the corresponding relative improvement. We still observe the same trends as in Figure 4.14. The improvement increases first and decrease with the processors increases. Also we observe that all the speedup points are above the linear line, and the improvement is up to more than 50% for 4 and 8 processors. It suggests that the adaptive partitioning scheme can dramatically reduce time on the communication time among processors when large amount of data need merging.

### 4.3.6 Experimental Evaluation of Combined Enhancements

Using the experimental platform described in Section 2.4, we combined and implemented the three enhancements: shared prefix pipeline processing, MSEMPP and the adaptive partitioning scheme, together in the external memory parallel data cube algorithm described in Algorithm 11. We also implemented a basic external memory parallel data cube method without any enhancement and in this method we used SSEMPP to handle external memory PipeSort. We compare these two methods for full cube generation, partial cube generation and cube generation on skewed data sets to evaluate the relative improvement from the three enhancements.

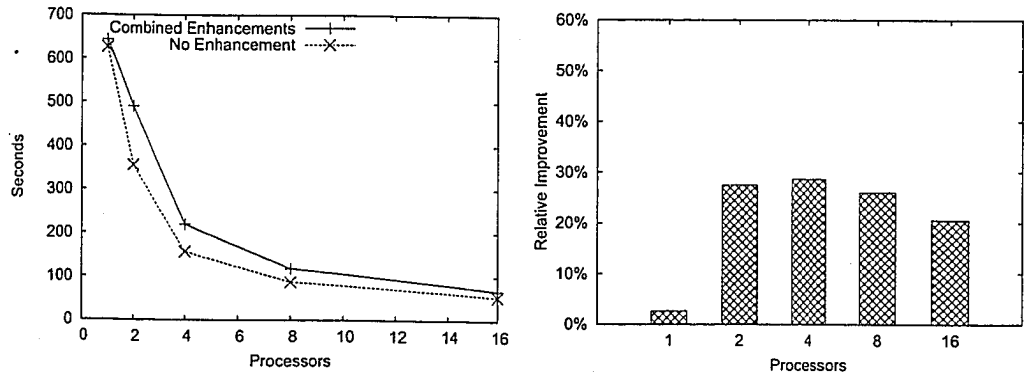



(a) (b)

Figure 4.16: (a) Parallel wall clock time in seconds as a function of the number of processors before and after the optimizations and (b) corresponding relative improvement. (Fixed parameters: The size of raw data, $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes. Balance threshold $\gamma = 5\%$)

Figure 4.16(a) shows the full cube generation running time of two methods and Figure 4.16(b) shows the corresponding relative improvement. We observe that the improvement for one processor is about 20%, however the improvement for multiple processors is up to 55%. The sequential cube generation only uses two enhancements: shared prefix pipeline processing and MSEMPP, while the parallel cube generation uses the adaptive partitioning scheme plus the above two enhancements. So we may observe that the improvement of the parallel generation is better than the sequential generation. Also, we observe that the improvement dips for 16 nodes to 30%. This dip is due to the fact that the relatively smaller data sizes per processor at 16 nodes cannot fully hide the additional communication costs.

Figure 4.17(a) shows for the same data as Figure 4.16 for the partial cube generation and Figure 4.17(b) shows the corresponding relative improvement. In this experiment, only 25% of the views are selected, so that the running time drops to 650 seconds from 2200 seconds for the sequential computation. We observe that the improvement curve is very similar to the one in Figure 4.16, although the improvement of the sequential generation is smaller due to the small size of the selected views. Also the overall improvement of the parallel generation drops to about 30%, smaller than




(a) (b)

Figure 4.17: (a) Parallel wall clock time in seconds as a function of the number of processors before and after the optimizations and (b) corresponding relative improvement. (Fixed parameters: The size of raw data, $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Percentage of views selected $k = 25\%$. The memory size $M = 100$ Megabytes. Balance threshold $\gamma = 5\%$)

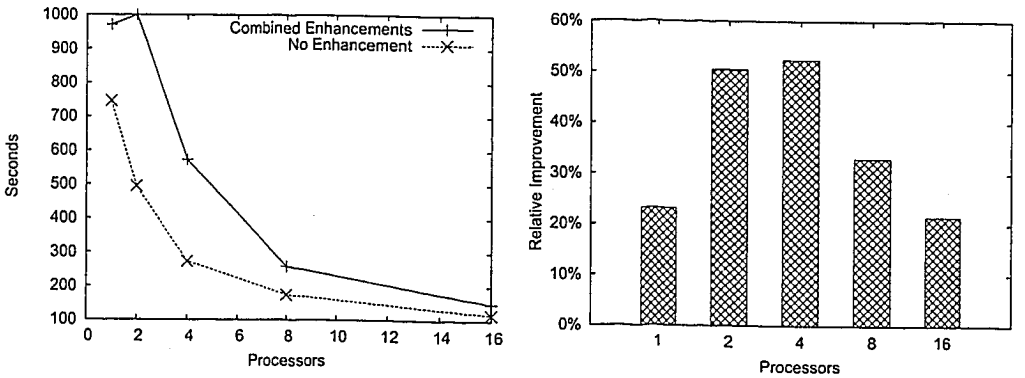
50% observed in Figure 4.16.

Figure 4.18(a) shows the full cube generation running time of the two methods and Figure 4.18(b) shows the corresponding relative improvement when the input data set is skew with $\alpha = 1$. We observe that the sequential running time drops sharply from 2200 seconds to 1000 seconds. We also observe that the improvement curve is also very similar to the one observed in Figure 4.16.

## 4.4 Performance Evaluation

We implemented the external memory parallel data cube generation algorithms with all the three enhancements, and evaluate them using the experiment platform described in Section 2.4 from the following aspects:

- Speedup. We evaluate the speedup of the enhanced external memory algorithms. Since the running time for the sequential cube generation is reduced dramatically, the speedup could decrease. However the running time for the parallel cube generation is reduced due to the adaptive partitioning, so the speedup could increase. From the experiments, we will find out the speedup trend under the effects of both sequential and parallel enhancements.




(a) (b)

Figure 4.18: (a) Parallel wall clock time in seconds as a function of the number of processors before and after the optimizations and (b) corresponding relative improvement. (Fixed parameters: The size of raw data, $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 1$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes. Balance threshold $\gamma = 5\%$)

- Scaleup. It is another key metric for evaluation our parallel algorithms. It indicates whether a constant running time can be maintained as the workload is increased by adding a proportional nodes.

- Sizeup. It is similar to scaleup but fixes the number of processors. It indicates whether a proportional running time can be maintained as the workload is increased.

- Data Dimensionality. We fix the size of data and the cardinality of each dimension, and then increase dimensions. By this way, we may evaluate the effects of dimensionality on our algorithms.

- Cardinality of Dimensions. We fix the size of data and the dimensions, and then change the cardinalities of dimensions. So that we may evaluate the effects of the sparsity of data cubes.

- Data Skew. We fix the size, dimensions and cardinalities of data, and then change the Zipf value to generate skew data sets. Therefore we may evaluate the effects of skew data on our algorithms.

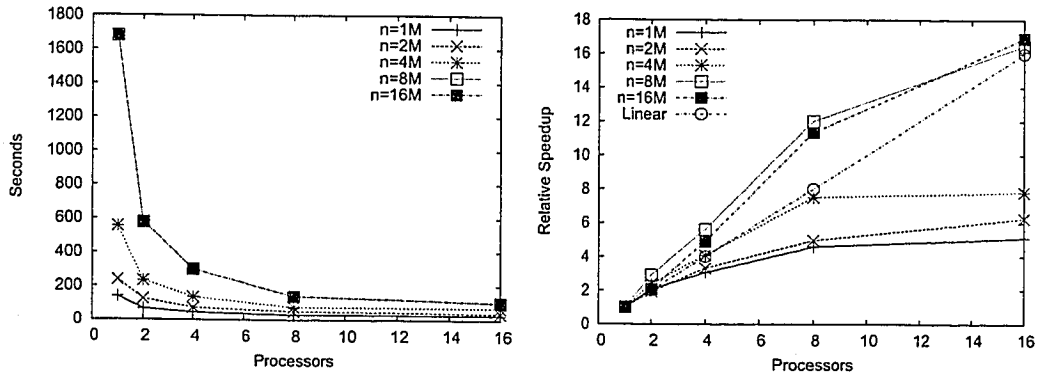
In the following experiments all sequential times were measured as wall clock times in seconds. All parallel times were measured as the wall clock time between the start of the first process and the termination of the last process. We will refer to the latter as *parallel wall clock time.* All times include the time taken to read the input from files and write the output into files. Furthermore, all wall clock times were measured with no other users on the machine.

Throughout these experiments, as we increased the number of processors we observed two countervailing trends. Increasing processors, while holding total data size constant, leads to less data per processor and therefore better relative speedup because each processor can fit more of its data in memory, thereby reducing disk related overheads. On the other hand, using standard 100M LAN and a standard MPI implementation, increasing the number of processors reduces the speed of communication, even when total data size communicated is held constant, and therefore tends to reduce relative speedup. The large super linear effects are observed in some of these experiments when the benefits of fitting data in memory outweigh the penalties associated with higher communication overheads.

### 4.4.1 Speedup

Speedup is one of key metrics for evaluation of parallel database systems [30] as it indicates the degree to which adding processors decreases the running time. The relative speedup for $p$ processors is defined as $S_p = \frac{t_1}{t_p}$, where $t_1$ is the running time of the parallel program using one processor, all communication overhead having been removed from the program, and $t_p$ is the running time using $p$ processors. An ideal $S_p$ is $p$, which implies that $p$ processors are $p$ times faster than one processor, so the curve of an ideal $S_p$ is a linear line.

Figure 4.19(a) shows the running time for external memory full cube generation for the data size of from 1M to 16M rows and Figure 4.19(b) shows the relative speedup. We observe that the speedup for 8M and 16M is super-linear. The speedup for 4M is close to the linear line on 2-8 nodes, and the speedup for 2M is good on 2-4 nodes. Basically, we observe good speedup when there is enough data (0.5M per processor in Figure 4.19) on each processors. When the input data size is too small, such as 1M or 2M rows, local computing costs are low and insufficient to hide the



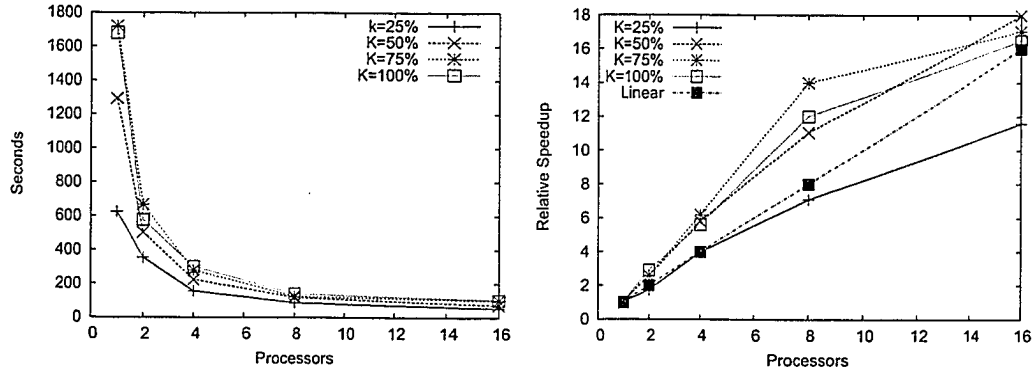

(a) (b)

Figure 4.19: (a) Parallel wall clock time in seconds as a function of the number of processors for the data size $n$ = from 1 million to 16 million rows and (b) corresponding speedup. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256$, 128, 64, 32, 16, 8, 4, 2. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes. The balance threshold parameter $\gamma = 5\%$)

communication overhead so the speedup on the large number of processors is poor for these small tasks.

In many applications, users do not require all of the $2^d$ views contained in a full data cube but rather only a selected subset. The challenge for a partial cube generation method is to efficiently construct the set of selected views, maintaining relative efficiency even as the number of views (and therefore total work) is decreased. Figure 4.20(a) shows the running time for full cube generation for the selected views of from 25% to 100% and Figure 4.20(b) shows the relative speedup.

We observe that for up to 8 processors, the speedup of 75% and 50% selected views is close to or better than the full cube. That is because the running time is not proportionally increased with the percentage of selected views, but at a faster rate. See Figure 4.20a. We also observe that when we use 16 processors, the speedup for 50% and 75% selected views drops below the speedup of full cube. That is because the local computation for each processor is decreased faster for 50% and 75% selected views than a full cube when we use 16 nodes. Except 25% selected views, the other speedup obtained is super linear speedup, because the benefits of fitting data in memory outweigh the penalties associated with higher communication overheads for 50%, 75% selected views and a full cube. For 25% selected views, the speedup is close




(a) (b)

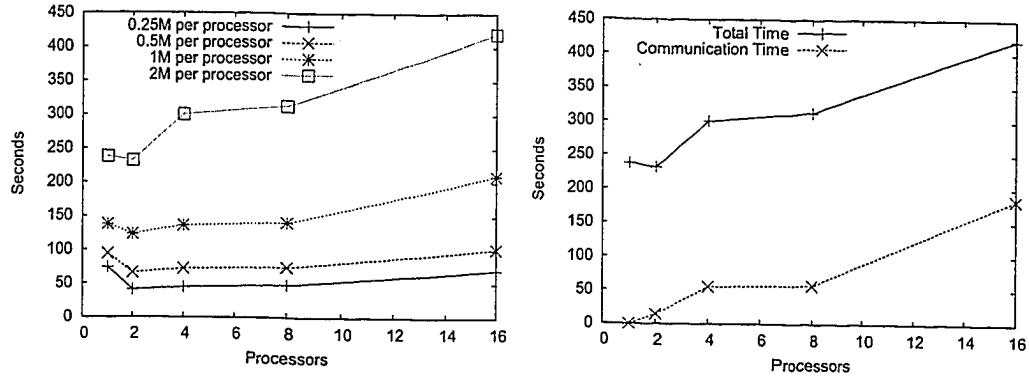
Figure 4.20: (a) Parallel wall clock time in seconds as a function of the processors for a range of different percentages of selected views and (b) corresponding speedup. (Fixed parameters: The data size $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. The memory size $M = 100$ Megabytes. The balance threshold parameter $\gamma = 5\%$)

to linear speedup for 2-8 nodes, and drops down for 16 nodes. The reason for that is no enough local work on each processor when we use 16 nodes for only 25% selected views.

### 4.4.2 Scaleup

Scaleup is another key metric for evaluation of parallel database systems [30]. It indicates whether a constant running time can be maintained as the workload is increased by adding a proportional numbers of processors and disks. So the curve of an ideal scaleup is a linear line, which parallels with x-axle.

Figure 4.21(a) shows for full cube generation the parallel wall clock time observed as a function of the number of processors used when $N/p = 0.25$M, 0.5M, 1M and 2M rows per processor. Overall, we observe good scaleup lines, which almost parallel with x-axle except the curves of 1M and 2M, where the scaleup is increased with the number of processors. When we double the number of processors and double the size of the input, we may spend less time on local computation than before. This is due to the fact that we are holding the cardinalities of dimensions constant as we increase the data size and therefore the relative density of the data cube is increasing which is beneficial for top-down generation methods. This increase in relative density leads to



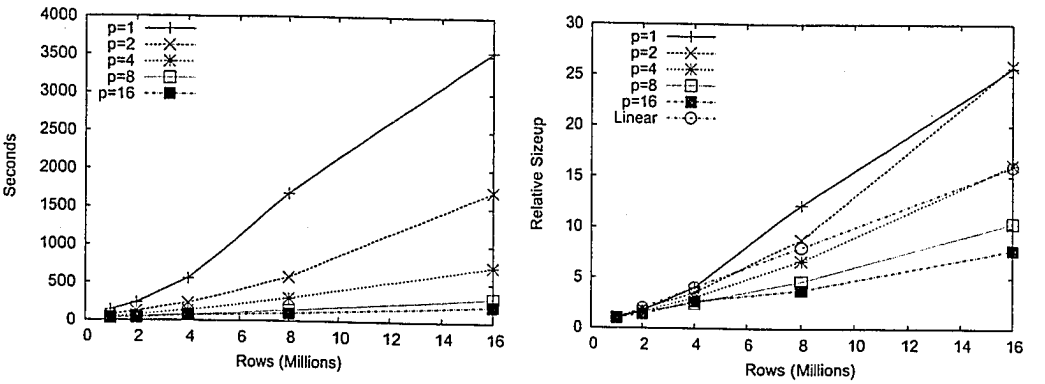

(a) (b)

Figure 4.21: (a) Parallel wall clock time in seconds as a function of the number of processors and (b) The total time and the communication time in seconds as a function of the number of processors for 2M rows per processor. (Fixed parameters: The size of raw data, $n = 8$ millions. Dimensions $d = 8$. Cardinalities $|D_i| = 256$, 128, 64, 32, 16, 8, 4, 2. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes. Balance threshold $\gamma = 5\%$)

more aggregates. We may observe this at the points for two nodes, where the running time is less than the sequential running time. However, this effect is offset by the fact that the network bandwidth is not being scaled as we increase the total input size. As we increase the data size per processor, more time has to be spent on communication across the network. When the communication time is not too much, we may still get linear scaleup, such as for 0.25M and 0.5M curves. But for more data on each node, such as 2M, high communication time degrades the global performance, as illustrated in Figure 4.21(b), where we observe the total time and the communication time are almost in parallel. It suggests the increasing time comes from communications when we increase data and processors at the same time.

### 4.4.3 Sizeup

Sizeup is similar to scaleup but fixes the number of processors. It indicates whether a proportional running time can be maintained as the workload is increased. The sizeup for $x$ units of workload is defined as $U_x = \frac{t_x}{t_1}$, where $t_1$ is the running time of one unit workload and $t_x$ is the running time of $x$ unit workload. An ideal $U_x$ is $x$, which implies that $x$ units of workload costs $x$ times more time than one unit of




(a) (b)

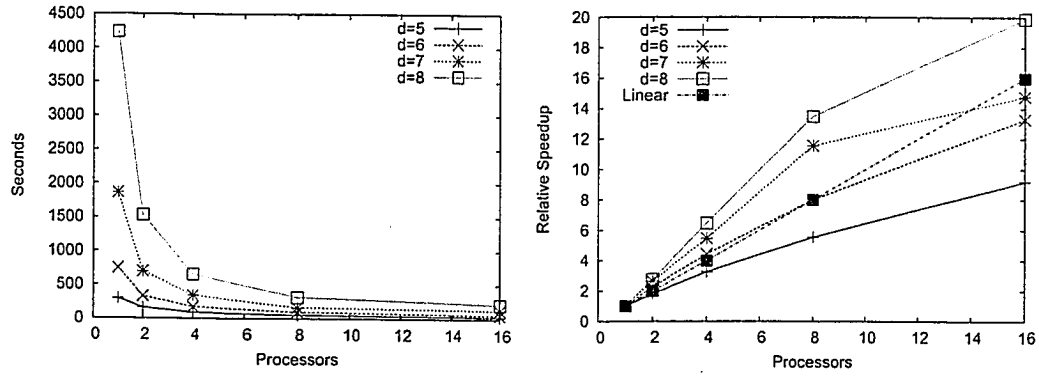
Figure 4.22: (a) Parallel wall clock time in seconds as a function of the size of input data for the processors $p = 1 - 16$ and (b) $p = 16$. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 4, 2$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$. The memory size $M = 100$ Megabytes. The balance threshold parameter $\gamma = 5\%$)

workload, so the curve of an ideal sizeup is a linear line.

Figure 4.22(a) shows the running time for full cube generation on the data sets of between 1M and 16M rows using $1 - 16$ processors and Figure 4.22(b) shows the corresponding sizeup. We observe that sizeup decreases when we chose more processors. This means we spend less time for one unit of workload when we use more processors. The reason for this is that each processor works on less data and has greater ratio of available memory to the data size when more processors are available. For $4 - 16$ processors, this effect is so significant that we observe super-linear sizeup Figure 4.22(b).

### 4.4.4 Data Dimensionality

Figure 4.23(a) shows the full cube running time for different dimensions from 5 to 8. Figure 4.23(b) shows the relative speedup. We observe that the speedup increases with the dimensions. The speedup of 8 dimensions is very good, even better than the linear speedup on 2-16 nodes. The speedup of 7 dimensions are better than the linear speedup on 2-8 nodes, and drops at 16 nodes, but still close to the linear line. The speedup of 5 dimensions is close to the linear line on 2-8 nodes, and drops at 16 nodes. The speedup of 5 dimensions is far away from the linear line. Note that,



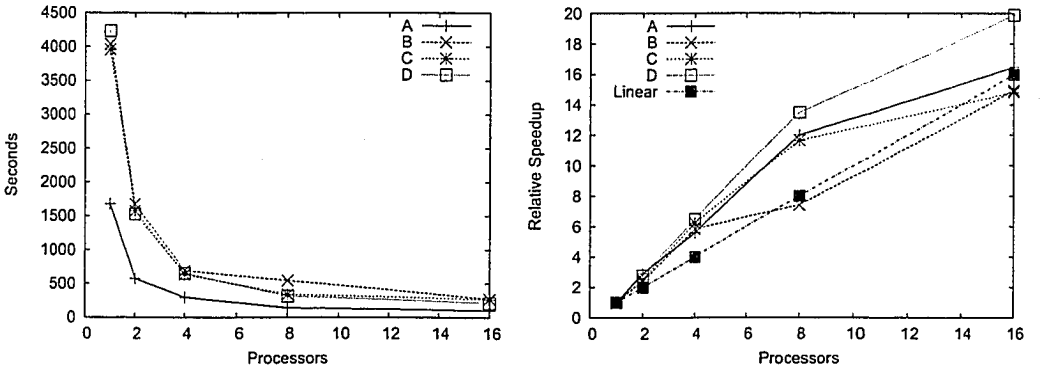

(a) (b)

Figure 4.23: Parallel wall clock time in seconds as a function of the the number of dimensions. (Fixed parameters: Data size $n = 8$ million rows. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. Percentage of views selected $k = 100\%$. Skew $\alpha = 0$. The memory size $M = 100$ Megabytes. The balance threshold parameter $\gamma = 5\%$)

the number of views grows exponentially with respect to the dimensionality of the data set, so that the workload increases very faster. We may see the running time increases faster than the linear speed in Figure 4.23(b). Therefore we may get good speedup at large number of dimensions. Only when the number of dimensions is very small, such as 5, the speedup is far away from linear speedup due to much less work to be parallelized.

### 4.4.5 Cardinality of Dimensions

Figure 4.24 shows the running time of full cube generation for different carnalities and the relative speedup. We observe that the speedup is all above the linear line or close to it. The speedup for $|A_i| = 64$ is not good as other after 8 nodes, even though the sequential time is similar to the others. The reason is that the small cardinality makes it harder to partition the data on large number processors evenly, so that we may take more time on merging data cubes and shift data across the processors. However our optimized partitioning still can find out the best partitions and make the speedup close to the linear line, as showed in Figure 4.24.




(a) (b)

Figure 4.24: (a) Parallel wall clock time in seconds as a function of the number of processors for data sets with different cardinality mixes, and (b) corresponding relative speedup. (Fixed parameters: Data size $n$ = 8 million rows. Dimensions $d$ = 8. Cardinalities and skews (A)$|D_i|$ = 256, 128, 64, 32, 16, 8, 4, 2. (B)$|D_i|$ = 64, $1 \leq i \leq d$. (C)$|D_i|$ = 128, $1 \leq i \leq d$. (D)$|D_i|$ = 256, $1 \leq i \leq d$. Percentage of views selected $k$ = 100%. Skew $\alpha = 0$. The memory size $M$ = 100 Megabytes. The balance threshold parameter $\gamma$ = 5%)

### 4.4.6 Data Skew

Data sets with skewed distributions can cause data reduction and then reduce the local computing time. To explore the effects of skew data, we generated data sets using the standard ZIPF [66] distribution in each dimension with $\alpha = 0$ (no skew) to $\alpha = 3$ (very high skew). Figure 4.25 shows the running time of full cube generation with the skew for $\alpha = 0, 1, 2, 3$, and the corresponding relative speedup. We observe that, in general, as skew is increased, the running time decreases due to data reduction and decreased local computation. Our data partitioning optimization appears to handle gracefully the resulting data imbalance by shifting data appropriately so that the speedup on the small number of processors is above or close to the linear line. However, if this data reduction is very large, as for $\alpha = 3$, it reduces the opportunities for speedup as there is simply much less work to be parallelized.

## 4.5 Summary

In this chapter, we present our external memory parallel ROLAP data cube generation algorithm on shared-nothing cluster and some enhancements. We address two

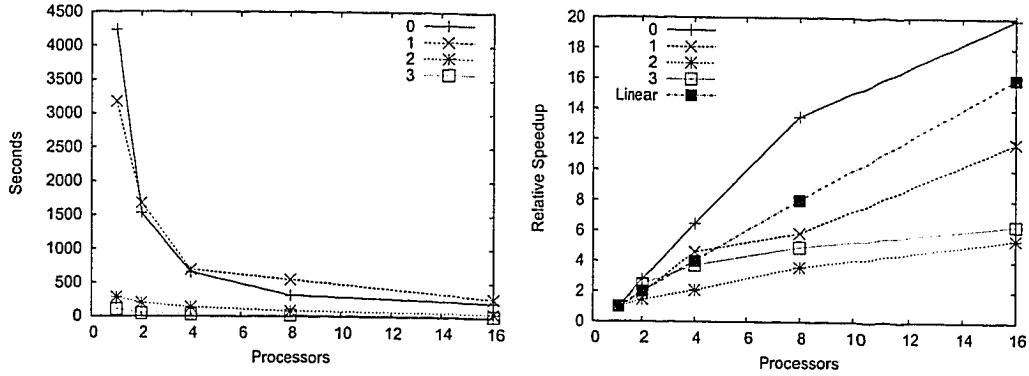



Figure 4.25: (a) Parallel wall clock time in seconds as a function of the skew of $\alpha = 0, 1, 2, 3$ for the running time and the communication time and (b) the corresponding relative speedup. (Fixed parameters: The data size $n$ = 8 millions. Dimensions $d$ = 8. Cardinalities $|D_i| = 256$, $1 \leq i \leq d$. The number of processors $p = 16$. The memory size $M$ = 100 Megabytes. The balance threshold parameter $\gamma = 5\%$)

critical problems: 1) how to compute data cube efficiently in external memory and 2) how to reduce the high cost of disk I/O and network I/O. For the first problem, we use a partitioning schema to reduce the memory for sorting and design an efficient external memory adoption of PipeSort, which is a popular in-memory data cube generation method. Our experiments show that the enhanced external memory data cube generation algorithm reduces the sequential running time by up to 40% compared with our basic algorithm. For the second problem, we design an adaptive data partitioning scheme, which uses a cost model to estimate the cost of computation, disk I/O and network I/O based on different parallel machines and computes a "best partitioning" to reduce the global cost. Our experiments show that the adaptive data partitioning reduces the parallel running time by up to 40% and increases the speedup by up to 50%. Overall our external memory parallel ROLAP data cube generation algorithm exhibits good speedup in most cases while reducing the total running time significantly. More importantly, our external memory algorithm can handle large data sets and maintains scalability in a linear time.

# Chapter 5

# External Memory and Parallel Generation of ROLAP Iceberg Data Cubes

In this chapter, we present a novel PnP operator and "Pipe 'n Prune" (PnP) algorithm for the computation of iceberg cube queries. Our PnP algorithm consists of a sequential PnP version, an external memory PnP version, and a parallel PnP version. In the performance evaluation, we compare the sequential PnP with other two sequential iceberg algorithms, BUC [20] and Star-cubing [64], and then evaluate the external memory PnP and the parallel PnP. These experiments show that PnP is an interesting new alternative method to BUC or Star-cubing, and the external memory PnP and parallel PnP algorithms also achieve good performance on large data sets.

## 5.1 Introduction

In the previous two chapters, we have described efficient parallel algorithms for computing both full and partial data cubes. However, the size of data warehousing keeps growing in recent years. In the Winter Corporation's report [63], the largest three databases exceed 20 terabyte in size. More importantly, it is estimated that by the end of 2004, the storage requirements of more than 40% of production data warehouses will exceed one terabyte [31].

One approach for dealing with the data cube size is to allow user-specific constraints. For iceberg-cubes (e.g. [20, 34, 64]), aggregate values are only stored if they have a certain, user specified, minimum support. Another possible approach is to introduce parallel processing which can provide two key ingredients for dealing with the data cube size: increased computational power through multiple processors and increased I/O bandwidth through multiple parallel disks (e.g. [25, 24, 28, 33, 29, 27, 39, 58, 40, 43, 55]). In the paper [56], Ng et.al. combined both of the above approaches and studied various algorithms for parallel iceberg-cube computation on PC clusters. The algorithm of choice in [56], referred to as *PT*, applies a *hybrid* approach

in that it combines top-down data aggregate with bottom-up data reduction.

Motivated by the work of Ng et.al. [56] and the recent success of another hybrid sequential method, Star-Cubing [64], in this chapter we further investigate the use of hybrid approaches for the *parallel* computation of iceberg-cube queries. We present a new hybrid method, called "Pipe 'n Prune" (PnP), for iceberg-cube query computation. Our approach combines top-down data aggregate through piping with bottom-up *Apriori* [18, 20] data reduction. The main difference to previous approaches is the introduction of a novel PnP operator which uses a piping approach to aggregate data and, at the same time, performs *Apriori* pruning for subsequent group-by computations. Our approach was motivated by the work of Ng et.al. [56] who presented a two phase hybrid parallel method, *PT*, which first partitions BUC bottom-up computation and then use top-down aggregate for building the startup group-by for each partition. Inspired by Star-Cubing [64], our new PnP operator extends this two phase approach towards a complete merge between data aggregate and *Apriori* pruning. PnP is very different from Star-Cubing [64] in that PnP retains top-down data aggregate through piping and interleaves it with iceberg bottom-up data reduction (pruning). An illustration of our approach is sketched in Figure 5.2. An important property of our PnP method is that it is composed mainly of linear data scans and does not require complex in-memory structures. This allows us to extend PnP to external memory computation of very large iceberg-cube queries with only minimal loss of efficiency. In addition, PnP is well suited for shared-nothing parallelization (where processors do not share any memory and all data is partitioned and distributed over a set of disks). Our new parallel, external memory, PnP method provides close to linear speedup particularly on those data sets that are hard to handle for sequential methods. In addition, parallel PnP scales well and provides linear speedup for larger number of processors, thereby also solving an open scalability problem observed in [56].

In the remainder of this chapter, we present a novel PnP operator and "Pipe 'n Prune" (PnP) algorithm for the computation of iceberg-cube queries. The novelty of our method is that it completely interleaves a top-down piping approach for data aggregation with bottom-up *Apriori* data pruning. A particular strength of PnP is that it is very efficient for *all* of the following scenarios: sequential iceberg-cube

queries, external memory iceberg-cube queries and parallel iceberg-cube queries on shared-nothing PC clusters with multiple disks.

## 5.2 The Sequential PnP Algorithm

PnP is a hybrid, sort-based, algorithm for the computation of very large iceberg-cube queries. The idea behind PnP is to fully integration data aggregation via top-down piping [59] with bottom-up (BUC [20]) *Apriori* pruning. We introduce a new operator, called the *PnP operator.* For a group-by $v$, the PnP operator performs two steps: (1) It builds all group-bys $v'$ that are a prefix of $v$ through one single sort/scan operation (piping [59]) with iceberg-cube pruning. (2) It uses these prefix group-bys to perform bottom-up (BUC [20]) *Apriori* pruning for new group-bys that are starting points of other piping operations. An example of a 5-dimensional PnP operator is showed in Figure 5.1. The PnP operator is applied recursively until all group-bys of the iceberg-cube have been generated. An example of a 5-dimensional *PnP Tree* depicting the entire process for a 5-dimensional iceberg-cube query is shown in Figure 5.2.

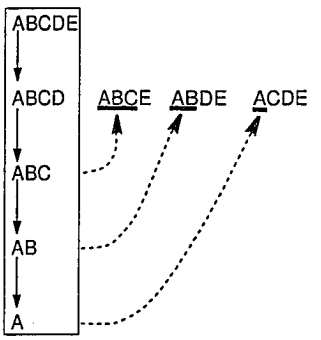



Figure 5.1: A PnP Operator.

### 5.2.1 PnP: Sequential In-Memory Version

In this chapter, we assume as input a table $R[1..n]$ representing a $d$-dimensional raw data set $R$ consisting of $n$ rows $R[i]$, $i = 1 \ldots n$. Because of the iceberg-cube constraint, a row in a view is only returned if its measure is greater than *minimum support*, *min_sup*.

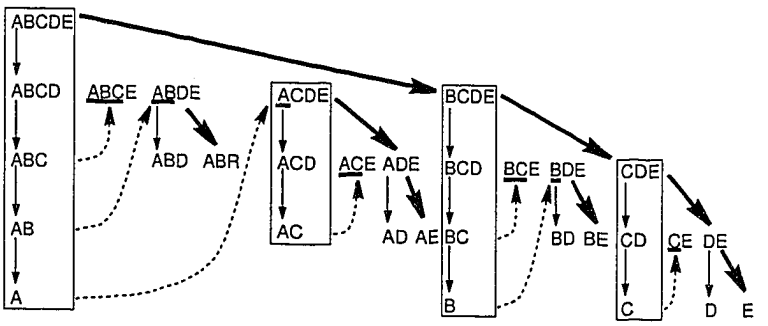



Figure 5.2: A PnP Tree. (Plain arrow: Top-Down Piping. Dashed Arrow: Bottom-up Pruning. Bold Arrow: Sorting.)

For a row $R[i]$ we denote with $\underline{R_j}[i]$ the prefix of $R[i]$ consisting of the first $j$ dimension values of $R[i]$, followed by the measure value of $R[i]$. We denote with $\hat{R}^j[i]$ the row $R[i]$ with its dimension value in dimension $j$ removed. We denote with $\emptyset$ the empty (0-dimensional) view. For a view $v$ we denote with $|v|$ the number of dimensions of $v$, and with $\hat{v}^j$ the view that is the same as $v$ but with dimension $j$ removed. We denote with $\underline{v_j}$ the view identifier consisting of the first $j$ dimensions of $v$.

Our PnP method for the sequential, in memory, case is shown in Algorithms 13 and 14. Algorithms 14 represents the main part, the implementation of the recursive PnP operator.

We explain our algorithm using the example in Figure 5.2 for a 5-dimensional iceberg-cube query. In Line 2 of Algorithms 13, we call PnP-1($R$, ABCDE, $\emptyset$). This will first result in the creation of the pipe ABCDE - ABCD - ABC - AB - A and then create pruned versions of ABCE, ABDE, and ACDE for subsequent piping operations. Table 5.1 shows a complete execution for the example raw data set $R$ indicated in the first column of Table 5.1. Buffers $b[5]$ ... $b[1]$ represent the results of piping operations, while $R_3$ ... $R_1$ show the result of pruning operations. Note that, the PnP operator uses only one single pass through the data set. The horizontal lines in Table 5.1 indicate cases where aggregate or pruning take place. The recursive call in Line 12 of Algorithms 14 initiates the PnP operator for group-bys ABCE, ABDE, and ACDE. The prefix passed as third parameter in Line 12 of Algorithms 14 is shown in Figure 5.2 as the underlined portions of ABCE, ABDE, and ACDE, respectively.

| R<br>**ABCDE** | $b$[5]<br>**ABCDE** | $b$[4]<br>**ABCD** | $b$[3]<br>**ABC** | $R_3$<br>**ABCE** | $b$[2]<br>**AB** | $R_2$<br>**ABDE** | $b$[1]<br>**A** | $R_1$<br>**ACDE** |
|---|---|---|---|---|---|---|---|---|
| 11111 1 | 11111 1 | | | pruned | | 1111 2 | | 1111 1 |
| 11112 1 | 11112 1 | 1111 2 | | | | 1112 1 | | 1112 1 |
| 11122 1 | 11122 1 | 1112 1 | 111 3 | | | 1122 1 | | 1122 1 |
| 11211 1 | 11211 1 | 1121 1 | 112 1 | pruned | **11 4** | | **1 4** | 1211 1 |
| 21111 1 | 21111 1 | 2111 1 | | pruned | | pruned | | pruned |
| 21121 1 | 21121 1 | | | | | | | |
| 21122 1 | 21122 1 | 2112 2 | 211 3 | | 21 3 | | 2 3 | |
| 31111 1 | 31111 1 | 3111 1 | | pruned | | 3111 1 | | 3111 1 |
| 31121 1 | 31121 1 | | | | | 3121 2 | | 3121 1 |
| 31122 1 | 31122 1 | 3112 2 | 311 3 | | | 3122 1 | | 3122 1 |
| 31221 1 | 31221 1 | | | pruned | | 3123 1 | | 3221 1 |
| 31223 1 | 31223 1 | 3122 2 | 312 2 | | **31 5** | | **3 5** | 3223 1 |
| 41111 1 | 41111 1 | 4111 1 | 411 1 | pruned | 41 1 | pruned | | 4111 2 |
| 42111 1 | 42111 1 | | | pruned | | pruned | | 4112 1 |
| 42112 1 | 42112 1 | 4211 2 | 421 2 | | 42 2 | | | 4121 1 |
| 43121 1 | 43121 1 | 4312 1 | 431 1 | pruned | 43 1 | pruned | **4 4** | |

Table 5.1: PnP Processing of $ABCDE$

It represents for those recursive calls the portion of the pipe that has already been computed. The recursive call in Line 20 of Algorithms 14 initiates the PnP operator for group-by BCDE and starts the iceberg-cube computation for all group-bys not containing A. The resulting entire process is depicted in Figure 5.2.

**Algorithm 13** Algorithm PnP: sequential, in memory

**Input:** $R[1..n]$: a table representing a $d$-dimensional raw data set consisting of $n$ rows $R[i]$, $i = 1 \ldots n$; *min_sup*: the minimum support.

**Output:** The iceberg data cube.

1: Sort $R$ and aggregate duplicates in $R$.

2: Call **PnP-1**($R$, $v_R$, $\emptyset$), where $v_R$ is the group-by containing all dimensions of $R$ (sorted by cardinality in decreasing order).

### 5.2.2 PnP: Sequential External Memory Version

Since PnP is sort based, it is easy to extend PnP to external memory, as shown in Algorithms 15 and 16. We discuss here only the main differences between Algorithm 14 and Algorithm 16. All sort operations are replaced by external memory sorts. Some care has to be taken with the scan and aggregate/pruning operations, as buffers may

**Algorithm 14** PnP-1($R$, $v$, $pv$)

**Input:** $R[1..n]$: a table representing the raw data set consisting of $n$ rows $R[i]$, $i = 1 \ldots n$; $v$: identifier for a group-by of $R$; $pv$: a prefix of $v$.

**Output:** The iceberg data cube.

1: **Local Variables:** $k = |v| - |pv|$; $R_j$: tables for storing rows of $R$; $b[1..k]$: a buffer for storing $k$ rows, one for each group-by $\underline{v}_1 \ldots \underline{v}_k$; $h[1..k]$: $k$ integers; $i$, $j$: integer counters. **Initialization:** $b[1..k]$ = [null .. null]; $h[1..k] = [1..1]$.
2: **for** $i = 1..n$ **do**
3:  **for** $j = k..1$ **do**
4:  **if** ($b[j]$ = null) OR (the feature values of $b[j]$ are a prefix of $R[i]$) **then**
5:  Aggregate $\underline{R}_j[i]$ into $b[j]$.
6:  **else**
7:  **if** $b[j]$ has minimum support **then**
8:  Output $b[j]$ into group-by $\underline{v}_j$.
9:  **if** $j \leq k - 2$ **then**
10:  Create a table $R_j = \hat{R}^{j+1}[h[j]]$ ... $\hat{R}^{j+1}[i-1]$.
11:  Sort and aggregate $R_j$.
12:  Call **PnP-1**($R_j$, $\hat{v}^{j+1}$, $\underline{v}_j$).
13:  **end if**
14:  **end if**
15:  Set $b[j] = null$ and $h[j] = i$.
16:  **end if**
17:  **end for**
18: **end for**
19: Create a table $R'[1..n']$ by sorting and aggregating $\hat{R}^1[1]$ ... $\hat{R}^1[n]$.
20: Call **PnP-1**($R'$, $\hat{v}^1$, $\emptyset$).

overflow and have to be saved to disk. The main difference between Algorithm 14 and Algorithm 16 is with respect to the recursive calls in Line 12 in Algorithm 14. In the external memory version, we have to save the tables $R_j$ into a file $F_j$ on disk as shown in Line 11 of Algorithm 16. A separate loop in Lines 18 to 22 of Algorithm 16 is then required to retrieve all $R_j$ and perform the recursive calls. Note that, these operations are independent and we can apply disk latency hiding through overlapping of computation and disk I/O. In order to make good use of this effect, we have implemented our own I/O manager which resulted in a significant performance improvement.

**Algorithm 15** Algorithm PnP: sequential, external memory

**Input:** $R[1..n]$: a table (stored on disk) representing a $d$-dimensional raw data set consisting of $n$ rows $R[i]$, $i = 1 \ldots n$; *min_sup*: the minimum support.

**Output:** The iceberg data cube (stored on disk).

1: Sort $R$, using external memory sorting, and aggregate duplicates in $R$.

2: Call **PnP-2**($R$, $v_R$, $\emptyset$), where $v_R$ is the group-by containing all dimensions of $R$ (sorted by cardinality in decreasing order).

## 5.3 The Parallel And External Memory PnP Algorithms

We now discuss how our PnP algorithm can be parallelized in order to be executed on a shared-nothing cluster as shown in Figure 2.27. Such a cluster consists of $p$ processors $P_0$ ... $P_{p-1}$, each with its own memory and disk. The processors are connected via a network or switch. Processors exchange information by sending and receiving messages each others. Our focus is on practical parallel methods that can be implemented on low-cost, *Beowulf* style, PC clusters consisting of standard Intel processor based Linux machines connected via Gigabit Ethernet. Since the speed of networks is much slower than processors and disk I/O, we try to reduce communications among processors as much as possible in parallel PnP.

We assume as input a $d$-dimensional raw data set $R$ stored in a table consisting of $n$ rows that are distributed over the $p$ processors as shown in Figure 3.1. More precisely, every processor $P_i$ stores on its disk a table $R_i$ consisting of $\frac{n}{p}$ rows of $R$. As indicated in Figure 3.1, each view of the output (iceberg-cube) will also be partitioned

**Algorithm 16** PnP-2($R$, $v$, $pv$)

**Input:** $R[1..n]$: a table (stored on disk) representing the raw data set consisting of $n$ rows $R[i]$, $i = 1 \ldots n$; $v$: identifier for a group-by of $R$; $pv$: a prefix of $v$.

**Output:** The iceberg data cube (stored on disk).

1: **Local Variables:** $k = |v| - |pv|$; $R_j$: tables for storing rows of $R$ (called *partitions*); $F_j$: disk files for storing multiple partitions $R_j$; $b[1..k]$: a buffer for storing $k$ rows, one for each group-by $\underline{v}_1 \ldots \underline{v}_k$; $h[1..k]$: $k$ integers; $i$, $j$: integer counters. **Initialization:** $b[1..k]$ = [null .. null]; $h[1..k] = [1..1]$.
2: **for** $i = 1..n$ (while reading $R[i]$ from disk in streaming mode...) **do**
3:  **for** $j = k..1$ **do**
4:  **if** ($b[j]$ = null) OR (the feature values of $b[j]$ are a prefix of $R[i]$) **then**
5:  Aggregate $\underline{R}_j[i]$ into $b[j]$.
6:  **else**
7:  **if** $b[j]$ has minimum support **then**
8:  Output $b[j]$ into group-by $\underline{v}_j$. Flush to disk if $\underline{v}_j$'s buffer is full.
9:  **if** $j \leq k - 2$ **then**
10:  Create a table $R_j = \hat{R}^{j+1}[h[j]]$ ... $\hat{R}^{j+1}[i-1]$.
11:  Sort and aggregate $R_j$ (using external memory sort if necessary). Write the resulting $R_j$ and an "end-of-partition" symbol to file $F_j$.
12:  **end if**
13:  **end if**
14:  Set $b[j] = null$ and $h[j] = i$.
15:  **end if**
16:  **end for**
17: **end for**
18: **for** $j = k..1$ **do**
19:  **for** each partition $R_j$ written to disk file $F_j$ in line 11 **do**
20:  Call **PnP-2**($R_j$, $\hat{v}^{j+1}$, $\underline{v}_j$).
21:  **end for**
22: **end for**
23: Create a table $R'[1..n']$ by sorting and aggregating $\hat{R}^1[1]$ ... $\hat{R}^1[n]$ (using external memory sort if necessary).
24: Call **PnP-2**($R'$, $\hat{v}^1$, $\emptyset$).

and distributed over the $p$ processors. We refer to this process as *striping* a view over the $p$ disks. When every view is striped over the $p$ disks, access to the view can be performed with maximum I/O bandwidth through full parallel disk access.

For a $d$-dimensional view $R_i$, we define views $T_i^j$, $j = 1, .., d$, as the views obtained by removing from each row of $R_i$ the first $j-1$ dimension values and performing aggregate to remove duplicates (but not performing iceberg-cube pruning). Note that, $T_i^1 = R_i$.

**Algorithm 17** Algorithm PnP: parallel, external memory

**Input:** $R$: a table representing a $d$-dimensional raw data set consisting of $n$ rows, stored on $p$ processors. Every processor $P_i$ stores (on disk) a table $R_i$ of $\frac{n}{p}$ rows of $R$ as shown in Figure 3.1. *min_sup*: the minimum support.

**Output:** The iceberg data cube (distributed over the disks of the $p$ processors as shown in Figure 3.1).

1: **Variables:** On each processor $P_i$ a set of $d$ tables $T_i^1$, ..., $T_i^d$.
2: **for** $j = 1..d$ **do**
3: Each processor $P_i$: Compute $T_i^j$ from $T_i^{j-1}$ via sequential sort. ($T_i^1 = R_i$)
4: Perform a parallel global sort on $T_1^j \cup T_1^j \cup ... \cup T_p^j$.
5: **end for**
6: **for** $j = 1..d$ **do**
7: Each processor $P_i$: Apply Algorithm 15 to $T_i^j$.
8: **end for**

Algorithm 17 shows the algorithm of parallel PnP. It consists two stages: data partitioning and iceberg cube computing. In the data partitioning stage, we partition the input data on processors and let each processor get the almost same amount of data. In the iceberg cube computing stage, each processor computes iceberg cube from the local data independently without further communications with each other.

### 5.3.1 Data Partitioning

In the data partitioning stage, we partition the input data on processors and let each processor get the almost same amount of data. In the iceberg cube computing stage, each processor computes iceberg cube from the local data independently without

further communications with each other. In order to make the computing stage independently for each processor, we partition the lattice into $d$ sub-lattices, where $d$ is the number of dimensions. All the views with the same first dimensions are grouped into one sub-lattice. See Figure 5.3. For each sub-lattice, there is a root view, such as $ABCDE$, $BCDE$, $CDE$, $DE$ and $E$. If each processor keeps the data of root views on its local disks, the iceberg cube computing on each processor is independent for sub-lattices. For example, Processor 0 may be computing on Sub-lattice $A$, while Processor 1 is computing on Sub-lattice $B$ without communication between them.

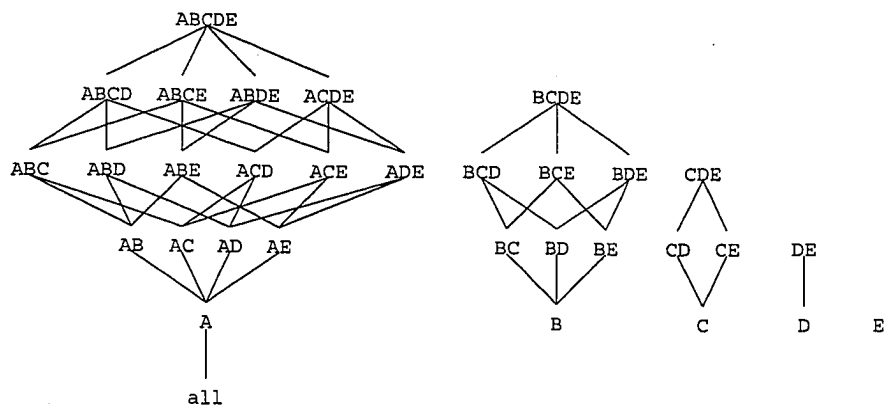



Figure 5.3: Five Dimension Sub-lattices.

However, when different processors are working on the same sub-lattices, communications could be necessary if we partition the root views evenly on processors. In Table 5.1, there are 16 rows of $ABCDE$, and each processor gets 8 rows if there is only two processors in total. When both processors compute (31) for $AB$, Processor 0 gets (311), and Processor 1 gets (314). They have to exchange data to generate the right aggradation value (315). But if we adjust partitioning between two processors to let them gets the different values of $A$ dimension, then no communication is needed between them. For example, Processor 0 gets the first 7 rows, and Processor 1 gets the rest of rows, so that all the rows where $A$ is 1 or 2, are on Processor 0, and all the rows where $A$ is 3 or 4, are on Processor 1. Therefore each processor can compute iceberg cube from its local data without communication.

The partitioning adjustment may cause slightly imbalance of workload on processors, and this increases the total computing time because the computing time is

determined by the slowest processor in the network. On the other hand, balanced partitioning could even cost more time than unbalanced partitioning, because we have to exchange data among processors, merge the data before computing measures and compare them with minimal support. Since the increased computing time is very small compared with the communication time and the merging time in the low speed network of PC cluster, The partitioning adjustment is better way to partition the data.

We partition the root views using the adaptive sample sort described in Algorithm 7. In this algorithm, each processor sorts its local data first, and then sends some sample data to Processor 0, which sorts the samples, and chooses pivots from the sorted samples data, sends them back to each processor. Then all processors partition the local data using the pivots, and send, receive partitions to or from other processors. At last, each processor merges the partitions locally. In Algorithm 7, there is a step to apply a global shift according to a threshold value. Here we do not need a shift for iceberg cube computing.

### 5.3.2 Parallel Iceberg Cube Computing

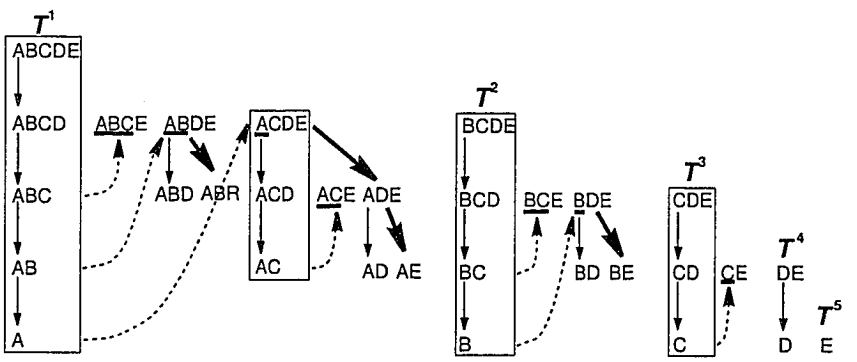



Figure 5.4: A PnP Forest.

The basic idea of the iceberg cube computing stage is illustrated in Figure 5.4. The figure shows a *PnP forest* obtained by converting each sub-lattice in Figure 5.3 into a PnP tree. Therefore we get $d$ PnP trees, one for each feature dimension. The data set for the root of the $j$th tree is the set $T^j = T_0^j \cup T_1^j \cup ... \cup T_{p-1}^j$. We start with $T^1 = R$ striped over the $p$ disks, where processor $P_i$ stores $T_i^1 = R_i$, and

execute on each processor $P_i$ the sequential Algorithm 15 with input $T_i^1$ (Line 7 of Algorithm 17). This creates the first tree in the PnP forest of Figure 5.4.

Next, we compute on each processor $P_i$ the table $T_i^2$ from $T_i^1$ by removing the first feature dimension and performing aggregate to remove duplicates (via a sequential sort); see Line 3 of Algorithm 17. Different data aggregate on different processors can lead to imbalance between processors, and the set $T_0^2 \cup T_1^2 \cup ... \cup T_{p-1}^2$ is therefore re-balanced through a global sort (Line 4 of Algorithm 17). We can then execute on each processor $P_i$ the sequential Algorithm 15 with input $T_i^2$ (Line 7 of Algorithm 17), creating the second tree in the PnP forest of Figure 5.4. This process is iterated $d$ times, until all views have been built.

## 5.4 Performance Evaluation

We have implemented the sequential (in-memory), external memory, and parallel versions of our PnP algorithm as presented in the previous section. Our sequential C++ code evolved from the code for top-down sequential PipeSort used in the previous chapters. Our external memory code evolved from the sequential code optimized using the partition sort and the external memory PipeSort in Chapter 4. Our parallel code evolved, in turn, from our the external memory code through the addition of communication operations drawn from the MPI communication library.

Our performance evaluation was conducted in three stages. In the first stage we evaluate the sequential version of PnP by comparing it with sequential implementations of BUC and Star-Cubing. The Microsoft Window's executables for these implementations were kindly provided by J. Han's research group to enable just such comparative performance testing of cube construction methods [64]. In the second stage, we evaluate the external memory version of PnP. For PnP codes, both the sequential version and external memory version, were compiled using Visual C++ 6.0. Both sequential and external memory experiments were conducted on a 2.8 GHz Intel Pentium 4 based PC running Microsoft Windows 2000 with 1 GB RAM and an 80 GB 7200 RPM IDE disk. In the third stage of our performance evaluation we explored the performance of our the parallel version of PnP on the 16 node cluster introduced in Section 2.4.

In the following experiments, all sequential times are measured as wall clock times

in seconds. All parallel times are measured as the wall clock time between the start of the first process and the termination of the last process. All times include the time taken to read the input from files and write the output into files. Furthermore, all wall clock times are measured with no other users on the machine. The running times for BUC and Star-Cubing that we show are those captured and reported by the executables obtained from [64].

### 5.4.1 Sequential Experiments

The performance results for our sequential experiments are shown in Figures 5.5 to 5.11. There are three groups of experiments in this section. The first group, Figure 5.5, compares PnP to BUC and Star-Cubing for the special case of full cube computation. Then the second group, Figures 5.6 to 5.7 compares the iceberg cube computation on raw data sets of varying sparsity. The last group, Figures 5.8 to 5.11 explore various settings of the size of raw data, dimensions, skew and the minimal support for both very dense and very sparse cubes.

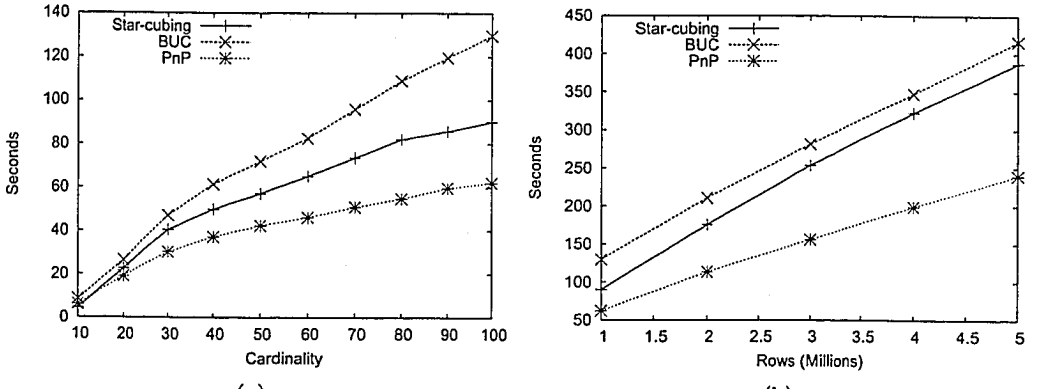



Figure 5.5: (a) Full cube running time in seconds as a function of the cardinality with the size of raw data, $n = 1$ million and (b) Full cube running time in seconds as a function of the size of raw data with the cardinalities $|D_i| = 256$, $1 \leq i \leq d$. (Fixed parameters: Dimensions $d = 6$. Skew $\alpha = 0$.)

Figures 5.5 shows for full cube computation (i.e. the minimal support is 1.) results for PnP compared to BUC and Star-Cubing on various cardinalities and growing data sizes. Note that varying cardinality, while holding the other parameters constant, amounts to varying the sparsity. We observe that for the special case of full cube

computation the sequential version of PnP performs better than BUC or Star-Cubing regardless of sparsity. In this case, PnP takes full advantage of pipeline processing and saves significant time by sharing sorts, while bottom-up *Apriori* data pruning is ineffectual.

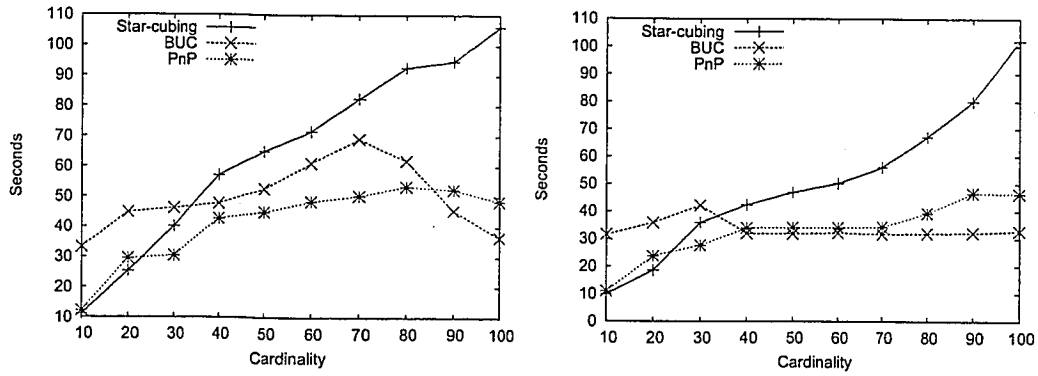



Figure 5.6: (a) Running time in seconds as a function of the cardinality with the minimal support, $m = 10$ and (b) $m = 100$. (Fixed parameters: The size of raw data, $n = 5$ million. Dimensions $d = 6$. Skew $\alpha = 0$.)

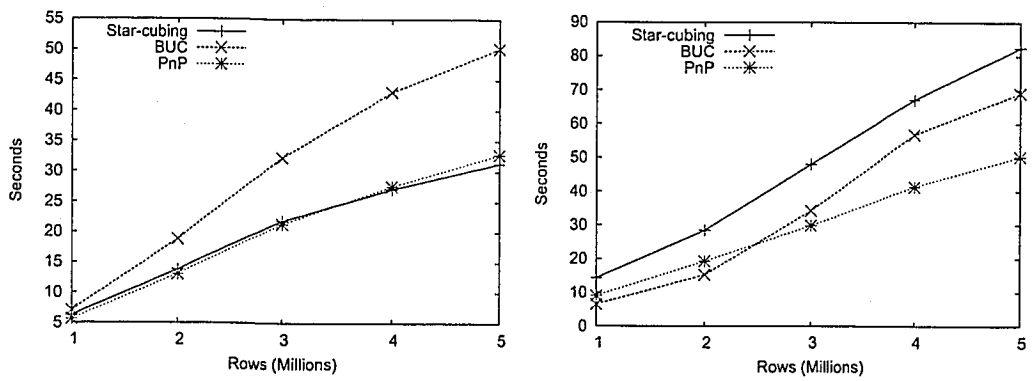



Figure 5.7: (a) Running time in seconds as a function of The size of raw data with the cardinality for each dimensions, $|D_i| = 22$ and (b) $|D_i| = 70$. (Fixed parameters: Dimensions $d = 6$. Skew $\alpha = 0$. The minimal support, $m = 10$.)

Figures 5.6 compares PnP to BUC and Star-Cubing for iceberg cube computation while varying sparsity. In order to measure the sparsity of a data cube, we use the

following formula to calculate it:

$$Sp = \frac{n}{\prod_{i=1}^{d} |D_i|}.$$

In Figure 5.6(a), Sequential PnP typically shows the best performance when the cardinality is between 25 and 80, or the sparsity of the data cube $Sp$ is between 0.02 and 0.00002. Star-Cubing is the best one when the cardinality is below 25, or $sp$ is larger than 0.02, which is a very dense case, while BUC is the best one when the cardinality is above 80, or $sp$ is larger than 0.00002, which is a very sparse case. Figure 5.6(b) shows the results when the minimal support is 100. We observe the similar trend as Figure 5.6(a). PnP and BUC are very close when $Sp$ is between 0.02 and 0.00002, and both of them are better than Star-Cubing. When $Sp$ is larger than 0.02, Star-Cubing is best, and when $Sp$ is less than 0.00002, BUC is the best.

In order to take a close look at these thresholds (0.02 and 0.00002), we select the cardinality as 22 and 70 respectively for a range of $n$ in order to examine at which point PnP switches position with BUC or Star-Cubing. Figure 5.7(a) shows running time verses varying data size with the cardinality for each dimensions, set to $|D_i| = 22$. In this dense cube case, we observe PnP and Star-cubing are better than BUC, and they switch the position between 3 million and 4 million rows. The sparsity at 3M is about 0.02. When the data size is smaller than 3M, or the sparsity is smaller than 0.02, PnP is versatilely better than Star-cubing. And when the sparsity is increase and larger than 0.02, Star-cubing becomes very slightly better than PnP. Figure 5.7(b) shows running time verses the varying data size with the cardinality for each dimensions, set to $|D_i| = 70$. This is the sparse cube case. We observe PnP and BUC are better than Star-cubing, and they switch the positions between 2 million and 3 million rows. The sparsity at 2M is about 0.00002. When the data size is smaller than 2M, or the sparsity is smaller than 0.00002, BUC is slightly better than PnP. And when the sparsity is increase and larger than 0.00002, PnP becomes better than BUC.

Finally, Figures 5.8 to 5.11 compare PnP to BUC and Star-Cubing for iceberg cube computation while varying the raw data size, dimensionality, minimum support, and skew, for both very dense ($|D_i = 10$ or $Sp = 5$) and very sparse ($|D_i| = 100$ or $Sp = 0.000005$) cubes. We observe that the sequential performance of PnP is highly stable even in these extreme cases. Sequential PnP performance is almost always

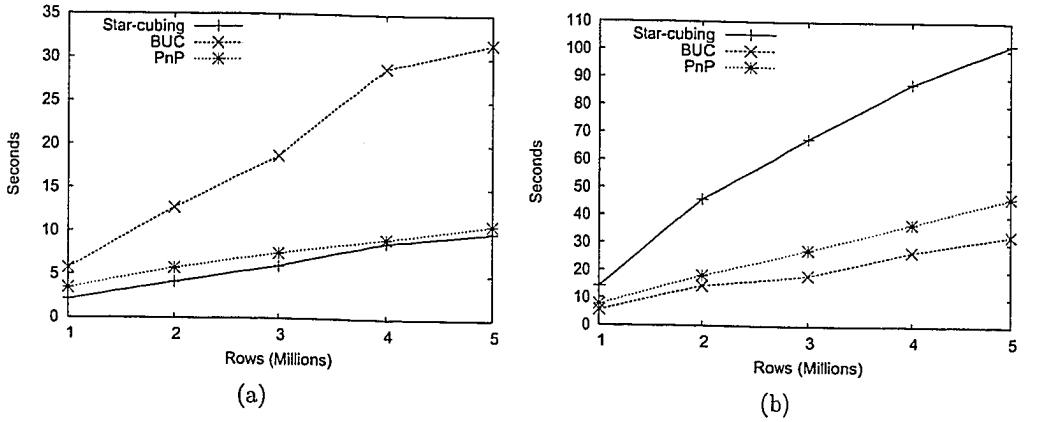



Figure 5.8: (a) Running time in seconds as a function of the size of raw data with the cardinality for each dimensions, $|D_i| = 10$ and (b) $|D_i| = 100$. (Fixed parameters: Dimensions $d = 6$. Skew $\alpha = 0$. The minimal support, $m = 10$.)

close to the best one, while BUC tends to perform best on very sparse data sets and Star-Cubing best on very dense data sets.

Overall, sequential PnP shows the best performance when the data sets are in the normal sparsity ($0.00002 \leq Sp \leq 0.02$), while still keeps close to the best one in the extremely sparse or dense cases. Therefore sequential PnP appears to be an interesting alternative to BUC and Star-Cubing especially in applications where performance stability over a large wide range of input parameters is important.

### 5.4.2 External Memory Experiments

The performance results for the external memory version of PnP are shown in Figures 5.12. Note that since PnP is composed mainly of linear scans and does not require complex in-memory data structures, it is reasonably easy to implement as an external memory method for very large iceberg-cube queries. For these experiments, in order to make good use of PnP's properties, we have implemented our own I/O manager to have full control over latency hiding through overlapping of computation and disk I/O.

In evaluating the external memory version of PnP we use larger data sets, ranging in size from 1 million to 20 million rows, while varying dimensionality $d$ and available memory $M$.

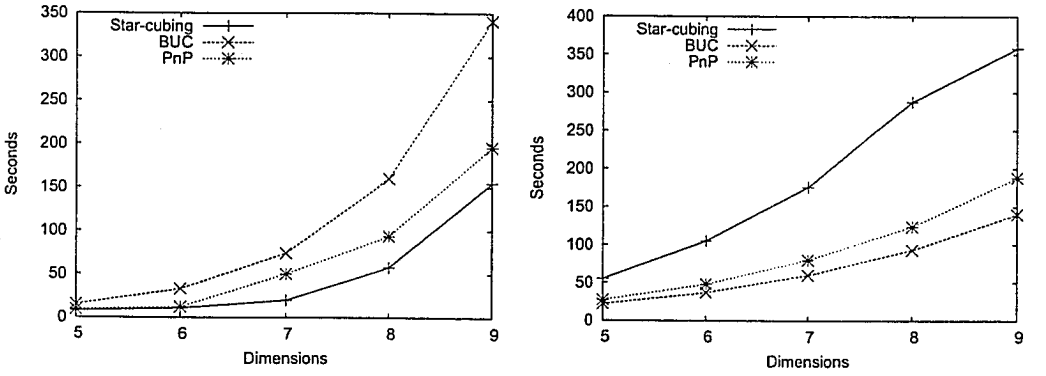



Figure 5.9: (a) Running time in seconds as a function of The size of raw data with the cardinality for each dimensions, $|D_i| = 10$ and (b) $|D_i| = 100$. (Fixed parameters: Dimensions $d = 6$. Skew $\alpha = 0$. The minimal support, $m = 10$.)

Overall, our experiments show minimum loss of efficiency when PnP switches from in-memory to external memory computation. The measured external memory running time (where PnP is forced to use external memory by limiting the available main memory) is less than twice the running time for full in-memory computation of the same iceberg-cube query. In Figure 5.12(a) we observe similarly shaped curves even as we increase the dimensionality of the problem due in large part to the effects of iceberg pruning. The location of the slight jump in time, corresponding to the switch to external memory, occurs between 5 million rows and 7 million rows depending on the dimensionality of the iceberg cube being generated. Figure 5.12(b) shows, not surprisingly, that there is a benefit to increasing the memory space $M$ available to the external memory algorithm. However, the gaps between the curves are very small. It suggests that the external PnP takes full use of memory and shows the good performance on the limit memory.

### 5.4.3 Parallel Experiments

The performance results for the parallel shared-nothing version of PnP are shown in Figures 5.13 to 5.17. This implement is based on the external memory PnP code base and uses external memory processing, in addition to parallelism, as needed.

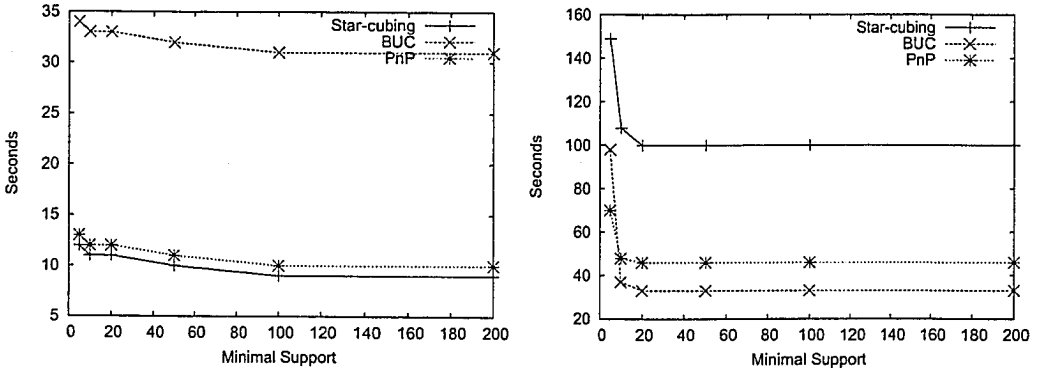



Figure 5.10: (a) Running time in seconds as a function of the minimal support with the cardinality for each dimensions, $|D_i| = 10$ and (b) $|D_i| = 100$. (Fixed parameters: The size of raw data, $n = 5$ million. Dimensions $d = 6$. Skew $\alpha = 0$.)

These experiments focus on speedup, since it is one of the key metrics for the evaluation of parallel database systems [30]. The experiments consist of incrementally increasing the number of processors available to the parallel version of PnP to determine the time and corresponding parallel speedup obtained while varying the other key parameters of input data size, dimensionality, cardinality, minimum support, and skew.

Figure 5.13(a) shows the running time of parallel PnP for input data sizes between 1 and 8 million rows and Figure 5.13(b) shows the corresponding speedup. As is typically the case, relative speedup improves as we increase the size of the input and consequentially the total amount of work to be performed. With the data size from 2M to 8M, near optimal linear speedup is observed all the way up to 8 processors. For 16 processors, speedup drops a little because the data on each processor reduces. With the 1M data size, speedup drops off slightly beyond 4 processors. Again, it suggests our parallel algorithm works well on large data sets.

Figure 5.14(a) shows the running time of the parallel version of PnP for increasing dimensionality and Figure 5.14(b) shows the corresponding speedup. We observe that the speedup increases with the dimensions increasing. When dimensions increase, there are exponentially increasing views in a data cube, so that the local computing spend much more time. However the communication time increases slowly. In fact, we just need one more parallel sort when dimensions increase by 1. Therefore, the

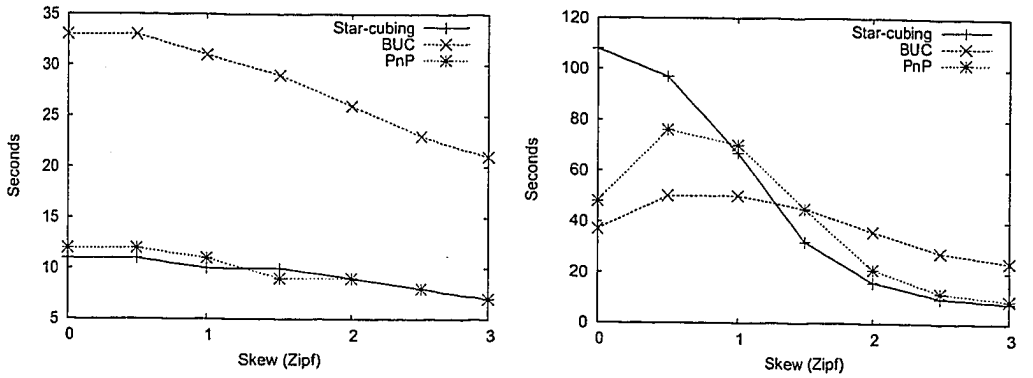
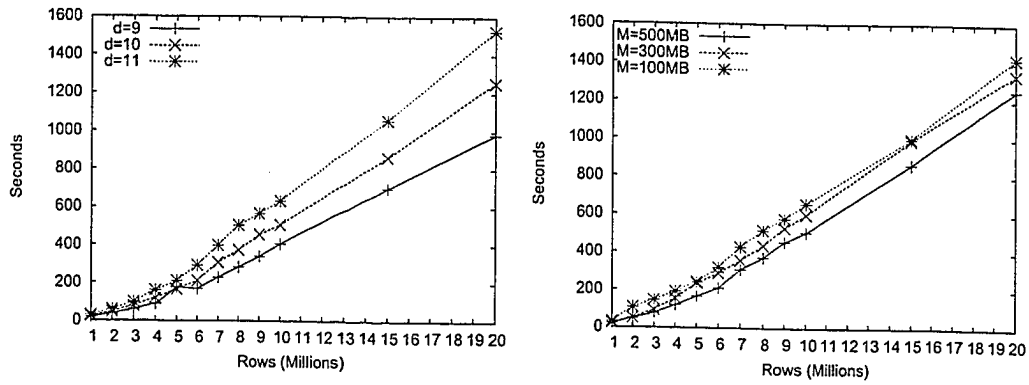



Figure 5.11: (a) Running time in seconds as a function of the data skew with the cardinality for each dimensions, $|D_i| = 10$ and (b) $|D_i| = 100$. (Fixed parameters: The size of raw data, $n = 5$ million. Dimensions $d = 6$. Skew $\alpha = 0$.)

speedup goes up with dimensions increasing. For 16 nodes, we observe that 11 dimension data shows super linear speedup, 10 dimension data speedup is below, but close to the linear line, and 9-8 dimension data speedup is farther away from the linear line. Note that the best speedup is achieved on the problems which are hardest to solve sequentially, that is those that involve the largest problems in terms of input size and/or dimensionality.

The cardinality of the dimensions in the input data can significantly effect performance. As cardinalities increase so does the sparsity of the data set, and this typically effects the size of the resulting iceberg query result. Figure 5.15(a) shows the running time of the parallel version of PnP for input data covering a range of cardinalities and Figure 5.15(b) shows the corresponding speedup. We observe that the running time increases when the cardinality increases. Generally, the aggregated measures become smaller when the cardinality increases, so that we might expect the less running time. However, in this experiment, we choose the small minimal support and a large data size. That means the most measures are still greater than the minimal support, and more rows are output. Therefore, in Figure 5.15(b), we observe that the speedup increases when the cardinality increases. The largest speedup is for the carnality of 200. It is even above the linear line. We notice its sequential running time is the too. Then the speedup for the carnality of 100 is below the linear line and still close to the linear line. The worst one is the data set for the cardinality of 50, because the cube



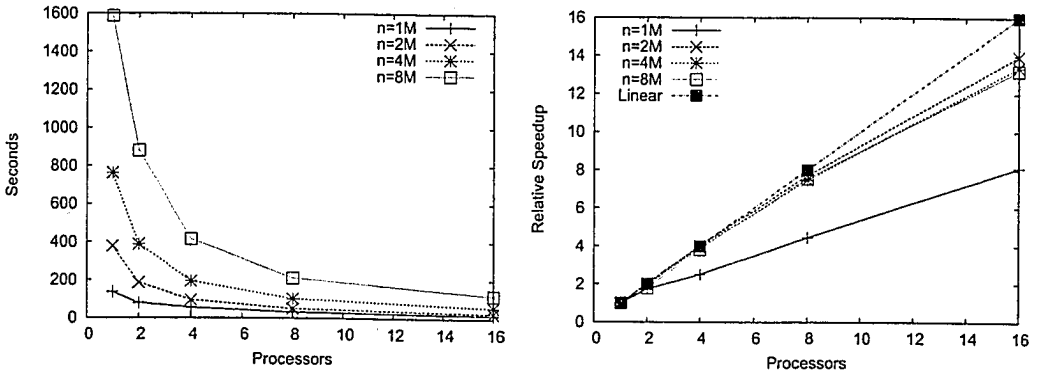

(a) (b)

Figure 5.12: (a) Running time in seconds as a function of the size of raw data with the memory size $M$=500MB and (b) with the 10 dimensions. (Fixed parameters: The cardinality $|D_i| = 300$, $1 \leq i \leq d$. Skew $\alpha = 0$. The minimal support, $m = 1000$.)

is dense, and output the less data. The data set with the various cardinality shows the speedup between the cardinality of 50 and 100, because most of its cardinalities are still small and it is still dense.

Figures 5.16 shows the effects on running time and speedup of varying minimum support. We observe that for smaller values of minimum support, $m = 100$ and $m = 500$, the computing time required is larger and the speedup obtained by our parallel PnP algorithm is near the linear line. And the curves of $m = 100$ and $m = 500$ are almost overlapped, because the most measures in the data cube are greater 500, also greater than 100 too, so that the two cases output almost the same amount of data. For the large minimal support, $m$=1000 and $m$=2500, the speedup is away from the linear line, because most of data are pruned and so lack of efficient local computing. Note that the running time for $m$=1000 is greater than $m$=2500, however the speedup for $m$=1000 is lower than $m$=2500. The reason is that PnP operator becomes an in-memory version when the data can fit in memory on each processor, so it takes much less time than the external version, which is used for the sequential case.

Figures 5.17 shows the effects on running time and speedup of skew. We observe that for smaller values of skew the time required is larger (as is typically the case in cube construction) and the speedup obtained by our parallel PnP algorithm is near linear. When skew is sufficiently large speedup falls off, however so does the wall




(a) (b)

Figure 5.13: (a) Parallel wall clock time in seconds as a function of the number of processors for the data size $n$ = from 1 million to 8 million rows and (b) corresponding speedup. (Fixed parameters: Dimensions $d = 10$. Cardinalities $|D_i| = 100$, $1 \leq i \leq d$. Skew $\alpha = 0$. The minimal support $m = 100$. The memory size $M = 100$ Megabytes.)

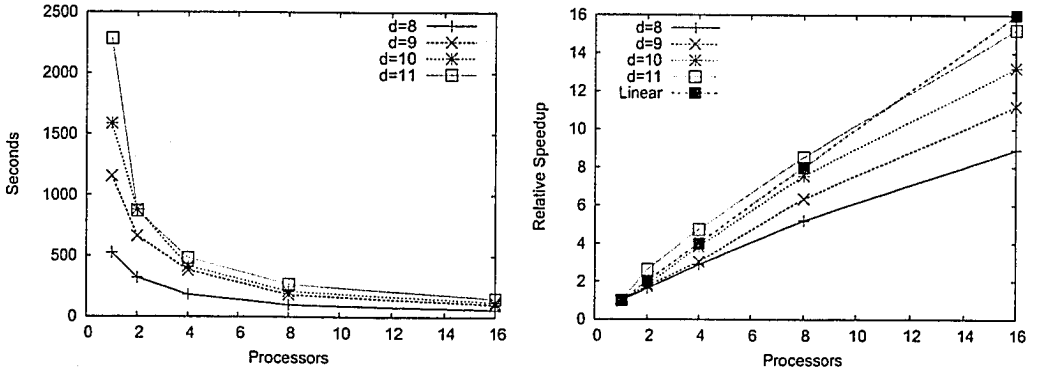
clock time required by parallel PnP to compute the iceberg cube.

Overall our experiments show that the parallel version of PnP provides close to linear speedup particularly on those data sets that are hard to handle for sequential methods, such as large size high dimensions and large cardinalities.

## 5.5 Summary

In this chapter, we have described the use of hybrid approaches for the *parallel* computation of iceberg-cube queries and presented a new hybrid method, "Pipe 'n Prune" (PnP), for iceberg-cube query computation. The most important feature of our approach is that it completely interleaves top-down data aggregate through piping with bottom-up *Apriori* data reduction.

We performed an extensive performance analysis of PnP for all of the above scenarios. For sequential iceberg-cube queries, PnP typically shows a best performance in the certain range of sparsity. While BUC and StarCube have ranges of data density and skew where BUC outperforms StarCube or vice versa. In both cases, PnP is close to the best one. This makes PnP an interesting new alternative method, especially in applications where performance stability over a wide range of input parameters is




(a) (b)

Figure 5.14: (a) Parallel wall clock time in seconds as a function of the number of processors for the dimensions $d =$ from 8 to 11 and (b) corresponding speedup. (Fixed parameters: The data size $n =$ 8M. Cardinalities $|D_i| = 100$, $1 \leq i \leq d$. Skew $\alpha = 0$. The minimal support $m = 100$. The memory size $M = 100$ Megabytes.)

important. For external memory iceberg-cube queries, we observe minimum loss of efficiency. The measured external memory running time is less than twice the running time for full in-memory computation of the same iceberg-cube query. For parallel iceberg-cube queries on shared-nothing PC clusters, PnP scales well and provides near linear speedup for larger numbers of processors. In general, PnP performs very well for both, dense *and* sparse data sets and it scales well, providing linear speedup for larger number of processors up to 16 processors.

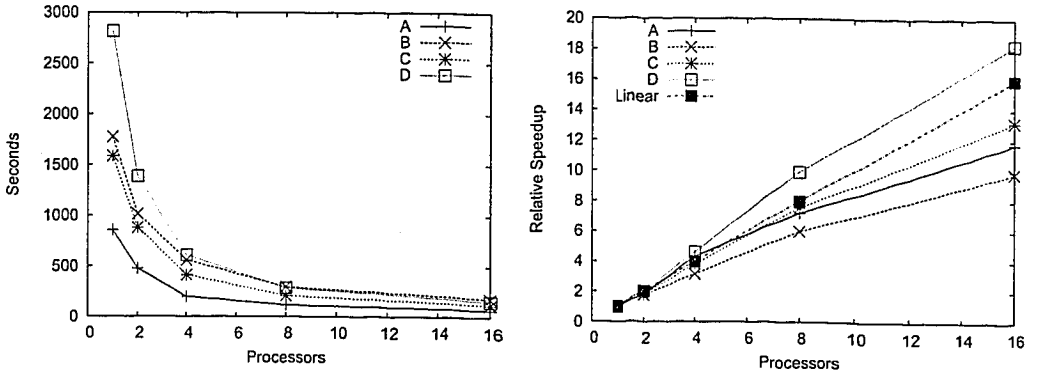



Figure 5.15: (a) Parallel wall clock time in seconds as a function of the number of processors for data sets with different cardinality mixes, and (b) corresponding relative speedup. (Fixed parameters: The data size $n = 8$M. Dimensions $d = 10$. Cardinalities (A)$|D_i| = 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2$. (B)$|D_i| = 50$, $1 \leq i \leq d$. (C)$|D_i| = 100$, $1 \leq i \leq d$. (D)$|D_i| = 200$, $1 \leq i \leq d$. The minimal support $m = 100$. Skew $\alpha = 0$. The memory size $M = 100$ Megabytes.)

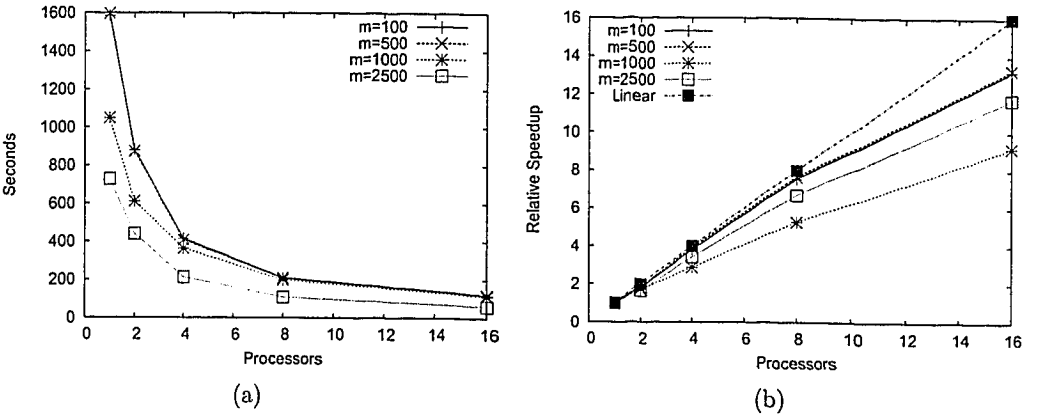



Figure 5.16: (a) Parallel wall clock time in seconds as a function of the minimal support and (b) corresponding relative speedup. (Fixed parameters: The data size $n = 8$M. Dimensions $d = 10$. Cardinalities $|D_i| = 100$, $1 \leq i \leq d$. Skew $\alpha = 0$. The memory size $M = 100$ Megabytes.)

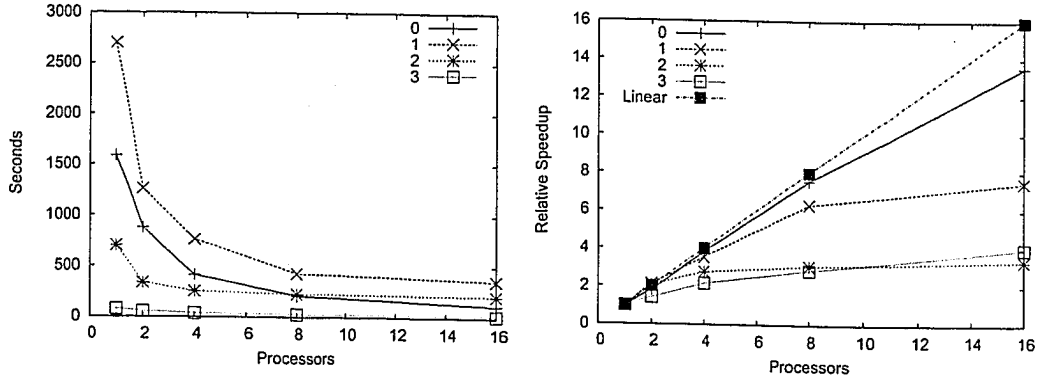



Figure 5.17: (a) (a) Parallel wall clock time in seconds as a function of the number of processors for data skew $\alpha = 0, 1, 2, 3$ and (b) corresponding relative speedup. (Fixed parameters: The data size $n = 8$ millions. Dimensions $d = 10$. Cardinalities $|D_i| = 100$, $1 \leq i \leq d$. The number of processors $p = 16$. The minimal support $m = 100$. The memory size $M = 100$ Megabytes.)

# Chapter 6

# The CgmOLAP System

In the previous three chapters we have focused on algorithm design issues for high performance sequential external memory and parallel data cube generation. The focus has been on the algorithms rather than the software engineering and system issues that must be tackled to realize them in a comprehensive robust extensible system. In this chapter, we introduce our cgmOLAP system, the first fully functional parallel OLAP system able to build data cubes at a rate of more than half terabyte per hour. We describe the system architecture, the main modules of the system, and how the system process files into the larger scheme of query processing. We also present a performance evaluation of the cgmOLAP system using real (i.e. non-synthetic) and very large synthetic data sets. This performance evaluation confirms our early results that were based on previous synthetic data.

## 6.1 Introduction

All of the key algorithms described in Chapters 3, 4 and 5 of the thesis have been incorporated into the cgmOLAP system. The cgmOLAP system, developed as part of the PANDA project [13] at Dalhousie, Concordia, Carleton and Griffith universities, employs parallel processing techniques to support highly scalable ROLAP data cube generation and queries on large data warehouses, where the size of a single data cube query can be massive. The cgmOLAP system is the first fully functional parallel OLAP system able to build data cubes at a rate of more than half terabyte per hour. The system incorporates a variety of novel algorithms, such as the parallel computation of full cubes, partial cubes, and iceberg cubes, which were discussed in the previous chapters. In particular, our cgmOLAP system has the following distinguishing characteristics:

1. **Fully parallel.** The cgmOLAP system is fully parallel and has been designed from the ground up to efficiently exploit the computational power of inexpensive,

shared-nothing, distributed memory clusters.

2. **Memory hierarchy adaptive.** All of the key algorithms that make up the cgmOLAP system are designed to be both cache friendly and capable of running fully in external memory. No requirement that even data stored on a single processor will fit in memory. Detailed engineering of I/O managers to manage local disk subsystems.

3. **Highly tunable.** All of the key algorithms that make up the cgmOLAP system are driven off an explicit cost model. This approach supports both hardware platform portability and application self tuning.

4. **Scalability.** The cgmOLAP system is highly scalable in terms of dimensions, processors, and input data size. All of the key parallel algorithms for data cube generation and query processing exhibit close to linear (optimal) speedup.

## 6.2 Software Architecture and Hardware Platform

Figure 6.1 shows the cgmOLAP software architecture. The key components in the architecture include:

1. **Application interface.** Provides the application interface to the cgmOLAP system.

2. **Parallel query engine.** Supports parallel OLAP operations, introduced in Section 2.2.4, such as roll-up, drill-down, slice, dice, and pivot queries.

3. **Parallel cube generation engine.** Supports parallel generation of full, partial and iceberg data cubes. The implement of the engine is based on the algorithms, which has been discussed in previous chapters.

4. **Meta data and cost model repositories.** Repositories for meta data describing original data sets and materialized data cubes, as well as hardware configuration and system performance parameters that drive the algorithmic cost models.

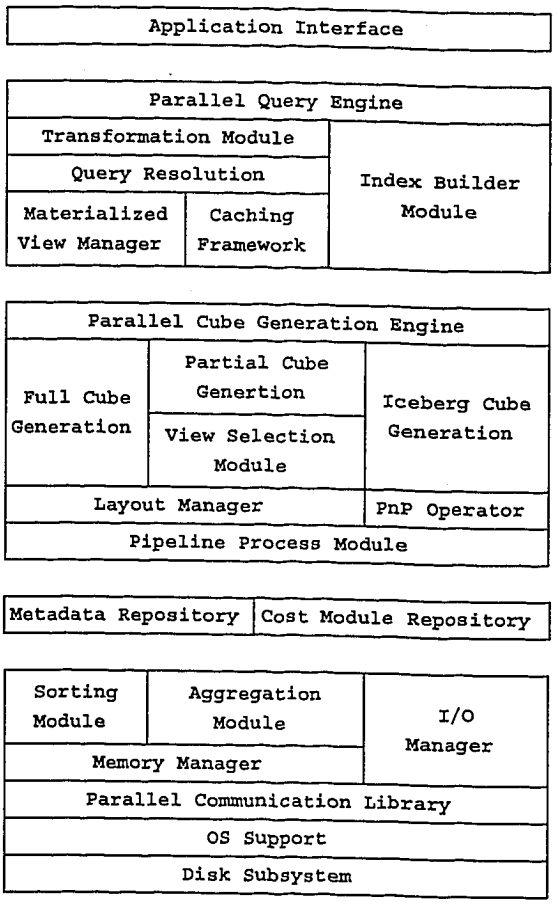



Figure 6.1: The cgmOLAP System Architecture.

5. **Shared server components.** These components provide shared modules used by upper components and uniform management of disk I/O, memory, communications and other resources. Of these components, the bottom three, the parallel cube generation engine, the metedata and cost model repository and the shared server components were all largely designed and implemented as part of this thesis. The components were built using C++, the MPI communication library and the LEDA data structure library. The core software components alone consist of over 80 files and 20,000 lines of codes. These components were designed for extensibility and heavily optimized. See Appendix A for the core systems' documentation.

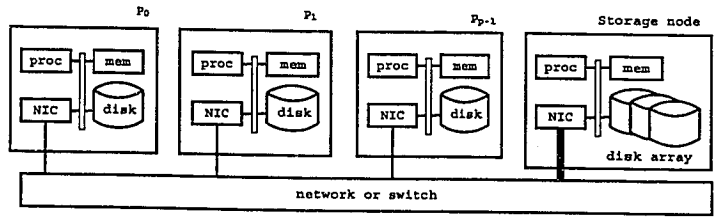



Figure 6.2: A Shared-Nothing Cluster with a Disk Array.

The hardware platform of the cgmOLAP system is low cost, shared nothing, commodity cluster architecture. A diagram of such an architecture is given in Figure 6.2. Consider a storage node as shown in Figure 6.2 that holds the initial, possibly very large, raw data set $R$ on a disk array. The cgmOLAP system fully distributes the data set $R$ over the $p$ local disks of processors $P_0$ to $P_{p-1}$ in striped format as shown in Figure 3.1. Similarly, each view created by the cgmOLAP system is generated over the $p$ local disks in striped format. The striped format ensures that subsequent accesses to an individual view generated by the cgmOLAP system can access all $p$ local disks in parallel, providing maximum I/O bandwidth, with balanced retrieval across the disks/processors. A particular feature of the cgmOLAP system is that it shows very high performance on this share-nothing cluster architecture.

## 6.3 Query Processing in cgmOLAP

While the parallel cube generation engine can be used to produce data cubes, the materialized views in the data cubes may still be quite large. Moreover, given the complex, multidimensional nature of the OLAP environment, a naive query implementation would significantly undermine the potential benefit of the materialized views. Therefore, it is crucial that an efficient, OLAP-centric query model be developed to complement the suite of cube generation algorithms developed in this thesis.

In the remainder of this section we briefly describe query processing in the cgmOLAP system not because it is a contribution of this thesis, but rather to show the interaction between cube generation algorithms and query processing in practice.

Figure 6.3 depicts the basic query model in our cgmOLAP system. In this example, the user requests a query on the view $ABC$ with the value ranges on each

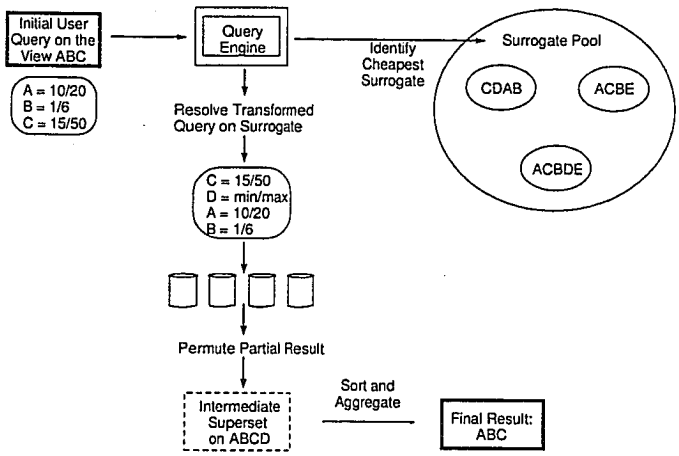



Figure 6.3: The Process of Resolving Queries Against Materialized Views.

dimension. If $ABC$ does not exist in the system cache, a surrogate view will be selected. Here the surrogate view is $CDBA$, and then the query will be transformed into a new query on the surrogate $CDBA$ and the ranges of $ABC$ are transformed into new ranges. After the result of the transformed query is answered, it is re-sorted and aggregated into the final answer on $ABC$.

In practice, OLAP queries tend to exploit dimension hierarchies such as year-month-day on a "time" dimension. The query engine supports hierarchical query resolution by way of an extensive dimension-aware caching subsystem. Not only does the caching framework improve the response time for general queries, but it can be used to efficiently resolve common hierarchical OLAP queries. Specifically, roll-up, drill-down, slice and dice, and pivot queries can directly manipulate the intermediate results sets associated with previous hierarchical queries. For example, if we need to answer a query on year, the query engine first searches for the intermediate cube including year. If there is no such cube, the query engine converts the query to a new query about the highest level of the "year-month-day" hierarchy available. Note that the base level in the hierarchy is always available, such as the day level in this example. After the result for the new query is found, the query engine converts the result from the day level back to the year level.

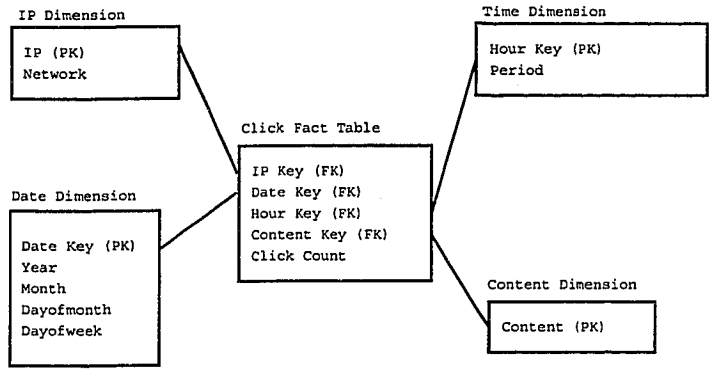



Figure 6.4: The Star Scheme for the Web Log Data Set.

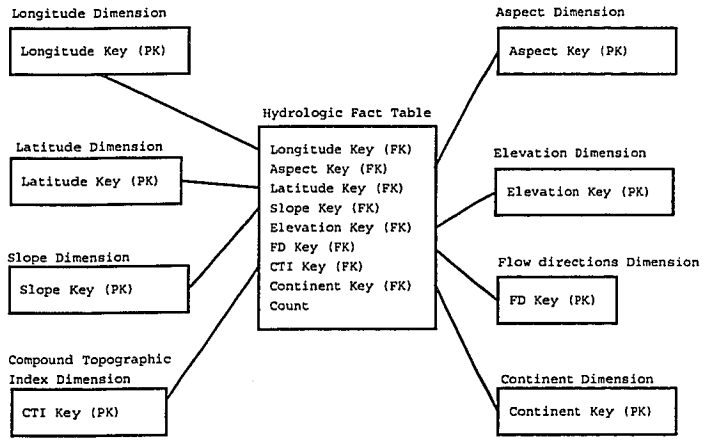



Figure 6.5: The Star Scheme for the World Hydrologic Data Set.

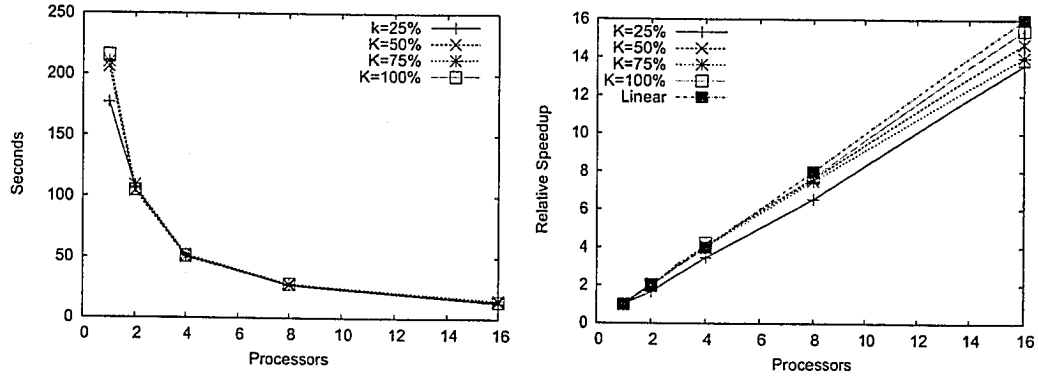
## 6.4 Experimental Evaluation of cgmOLAP on Real Data Sets

In this section we describe an experimental evaluation of the cgmOLAP cube generation algorithms using *real* (i.e non-synthetic) data sets. In order to convert these data sets to cgmOLAP internal format, an ETL process was employed. Using ETL tools provided in Oralce 9 for Linux, we converted two large real data sets into cgmOLAP internal format and we evaluate the parallel cube generation engine on these real data sets. The internal format is a structured binary file. The first integer indicates the number of rows in this file and the second integer indicates the number of columns in this file. The rest of the file is a two dimension integer array to store data. We first store the first row, then the second row, ..., and the last row. Binary files can be read into memory fast and data in memory can be written into binary files fast.

The first data set comes from the web logs of a local newspaper web site. For each day, there is an individual log file recoding every click event from users. Each line in the log file indicates a click event, and consists of the IP address, date, time and page accessed in this click event. We converted the log files into a click fact table and four dimension tables stored in an Oracle database. In this fact table, there were four dimensions (IP, date, time and content) and one measure (click counts). The tables were organized as a star schema, illustrated in Figure 6.4. The IP dimension includes 833471 unique addresses. The date dimension includes 366 unique days. The time dimension includes 24 unique hours. And the content dimensions includes 25 unique categories of contents. The fact table includes 154,729,804 rows and each row includes four dimensions and one measure.

The second data set comes from the *HYDRO1k Elevation Derivative Database* [5]. It is a geographic database developed to provide hydrologic information on a continental scale. The database includes data of six continents: North America, South America, Europe, Africa, Asia and Australia. For each continent, it divides the ground into the same size squares based on longitude and latitude, and then records elevation, aspect, compound topographic index, flow directions and slope for each square. We organized the data as a star schema as illustrated in Figure 6.5. In the hydrologic information fact table, there are eight dimensions: longitude, aspect, slope, elevation, flow directions, compound topographic index and continents. The number of unique values (cardinalities) for each dimension is 360, 360, 180, 90, 82, 8, 7 and 6 respectively. The fact table includes one measure: square counts. The total number of rows in the fact table is 124,676,260.

Figure 6.6 shows the running time and the relative speedup for generating full and partial data cubes from the web log data set. In Figure 6.6(b), we observe that the speedup is very close to linear for both full and partial data cubes. The full cube generation shows the best speedup, since much time is spent on local computing. The speedup for 50% selected views is better than that for 75% selected views, because both of them compute the similar number of views, while 75% selected views spend more time on communications. The speedup for 25% selected views drops because of insufficient local computing. Note that we cannot use the zipf value to indicate the data skew in the web log data set. However, the skew does exist in this real



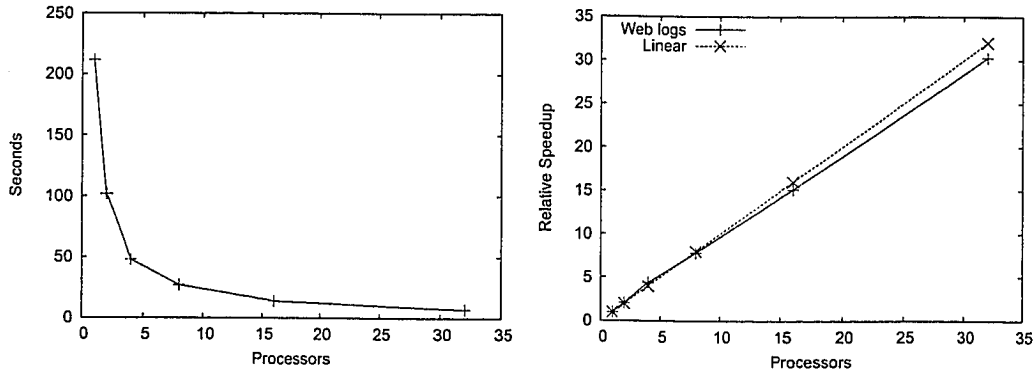

(a) (b)

Figure 6.6: (a) Parallel wall clock time in seconds as a function of the number of processors (b) corresponding speedup. (Fixed parameters: The data size $n$ = 32,631,392. Dimensions $d = 4$. Cardinalities $|D_i|$ = 833471, 366, 24, 25. Skew $\alpha$ = *Unknown*. The memory size $M$ = 100 Megabytes. The balance threshold parameter $\gamma = 5\%$)

data set. For example, some IP addresses belong to proxy servers, which have much higher access frequency than personal IP addresses. Another example is that users prefer one or two content pages over other contents on this web site. We know that about 40% clicks are about the obituary and birth pages on this local newspaper web site. In spite of these skew data, our parallel cube generation algorithm shows good speedup. We also observe that the running time for full cube, 75% selected views and 50% selected views is very close in Figure 6.6(a). The reason for this is that only four dimensions or 16 views are in this data sets. And when we compute 75% selected views and 50% selected views, we may add some intermediate views. This makes the total number of computed views is close to 16 for partial data cubes. For 25% selected views, only 4 views are selected so that the running time decreases.

Figure 6.7 shows the running time and the relative speedup for generating iceberg data cubes from the web log data set. We observe that the speedup is almost linear in Figure 6.7(b). In the web log data set, the first dimension and the second dimension have the relatively large cardinalities compared with the number of processors, so the work load is well balanced across the processors in our parallel PnP algorithm. This makes the speedup is almost linear even though the data set is skew.

Figure 6.8(a) shows the running time for full and partial data cube generation on



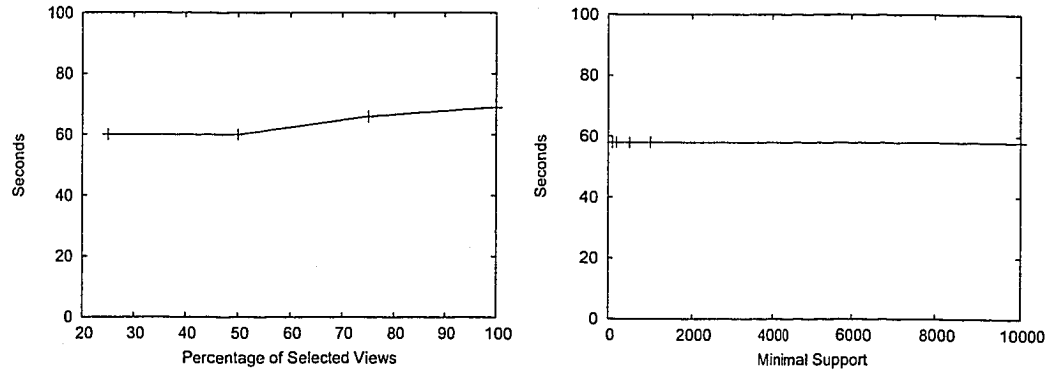

(a) (b)

Figure 6.7: (a) Parallel wall clock time in seconds as a function of the number of processors (b) corresponding speedup. (Fixed parameters: The data size $n$ = 32,631,392. Dimensions $d = 8$. Cardinalities $|D_i|$ = 833471, 366, 24, 25. The minimal support $m = 100$. Skew $\alpha = Unknown$. The memory size $M = 100$ Megabytes. )

the web log data set. We observe that the computing time is only one minute for the full data cube and less than one minute for partial data cube for this large web log data set, which is 3 gigabyte in size. We also observe that the running time for full and partial data cubes is very close because there are only four dimensions or 16 views in total. Therefore, partial data cubes might compute the similar number of views with the full cube, since our sequential partial data cube algorithm might add some intermediate views.

Figure 6.8(b) shows the running time for iceberg cube generation on the web log data set. We observe that the running time is less than one minute for this large data set with 150M rows. We also observe that the running time is constant for different minimal support values, because the aggregated measures are very large in this large size data set with small dimensions, and then as a consequence most of the data are output for all of the different minimal support values. That is also the reason that the running time of iceberg cube generation is so close to that of full and partial cube generation.

Figure 6.9 shows the running time and the relative speedup for generating full and partial data cubes from the world hydrologic data set. We observe that the speedup is close to linear when we use a small number of processors, such as two or




(a) (b)

Figure 6.8: (a) Parallel wall clock time in seconds as a function of different percentages of selected views and (b) Parallel wall clock time in seconds as a function of the minimal support. (Fixed parameters: The data size $n$ = 154,729,804. Dimensions $d$ = 4. Cardinalities $|D_i|$ = 833471, 366, 24, 25. The number of processors, $p$ = 16. Skew $\alpha = Unknown$. The memory size $M$ = 100 Megabytes. The balance threshold parameter $\gamma$ = 5%)

four, but the speedup drops a little when we use more processors, such as 8 or 16, however the speedup is still more than 80% of linear except in that case. For 25% selected views, the speedup drops to 60% of linear speedup because of insufficient local data on each processor. Note that we cannot use the zipf value to indicate the data skew in this world hydrologic data set either. However, the skew does exist in this world hydrologic data set. For example, deserts and lakes have the totally different hydrologic characteristics. In spite of these skew data, our parallel cube generation algorithm shows the good speedup.

Figure 6.10 shows the running time and the relative speedup for generating iceberg data cubes with different minimal support values from the world hydrologic data set. We observe that the relative speedup is typically super linear. The reason for this is twofold. Firstly, the parallel algorithm benefits from the large cardinality of the dimensions and is able to get a near perfect load balance. There is also sufficient local computation involved in this very large data set to overwhelm the overhead due to communication. Secondly, for this large size data set, the sequential algorithm has to extensively use disk based external memory operations while the parallel version deals with less data on each processor and can use more in-memory operation. We also

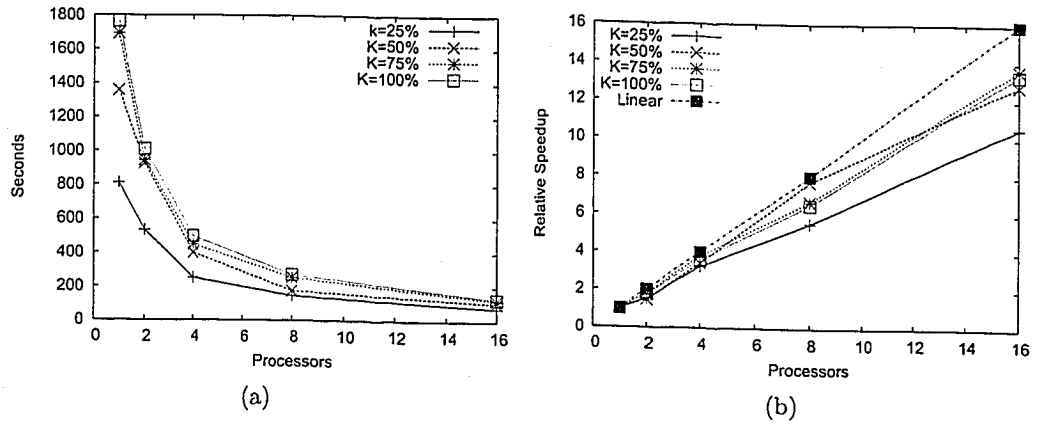


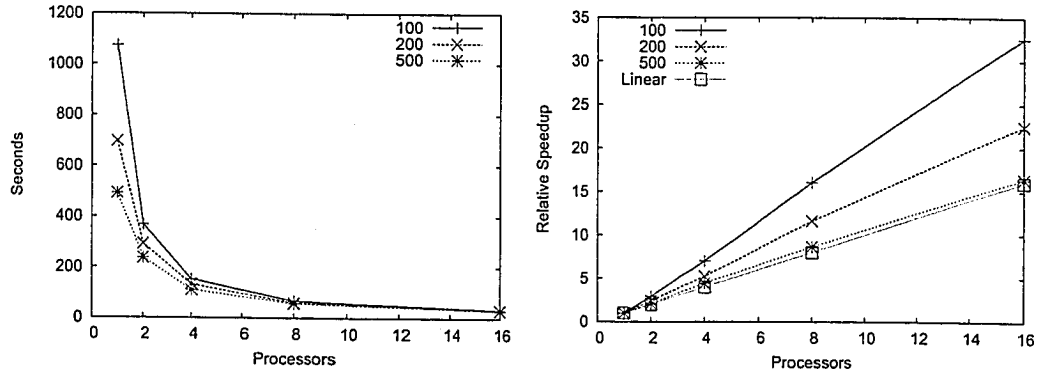

Figure 6.9: (a) Parallel wall clock time in seconds as a function of the number of processors (b) corresponding speedup. (Fixed parameters: The data size $n$ = 17,845,529. Dimensions $d = 8$. Cardinalities $|D_i| = 360, 360, 180, 90, 82, 8, 7, 6$. Skew $\alpha = Unknown$. The memory size $M = 100$ Megabytes. The balance threshold parameter $\gamma = 5\%$)

observe that the speedup increases when the minimal support values decrease. This is because the local computing cost goes up as minimal support decreases, making it easier to hide communication overhead.

Figure 6.11(a) shows the running time for full and partial data cube generation on the world hydrologic data sets. We observe that the running time increases with the percentage of selected views, since more time is needed to process the larger number of views. We also observe that the running time for 75% views is slightly larger than that for the full cube. This is unexpected. Apparently,, it is due to the fact that the sequential partial data cube algorithm cannot compute every view using a pipeline for 75% selected views, instead it sorts some parent views at a high cost. But for the full cube, the algorithm can always find a best parent view in a pipeline to compute every view.

Figure 6.11(b) shows the running time for iceberg cube generation on the world hydrologic data sets. We observe that the running time decreases when the minimal support value increases, because the output is smaller when we choose larger minimal support values.

The above experiments shows our algorithms run very efficiently on these large real data sets. It only takes one minute to generate full, partial or iceberg data cube




(a) (b)

Figure 6.10: (a) Parallel wall clock time in seconds as a function of the number of processors (b) corresponding speedup. (Fixed parameters: The data size $n$ = . Dimensions $d = 8$. Cardinalities $|D_i| = 360, 360, 180, 90, 82, 8, 7, 6$. The minimal support $m = 100$. Skew $\alpha = Unknown$. The memory size $M = 100$ Megabytes. )

for 150M row and 4 dimension data sets. It takes only 30 minutes to generate full cube for a 124M row and 8 dimension data set, and 15 minutes for the corresponding iceberg cubes. This is strong evidence that our parallel algorithms work well in practice on large scale real data sets.

## 6.5 Experimental Evaluation of cgmOLAP on Large Data Sets

In this section we evaluate the cgmOLAP cube generation algorithms using large synthetic data sets. Figure 6.12 shows the running time for generating full data cubes from large data sets of up to 250 million rows and the running time for the corresponding output data of up to one terabyte. We observe that running time is almost linear with both input data size and output data size. We also observe that for the output data size of 980 megabytes the running time is about 115 minutes. It shows our cgmOLAP system is able to build data cubes at a rate of more than half terabyte per hour on 32 nodes.

## 6.6 Summary

In this chapter, we introduced our cgmOLAP system, which is the first fully functional parallel OLAP system able to build data cubes at a rate of more than half terabyte per

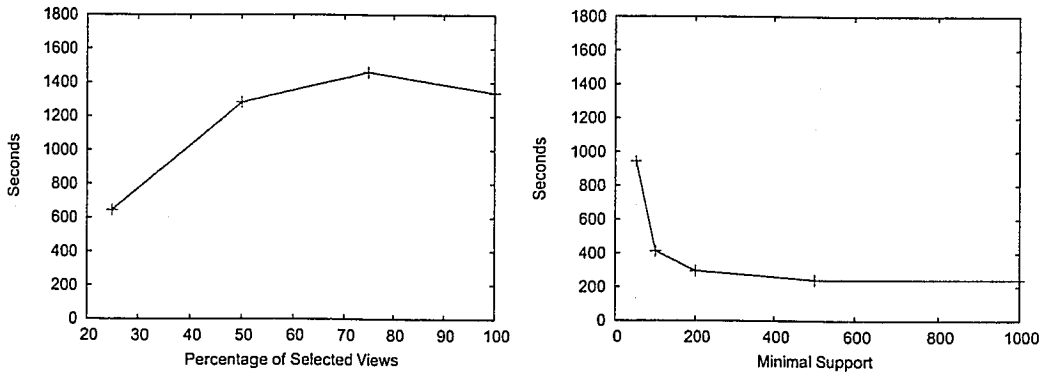



Figure 6.11: (a) Parallel wall clock time in seconds as a function of percentages of selected views and (b) Parallel wall clock time in seconds as a function of the minimal support. (Fixed parameters: The data size $n$ = 124,676,260. Dimensions $d$ = 4. Cardinalities $|D_i|$ = 360, 360, 180, 90, 82, 8, 7, 6. The number of processors, $p$ = 16. Skew $\alpha = Unknown$. The memory size $M$ = 100 Megabytes. The balance threshold parameter $\gamma$ = 5%)

hour. cgmOLAP integrates all of the algorithms, discussed in the previous chapters. Our experiments show that the parallel cube generation engine works well on large scale real data sets. Along with the parallel query engine, the cgmOLAP system can support massive hierarchical queries on terabyte data sets.

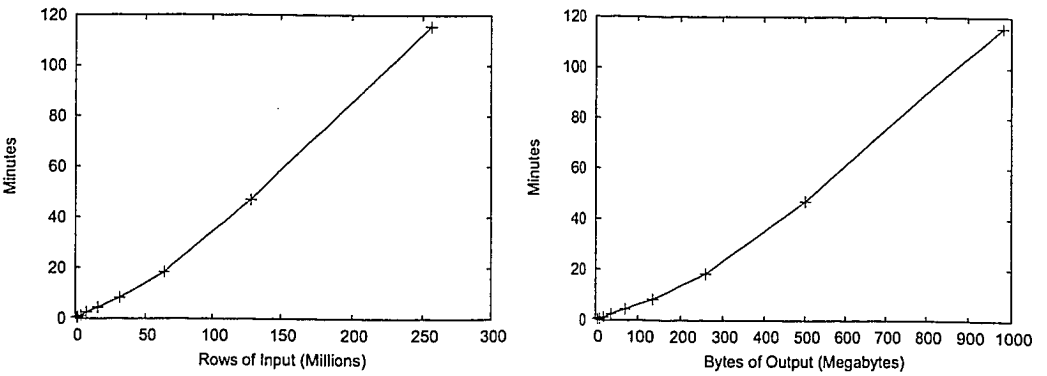



Figure 6.12: (a) Parallel wall clock time in minutes as a function of input data size and (b) the corresponding output data size. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256$ for $1 \leq i \leq d$. The number of processors, $p = 32$. Skew $\alpha = 0$. The memory size $M = 300$ Megabytes. The balance threshold parameter $\gamma = 5\%$)

# Chapter 7

# Conclusions and Future Work

This thesis has combined 1) the design of efficient parallel cube generation algorithms for the three basic types of data cubes: full cubes, partial cubes and iceberg cubes, 2) careful system work associated with parallelism and external memory issues, and 3) extensive experiments and evaluation. The resulting techniques are first reported in the literature that exhibit new linear speedup and good scalability for cube generation problems on shared nothing clusters of up to 16 processors.

In this thesis, we have designed and implemented parallel data cube generation algorithms for shared-nothing clusters. Based on these algorithms and implementations, here are a number of potential research directions that could be pursued:

1. **View Selection** In this thesis we assume that the user provides a list of views to be generated. In practice, a OLAP system may need to determine which views to generate in order to improve querying response itself. This is called the view selection problem. It would be interesting to explore view selection in the parallel context.

2. **Grid Computing** The Grid is a new type of parallel and distributed system. It can use geographically distributed autonomous resources to solve large problems. Grids are in many ways similar to clusters but exhibit much higher possible communication latency. It would be interesting to try to extend the parallel algorithms described in this thesis to the grid environment.

3. **Integration with Data Mining** Data Mining is another important decision support tools in data warehousing systems. Since data mining tasks can be executed on various data sources, including OLAP data cubes. It would be interesting to consider if we can use data cubes to speed up data mining tasks.

## Bibliography


[1] Beowulf Cluster. http://www.beowulf.org/. Last visited: May 9th, 2005.

[2] Business Intelligence Definition. WHATIS.COM. http://whatis.techtarget.com/. Last visited: May 9th, 2005.

[3] Business intelligence with SAS. SAS Corp. http://www.sas.com. Last visited: May 9th, 2005.

[4] Data warehousing guide. release 2(9.2). Oracle Corp. http://www.oracle.com. Last visited: May 9th, 2005.

[5] HYDRO1k Elevation Derivative Database. http://edcdaac.usgs.gov/gtopo30/hydro/index.asp. Last visited: May 9th, 2005.

[6] Hyperion Essbase OLAP Servers. http://www.hyperion.com/. Last visited: May 9th, 2005.

[7] Informix Dynamic Server. http://www-3.ibm.com/software/data/informix/ids/. Last visited: May 9th, 2005.

[8] Microsoft SQL Server OLAP Services. http://www.microsoft.com/sql/default.asp. Last visited: May 10th, 2005.

[9] MicroStrategy. http://www.microstrategy.com/Software/Products/Service_Modules/OLAP_Services/. Last visited: May 9th, 2005.

[10] OLAP and OLAP Server Definitions. http://www.olapcouncil.org/research/glossary.htm. Last visited: Match 6th, 2004.

[11] Open database connectivity (ODBC). Microsoft Corp. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/odbc/htm/ dasdkodbcoverview.asp. Last visited: May 9th, 2005.

[12] Oracle 9i OLAP. http://otn.oracle.com/products/bi/9iolap.html. Last visited: May 9th, 2005.

[13] The PANDA project. http:// www.cs.dal.ca/~panda/ Last visited: May 9th, 2005.

[14] Pilot Decision Support Suite. http://www.auriga.com/pilot_decision_.html. Last visited: May 10th, 2005.

[15] The Message Passing Interface (MPI) Standard. http://www-unix.mcs.anl.gov/mpi/. Last visited: May 9th, 2005.

[16] USING OLAP TO IMPROVE YOUR PERFORMANCE. Cognos Corp. http://www.cognos.com. Last visited: May 10th, 2005.

[17] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 506–521. Morgan Kaufmann Publishers Inc., 1996.

[18] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[19] S. Ceri S. Kant Batini, C. and B. Navathe. *Conceptual Database Design: An Entity Relational Approach.* The Benjamin/Cummings Publishing Company, 1991.

[20] Kevin Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and Iceberg CUBE. pages 359–370, 1999.

[21] Rajkumar Buyya. *High Performance Cluster Computing.* Prentice Hall PTR, Upper Saddle River, Ner Jersey 07458, 1999.

[22] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74.

[23] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel ROLAP data cube construction on shared-nothing multiprocessors. In *International Parallel and Distributed Processing Symposium (IPDPS2003)*, 2003.

[24] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Building large ROLAP data cubes in parallel. In *Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS '04)*, pages 367–377, 2004.

[25] Y. Chen, F.Dehne, T.Eavis, and A.Rau-Chaplin. Parallel ROLAP data cube construction on shared-nothing multiprocessors. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 70. IEEE Computer Society, 2003.

[26] Y. Chen, F.Dehne, T.Eavis, and A.Rau-Chaplin. Pnp: Parallel and external memory iceberg cube computation. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, 2005.

[27] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the data cube. *International Conference on Database Theory*, 2001.

[28] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the data cube. *Distributed and Parallel Databases*, 11(2):181–201, 2002.

[29] F. Dehne, T. Eavis, and A. Rau-Chaplin. Computing partial data cubes for parallel data warehousing applications. *Euro PVM/MPI 2001*, 2001.

[30] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[31] The Rising Storage Tide. DataWarehousing.com http://www.datawarehousing.com/papers/ storage_tide.asp. Last visited: May 10th, 2004.

[32] Todd Eavis. *PARALLEL RELATIONAL OLAP.* PhD thesis, DALHOUSIE UNIVERSITY, 2003.

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

[34] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. *in Proceedings VLDB*, pages 299–310, 1998.

[35] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17:37–54, 1996.

[36] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[37] Andreas Fabri Frank Dehne and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *SCG '93: Proceedings of the ninth annual symposium on Computational geometry*, pages 298–307, New York, NY, USA, 1993. ACM Press.

[38] Todd Eavis Frank Dehne and Andrew Rau-Chaplin. Top-down computation of partial ROLAP data cubes. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, 2004.

[39] S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, 1(4):391–417, 1997.

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

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

[42] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[43] L. Feng H. Lu, J.X. Yu and X. Li. Fully dynamic partitioning: Handling data skew in parallel data cube computation. *Distributed and Parallel Databases*, 13:181, 2003.

[44] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 311–322. Morgan Kaufmann Publishers Inc., 1995.

[45] Jiawei Han, Yongjian Fu, Wei Wang, Jenny Chiang, Wan Gong, Krzysztof Koperski, Deyi Li, Yijun Lu, Amynmohamed Rajan, Nebojsa Stefanovic, Betty Xia, and Osmar R. Zaiane. DBMiner: A system for mining knowledge in large relational databases. In *Proc. 1996 Int'l Conf. on Data Mining and Knowledge Discovery (KDD'96)*, pages 250–255, Portland, Oregon, 1996.

[46] W.H. Inmon. *Building the Data Warehouse.* John Wiley, 1992.

[47] Max Planck Institute. *LEDA.* http://www.mpi-sb.mpg.de/LEDA/. Last visited: May 9th, 2005.

[48] M. Kamber J. Han. *Data Mining: Concepts and Techniques.* Morgan Kaufmann, San Francisco, CA, 2001.

[49] Thomas P. Nadeau Kanda Runapongsa and Toby J. Teorey. Storage estimation for multidimensional aggregates in OLAP. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 10. IBM Press, 1999.

[50] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit. The Complete Guide to Dimensional Modeling.* John Wiley and Sons, Inc., 2002.

[51] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, 1993.

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

[53] Joerg Reinschmidt Maria Sueli Almeida, Missao Ishikawa and Torsten Roeber. *Getting Started with DataWarehouse and Business Intelligence.* IBM International Technical Support Organization. http://www.redbooks.ibm.com, August 1999.

[54] Michael W. Hawkins Mark Humphries and Michelle C. Dy. *Data Warehousing Architecture and Implementation.* Prentice Hall PTR.

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

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

[57] Kenneth A. Ross and Divesh Srivastava. Fast computation of sparse datacubes. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 116–125. Morgan Kaufmann Publishers Inc., 1997.

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

[59] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

[60] Abraham Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman. Database research: Achievements and opportunities into the 21st century. *SIGMOD Record*, 25(1):52–63, 1996.

[61] C. Leiserson T. Cormen and R. Rivest. *Introduction to Algorithms.* The MIT Press, 1996.

[62] Alejandro Ariel Vaisman. Olap, data warehousing, and materialized views: a survey. http://citeseer.nj.nec.com/vaisman98olap.html. Last visited: May 9th, 2005.

[63] The Winter Report. http://www.wintercorp.com/vldb/ Last visited: May 10th, 2005.

[64] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. *in Proceedings Int. Conf. on Very Large Data Bases (VLDB'03)*, 2003.

[65] Yihong Zhao, Prasad M. Deshpande, and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997*, pages 159–170, 1997.

[66] G.K. Zipf. *Human Behavior and The Principle of Least Effort.* Addison-Wesley, 1949.

# Appendix A

# Parallel Data Cube Generation Library

In this appendix we describe the *Parallel Data Cube Generation Library.* The library implements the bottom three components in our cgmOLAP system: parallel cube generation engine, meta data and cost model repositories and shared server components. We first present modules and classes in the library, and then give more details about implementations of core algorithms. Lastly, we introduce the steps for building the library and execute the main applications of the library.

## A.1 Modules and Classes

The library is organized in a hierarchy structure, illustrated in Figure A.1. There are nine modules in it. More modules could be added into the library in the future. In each module, there are one or more C++ classes, which support functions of the module, illustrated in Figure A.2.

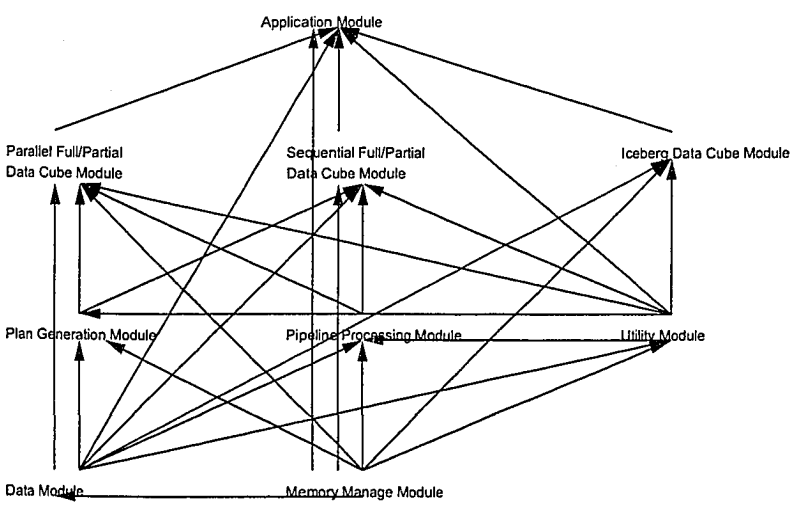



Figure A.1: Module Structure

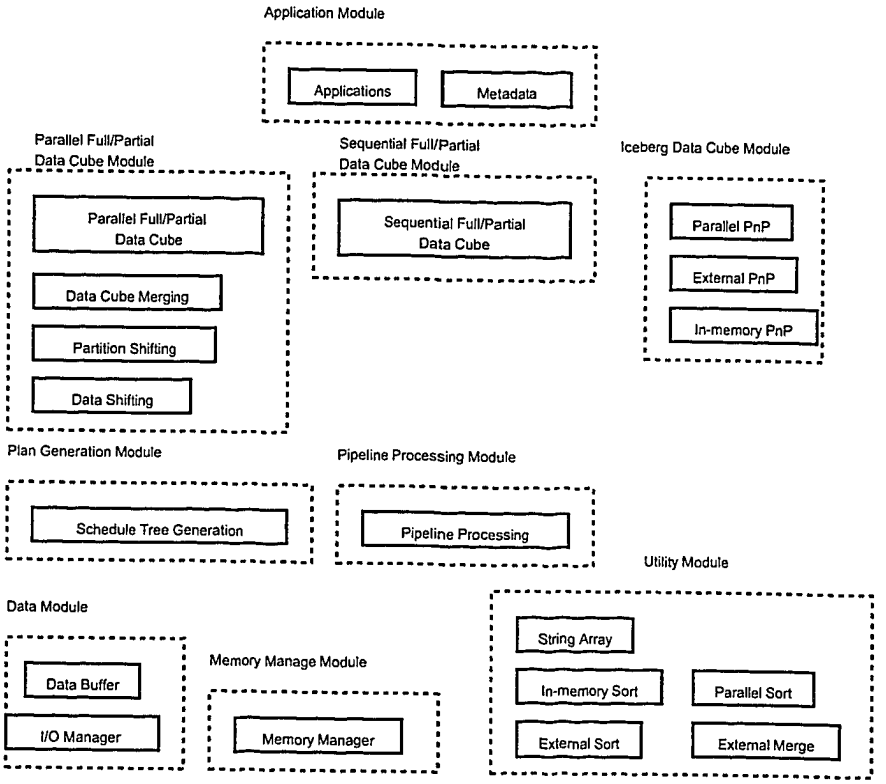



Figure A.2: Modules and Classes

### A.1.1 Data Module

One of basic modules is the data module. It provides all functions to input, output and cache the data in relational tables for ROLAP.

The base class in this module is *Data Buffer*, which implements most manipulation functions on the data of a single table, such as reading data from a file, writing results to a file, caching data, and swapping data between disks and main memory. The data buffer also supports multiple measures, and multiple aggregation functions by defining a set of comparing and aggregating function. The source files of this class are *databuffer.cpp* and *databuffer.h*.

The second class in this module is *I/O manager*. It implements an asynchronous disk I/O, so that the time for disk I/O can be overlapped with then time for computing and communications. The source files of this class are *iomanager.cpp* and *iomanager.h*.

### A.1.2 Memory Manage Module

The memory manage module is in charge of memory allocation for all other classes. It keeps the size of available memory in the system, the size of working space for some external memory algorithms, and the size of cache for data buffers. The only class in this module is *memory manager*. The source files of this class are *memorymanager.cpp* and *memorymanager.h*.

### A.1.3 Utility Module

The utility module provides some basic functions for other modules, such as sorting, merging and string array.

The class of *string array* is a simple array, whose elements are strings. This class are always used to store sets of view name. The source files of this class are *stringarray.cpp* and *stringarray.h*.

The classes of sorts are *in-memory sort*, *external memory sort*, and *parallel sort*. The in-memory sort is implemented by quick sort. In external memory sort, data are read into memory block by block, and then are sorted in memory and output into disks. After that the data are merged into a single data set by executing *external merge*. The source files of these class are *quicksort.cpp* and *quicksort.h*, *localsort.cpp* and *localsort.h*, *parallelsort.cpp* and *parallelsort.h*, *mergesort.cpp* and *mergesort.h*

### A.1.4 Plan Generation Module

The plan generation module is used in full/partial data cube generation only. It generates a schedule tree to compute full/partial data cube. The classes in this module are *pipeline tree* and some other classes related with LEDA library. The source files of these class are *pipelinetree.cpp* and *pipelinetree.h*, and *balance_data.cpp*,*balance_data.h*, *edges.cpp*, *edges.h*, *lattice.tmpl*, *nodes.cpp*, *nodes.h*, *partial_base.tmpl*, *partial_lattice.tmpl*, *partial_pipesort.tmpl*, *plan.cpp*, *plan.h*, *plan_data.cpp*, *plan_data.h*, *sort_lattice.tmpl*, *utility.cpp*, *utility.h*.

### A.1.5 Pipeline Processing Module

Pipeline Processing Module is used in full/partial data cube generation only. It executes a schedule tree generated in the plan generation module. The only class in this module is *pipeline*. The source files of this class are *pipeline.cpp* and *pipeline.h*.

### A.1.6 Parallel Full/Partial Data Cube Module

The parallel full/partial data cube module provides all functions for parallel full or partial data cube generation. The algorithms used in this module are based on our published papers. The classes in this module are *data cube merging*, *partition shifting* and *data shifting*. The source files of this class are *mergecube.cpp* and *mergecube.h*, *shiftpartition.cpp* and *shiftpartition.h*, *shiftdata.cpp* and *shiftdata.h*. Another file, which defines "main" function, is *papcube.cpp*.

### A.1.7 Sequential Full/Partial Data Cube Module

The sequential full/partial data cube module implements sequential full or partial data cube generation. The algorithms used in this module are based on our published papers. The only file, which defines "main" function, is *expcube.cpp*.

### A.1.8 Iceberg Data Cube Module

The iceberg data cube module provides all functions for iceberg data cube generation, including parallel iceberg data cube generation, sequential external memory data cube generation and sequential in-memory data cube generation. The methods used in this module are based on our PnP algorithm for parallel and external memory data cube generation. The source files of these class are *iceberg.cpp* and *iceberg.h*, *exiceberg.cpp* and *exiceberg.h*, *paiceberg.cpp* and *paiceberg.h*.

### A.1.9 Applications Module

The application module provides some applications by using all other modules, and metadata used to describe data cubes. The applications module is included in the library.

## A.2 Implementation of Algorithms

The library implements five main algorithms, which are external memory full/partial data cube generation, parallel full/partial data cube generation, in-memory iceberg data cube generation, external memory iceberg data cube generation and parallel iceberg data cube generation. The two parallel algorithms are based on the two external algorithms respectively.

For each algorithm, there a C++ file to implement the mainframe of the algorithm in the "main" function of C++ language. The files are expcube.cpp for external memory full/partial data cube generation, papcube.cpp for parallel full/partial data cube generation, pnp.cpp for in-memory iceberg data cube generation, expnp.cpp for external memory iceberg data cube generation, and papnp.cpp for parallel iceberg data cube generation.

### A.2.1 External Memory Full/Partial Data Cube Generation

The algorithm of external memory full/partial data cube generation includes two steps: Generating Processing Tree and Executing Processing Tree. The class "CPipelinetree" implements the first step, and the class "CPipeline" implements the second step. The C++ file "expcube.cpp" implements the algorithm. It calls "CPipelinetree" to generate the processing tree first, and then calls "CPipeline" to execute the processing tree. At last it outputs data to disks.

### A.2.2 External Memory Parallel Full/Partial Data Cube Generation

The C++ file "papcube.cpp" implements the mainframe of this algorithm. First it reads configurations, and then does some testing for system parameters using the class "CTestio". Before beginning the algorithm, it distributes the raw data to each node from node 0 using the class "CCutdata", so that each node gets almost the same amount of data.

The first step of the algorithm is to generation root views on each node using the function of "rootview" in the class of "CLocalsort". The next step is a big loop to generate a number of partial data cubes for the corresponding root view in parallel one by one. The turns of the loop is the number of the dimensions.

In each loop, a sub-lattice is generated first to consist of all the views to be computed in the loop. Then the corresponding root view is sorted in parallel and is partitioned using the class of "CParallelsort". Next the main C++ file calls the class of "CPipelinetree" to generate the processing tree, and a shifting processing is executed to optimize the partitions using the class of "CShiftpartition". After that, the main file calls the class of "CPipeline" to execute the processing tree locally. Next the class of "CMergecube" is executed to merge data cubes across the nodes. At last "CShiftdata" is executed to balance the amount of data for each view across the nodes, and then the results are written to disks.

### A.2.3 In-memory Iceberg Data Cube Generation

The algorithm of in-memory iceberg data cube generation is implemented in the files of iceberg.cpp and iceberg.h, although the file pnp.cpp consists of "main" function for the in-memory iceberg data cube application.

The main function in the iceberg.cpp is "computeiceberg". It sorts the input data set, and then calls the recursive function of "iceberg", which implements the PnP operator of in-memory PnP algorithm. "iceberg" scans the input view, and aggregates the measures for each view in the pipeline beginning with the input view. If the measure of a row in a view is equal or greater than the minimal support, the row is output and a partition of the child view of the view is generated and sorted. Immediately after that "iceberg" is called on the partition. The recursive calls continue until there is no child view available.

#### The Iceberg Function

The "iceberg" function is the core procedure to implement the in-memory PnP operator. In order to compete with other algorithms, we optimized it heavily so that it is hard to read. The following is a brief description about this function.

The variable *pipelen* is the length of pipelines.

The variable *agg* is the array of aggregation values. *agg*[0] stores the aggregation of the first view in the pipeline, and *agg*[*pipelen* − 1] stores the aggregation of the last view. For example, if the input view is $ABCDE$, agg[0] is the aggregation of $ABCDE$, and agg[pipelen-1] is the aggregation of $A$

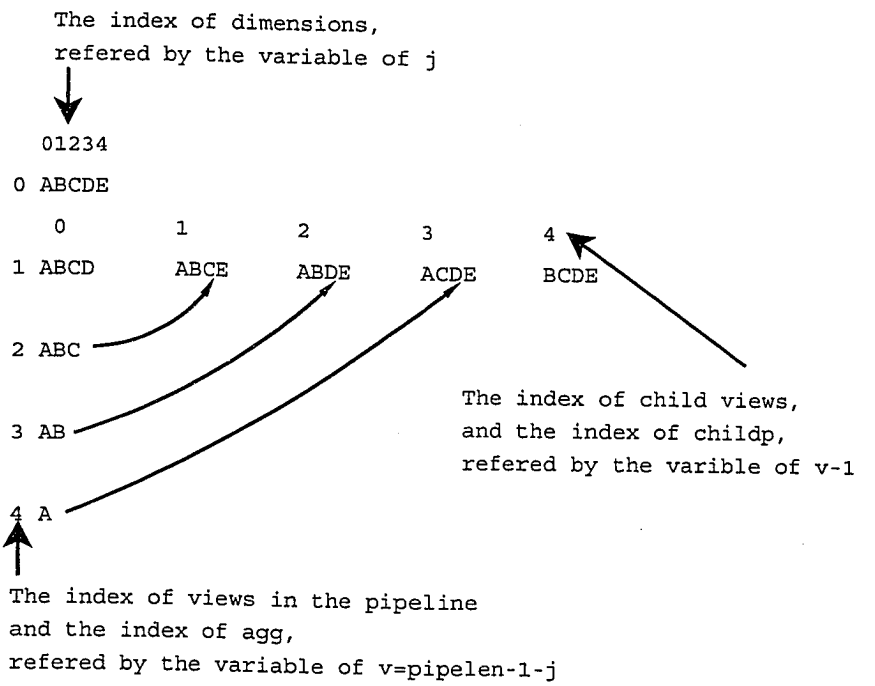



Figure A.3: Variables in the Iceberg Functions

The variable $v$ is the index of *agg*. It is always equal to $pipelen - 1 - j$, where $j$ is the index of the dimensions.

The variable *childp* is an array of the start indexes of the partitions of the child views. *childp*[0] and *childp*[$pipelen - 1$] are NULL. For example, if the input view is $ABCDE$, *pipelen* is 5. And the start index of the partitions of $ABCE$ is (*childp*[1]), and the start index of the partitions of $ACDE$ is (*childp*[3]). *childp*[0] and *childp*[4] are NULL, because there are no partitions to be generated for $ABCD$ and $BCDE$. We use $v - 1$ as the index of *childp*. *childp* is updated as soon as a partition is found.

In *iceberg*, the main loop ($i$) is to read rows from the input data, and the secondary loop ($j$) is to check the value of each dimension. If the value of the dimension $j$ does not change, we add the measure of the current row to agg[v], where $v$ is $pipelen-1-j$. If it changes, we start a new loop from $v = 0$ to $v = pipelen - 1 - j$. In this loop, we compare agg[v] with the iceberg condition. If agg[v] is greater than the iceberg condition, we output the rows for the view of $v$ and generate a partition for the child views of $v - 1$ when $v >= 2$. And then we call *iceberg* on the partition immediately.

After all the rows are finished, we sort the input view to the last child view, and call *iceberg* on the last child view immediately.

### A.2.4 External Memory Iceberg Data Cube Generation

The algorithm of external Memory iceberg data cube generation is implemented in the files of exiceberg.cpp and exiceberg.h, although the file expnp.cpp consists of "main" function for the external memory iceberg data cube application.

The main function in the exiceberg.cpp is "computeiceberg". It sorts the input data set, and then calls the recursive function of "iceberg", which implements the PnP operator of external memory PnP algorithm. "iceberg" calls the function of "pipeline" to execute the PnP operator in external memory. "pipeline" scans the input view, and aggregates the measures for each view in the pipeline beginning with the input view. If the measure of a row in a view is equal or greater than the minimal support, the row is output and a partition of the child view of the view is generated and sorted. And the partition is stored in memory or in disks if memory is not enough. After "pipeline" finish the current input view, it returns to "iceberg". "iceberg" picks up partitions and call "pipeline" on them one by one until no partition is available.

### A.2.5 External Memory Parallel Iceberg Data Cube Generation

The C++ file "papnp.cpp" implements the mainframe of this algorithm. First it reads configurations and distributes the raw data to each node from node 0 using the class "CCutdata", so that each node gets almost the same amount of data.

The first step of the algorithm is to generation root views on each node using the function of "rootview" in the class of "CLocalsort". The next step is a loop, which runs $m$ times, where $m$ is the number of the dimensions. In this loop, the corresponding root view is sorted in parallel and be partitioned using the class of "CParallelsort". The last step is another loop, where iceberg data cubes are generated from the corresponding root views. In this loop, the functions in "paiceberg.cpp" and "paiceberg.h" are used to execute PnP operator. They are similar to ones in "exiceberg.cpp", except that only part of iceberg cubes are computed. For examples, for the root view $ABC$, only $ABC$,$AB$,$AC$ and $A$ are computed.

## A.3 Build the Library

The library is written in ANSI C/C++. It can be compiled on any operation system with C/C++ compiler, such as Windows, Linux and other UNIX system. The library uses MPI to exchange message among nodes, so an MPI library in C language is needed to link objective files into executable files. In the plan generation module, some classes come from LEDA library in C/C++ language. Before you compile the library, make sure the following components are in your system.

- The source codes of the library.
- MPI library in C language.
- LEDA library in C/C++ langauge.
- C/C++ compiler and linker.

A free version of MPI library in C language can be downloaded from http://www.lam-mpi.org/. And a free LEDA library can be downloaded from (http://www.mpi-sb.mpg.de/LEDA/leda.html).

Executable files for Linux are included in the source codes already. You may run them without any change.

To compile the source codes, you may go to the directory of source codes. There five make files for individual five applications.

**expcube.mak** It generates "expcube", which is for sequential full/partial data cube generation.

**papcube.mak** It generates "papcube", which is for parallel full/partial data cube generation.

**pnp.mak** It generates "pnp", which is for in-memory iceberg data cube generation.

**expnp.mak** It generates "expnp", which is for external memory iceberg data cube generation.

**papnp.mak** It generates "papnp", which is for parallel iceberg data cube generation.

You may use the command "make -f makefilename" to generate five executable files respectively. The objective files are in the directory of "objfile", and the executive files are in the directory of "exefile".

## A.4 Execute Applications of the Library

After you generate executable files, you may run them to generate full/partial or iceberg data cube in parallel or in sequence. All the five executive files share one configuration file, whose name is "conf.dat". It must be in the directory, where executable files are. A sample conf.dat is in the directory of "exefile" of source codes. You may find more details from "config.cpp" and "config.h" in this manual about the entries of "conf.dat".

In full/partial data cube generation, we need to set up a flat file to list target views to be generated. In this file, each line includes one view name, represented by a letter string. A sample flat file is in the directory of "exefile" of source codes. The file name is "viewfile".

The library uses synthetic data as input data, which can be generated by a tool "dg". There is a Linux executable file "dg" in the directory of "exefile" of source codes. "dg" needs a flat file as input parameters. Here is an example:

r 10000
d 8
w input.dat
z 0
c01 1024
c02 512
c03 256
c04 128
c05 64
c06 32
c07 16
c08 8
c09 100

"r 10000" means 10000 rows. "d 8" means 8 dimensions. "w input.dat" means the output file name. "z 0" means the skew of data. "z" can be from "0" to "2". "0" means no skew and "2" means high skew. The rest of lines are cardinalities od each columns.

A sample of the flat file is in the directory of "exefile" of source codes. The file name is "g".

Another Linux executable in the directory of "exefile" of source codes is "peek", which can display the data in data files. Running "peek" without parameters may display details about "peek".

For sequential executable files, you may run it directly. For parallel executable files, you need to use "mpirun", like "mpirun -np 16 papcube". The output files for full/partial data cubes can be check using "peek". The output files for iceberg data cubes are flat files, which can be display by any text editors.

## A.5 Summary

The parallel data cube generation library implements the core components of the cgmOLAP system. It provides functions and data strictures used by the upper components in cgmOLAP. Most experiments in the thesis are also done by using functions and strictures provided by the library. These experiments show that the library is well scalable and very reliable.