Datacube:
Its Implementation and Application in OLAP Mining

by

Yin Jenny Tam

B.Sc., Simon Fraser University, 1995

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Computer Science

SIMON FRASER UNIVERSITY
September 1998

APPROVAL

Name: Yin Jenny Tam

Degree: Master of Science

Title of thesis: Datacube: Its Implementation and Application in OLAP Mining

Examining Committee: Dr. Robert Hadley
Chair

Dr. Jiawei Han
Senior Supervisor

Dr. Veronica Dahl
Supervisor

Dr. Wo-Shun Luk
External Examiner

Date Approved:

Abstract

With huge amounts of data collected in various kinds of applications, data warehouse is becoming a mainstream information repository for decision support and data analysis mainly because a data warehouse facilitates on-line analytical processing (OLAP). It is important to study methods for supporting data warehouses, in particular its OLAP operations, efficiently. In this thesis, we investigate efficient methods for computing datacubes and for using datacubes to support OLAP and data mining. Currently, there are two popular datacube technologies: Relational OLAP (ROLAP) and Multidimensional OLAP (MOLAP). Many efficient algorithms have been designed for ROLAP systems, but not so many for the MOLAP ones. MOLAP systems, though may suffer from sparsity of data, are generally more efficient than ROLAP systems when the sparse datacube techniques are explored or when the data sets are small to medium sized. We have developed a MOLAP system which combines nice features of both MOLAP and ROLAP. Our performance study shows that such an integrated system is faster than plain MOLAP or ROLAP systems in many cases. We also discuss the promising direction of OLAP mining (data mining integrated with OLAP). On top of this MOLAP system, we developed some OLAP mining modules to assist users to discover different kinds of knowledge from the data. Decision makers often do not know exactly what they want when they do OLAP or mining, so OLAP mining helps them explore the data flexibly from different angles and at multiple abstraction levels.

Acknowledgments

I would like to thank my senior supervisor, professor Dr. Jiawei Han, for introducing me to the data warehousing and data mining fields, which are very exciting and booming. My gratitude goes to him for his confidence in me to have assigned me as one of the chief designers and programmers of the OLAP (core) engine of DBMiner. He has been more than available throughout my research, development, and the writing of my thesis. His continuing support and guidance has been invaluable to me. Thanks also goes to my supervisor, professor Dr. Veronica Dahl, and external examiner, professor Dr. Wo-Shun Luk, for reading my thesis and making useful suggestions.

In addition, I would like to thank Sonny Chee for his observant and valuable comments that inspired and focused my research work. Moreover, it was great and fun working with him on the design and implementation of DBMiner.

I would also like to thank my parents and my family for their continuous love, patience, confidence, support and prayers. Without their understanding my research could not have been realized.

Appreciation and love also go to my husband, Dominic Tam, who has been lovingly and prayerfully supporting, understanding, and encouraging me since the beginning of my research. I also cherished his reminders and technical assistance while developing DBMiner.

Finally, and most important of all, I love to thank God for giving me the chance to continue my studies in Simon Fraser University as a master student. Besides, He has been listening to our prayers and has been inspiring my research and given me the wisdom. Without Him, my thesis could not have been written.

Dedication

To Mom, Dad, and Dominic.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The research and development in database systems since the 1970's has led to the development of sophisticated relational database systems from simple file processing systems. The low-cost, high-speed microprocessors allow different data types such as multimedia "documents", images, time-series, procedural or "active" data, and other complex data forms [SSU96] to be stored in relational table structures. As a result, many companies in the business world store, organize, and update their inventories, sales history, customers information, marketing information, etc. in a collection of large databases. For example, one of the largest databases in the world was created by WalMart (a U.S. retailer), which handles over 20 million transactions a day. Other non-business organizations have also collected tremendous amounts of data. Most health-care transactions in the U.S. are being stored in computers, which yields multi-gigabyte databases. The NASA Earth Observing System (EOS) of orbiting satellites and other space borne instruments is projected to generate on the order of 50 gigabytes of remotely sensed image data per *hour* in their scientific databases [FPSS96a].

If the accumulated data is analyzed accurately, it can be turned into competitive weapons. Unfortunately, the fast-growing, tremendous amounts of data collected has far exceeded the human ability for comprehension and examination. They clearly overwhelm the traditional manual methods of data analysis such as spreadsheets and ad-hoc queries [Tho97]. Thus, a lot of important decisions are made based on the decision makers' intuition, experience or expertise rather than on the information rich data stored in databases. As a result, large databases become "data tombs" and this phenomenon has been described as "data rich but information poor" problem [FPSS96a].

In today's competitive, fast evolving world, the decision makers definitely require a new generation of computational techniques and tools to assist them in extracting useful information (knowledge) embedded in the data collected and accumulated at a dramatic pace [FPSS96a]. Yet these decision makers are often non-technical. They would desire the decision support tools to be in an integrated environment such that they can create their own standardized or ad-hoc reports for complex data analysis. By taking advantage of the large volume of data, they can probably identify potential cost-savings, reach new markets, and track revenue effectively.

A new breed of sophisticated decision support systems — the **data warehouse** — have begun to appear. They are valuable tools to many organizations because they provide a collection of decision support technologies to enable the executives, managers, analysts, etc., to make decisions faster and more accurately. In the past three years, we have seen explosive growth, both in the number of products and services offered, and in the adoption of these technologies by industry. In fact, the data warehousing market is projected to grow from $2 billion in 1995 to $8 billion in 1998 [CD97].

1.1 Data Warehousing

1.1.1 What is a Data Warehouse?

Data warehouses have been defined in many ways, and there is not yet a standard definition. Some people say that a data warehouse is an *instantiated view* of some integrated information to build mediators, which are in effect views of many integrated information sources. The warehouse can physically store the integrated information, which is made available by forwarding queries [SSU96].

In general, a data warehouse refers to a database that is maintained separately from an organization's operational databases. It is often integrated with a variety of application systems to support information processing and data analysis by providing a solid platform for consolidated, historical data. W. H. Inmon, a leading architect of data warehouse construction, defined a data warehouse as a "**subject-oriented**, **integrated**, **time-variant**, and **nonvolatile** collection of data in support of management's decision-making process" [Inm92].

The term **data warehousing** refers to the process of constructing and using data warehouses. Decision support tools usually require consolidating data from many heterogeneous sources, which might include external sources such as stock market data. These sources have to be reconciled because they might contain data of varying quality or use inconsistent representations, codes and formats. Thus, the construction of a data warehouse requires data integration, data cleaning, and data consolidation [CD97].

1.1.2 A Multidimensional Data Model

Business managers tend to think multidimensionally. For example, they tend to describe what the company does as follows [Kim96]:

"We sell products in various markets, and we measure our performance over time."

Data warehouse designers often listen carefully to these words and they add their own special emphasis:

"We sell *Products* in various *Markets*, and we measure our performance over *Time*."

Most people find it intuitive to think of the business as a **cube** of data, with labels on each edge of the cube, as shown in Figure 1.1 [Kim96]. Any point inside the cube is at the intersection of the coordinates defined by the edges of the cube. For the business described above, the edges of the cube are *Product*, *Market*, and *Time*. Most people can quickly understand and imagine that the points inside the cube are where the measurements of the business for that combination of *Product*, *Market*, and *Time* are stored.

Data warehouses are constructed based on the **multidimensional data model**.

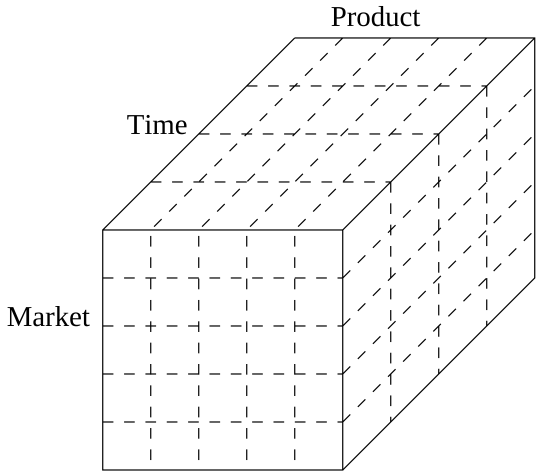


Figure 1.1: The dimensional model of a business [Kim96].

A datacube in data warehousing is not necessarily a three-dimensional (3-D) geometric structure, but is essentially N dimensional (N-D). The edges of the cube are called **dimensions**, which are the perspectives or entities with respect to which an organization wants to keep records. Each dimension may be associated with a **dimension table**, which describes the dimension. For example, a dimension table for *Product* may contain such attributes as *product_key*, *description*, *brand*, *category*, etc., which can be specified by managers or data analysts. For those dimensions that are non-categorical, such as *Time*, the data warehouse system should be able to automatically generate the corresponding dimension table based on the data distribution. As a side note, the *Time* dimension is in fact of particular significance to decision support for trend analysis. Often it is desirable to have some built-in knowledge of calendars and other aspects of the time dimension [CD97].

In addition, a datacube in data warehousing is mainly constructed to measure the company's performance. Thus, a typical multidimensional data model is organized around a theme, which is represented by a **fact table** of some numerical measures — the objects of analysis. For example, a fact table may contain *sales*, *budget*, *revenue*, *inventory*, *number of items sold*, etc. Each of the numerical measures depends on a set of dimensions, which provide the context for that measure. Therefore, the dimensions together are assumed to *uniquely* determine the measure, which is a value in the multidimensional space of dimensions [CD97, Sar97].

Dimensions are hierarchical by nature [Sar97]. For example, dimension *Time* can be described by the attributes *Year, Quarter, Month*, and *Day*. Alternatively, the attributes of a dimension may be organized into a **lattice**, which indicates a partial order for the dimension. That is, the same *Time* dimension can have *Year, Quarter, Month, Week*, and *Day* instead. With this scheme, the *Time* dimension is no longer a hierarchy because some weeks in the year may belong to different months [HCC93].

Therefore, if each dimension contains multiple levels of abstraction, the data can be viewed from different perspectives flexibly. A number of typical datacube operations: **roll-up** (increasing the level of abstraction), **drill-down** (decreasing the level of abstraction or increasing detail), **slice and dice** (selection and projection), and **pivot** (re-orienting the multidimensional view of data), exist to allow interactive querying and analysis of the data at hand. These operations are known as **On-Line Analytic Processing** (OLAP).

Decision makers would like to ask questions like "compute and rank the *total sales* by each country (or by each year)". They would also like to *compare* two numerical measures such as *sales* and *budget* aggregated by the same dimensions. Thus, another distinctive feature of the multidimensional data model is its stress on aggregation of numerical measures by one or more dimensions, which is one of the key operations mainly to speed up query processing time.

1.1.3 Implementation

Data warehouses might be implemented on standard or extended relational DBMSs, called **Relational OLAP** (ROLAP) servers. These servers support extensions to SQL and special access and implementation methods to efficiently implement the multidimensional data model and operations. They assume that data is stored in relational databases. On the other hand, **Multidimensional OLAP** (MOLAP) servers directly store multidimensional data in some special data structures (such as arrays) and implement the OLAP operations over these special data structures [CD97].

1.1.4 References

For readers who are not familiar with "OLAP" or multidimensional data analysis, perhaps the best place to start is the OLAP council Web site[1], which is a good source of information on standardization efforts across the industry, and a paper written by Codd, et al [CCS93], which defines twelve rules for OLAP products. Moreover, a good source of references on data warehousing and OLAP is the Data Warehousing Information Center[2].

1.2 Knowledge Discovery and Data Mining

Data mining, often also called **knowledge discovery in databases** (KDD), has been known as a process of nontrivial extraction of implicit, previously unknown and potentially useful information (includes interesting knowledge rules, constraints, regularities, etc. [FPSSU96]) from large databases. It has been recognized by many researchers as a key research topic in database systems and machine learning, which is in fact an integration of techniques from multiple disciplines such as database technology, statistics, machine learning, high performance computing, pattern recognition, neural networks, data visualization, information retrieval, image database, signal processing, and spatial data analysis.

Fayyad, et al [FPSS96a], however, claimed that "knowledge discovery" and "data mining" are different. "The term *knowledge discovery* in databases, or KDD for short, was coined in 1989 to refer to the broad process of finding knowledge in data, and to emphasize the 'high-level' application of particular *data mining* methods. The term *data mining* has been commonly used by statisticians, data analysts and the MIS (Management Information Systems) community, while KDD has been mostly used by artificial intelligence and machine learning researchers ... KDD refers to the overall *process* of discovering useful knowledge from data while *data mining* refers to the application of algorithms for extracting patterns from data without the additional steps of the KDD process."

[1] http://www.olapcouncil.org/
[2] http://pwp.starnetinc.com/larryg/articles.html

1.2.1 The KDD Process

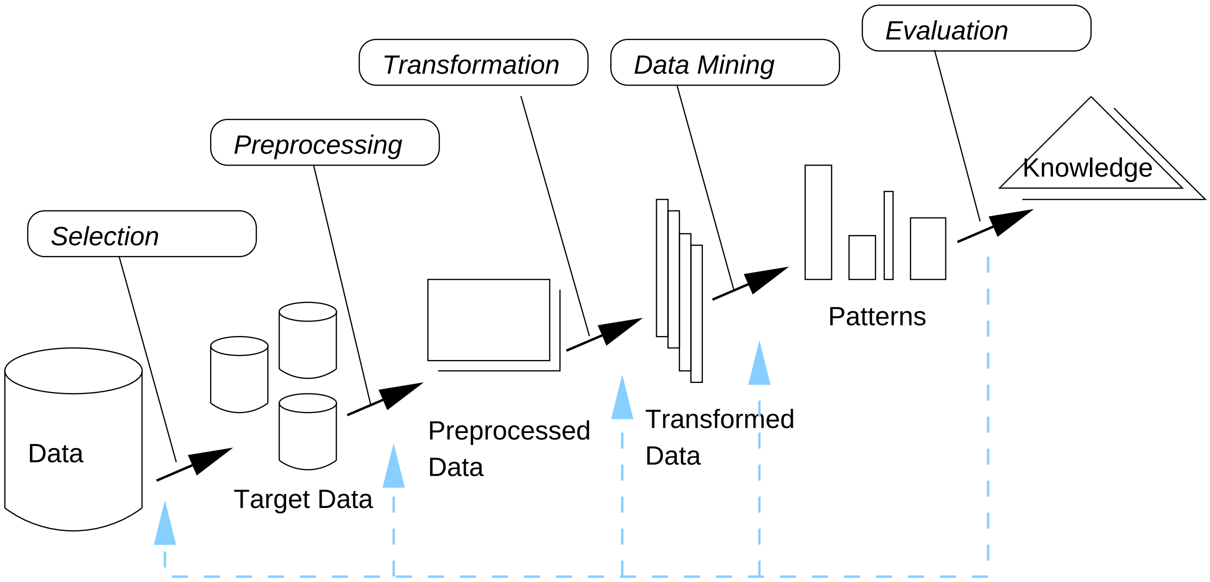


Figure 1.2: An overview of the steps comprising the KDD process [FPSS96a].

As shown in figure 1.2, the KDD process [FPSS96a] is interactive and iterative, which involves many user interaction and decisions. Some of the basic steps are outlined in the following.

The Process

1. Understanding the application domain, the relevant prior knowledge, and the goals of the end-user.
2. Creating a target data set for discovery by selecting a data set, or focusing on a subset of variables or data samples.
3. Preprocessing by cleaning the data using basic operations such as the removal of noise or outliers if appropriate, deciding on strategies for handling missing data fields, and so on.
4. Transforming the data to reduce the number of variables under consideration or to find invariant representations for the data.

5. Choosing the data mining task by deciding whether the goal of the KDD process is classification, regression, clustering, etc. and choosing the appropriate data mining algorithm(s).

6. Mining the data by searching for patterns of interest in a particular representational form or a set of such representations. Examples are classification trees, regression, clustering, and so on.

7. Interpreting or evaluating the mined patterns. It is possible to return to any of the steps above for further iteration.

8. Consolidating discovered knowledge by incorporating it into the performance system, or simply documenting it and reporting it to the interested parties.

Applications

More and more industrial companies realize that the KDD process can be used in information management, query processing, decision making, process control, and many other applications [CHY96]. In the business world, the most successful application of KDD is "Database Marketing", which is "a method of analyzing customer database, looking for patterns among existing customer preferences and using those patterns for more targeted selection of future customers" [FPSS96a]. Another major business use of data mining methods is the analysis and selection of stocks and other financial instruments [FPSS96a]. Besides business world, there are a number of interesting and important scientific applications of KDD which include astronomy, molecular biology, global climate change modeling, etc. [FPSS96a].

1.2.2 Data Mining Methods

As a matter of fact, the two "high-level" primary goals of data mining tend to be *prediction* and *description*. The former involves using some variables in the database to predict unknown or future values of other variables of interest. The latter, however, focuses on finding human-interpretable patterns describing the data [FPSS96a].

Yet, there exist a wide variety of data mining algorithms. For a brief review of the most popular of these see [FPSS96a]. For interested readers, some popular data mining methods are categorized as follows: mining association rules in transactional or relational databases [AIS93, AS94, HF95, PCY95, SON95, SA95], summarizing and generalizing data using datacube approach [GHQ95, HRU96, Wid95, YL95] and attribute-oriented induction approach [HCC93, HF96], mining classification rules [Qui93, CS96, IP96, PS91, Zia94], data clustering analysis [CS96, EKX95, NH94, ZRL96], etc.

An important point is that each technique typically suits some problems better than others. Thus, there is no 'universal' data mining method and choosing an appropriate algorithm for a particular application is something of an art [FPSS96b].

1.3 OLAP and Data Mining

In February of 1990, a group of database researchers met to examine the prospects for future database research efforts. In May of 1995, a second workshop was held to consider anew the prospects for database research, and their discussions were summarized by Silberschatz, et al [SSU96]. They described data mining as the extraction of information from large bodies of data often accumulated for other purposes. Also, the decision makers cannot really formulate a precise query and their real question is probably "Find me something interesting". Thus, data mining queries tend to be ad-hoc, and they often involve aggregations of huge amounts of data. The most demanding requirement is that there is a need for extremely fast response because any long delay might severely hinders productivity [SSU96, HRU96]. The figure of merit is the total elapsed time, including the writing, debugging, and execution of the query, especially in such applications as trading of commercial instruments [SSU96].

Moreover, from the KDD process described above [FPSS96a], we know that data mining tools need to work on the integrated, consistent, and cleaned data, which often require the costly preprocessing steps such as data cleaning and data integration. The top researchers concluded that optimization for complex queries, such as those involving aggregations and grouping, and the techniques for supporting "multidimensional" queries, where the data is organized into a "datacube", will further enhance the flexibility and ability of the data mining tools [SSU96].

Clearly, a data warehouse provides a valuable source for data mining. If the data mining tools can work with the OLAP engine of a data warehouse, the tools will provide more navigation power in the cleaned and aggregated data such that data mining can be an interesting exploratory experience to the user [HCC98]. We call the combination of OLAP and data mining **OLAP mining** or **On-Line Analytical Mining**(OLAM).

1.3.1 Motivation of this Research

OLAP mining is a promising direction, and we have developed an OLAP mining system, DBMiner [HFW+96, HCC+97]. Efficient computation, construction, and retrieval of datacubes for DBMiner are very important for OLAP and especially data mining because data mining tools often cost a lot more than simple OLAP operations. Efficient implementation and fast response of the datacube are thus the major challenges in the realization of OLAP mining in large databases or data warehouses.

1.3.2 Major Results and Progress

Our major results and progress include:

1. the design of a chunk-based compressed multidimensional datacube and its aggregation computation,

2. the design of an OLAP system by integration of datacube computation and relational database technology,

3. the design of an OLAP mining method which integrates OLAP with data mining, and

4. the implementation of integrating datacube and OLAP mining mechanisms successfully with high performance.

The core engine of DBMiner is the datacube, which is constructed based on the chunking ideas described in [ZDN97]. It has been tested by building cubes from a table of more than 300,000 tuples. We also have constructed a 20 dimensional cube with a success. DBMiner works very well with small to medium sized cubes (less than 10 dimensions), but suffers a little with bigger cubes. The core engine can still be improved such that it can handle cubes of more dimensions efficiently.

Recently, the educational version DBMiner E1.0 has been released and more than 700 users have downloaded the demo version. We also have an active customer using DBMiner E1.0 to do the OLAP-based association rule mining.

1.3.3 Outline of the Thesis

We have introduced the concepts of data warehousing and data mining. These two booming fields explain why we need efficient computation of datacubes, which motivated this research. In the next chapter, we will discuss in detail different underlying implementation approaches of data warehouses. Then we will discuss some previous work on this topic. In chapter 3, we will thoroughly analyze the design, architecture, technical issues, and innovations of the datacube created for DBMiner. In chapter 4, we will further discuss how to implement the OLAP operations with our datacube. We will also use an example of OLAP mining (datacube-based association rules) to illustrate the idea of data mining using the OLAP operations. In the following chapter, we will analyze the performance results of comparing our approach with the simulation of a previously published method, and comparing DBMiner with an existing OLAP product. We also will discuss the strengths and limitations of our approach. Finally, in the last chapter, we will briefly summarize the technical contributions of this thesis and discuss some future work.

Chapter 2

Related Works

We will first discuss different approaches of data warehouse implementations. Then we will look at some previous work on how to improve the performance of aggregation computation for datacubes.

2.1 Data Warehouse Model

There are many reasons to separately maintain a data warehouse from the organization's operational databases [CD97]. The functional and performance requirements of On-Line Analytical Processing (OLAP) supported by data warehouses are quite different from those of the **On-Line Transaction Processing** (OLTP) applications [CD97, Sar97] traditionally supported by the operational databases.

2.1.1 OLAP vs. OLTP

OLTP applications typically automate clerical data processing tasks of an organization such as order entry and banking transactions that are the "bread-and-butter day-to-day operations of an organization" [CD97], which read or update a few records accessed on their primary keys. These tasks are structured and repetitive, and consist of short, atomic, isolated transactions, that require detailed and up-to-date data. The operational databases tend to be hundreds of megabytes to gigabytes in size and store only current data [CD97]. Consistency and recoverability of the database are critical, and maximizing transaction throughput is the key performance metric. Therefore, the database is designed to minimize concurrency conflicts [CD97].

Data warehouses, however, are targeted for decision support for managers. Detailed and individual records are less important than historical, summarized, and consolidated data. Thus, data warehouses usually contain consolidated data from one or more operational databases collected over long periods of time. As a result, data warehouses tend to be orders of magnitude larger, say hundreds of gigabytes to terabytes in size, than operational databases. The workloads are query intensive with mostly ad-hoc, complex queries that can access millions of records and perform a lot of scans, joins, and aggregates. Query throughput and response times are more important than transaction throughput [CD97]. Given that operational databases are finely tuned to support known OLTP workloads, trying to execute complex OLAP queries against the operational databases would result in unacceptable performance.

2.1.2 Data Warehouse Design

Entity Relationship diagrams and normalization techniques are popularly used for database design in OLTP environments. Thus, the database designs recommended by ER diagrams are inappropriate for decision support systems because they mainly need efficiency in querying and in loading data (including incremental loads) [CD97].

Most data warehouses use a **star schema** to represent the multidimensional data model [Kim96, CD97]. Figure 2.1 shows a typical star schema representation. The database consists of one large dominant table in the center, called the **fact table**, which is the only table with multiple joins connecting it to other tables. The other tables, called the **dimension tables**, all have only a single join attaching them to the fact table. Each tuple in the fact table consists of a pointer (i.e. foreign key which may be generated for efficiency) to each of the dimension tables that provide its multidimensional coordinates, and stores the numeric measures for those coordinates. Each dimension table consists of columns that correspond to attributes of the dimension.

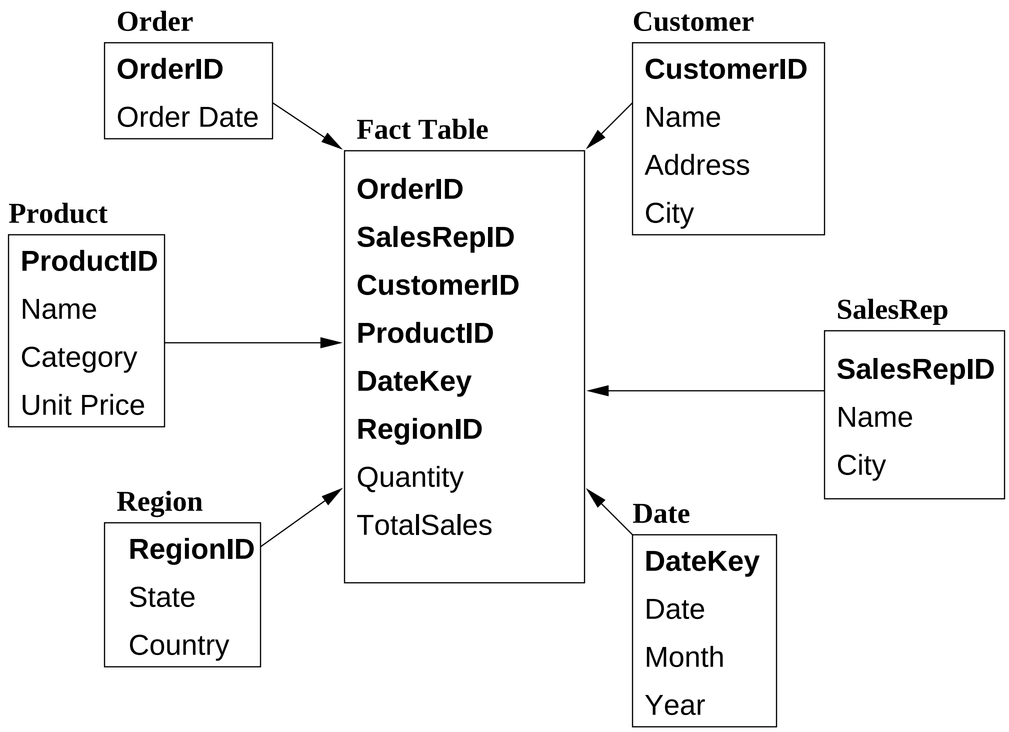


Figure 2.1: A Star Schema [CD97].

Star schemas do not explicitly provide support for attribute hierarchies. **Snowflake schemas** (shown in figure 2.2) provide a refinement of star schemas where some dimensional hierarchies are explicitly represented by normalizing the dimension tables. This leads to advantages in maintaining the dimension tables. However, the denormalized structure of the dimensional tables in star schemas may be more appropriate for browsing the dimensions [CD97].

In addition to the fact and dimension tables, data warehouses store selected summary tables containing pre-aggregated data. In the simplest cases, the pre-aggregated data corresponds to aggregating the fact table on one or more selected dimensions.

2.1.3 Data Warehouse Architecture

Figure 2.3 [CD97] shows that the data warehouses adopt a three-tier architecture. The bottom tier is a **warehouse database server** which is almost always a relational

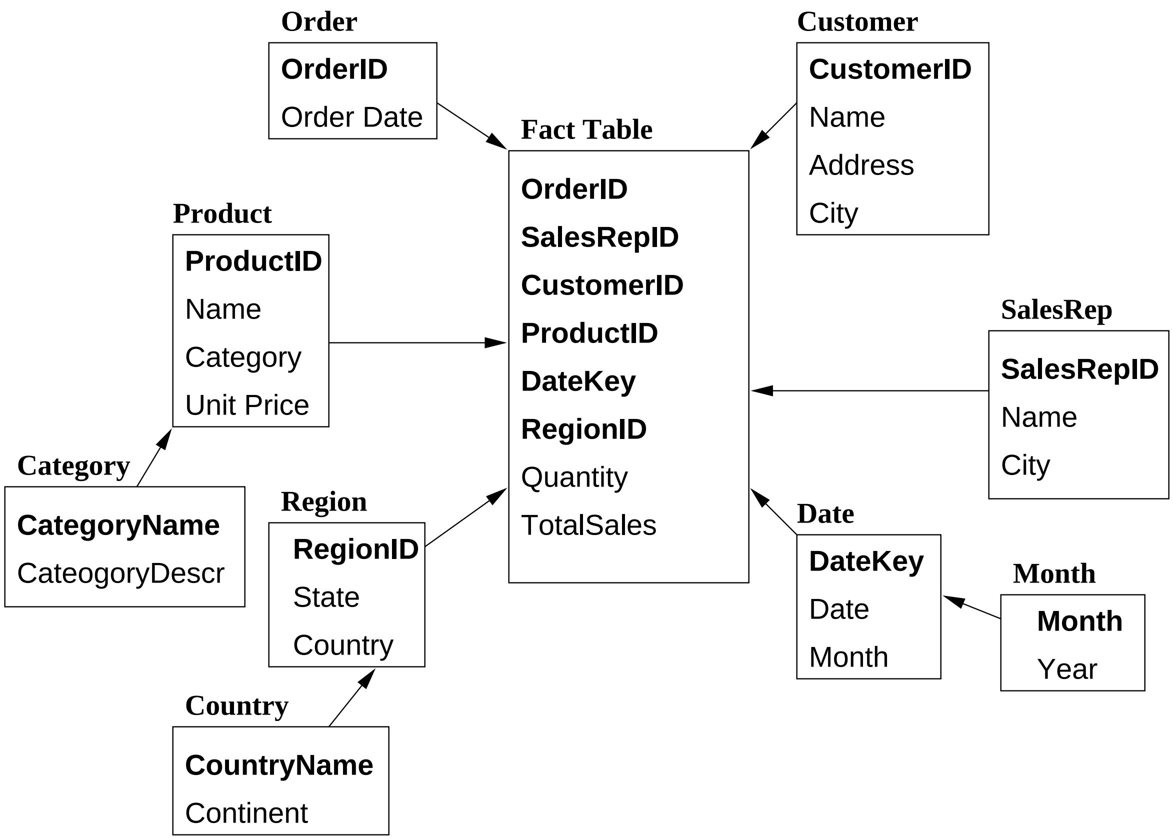


Figure 2.2: A Snowflake Schema [CD97].

database system. The middle tier is an **OLAP server** which can be implemented by either a Relational OLAP (ROLAP) model (store data in tables) or a Multidimensional OLAP (MOLAP) model (store data in arrays). Finally, the top tier is a **client**, which contains query and reporting tools, analysis tools, and/or data mining tools for trend analysis, prediction, and so on.

The popular conceptual model that influences the front-end tools, database design, and the query engines for OLAP is the *multidimensional* view of data in the warehouse and its stress on *aggregation* of measures by one or more dimensions as one of the key operations.

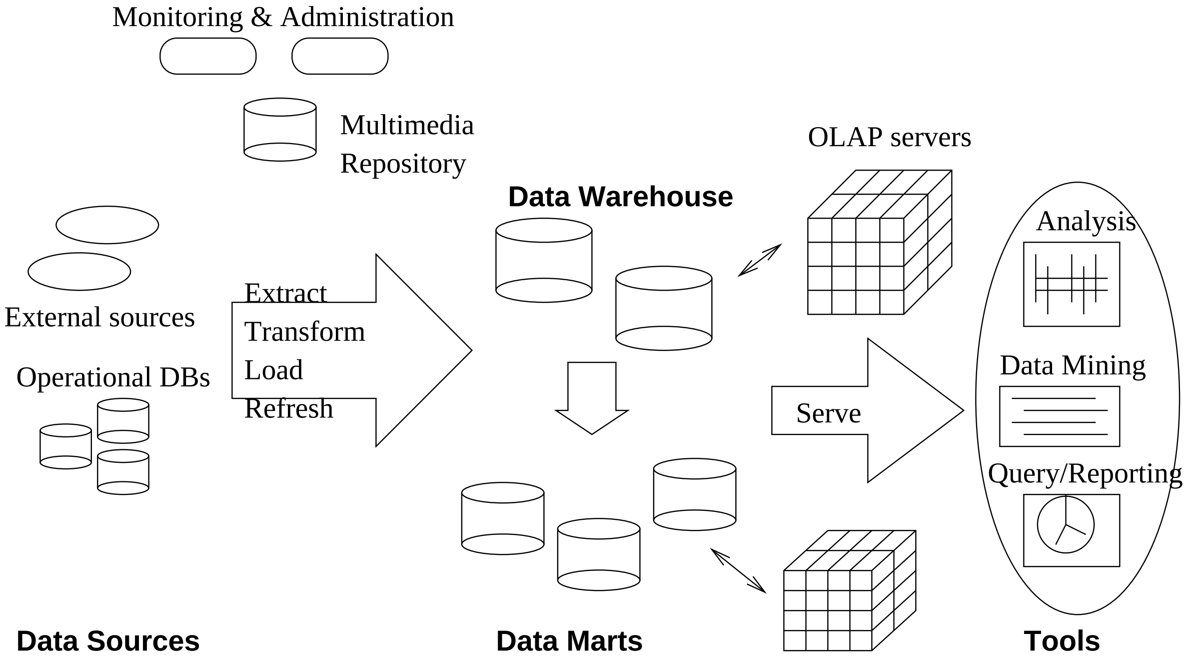


Figure 2.3: Data Warehousing Architecture [CD97]

2.1.4 OLAP server: ROLAP vs. MOLAP

As stated above, there are two main options for the implementation of the OLAP server [CD97]. For example, the DSS server of MicroStrategy[1] [Incb] and MetaCube of Informix [2] [Inca] adopted the Relational OLAP (ROLAP) servers assuming data is stored in the tables of relational databases or extended-relational databases [Sar97, CD97]. They use the relational database system to manage the warehouse data and aggregations using star schemas [Sar97]. They also support extensions to SQL and special access and implementation methods as the OLAP middleware to efficiently implement the multidimensional data model and operations [CD97]. Thus, the relational tables are their data structure and a "cell" in the conceptual multidimensional space is represented as a tuple. This tuple carries some attributes that identify the location of the cell in the multidimensional space and other attributes that contain the data value corresponding to that data cell [ZDN97]. Since they use the relational database to

[1] http://www.strategy.com
[2] http://www.informix.com

implement the multidimensional data model, they need to rephrase given user queries in terms of the appropriate materialized views and generate multi-statement SQL for the database server [CD97]. ROLAP servers claim that they are well-suited to larger OLAP data sets because sparse data sets may be stored more compactly in tables than in arrays [DNR+97]. Moreover, ROLAP servers can exploit the scalability and the transactional features of the relational systems [DNR+97, CD97]. However, the intrinsic mismatches between the OLAP-style querying and SQL (e.g., lack of sequential processing, column aggregation) can cause performance bottlenecks for ROLAP servers [CD97].

Another popular option is the Multidimensional OLAP (MOLAP) servers, which is adopted by Essbase of Arbor[3] [Cor]. MOLAP servers directly implements the multidimensional views by storing data in some special data structures (e.g. sparse arrays) and the OLAP operations over these special data structures [ZDN97, CD97]. The multidimensional datacube is implemented by the arrays with the dimensions forming the axes of the cube [Sar97]. Thus, only the data value corresponding to a data cell is stored. The position within the sparse array would encode the exact location of this cell in the multidimensional space [ZDN97]. MOLAP servers have excellent indexing properties due to the fact that looking for a cell is simple array lookups rather than associative lookups in tables. Unfortunately, when data is sparse, MOLAP servers may suffer from poor storage utilization [CD97] and some sparse matrix compression techniques should be explored. Yet they provide small and medium OLAP data sets with more efficient storage and retrieval.

Typically 20% of the data in the logical OLAP cube are non-zero [Col96]. Thus, some OLAP servers adopt a two-level storage representation to handle sparse and dense data sets. A set of one or two dimensional subcubes that are likely to be dense are identified and are represented in the array format. Then the traditional indexing structure is used to index onto these "smaller" arrays [CD97]. The rest of the subcubes that are sparse employ the compression technology for efficient storage utilization. This results in a hybrid OLAP storage method in which data is maintained in a relational database while aggregations are held in a multidimensional form. This

[3] http://www.arborsoft.com

is called **Hybrid OLAP (HOLAP)** [PSF91].

There is yet another option, the **Specialized SQL servers**, which is adopted by Redbrick[4]. They implemented these servers to meet the growing demand of OLAP processing in relational databases. The objective is to provide advanced query language and query processing support for SQL queries over star and snowflake schemas in read-only environments [CD97].

2.2 Multiple related Groupbys and Aggregates

Users of decision support queries are often interested in identifying trends rather than looking at individual records in isolation. DSS queries thus make heavy use of aggregations, and the ability to simultaneously aggregate across many sets of dimensions (in SQL terms, this translates to many simultaneous groupbys) is very crucial for OLAP or multidimensional data analysis applications.

Gray, et al [GBLP96] introduced the `CUBE` operator as the extension to the SQL's `SELECT-GROUP-BY-HAVING` syntax. It is equivalent to the union of a number of standard groupby operations, which computes groupby aggregations over all possible subsets of the specified dimensions. Its rapid acceptance has made the SQL community extend the SQL syntax to include this `CUBE` operator.

Yet, the size of the data warehouse and the complexity of DSS queries may cause queries to take very long to complete, which is unacceptable in most DSS environments. To speed up multidimensional data analysis, database systems frequently precompute aggregates on some subsets of dimensions and their corresponding hierarchies that help accelerate many common queries. Another commonly used technique is to materialize frequently-asked queries. For example, the data warehouse at the Mervyn's department store chain has a total of 2400 precomputed tables to improve query performance [HRU96].

Since enterprise data warehouses can be in hundreds of gigabytes to terabytes in size, to materialize all possible combinations of groupbys is definitely infeasible. The decision of what and how much to precompute is difficult. It is obvious that the more

[4]http://www.redbrick.com

that is precomputed, the faster the queries can be answered. The selection of summary data (views) to materialize must consider the workload characteristics, the costs for incremental update, and the upper bounds on the storage requirements [CD97]. Harinarayanan, et al [HRU96] considered this problem of which aggregates are best to precompute and store if only a certain number of them can be stored. Under simplifying assumptions, a greedy algorithm was shown to have good performance [HRU96].

2.2.1 The Data CUBE Operator

The `CUBE` operator facilitates the construction of *histograms* and *cross tabulation* (or *cross tab* for short). It is the N-dimensional generalization of simple aggregate functions as shown in figure 2.4. The 0D datacube is a point. The 1D datacube is a line with a point. The 2D datacube is a cross tab, a plane, two lines, and a point. The 3D datacube is a cube with three intersecting 2D cross tabs.

In general, if there are N attributes, there will be $2^N - 1$ super-aggregate values. If the cardinalities of the N attributes are $C_1, C_2, \ldots, C_N$, then the cardinality of the resulting cube relation is $\prod(C_i + 1)$. The extra value in each domain is `ALL`. Each `ALL` value really represents a set over which the aggregate was computed. For example, if a table only records sales from 1990 to 1992, then `Year.ALL = ALL(Year) =` {`1990, 1991, 1992`}. One can think of the `ALL` value as a token representing these kind of sets. However, the `ALL` value creates substantial complexity in SQL, so the `NULL` value is used instead in the real implementation. However, the `ALL` value is stilled used as the aggregate token.

Gray, et al [GBLP96] also introduced the syntax to address the cube cells. For example, to select a cell from a 2D `cube` with fields `row = i` and `column = j` would be

```
cube.v(:i, :j).
```

Aggregate functions can be classified into three categories considering aggregating a two dimensional set of values $\{X_{i,j} | i = 1, \ldots, I; j = 1, \ldots, J\}$:

1. **Distributive.** Aggregate function $\mathcal{F}()$ is distributive if there is a function $\mathcal{G}()$ such that $\mathcal{F}(\{X_{i,j}\}) = \mathcal{G}(\{\mathcal{F}(\{X_{i,j} | i = 1, \ldots, I\}) | j = 1, \ldots, J\})$. e.g. `COUNT()`,

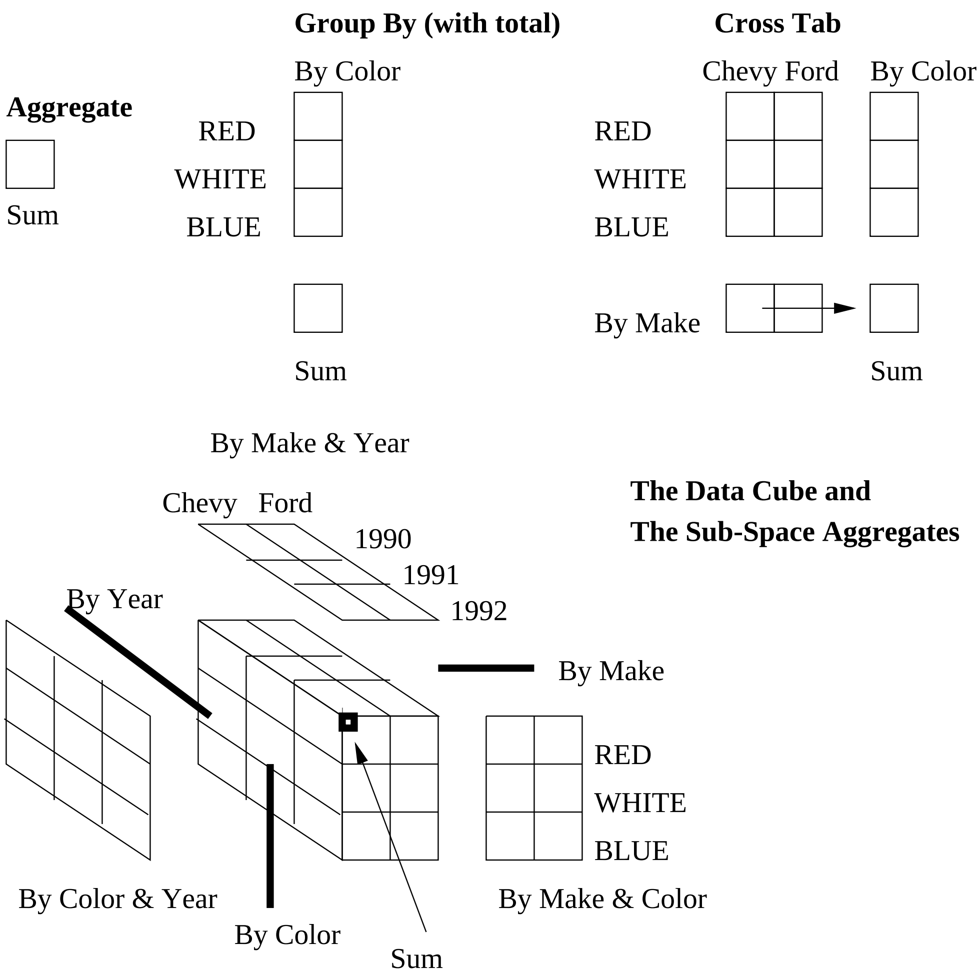


Figure 2.4: The CUBE operator for 0D, 1D, 2D, and 3D datacubes [GBLP96].

`MIN()`, `MAX()`, and `SUM()`. In fact, $\mathcal{F} = \mathcal{G}$ for all except `COUNT()`. $\mathcal{G} =$ `SUM()` for the `COUNT()` function. Once order is imposed, the cumulative aggregate functions also fit in the distributive class. That is, the input set can be partitioned into disjoint sets that can be aggregated separately and later combined [AAD+96].

2. **Algebraic.** Aggregate function $\mathcal{F}()$ is algebraic if there is an M-tuple valued function $\mathcal{G}()$ and a function $\mathcal{H}()$ such that $\mathcal{F}(\{X_{i,j}\}) = \mathcal{H}(\{\mathcal{G}(\{X_{i,j} | i = 1, \ldots, I\}) | j = 1, \ldots, J\})$. e.g. `Average()`, standard deviation, `MaxN()`, `MinN()`,

etc. For `Average()`, the function $\mathcal{G}()$ records the sum and count of the subset. The $\mathcal{H}()$ function adds these two components and then divides to produce the global average. The key to algebraic functions is that a fixed size result (an M-tuple) can summarize the sub-aggregation. That is, algebraic aggregate functions can be expressed in terms of other distributive functions [AAD+96].

3. **Holistic.** Aggregate function $\mathcal{F}()$ is holistic if there is no constant bound on the size of the storage needed to describe a sub-aggregate. That is, there is no constant M, such that an M-tuple characterizes the computation $\mathcal{F}(\{X_{i,j} | i = 1, \ldots, I\})$. e.g. Median(), Mode(), Rank(), etc. Therefore, holistic aggregate functions cannot be computed in parts and combined [AAD+96].

Gray, et al [GBLP96] claimed that there is no more efficient way of computing super-aggregates of holistic functions than the 2^N-algorithm using the standard `GROUP BY` techniques. On the other hand, cubes of distributive functions are relatively easy to compute. Given that the core is represented as an N-dimensional array in memory and each dimension has size $C_i + 1$, the $N - 1$ dimensional slabs can be computed by projecting (aggregating) one dimension of the core. For example, the following computation aggregates the first dimension. $\texttt{CUBE}(\texttt{ALL}, x_2, \ldots, x_N) = \mathcal{F}(\{\texttt{CUBE}(i, x_2, \ldots, x_N) | i = 1, \ldots, C_1\})$. N such computations compute the $(N - 1)$ dimensional super-aggregates. The distributive nature of the function $\mathcal{F}()$ allows aggregates to be combined. The next step is to compute the next lower dimension — an (…, `ALL`, …, `ALL`, …) case. Thinking in terms of the cross tab in figure 2.4, one has the choice of computing the result by aggregating the lower row or aggregating the right column (aggregate (`ALL`, *) or (*, `ALL`)). Either approach will give the same answer. The algorithm will be most efficient if it aggregates the smaller of the two (pick the * with the smallest C_i.) In this way, the super-aggregates can be computed dropping one dimension at a time. Algebraic aggregates are more difficult to compute than distributive aggregates. The super-aggregate needs the intermediate results rather than just the raw sub-aggregate.

If the datacube does not fit into memory, array techniques do not work. Rather one may either partition the cube with a hash function or sort it. The super-aggregates are likely to be orders of magnitude smaller than the core, so they are very likely to fit in memory.

It is possible that the core of the cube is sparse. In that case, only the non-null elements of the core and of the super-aggregates should be represented. This suggests a hashing or a B-tree be used as the indexing scheme for aggregation values.

2.2.2 Selection of Groupbys for Precomputation

Choosing the right set of frequently-asked queries to materialize is a nontrivial task. For example, a relatively infrequently asked query may be materialized if it helps to answer many other queries quickly. Harinarayanan, et al [HRU96] presented a framework and algorithms that help to pick a good set of queries to materialize.

Harinarayanan, et al [HRU96] assumed that the datacube is stored in *summary* tables in a relational system (ROLAP model). The cells of the datacube are organized into different sets based on the positions of `ALL` in their addresses, and each set is assigned to a different table. For example, all cells whose addresses match the address `(-,ALL,-)` are placed in the same set. Here, '-' is a placeholder that matches any value but `ALL`. The raw-data relation is referred as `R`.

In a datacube, the values of many cells are computable from those of other cells in the same datacube. Any cell that has the `ALL` value as one of the components of its address is one of those cells. These cells are called *dependent* cells or *cuboid* cells. That is, any cell that has no `ALL` value in its components are raw data cells, and its value has to be fetched from the data table `R`. In fact, the number of cells with `ALL` as one of their components is usually a large fraction of the total number of cells in the datacube.

They gave an example on the TPC-D database that has three attributes: **part**, **supplier**, and **customer**. Thus there are eight (2^3) possible groupings of all attributes. They are

1. **part**, **supplier**, **customer**

2. **part**, **customer**

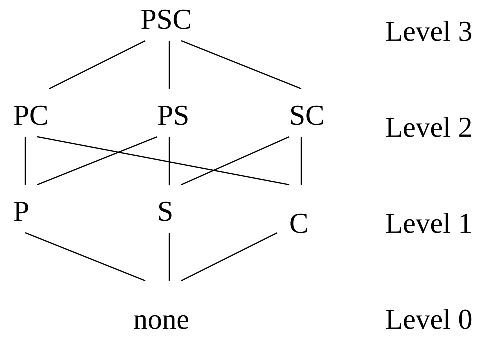


Figure 2.5: The eight TPC-D views where P for part, S for supplier, and C for customer [HRU96].

3. **part**, **supplier**
4. **supplier**, **customer**
5. **part**
6. **supplier**
7. **customer** and
8. **none**.

The last one **none** indicates that there are no attributes specified. Therefore, including **none**, a datacube with N dimensions will have 2^N possible groupbys (thus $2^N - 1$ super-aggregates). Figure 2.5 shows the eight views organized as a lattice. If the user asks for the sales grouped by **part**, and if the view that groups only by **part** (view 5) is materialized, just scan this view and output the answer. Moreover, this query can be answered by using the view that groups by **part** and **customer** (view 2). In this case, since the total sales for each customer exists, for each part sum the sales across all customers to get the result.

Consider two queries Q_1 and Q_2. $Q_1 \preceq Q_2$ if Q_1 can be answered using only the results of Q_2. That is, Q_1 is *dependent* on Q_2. The $\preceq$ operator imposes a partial ordering on the queries. Harinarayanan, et al [HRU96] talked about the views of a datacube problem as forming a lattice. They assume that $\preceq$ is a partial order, and that there is a *top* element, a view upon which every view is dependent.

They also introduced the *Linear Cost Model.* Let $< L, \preceq >$ be a lattice of queries (views). To answer a query Q we choose an ancestor of Q, say Q_A, which has been materialized. The cost of answering Q is the number of rows present in the table for that query Q_A used to construct Q.

Their important objective is to develop techniques for optimizing the space-time tradeoff when implementing a lattice of views, which is an NP-complete problem (there is a straightforward reduction from Set-Cover). The obvious choice of heuristic is a *greedy* algorithm, where they select a sequence of views, each of which is the best choice given what has gone before. They showed that no matter what lattice is given, the greedy algorithm never performs too badly. The *benefit* of the greedy algorithm is at least 63% of the benefit of the optimal algorithm.

2.3 Computation of Multidimensional Aggregates

Computing the multidimensional aggregates can be a performance bottleneck for OLAP applications. Gray, et al [GBLP96] gave some rules of thumb for computing them in an efficient implementation of the `CUBE` operator. Yet, their primary focus was to define the semantics of the `CUBE` operator. Therefore, several efficient algorithms for Relational OLAP (ROLAP) have been developed to compute the `CUBE`. A team at IBM Almaden independently explored some approaches to this problem [AAD+96] using sorting and hashing. Several other researchers proposed some other algorithms to this same problem for MOLAP [ZDN97] using chunking and for ROLAP [RS97] using partitioning.

2.3.1 Sorting or Hashing

Agarwal, et al [AAD+96] was the first published work on methods for optimizing the computation of a collection of groupbys. Based on the `CUBE` operator, Agarwal, et al [AAD+96] assumed that the aggregating functions are *distributive* or *algebraic* because both can be computed in parts and combined.

Computing the cube stored in some relational tables (ROLAP) requires a generalization of standard relational aggregation operators [AAD+96]. Two basic methods have been studied for computing *single* groupbys and they are: (1) the sort-based method `PipeSort` and (2) the hash-based method `PipeHash` [Gra93]. Agarwal, et al [AAD+96] adapted these methods and incorporated them with the following optimizations:

1. **Smallest-parent** — computing a groupby from the smallest previously computed groupby,
2. **Cache-results** — caching in memory the results of a groupby from which other groupbys are computed to reduce disk I/O,
3. **Amortize-scans** — amortizing disk reads by computing as many groupbys as possible together in memory,
4. **Share-sorts** (for sort-based algorithms only) — sharing sorting cost across multiple groupbys, and
5. **Share-partitions** (for hash-based algorithms only) — partitioning the hash table if it is too large to fit in memory, and the aggregation is done for each partition that fits in memory.

Unfortunately, the above optimizations are often contradictory for OLAP databases. For example, a groupby can be computed from one of the several parent groupbys, but the bigger one is in memory and the smallest one is not. In this case, the second optimization **Cache-results** may win.

Both `PipeSort` and `PipeHash` compute the groupbys in a pipelined fashion such that each pipeline is a list of groupbys all of which are computed in a single scan of the sort input stream. In order to decide what groupby is computed from what and the attribute order in which it is computed, they need some global planning, which uses the **search lattice** introduced in [HRU96].

The Search Lattice

The search lattice is a graph where a vertex represents a groupby of the cube. A directed edge connects groupby i to groupby j whenever j can be generated from i and j has exactly one attribute less than i (i is called the parent of j). Thus, the out-degree of any node with k attributes is k. Level k of the search lattice denotes all groupbys that contain exactly k attributes. They use the keyword `ALL` to denote the empty groupby instead of `none` in [HRU96]. Each edge in the search lattice e_{ij} is labeled with two costs. The first cost $S(e_{ij})$ is the cost of computing j from i when i is not already sorted. The second cost $A(e_{ij})$ is the cost of computing j from i when i is already sorted.

PipeSort

`PipeSort` calls `Generate-Plan` whose objective is to find an output $\mathcal{O}$ that has the minimum sum of edge costs. This algorithm proceeds level-by-level, from level $k = 0$ to level $k = N - 1$, where N is the total number of attributes. For each level k, it finds the best way of computing level k from level $k + 1$ by reducing the problem to a weighted bipartite matching problem. For details, please refer to [AAD+96]. The complexity of this algorithm is $\mathcal{O}(((k + 1)M_{k+1}))$, where M_{k+1} is the number of groupbys in level $k + 1$.

The output of `Generate-Plan` is a subgraph of the search lattice where each groupby is connected to a single parent groupby from which it will be computed and is associated with an attribute order in which it will be sorted.

The algorithm `PipeSort` is shown in table 2.1. In general, after the plan is generated, the data will be sorted according to the plan and a collection of groupbys linked by the A() edges will be computed in a pipelined fashion. Some groupbys in level k might need to be re-used or even re-sorted to compute more than one groupby in level $k + 1$. The sorting cost might be reduced by taking advantage of the partial sorting order [AAD+96].

The main limitation of `PipeSort` is that it does not scale well with respect to the number of attributes in the datacube query [RS97]. When the number of attributes

```
PipeSort
(Input: search lattice with the A() and S() edges costs)
For level k = 0 to N − 1
/* find how to generate level k from level k + 1 */
Generate-Plan(k + 1 → k);
For each groupby g in level k + 1
    Fix the sort order of g as the order of the
        groupby connected to g by an A() edge;
```

Table 2.1: Algorithm `PipeSort` [AAD+96]

in the datacube is k, a lower bound on the number of sorts performed by `PipeSort` is given by

$$\binom{k}{\lceil k/2 \rceil},$$

which is exponential in k. When the underlying relation is sparse and much larger than the memory available, then many of the groupbys that `PipeSort` sorts are also larger than the available memory. Consequently, sorting these cuboids requires external sorts, which results in considerable amount of I/O.

PipeHash

The data to be aggregated is usually too large for the hash-tables to fit in memory. `PipeHash` algorithm (shown in table 2.2) partitions the data on one or more attributes. When data is partitioned on some attribute, say B, then all groupbys that contain B can be computed by independently grouping on each partition. That is, the results across multiple partitions need not be combined. The cost of data partitioning can be shared across all groupbys that contain the partitioning attribute.

The input to the algorithm is the search lattice described above. It first chooses the parent groupby with the smallest estimated total size for each groupby. The outcome is a minimum spanning tree (MST) where each vertex is a groupby and an edge from groupby a to b shows that a is the smallest parent of b. Figure 2.6 shows

```
PipeHash
(Input: search lattice with groupby estimated sizes)
Initialize worklist with MST of the search lattice;
While worklist is not empty
    Pick any tree T from the worklist;
    T' = Select-subtree of T to be executed next;
    Compute-subtree T';
```

Table 2.2: Algorithm `PipeHash` [AAD+96]

the MST for a four attribute search lattice and the size of each groupby is indicated below the groupby.

In most cases the available memory will not be sufficient to compute all the groupbys in the MST together, so the next step is to decide what groupbys to compute together, when to allocate and deallocate memory, and what attribute to choose for partitioning data. This problem is NP-complete because solving this problem is similar to the well-known NP-complete partitioning problems [AAD+96]. The heuristic solution is to choose the partitioning attribute that allows the choice of the largest subtree.

`PipeHash` computes all groupbys from the same partition that contain the partitioning attribute. When computing a subtree, all hash-tables of groupbys in the subtree (except the root) are maintained in the memory until all its children are created. Moreover, each groupby is scanned once and all its children are computed.

2.3.2 Array Chunking

MOLAP systems have a different sort of challenge in computing the cube than ROLAP systems do. The fundamental difference is the different data structures used to store the data. For MOLAP systems where the cube is stored in sparse arrays, we can still use the trick of computing an aggregate from a subaggregate employed by most ROLAP systems. Unfortunately, none of the other techniques developed for ROLAP cube computations can apply. The main reason is that there is no equivalent of

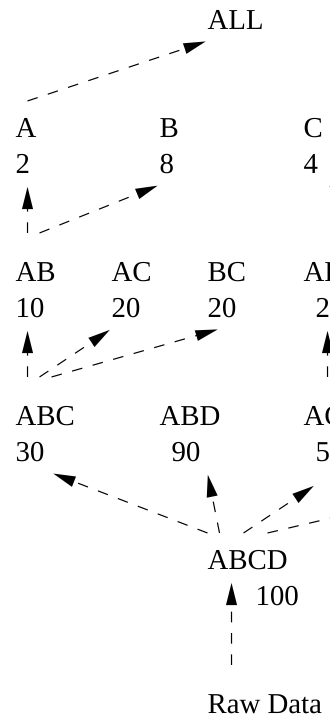


Figure 2.6: A four attribute Minimum Spanning Tree [AAD+96].

"reordering to bring together related tuples" [ZDN97] based upon their dimension values. The trick MOLAP can use is to visit those dimension values in the right order so that the computation is efficient. Similarly, we cannot use an order generated by one subaggregate in the computation of another. The trick is to simultaneously compute "spatially-delimited partial aggregates so that a cell does not have to be revisited for each subaggregate" [ZDN97].

Zhao, et al [ZDN97] introduced the `Multi-Way Array Cubing` algorithm in which the cells are visited in the right order so that a cell does not have to be revisited for each sub-aggregate. The goal is to overlap the computation of all these groupbys and finish the cube in one scan of the array with the requirement of memory minimized.

Zhao, et al [ZDN97] indicated two main issues regarding the Multidimensional OLAP (MOLAP) systems. First, it is very likely in a multidimensional application that the array itself is far too large to fit in memory. In this case, the array must be split up into *chunks* each of which is small enough to fit comfortably in memory. This ensures a uniform treatment for all the dimensions, as suggested by Sarawagi [SS94]. Secondly, it is likely that many of the cells in the array are empty, meaning that there is no data for that combination of coordinates. Therefore, there will be chunks that are sparse (less dense than some preset threshold).

Chunking is a way to divide an N-dimensional array into small size N-dimensional chunks and store each chunk as one object on disk. Moreover, each of the 2^N groupbys are also represented as separate arrays and will be chunked. Each array chunk has N dimensions and will correspond to the blocking size on the disk. Zhao suggested using chunks which have the same size on each dimension. However, those chunks that are sparse can be compressed such that each cell in a compressed chunk is associated with an integer indicating its offset in this chunk and empty cells will not be stored. Therefore, it is likely that chunks will be of variable length and some meta data is needed to store such information.

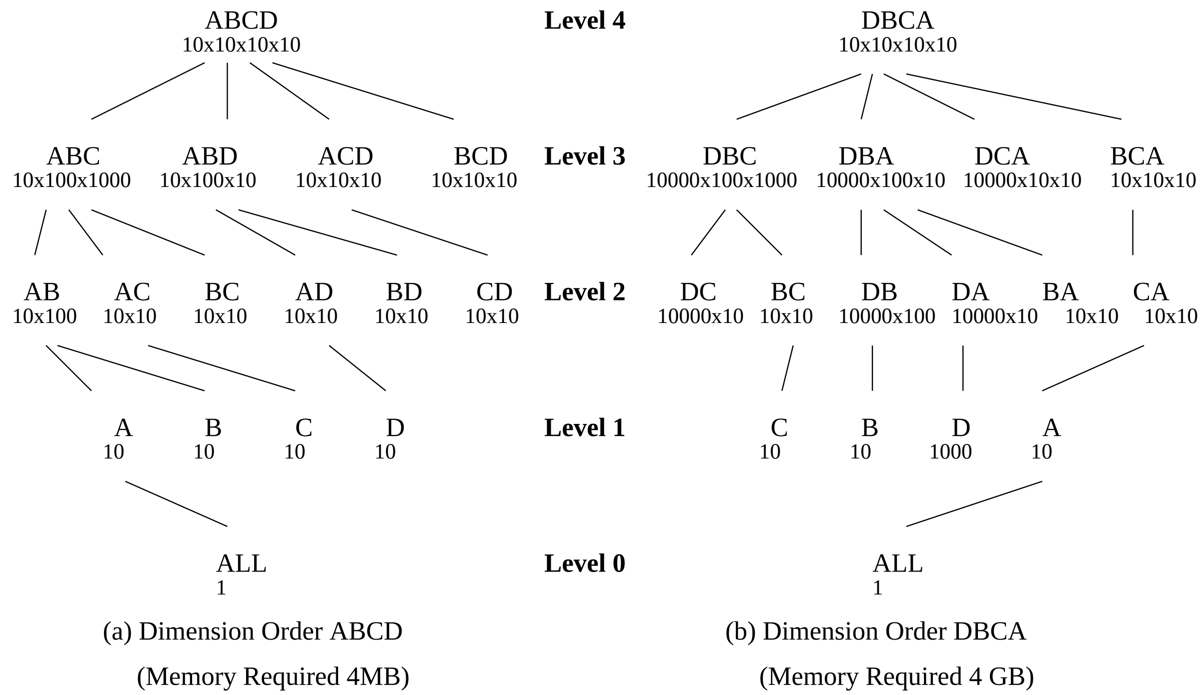


Figure 2.7: (a) MMST for dimension order ABCD (b) MMST for dimension order DBCA [ZDN97].

Chunking ensures efficiency in loading and storing the cube cells, but in order to minimize the memory requirement when computing the groupbys, Zhao, et al [ZDN97] introduced the concepts of **Optimal Dimension Order** and **Minimum Memory Spanning Tree**. A dimension order of the array chunks is a row major order of the chunks with the N dimensions $D_1, D_2, \ldots, D_N$ in some order $\mathcal{O}= (D_{j1}, D_{j2}, \ldots, D_{jN})$. A Minimum Memory Spanning Tree (MMST) for a cube $(D_1, \ldots, D_N)$ in a dimension order $\mathcal{O}= (D_{j1}, \ldots, D_{jN})$ has $N+1$ levels with the root $(D_{j1}, \ldots, D_{jN})$ at level N. Here the MMST is quite similar to the MST as an input to the algorithm `PipeHash` in [AAD+96]. An MMST, for a given dimension order, is minimum in terms of the total memory requirement for that dimension order. Thus, for different dimension orders of the array $(D_1, \ldots, D_N)$, different MMSTs will be generated and the memory requirements can be very different. Take a four dimensional array ABCD as an example. It has $10 \times 10 \times 10 \times 10$ chunks. The sizes of dimensions A, B, C, and D are 10,100,1000,and 10000 respectively. The MMSTs for the dimension order (A,B,C,D) and for the dimension order (D,B,C,A) are shown in figure 2.7 [ZDN97]. The number below each groupby node in the figure is the number of units of array element required by the node. For more details how these numbers are resulted and how the MMSTs are formed, please refer to [ZDN97].

The optimal dimension order is the one whose MMST requires the least amount of memory. Zhao, et al [ZDN97] proved that the optimal dimension order $\mathcal{O}$ is (D_1, D_2,. . ., D_N), where $|D_1| \leq |D_2| \leq \ldots \leq |D_N|$ and $|D_i|$ denotes the size of the dimension D_i.

Zhao, et al [ZDN97] stressed the fact that the related groupbys can be computed when the raw data is being scanned. Therefore, they made the assumption that they would have enough memory to allocate the required memory for the related groupby nodes in the MMST. Take a 3-D cube $16 \times 16 \times 16$ (ABC) as an example. Each dimension consists of four segments. Thus, there are $4 \times 4 \times 4$ chunks. Once its MMST in a dimension order $\mathcal{O}$ is built, the computation of the MMST subtrees can be overlapped as follows. The MMST for array ABC in the dimension order (A, B, C) is shown in Figure 2.8. Chunks of BC, AC, and AB are calculated in dimension orders (B, C), (A, C) and (A, B) in memory since we read ABC chunks in dimension order

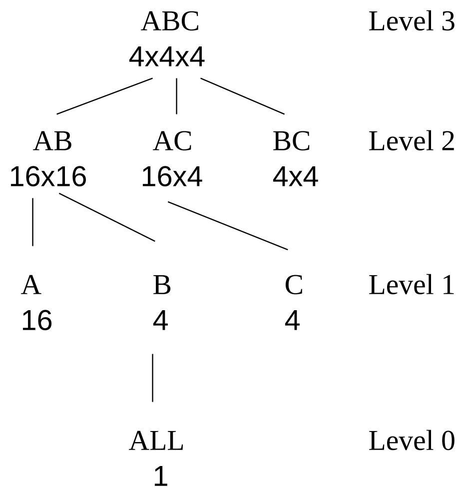


Figure 2.8: 3-D array MMST in dimension order ABC [ZDN97].

(A, B, C). To each node A, B, and C, this is equivalent to reading in chunks of groupby AB and AC in the dimension order (A, B) and (A, C). Similar to the nodes in level 2, the chunks of the nodes A, B, and C are generated in the proper dimension orders. In general, the chunks of each tree node are generated in a proper dimension order. Moreover, memory is allocated to only the minimum number of chunks to each nodes instead of all chunks.

The algorithm succeeds in overlapping the computation of multiple subaggregates using the MMST data structure, and it makes good use of available main memory. Zhao, et al [ZDN97] claimed that this array-based algorithm for MOLAP systems may be useful for ROLAP systems as well. They did experiments to show that it is faster to first convert the table to an array, cube the array, then convert the result back to a table. Furthermore, the arrays are stored efficiently on disk because they are *chunked* and the chunks are compressed if sparsed.

2.3.3 Partitioning

Ross, et al [RS97] considered real-world data sets, where it is orders of magnitude more sparse than the synthetic data sets considered in [ZDN97]. For extremely sparse data, the array representation of [ZDN97] cannot fit into memory, so a more costly data structure would be necessary. Two algorithms, `Partitioned-Cube`, and `Memory-Cube`, are proposed for the fast computation of datacubes for ROLAP over large and sparse relations. A relation is *sparse* with respect to a set of attributes if its cardinality is a small fraction of the size of the cross-product of the attribute domains. Real-world data in application domains is often very large and sparse. Therefore, efficiently computing datacubes over large and sparse relations is important.

The idea is to partition the large relations into fragments that fit in memory, and perform the complex operation over each memory-sized fragment independently. `Partitioned-Cube` assumes the existence of a subroutine `Memory-Cube` that computes the datacube of a relation that fits in memory. Larger partitions that do not fit in memory will be repartitioned.

Given a relation with attributes $B_1, B_2, \ldots, B_k$, `Partitioned-Cube` uses a divide-and-conquer strategy that divides the problem of computing the datacube over the relation with T tuples and k attributes into $n+1$ sub-datacubes, for a large number n. The first n of these sub-datacubes each has approximately T/n tuples and k attributes. The final sub-datacube has no more than T tuples and has $k-1$ attributes. `Partitioned-Cube` does not specify exactly how to partition, or how many partitions to create. Given a partitioning attribute B_j, the number n of partitions is bounded above by both the cardinality of the domain of B_j, and the number of buffers that can be allocated in memory.

`Memory-Cube` is formulated for the efficient computation of datacubes over relations that fit in memory. It performs multiple in-memory sorts, and does not incur any I/O beyond the input of the relation and the output of the datacube itself. It also performs as few sorts as possible, and it takes advantage of common prefixes between multiple sort orders to optimize the CPU cost.

`Memory-Cube` computes the datacube as follows: it takes a prefix-ordered path, and sorts the in-memory relation according to the attribute ordering of the initial node of the path. It then makes a single scan through the data, accumulating aggregates along the way at all granularities on the path. Aggregates from finer granularities are combined with those at coarser granularities when the corresponding grouping attributes change. Datacube results are output immediately.

Similar to `PipeSort` [AAD+96], `Memory-Cube` computes the various cuboids (groupbys) of the datacube using the idea of pipelined paths. The search lattice in [AAD+96] determine which of the 2^k cuboids can be computed from which others. To minimize the number of expensive sort operations performed, it is desirable to minimize the total number of paths in the search lattice that are generated to cover all the nodes. `Memory-Cube` constructs the desired set of paths using the algorithm `Paths`$(B_1, \ldots, B_k)$. For details, please refer to [RS97].

The number of in-memory sorts needed is exponential in k. This exponential factor is unavoidable, because the width of the search lattice of the datacube is exponential in k. `Partitioned-Cube` combined with `Memory-Cube` is the first solution where the total I/O overhead is linear in the number of groupby attributes when the partitions fit in memory. Previous techniques were either exponential or had a quadratic approximate lower bound.

Chapter 3

Chunk-Based Multidimensional Datacube

Motivated by the CUBE operator [GBLP96] and the array-based algorithm [ZDN97], we have designed a chunk-based compressed datacube using the MOLAP approach together with some nice relational features. We have also invented an efficient aggregation computation algorithm (`R-cubing`) which incorporates the following optimizations: (1) scanning the relation only once, (2) computing a groupby from the smallest computed groupby, (3) computing as many groupbys as possible together in memory, and (4) saving the groupbys as soon as not needed to free up memory for computing other groupbys. We will focus on the distributive aggregate functions only [GBLP96][1], and we will consider materializing all possible groupby combinations [HRU96].

3.1 Introduction

Based on the multidimensional data model, the cube is N dimensional. Let its dimensions be $D_0, D_1, \ldots, D_{N-1}$. We add one more value, `ALL` to each dimension D_i, for i from 0 to $N-1$ as described in [GBLP96]. Thus, any cell with the `ALL` value as one of its address components corresponds to a groupby. From here on, these cells

[1] Note that algebraic functions can be computed easily from the computed distributive aggregate values

will be called *cuboid cells* and those from the raw data are called *core cells*. Notice that all our enumeration start from zero.

3.1.1 Architecture

Each dimension values are mapped to coordinates. For i from 0 to $N-1$, let dimension D_i have the values $\{d_{i_0}, d_{i_1}, \ldots, d_{i_{|D_i|-1}}, d_{i_{\texttt{ALL}}}\}$, and for convenience, $d_{i_{\texttt{ALL}}}$ is simply implemented or denoted as $d_{i_{|D_i|}}$. Thus, each dimension D_i has $|D_i| + 1$ values in total.

To denote a cube cell, we use vector $V(v_0, v_1, \ldots, v_{N-1})$ to denote its position in the multidimensional space. The magnitude of the vector must be the same as the number of dimensions in the cube. For a cuboid cell, at least one of its address components v_j must equal $|D_j|$, where $j \geq 0$ and $j < N$. Thus, the cell addressed by vector $V(|D_0|, \ldots, |D_{N-1}|)$ is the cuboid cell corresponding to the empty groupby `ALL`.

From figure 3.1, we see a 3-D cube call C, which will be used throughout this chapter. The dimensions D_0, D_1, D_2 have sizes as follows: $|D_0| = 5$, $|D_1| = 4$, and $|D_2| = 3$. For convenience, the values for each dimension are just numbers in such a way that d_{i_0} is 0, d_{i_1} is 1, and so on, for $i = 0, 1, 2$. Therefore, the cuboid cell with address $V(5, 4, 3)$ is actually the cell $V(\texttt{ALL}, \texttt{ALL}, \texttt{ALL})$.

According to our convention, the cuboid cell with the address $V(5, 0, 0)$ stores the sum of the measurements of the core cells $\{V(0, 0, 0)$, $V(1, 0, 0)$, $V(2, 0, 0)$, $V(3, 0, 0)$, $V(4, 0, 0)\}$. Similarly, another cuboid cell with address $V(5, 4, 0)$ stores the sum of the measurements of the cuboid cells from set S_1 : $\{V(5, 0, 0)$, $V(5, 1, 0)$, $V(5, 2, 0)$, $V(5, 3, 0)\}$, or set S_2 : $\{V(0, 4, 0)$, $V(1, 4, 0)$, $V(2, 4, 0)$, $V(3, 4, 0)$, $V(4, 4, 0)\}$. Thus, the cuboid cell $V(5, 4, 0)$ is *dependent* on the cuboid cells in sets S_1 and S_2 [HRU96]. In other words, the cuboid cells in sets S_1 or S_2 are potential *subcuboids* for cuboid cell $V(5, 4, 0)$.

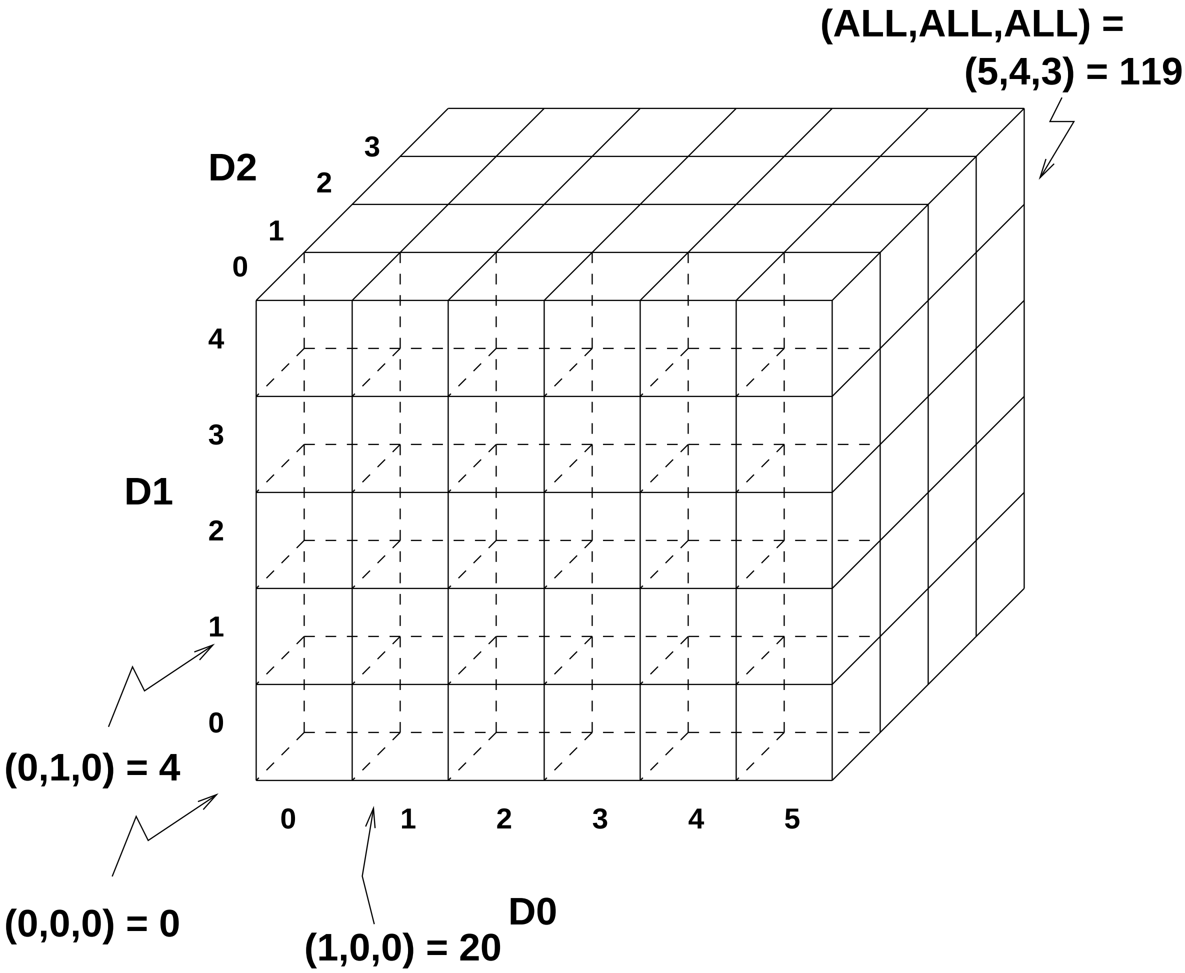


Figure 3.1: Cube C of 3 Dimensions

3.1.2 Design

Our cube is designed for a query-driven setting, so the number of dimensions in the cube is unknown until the request for cube construction is submitted. Therefore, conceptually, our cube is a multidimensional cube, but it is implemented using a one dimensional array.

For example, the cells of cube C in figure 3.1 are ordered as shown in table 3.1 such that the address $V(0,0,0)$ and $V(0,1,0)$ are located at positions 0 and 4 in the array. The last cell of cube C ($V(5,4,3)$) is located at position 119 because there are $(5+1) \times (4+1) \times (3+1) = 120$ cells in total.

$V(0,0,0)$	$V(0,0,1)$	$\cdots$	$V(0,0,\|D_2\|)$
$V(0,1,0)$	$V(0,1,1)$	$\cdots$	$V(0,1,\|D_2\|)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$V(0,\|D_1\|,0)$	$V(0,\|D_1\|,1)$	$\cdots$	$V(0,\|D_1\|,\|D_2\|)$
$V(1,0,0)$	$V(1,0,1)$	$\cdots$	$V(1,0,\|D_2\|)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$V(\|D_0\|,2,0)$	$V(\|D_0\|,2,1)$	$\cdots$	$V(\|D_0\|,2,\|D_2\|)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$V(\|D_0\|,\|D_1\|,0)$	$V(\|D_0\|,\|D_1\|,1)$	$\cdots$	$V(\|D_0\|,\|D_1\|,\|D_2\|)$

Table 3.1: The ordering of the cube cells in an array

Direct mappings from the array position indexes to the vector in the multidimensional space or vice versa are quite straightforward. Algorithm Vector-To-Index in table 3.2 shows how to map a multidimensional vector to the corresponding array position index. It is an iterative function (which can be easily converted to a recursive function), and is initially invoked by Vector-To-Index(V($v_0, \ldots, v_{N-1}$)).

Algorithm Vector-To-Index assumes the cube cells are ordered as in table 3.1 such that all values of dimension D_j will be enumerated before the value of dimension D_i is changed, for any $0 \leq i < j < N$. Once the value of dimension D_i is changed, the enumeration of all values of D_j will start again. Therefore, the formula to get the array position index from the 3-dimensional vector $V(v_0, v_1, v_2)$ is

$$index = ((v_0(|D_1| + 1) + v_1)(|D_2| + 1)) + v_2.$$

In cube C, $V(0,1,0)$ is $((0(4+1)+1)(3+1))+0 = 4$, $V(1,0,0)$ is $((1(4+1)+0)(3+1))+0 = 20$, and $V(5,4,3)$ is $((5(4+1)+4)(3+1))+3 = 119$. Note that the last cell in the array always corresponds to the empty groupby.

Similarly, Algorithm Index-To-Vector in table 3.3 converts an array position index to the corresponding multidimensional vector. For cube cell retrieval and construction, Algorithm Vector-To-Index is used a lot more than Algorithm Index-To-Vector, but it is sometimes very useful.

```
PROCEDURE Vector-To-Index(Vector V(v_0, ..., v_{N-1})) {
    // INPUT:     multidimensional vector V(v_0, ..., v_{N-1})
    // OUTPUT:    The array position index
    index = 0;
    // N is the number of dimensions in the cube
    for (i = 0; i < N - 1; i++) {
        index = index + v_i;
        index = index × (|D_{i+1}| + 1);
    } // end for
    index = index + v_{N-1};
    return index;
} // end PROCEDURE
```

Table 3.2: The Algorithm Vector-To-Index.

3.2 Datacube Representation

When the cube is small, the array guarantees efficient look-up of cube cells. Unfortunately, it will grow rapidly with the increase of the number of dimensions in the cube. Let us look at some different ways to handle this sparse matrix array problem.

3.2.1 Compressed Cubes in One Dimensional Array

Typically 20% of the data in the logical OLAP cube are non-zero [Col96], so a simple way is to compress this array such that the empty cells are removed.

Originally, the conceptual cube cell location (offset) in the multidimensional space is implicit given the array position. Therefore, each cell (assume non-empty) in the compressed array is associated with its offset. For example, if the cube has only 5 cells, the array A also has 5 elements: $a[i] = cell[i]$, where $0 \leq i < 5$. If only $a[1]$ and $a[3]$ are not empty, a compressed array A' is formed such that A' only has two cells: $a[0]'$ ($cell[1]$ and the offset 1) and $a[1]'$ ($cell[3]$ and the offset 3).

Binary search algorithm can be used for cell lookup using the cell offsets. Unfortunately, the sparse array has to be created and filled before we can compress it because in this case the cells in the compressed array will have been ordered according to the

```
PROCEDURE Index-To-Vector(int index) {
    // INPUT:      The array position index
    // OUTPUT:     Multidimensional vector V(v_0, ..., v_{N-1})
    last = N - 1;
    // N is the number of dimensions in the cube
    div = 1;
    for (i = 0; i < N; i++) {
        if (i == last)
            v_i = index mod(|D_last| + 1);
        else {
            j = i + 1;
            div = 1;
            while (j ≤ last) {
                div = div × (|D_j| + 1);
                j++;
            } // end while
            v_i = (index / div) mod(|D_i| + 1);
        } // end else
    } // end for
    return V;
} // end PROCEDURE
```

Table 3.3: The Algorithm Index-To-Vector.

cell offsets. Otherwise, we have to either sort and/or eliminate duplicate cells later or insert the cells in the increasing order. Therefore, this method is not very efficient.

3.2.2 Compressed Cubes Using B+-tree

This method eliminates the step of allocating a big chunk of memory. We use the B+-tree structure with the cell offsets as keys. Let us first give a brief introduction to B+-trees.

Each node in a B+-tree contains up to $n-1$ search-key values and n pointers. A B+-tree contains both leaf nodes and zero or more non-leaf nodes. In a leaf node, the first $n-1$ pointers point to the corresponding cube cells while the last pointer points to the next leaf node. That is, the leaf nodes are chained together in the search-key

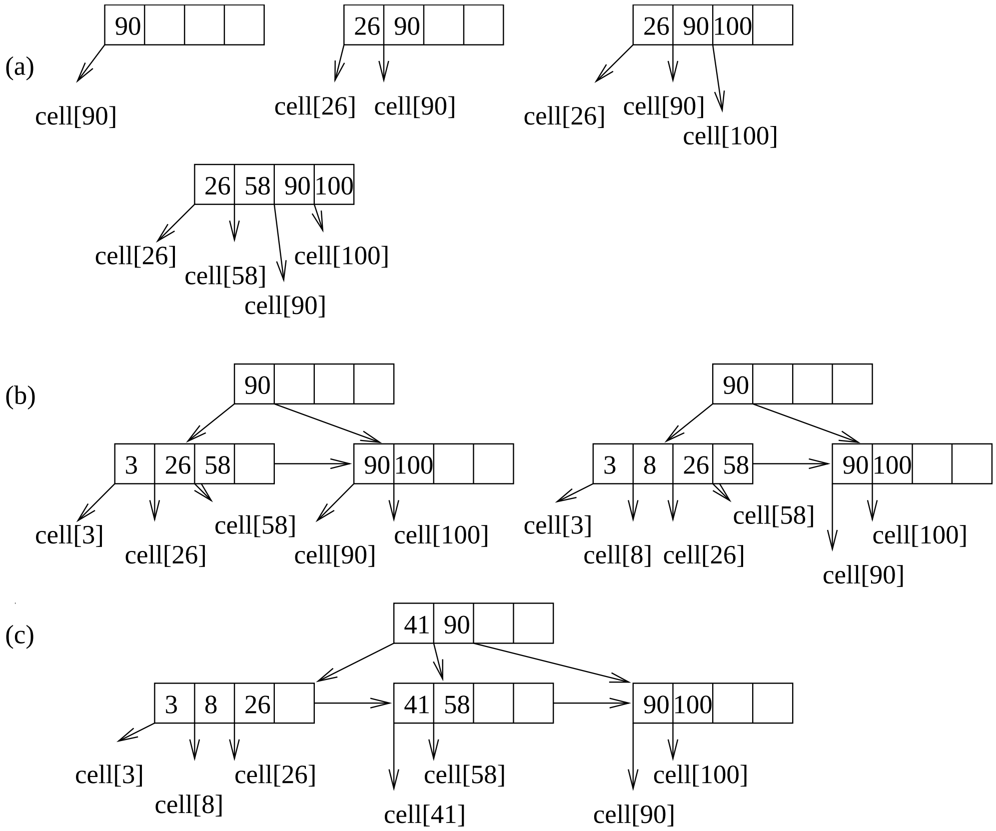


Figure 3.2: Insertion of (a) 90, 26, 100, 58 (b) 3, 8 (c) 41 to a B+-tree

order. Each leaf can hold up to $n-1$ keys and as few as $\lceil (n-1)/2 \rceil$ keys. Each key is associated with a pointer that points to a unique cell.

The non-leaf nodes form a multilevel (can be sparse) index on the leaf nodes. The structure is the same as that of leaf nodes except that all pointers point to tree nodes only. The extra requirement for a non-leaf node compared to a leaf node is this. The i-th pointer of a non-leaf node must point to a tree node such that all keys in that node must be smaller than the i-th key in the non-leaf node and the $(i+1)$-th pointer must point to a tree node such that all keys in that node must be greater than or equal to the i-th key. A non-leaf node may hold up to n pointers, but must hold at least $\lceil n/2 \rceil$ pointers. This requirement is imposed at all levels of the tree except for the root.

One important characteristic of B+-trees is that the length of every path from the root to a leaf node is the same. Indeed, the "B" in B+-tree stands for "balanced". It is the balance property of B+-trees that ensures good performance for lookup and insertion.

The cube in figure 3.1 have 120 cells in total. Let the cells at offsets 3, 8, 26, 41, 58, 90, 100 be non-empty. Figure 3.2 shows how a B+-tree with $n = 5$ is formed when the cells are inserted in the order of 90, 26, 100, 58, 3, 8, and 41.

There is a little overhead when inserting cells into a B+-tree structure, but the cells pointed by the leaf node pointers are always ordered by the cell offsets in the ascending order. Thus, binary search can be used for a cell lookup. For more details on B+-trees, please refer to the chapter on Indexing and Hashing in [KS91].

3.2.3 Disk-based Compressed Cubes

When a compressed format of a cube, either a compressed array or a B+-tree cannot fit in the memory, the cube has to reside on disk. One option is to put the compressed array format onto a flat file and then access from the file directly. However, this is not very efficient once the cube has become very huge.

Another way is to adopt the ROLAP approach — put the compressed array in a relational table, which consists of a few columns, the *cell offset* plus the measurements of this cell, *measure 1*, *measure 2*, etc. Therefore, each tuple is a cube cell in compressed array format. After all cells are inserted into the table, the cells will be indexed and we can utilize the available database operations to ensure fast access to any cells.

When the total number of cells in the cube is larger than a long integer (e.g. more than 2^{31} cells), the offsets can be represented as strings of characters instead of integers. For example, if the cell offset is 9000, the character string for this offset can be "9000". This is very straightforward, but we have to provide our own integer calculation routines.

Alternatively, we can form character strings in a different way. The cell offset is calculated by Algorithm Vector-To-Index in table 3.2, but instead of returning a long integer offset, a character string offset is returned. Given the vector $V(v_0, v_1, \ldots, v_N)$, the steps are as follows. (1) Convert each v_i, $0 \leq i < n$, from decimal integer to a binary string b_i. The number of bits in b_i should be enough to hold all distinct values of dimension D_i. We may need to pad zeroes to the left sometimes. (2) Concatenate all these binary strings from b_0 to b_{N-1} to form a long binary string B. (3) Pad extra zeroes to the left of the binary string B if the number of bits in B is not divisible by 4. (3) Convert this binary string B into a hexadecimal character string H.

To illustrate, consider cube C again in figure 3.1. For dimension D_0, 3 bits are required to represent all its different values. Similarly, 3 bits and 2 bits are required for dimensions D_1 and D_2 respectively. We need 3 bits for dimension D_1 because it has $4 + 1$ distinct values in total including the value `ALL`. In total, the number of bits in the concatenated binary string B will be $3 + 3 + 2 = 8$. Given a vector $V(3, 3, 2)$, b_0 is "011", b_1 is "011" and b_2 is "10". Thus, the concatenated string B is "01101110". Since the number of bits in B is already divisible by 4, we do not need to add padded zeroes to the left. We convert four bits of B at a time to a hexadecimal digit. Thus, "0110" is converted to "6" and "1110" is converted to "E". The resulting H is therefore "6E".

This method is completely disk-based, so it might scale well, but it is definitely slower than the above methods and the following method.

3.2.4 Chunk-based Compressed Cubes

Based on the array-based algorithm in [ZDN97], we divide the array into *chunks*, each of which should be small enough to fit in the main memory. Each chunk is itself an array, and it can be sparse too. Unlike the cube, a chunk will not have both cuboid cells and core cells. Hence, a chunk will either be a *core chunk* or a *cuboid chunk*. Most cuboid chunks are significantly smaller than the core ones, and will likely be accessed relatively more frequently than the core chunks. To ensure fast access to these cuboid chunks, we separate them from the core ones.

Chunks and Segments

Given an N-dimensional cube, *chunking* divides it into small size N-dimensional chunks and store each chunk as one object on disk. Most of the chunks have the same size on each dimension. Figure 3.3 shows cube C after chunking. It has 3 dimensions D_0, D_1, and D_2, and again $|D_0| = 5$, $|D_1| = 4$, and $|D_2| = 3$. There are 120 cells in total and 60 of which are core cells. If each chunk can have at most 27 cells, then the dimension size of each chunk is 3. In this case, dimensions D_0, D_1, and D_2 are broken into 2, 2 and 1 *segments* respectively. Again as before, we add one more segment of size 1 to each dimension to form cuboid chunks. Therefore, cube C in figure 3.3 have $(2+1) \times (2+1) \times (1+1) = 18$ chunks in total. If the limit is 64 cells instead of 27, then each dimension is allowed to have segments of size 4. Note that dimension D_2 will still have only 1 segment of size 3.

We treat each chunk as if it was a cube cell, and the chunks are ordered by D_2, then D_1 and lastly D_0 also in an array. Thus, we can still use Algorithm Vector-To-Index to assign IDs (array positions) to different chunks except that we use number of core segments (excluding the cuboid segment) in each dimension ($|D_0|'$, $|D_1|'$ and $|D_2|'$) instead of number of core distinct values. Therefore, the corresponding multidimensional vector for the first core chunk is $V(0,0,0)$ and its ID is 0. For a cuboid chunk, one of its vector components v_i must equal $|D_i|'$, $0 \leq i < N$. Thus, the ID for the last cuboid chunk is 17, and its corresponding vector is $V(2,2,1)$.

It is obvious that the chunks of cube C are not of the same volume. In general, for any N-dimensional cubes, we set a limit to the largest chunk size, say S. For the largest core chunk, each of its dimension size is at least $s = \lfloor \sqrt[N]{S} \rfloor \simeq \sqrt[N]{S}$. The number of core segments for any dimension D_i, $0 \leq i < N$, is determined as follows. If $|D_i| \leq s$, then $|D_i|' = 1$. Otherwise, $|D_i|' = (|D_i|/s)$ if $|D_i|$ is divisible by s. If not, then $|D_i|' = (|D_i|/s) + 1$, and the last segment size s' of D_i is $(|D_i| \bmod s)$. Note that all chunks in a cube should be at most as large as the first core chunk.

Alternatively, the chunk segment sizes can be computed according to the dimension sizes. For example, when one of the dimensions has size way more than the other, if we use the above method, we might end up with a very small segment size of that

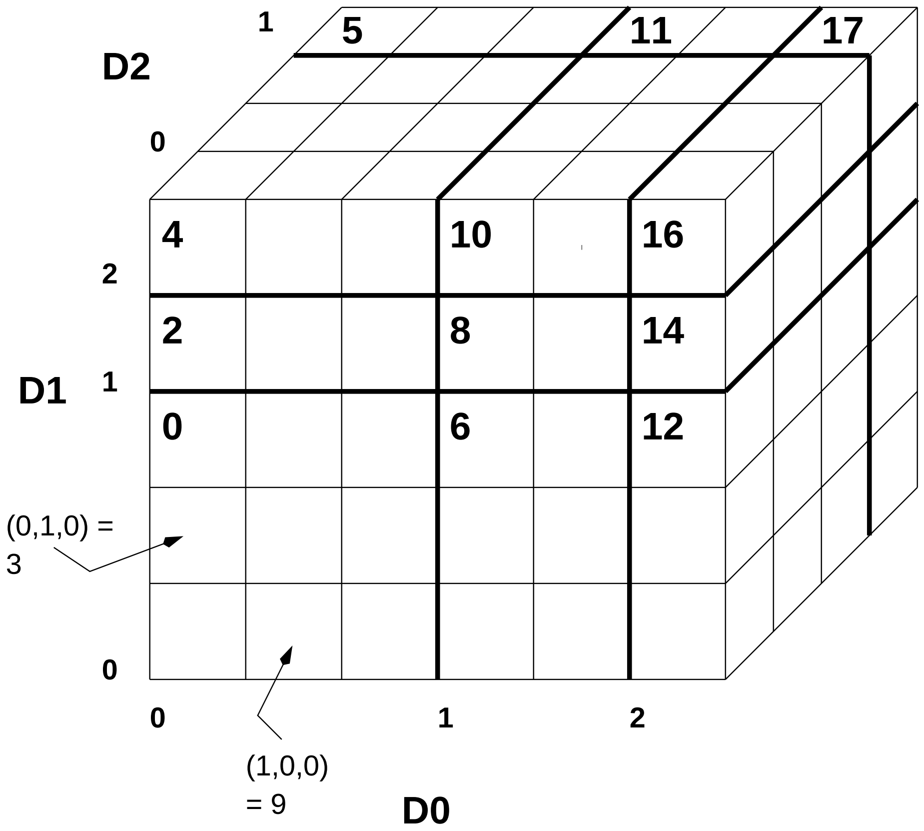


Figure 3.3: Cube C with 18 chunks

dimension and it will have a lot more segments than the other dimensions. Since our chunks do not necessarily have the same volume anyway, we can consider another way to segment the dimensions and the steps are illustrated in table 3.4. We will now describe the method using an example. First, we find out which dimension has the largest size and let it be s_{max}. Then we loop through the rest of the dimensions to see whether we can simply use s_{max} as the segment size for all dimensions. Again, let the limit be 27 cells, and $s_{max} = |D_0| = 5$. As stated above, both dimensions D_1 and D_2 will have segment sizes equal to 4 and 3 respectively. The number of cells in the first core chunk is $5 \times 4 \times 3 = 60$. It is way over our limit, so the segment size cannot be 5.

Then what we do is to set

$$s_{max} = \lceil s_{max}/2 + 1 \rceil$$

because we assume this dimension is much larger than the others. Hence, s_{max} is now 3, and the resulting cube is the same as that in figure 3.3. However, if our limit is 64 cells instead of 27 cells, we could have used $s_{max} = 5$ such that there are only $(1+1) \times (1+1) \times (1+1) = 8$ chunks in total. If the memory is sufficient to hold larger chunks, it is always more desirable to have fewer chunks instead. We will know why in section 3.3.7.

Cube space and Chunk space

Each chunk is like a cube, it has its own conceptual multidimensional space. We can use a slightly different version of Algorithm Vector-To-Index in table 3.2 to handle this. We use the number of cells along dimension D_{i+1} in a chunk ($|D_{i+1}|''$) instead. Moreover, there is no need to add 1 to $|D_{i+1}|''$ because a chunk is either a core chunk or a cuboid chunk. For example, chunk 0 of cube C has $3 \times 3 \times 3 = 27$ cells in total. The formula to get the chunk cell offsets (chunk's array positions) from the 3-dimensional vector $V(v_0, v_1, v_2)$ is

$$index = (v_0(|D_1|'') + v_1)(|D_2|'') + v_2.$$

Thus, $V(0,1,0)$ is $(0(3)+1)(3)+0 = 3$, and $V(1,0,0)$ is $(1(3)+0)(3)+0 = 9$.

Consequently, there exist two conceptual multidimensional models for a chunk-based datacube. Let us call the one that determines the chunk IDs the *global* model and the one that determines the cell offsets within each chunk the *local* model. Hence, a cell in the chunk-based datacube is always associated with a *global* and a *local* vectors. For example, the last cuboid cell in cube C at $V(5,4,3)$ now is the only cell in chunk 17. The associated vectors are then $GV(2,2,1)$ and $LV(0,0,0)$.

We can still retrieve a cube cell using its cell-based multidimensional vector. The cube will convert it into the corresponding global and local vectors. Given the vector $V(5,4,3)$, the following steps are taken: (1) From V, we know v_0 is 5. Since the

```
PROCEDURE Compute-Segment-Size() {
    // INPUT:      None
    // OUTPUT:     The desirable segment size for each chunk s_max
    // N is the number of dimensions in the cube
    s_max = |D_0|;
    for (i = 1; i < N; i++) {
        if (|D_i| > s_max)
            s_max = |D_i|;
    } // end for
    Done = FALSE;
    while (not Done) {
        nCells = s_max;
        for (i = 1; i < N; i++) {
            s = min(s_max, |D_i|);
            nCells = nCells × s;
            if (nCells > LIMIT) {
                s_max = ⌈s_max/2 + 1⌉;
                break;
            } // end if
        } // end for
        if (i == N)
            Done = TRUE;
    } // end while
    return s_max;
} // end PROCEDURE
```

Table 3.4: The Algorithm Compute-Segment-Size.

first segment size of dimension D_0 is 3 and $5 \geq 3$, we proceed to the next segment, which has size 2. Again, $5 \geq (3+2)$, so we end up in the last segment, and $gv_0 = 2$. Similarly, for dimensions D_1 and D_2, we end up in segments 2 and 1, so $gv_1 = 2$ and $gv_2 = 1$. Therefore, $V(5,4,3)$ refers to the chunk denoted by $GV(2,2,1)$. (2) The next step is to determine the local vector LV for $V(5,4,3)$ by using the formula

$$lv_i = v_i - \sum_{j=0}^{gv_i - 1} s_{i_j},$$

where $0 \leq i < N$ and s_{i_j} is the number of cells in the j-th segment of dimension D_i.

As a result, $lv_0 = 5 - (3+2) = 0$, $lv_1 = 4 - (3+1) = 0$, and $lv_2 = 3 - (3) = 0$.

We can use a simpler method to compute the global and local vectors for the core cells. Given any cell-based vector $V(v_0, v_1, v_2)$ for cube C, $gv_i = v_i/s_i$ and $lv_i = v_i \bmod s_i$ for $i = 0, 1, 2$, and s_i is the segment size of dimension i. Unfortunately, this method does not apply to the cuboid cells at all. For example, the cuboid cell at $V(5,4,3)$ (cell 119) is associated with $GV(2,2,1)$ and $LV(0,0,0)$, and the segment size for each dimension is 3. Yet $gv_0 = 2 \neq 5/3$ and $lv_1 = 0 \neq 5 \bmod 3$. On the other hand, for the core cell at $V(1,0,0)$ (cell 20 in figure 3.1), $gv_i = v_i/3$ and $lv_i = v_i \bmod 3$ ($GV(0,0,0)$ and $LV(1,0,0)$ in figure 3.3). Therefore, if we know that we will be dealing with core cells only, we can use this simpler method. In general, we use the original method to compute the global and local vectors.

Data Input

Given the source data relation R, each tuple is first mapped to the cell-based multidimensional vector. Then we apply the simpler method to determine its global and local vectors because each tuple is by definition mapped to a core cell. Once the cube cell is identified, the data values (measurements) of the tuple will be added to it.

After the relation has been scanned, the core chunks are done and will be used to compute the cuboid chunks (aggregation).

Compress the Chunks

When the cube is sparse, we can use the compressed methods illustrated in sections 3.2.1 and 3.2.2 to both the cube and its chunks. For example, it is very likely that some chunks of a sparse cube are completely empty. We simply do not include them in the cube. Therefore, we could apply the methods described above to compress the cube. Similarly, most non-empty chunks could be sparse too and these chunks will also be compressed. We can set up a threshold to determine when to compress a chunk (same as compressing a cube) by looking at the chunk density or we can simply compress all non-empty chunks.

Currently, we simply compress all the chunks. Therefore, when the cube is very sparse, we could have a compressed array of chunks each of which is a compressed array of cells. We could also use the B+-tree structure to index a chunk in the cube or a cell in a chunk.

3.3 Aggregation

Computing the aggregates is the core part of constructing a datacube for a data warehouse. Our goals are to scan the relation only once, compute a groupby from the smallest computed groupby, compute as many groupbys as possible together in memory, and save the groupbys as soon as not needed to free up memory for computing other groupbys.

To look at aggregation in general, we know, from figure 3.1, that the cell at

$$V(|D_0|, v_1, v_2) = \sum_{j=0}^{|D_0|-1} V(j, v_1, v_2)$$

and the cell at

$$V(|D_0|, v_1, |D_2|) = \sum_{j=0}^{|D_0|-1} V(j, v_1, |D_2|)$$

or preferably

$$V(|D_0|, v_1, |D_2|) = \sum_{k=0}^{|D_2|-1} V(|D_0|, v_1, k).$$

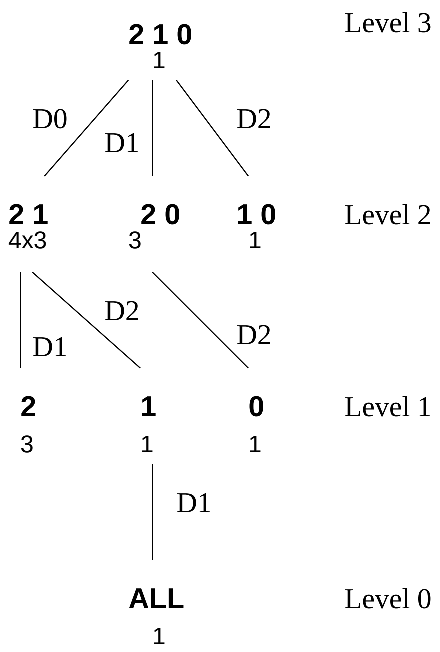
The reason why we prefer the second one to compute $V(|D_0|, v_1, |D_2|)$ is that the number of cells along dimension D_2 is less than that along dimension D_0. For any cuboid cell of the similar type as $V(|D_0|, v_1, |D_2|)$ (i.e. more than one address component v_i of vector V equals $|D_i|$), we have to determine the dimension along which the summation takes place. Instead of determining this at every cell vector, we use the concepts of both the Minimum Memory Spanning Tree (MMST) and the Optimal Dimension Order introduced in [ZDN97].

We have implemented the aggregation computation method, `R-cubing`, which is similar to `Multi-way Array Cubing` in [ZDN97]. We will first describe the aggregation process by a detailed example using the MMST. Then we will explain why

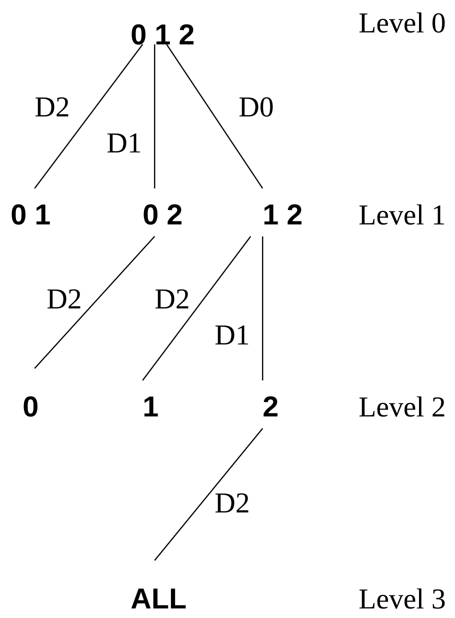
`R-cubing` can aggregate without it. Throughout this section, we will often refer to both figure 3.1 and figure 3.3.

3.3.1 A Brief Introduction

Zhao, et al [ZDN97] stressed the fact that multiple related groupbys were computed simultaneously while scanning the relation once with the least amount of memory used. They used a data structure called MMST to coordinate the overlapped computation.



(a) The MMST for cube C.



(b) The MNST for cube C.

Figure 3.4: (a) The MMST for cube C (b) The MNST for cube C

The MMST for cube C in figure 3.1 is shown in figure 3.4(a). According to [ZDN97], since $|D_0| > |D_1| > |D_2|$, the optimal dimension order is (D_2, D_1, D_0). Let us use figure 3.5 to briefly explain the aggregation process and its memory requirement using the MMST structure.

The numbers below each node in the MMST denotes the number of groupby cells required to stay in the memory. For example, when the cells at $V(0,0,0)$, $V(0,0,1)$, $V(0,0,2)$ are read in order, the cuboid cell at $V(0,0,3)$ (correspond to node **10** of the MMST) is needed. When we scan the cell at $V(0,1,0)$, we no longer need the cuboid cell $V(0,0,3)$. Instead, it can be saved and replaced by cell $V(0,1,3)$ in the memory. Thus, we only need one cell for node **10** of the MMST. On the other hand, at most three cuboid cells are needed in the memory for node **20**. For example, when the above core cells are read, their counts are summed to three cuboid cells at $V(0,4,0)$, $V(0,4,1)$, $V(0,4,2)$ respectively. These cells will stay in memory until the core cell $V(1,0,0)$ is read. In that case, the cuboid cells $V(1,4,0)$, $V(1,4,1)$, $V(1,4,2)$ will be loaded into memory instead. However, for node **21** of the MMST, we need a lot more cuboid cells in memory when the core cells are being scanned. From figure 3.5, we know that the cuboid cell $V(5,0,0)$ is computed by the sum of the core cells $V(0,0,0)$, $V(1,0,0)$, $\ldots$, $V(4,0,0)$. Therefore, the cuboid cells at $V(5,0,0)$, $V(5,0,1)$, $V(5,0,2)$, $V(5,1,0)$, $\ldots$, $V(5,3,2)$ ($4 \times 3 = 12$ cells in total) are all needed in the memory. When one of these cuboid cells have been computed (say $V(5,0,0)$), the cuboid cell at $V(5,4,0)$ and $V(5,0,3)$ can be loaded into memory. With similar arguments as above, at most 3 cells for node **2** and 1 cell for node **1** are required in memory.

We have employed a slightly different MMST to guide the aggregation process. We call it the **Minimum Number Spanning Tree** (MNST). Let us first describe it and then discuss the aggregation computation in detail.

3.3.2 The Minimum Number Spanning Tree

Figure 3.4(b) shows the MNST. Compare it with the MMST, the differences are as follows. The dimensions are ordered such that $|D_0| \geq |D_1| \geq |D_2| \geq \cdots |D_{N-1}|$, and by definition, the optimal dimension order $\mathcal{O}= \{D_{N-1}, \ldots, D_2, D_1, D_0\}$. Thus, the MNST like a mirror image of the MMST. Also, the level numbers are changed such that the root level is now level 0, not level N. That is, the level number increases when we go down to the levels below the root level. The MNST for a cube $(D_0, \ldots, D_{N-1})$ in a dimension order $\mathcal{O}= (D_{j_0}, \ldots, D_{j_{N-1}})$ has $N+1$ levels with the root $(D_{j_0}, \ldots, D_{j_{N-1}})$

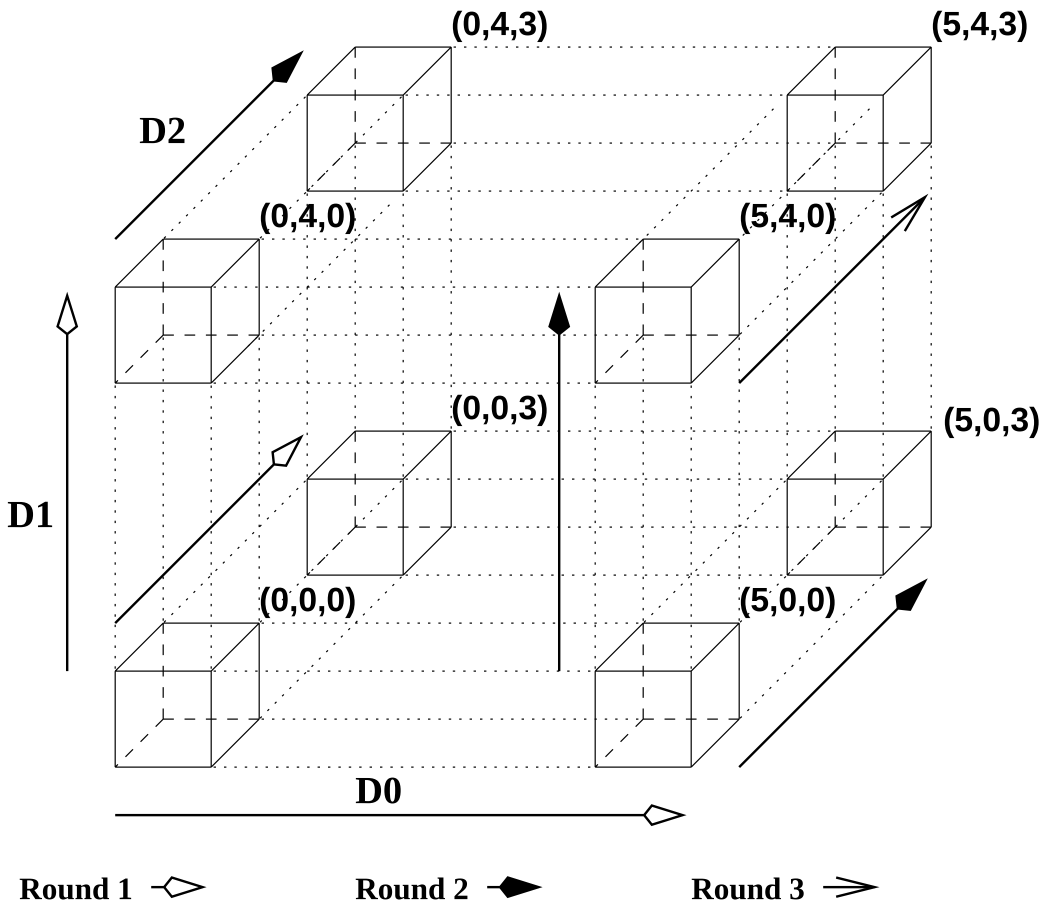


Figure 3.5: The relationship between cuboids and their subcuboids with respect to cell $V(0,0,0)$ using `R-cubing`

at level 0. Any tree node M at level i below level 0 may be computed from those nodes at one level up whose dimensions contain the dimensions of node M. For any node M at level i, there may be more than one node at level $i-1$ from which it can be computed. Instead of choosing the node that makes the node M require the minimum memory, we choose the node that requires the minimum *number* of subcuboid cells to compute the groupby aggregation of node M. Therefore, in figure 3.4(b), to compute groupby node ALL, we choose node 2 instead of node 1 because node 2 requires $|D_2|$ cells only where node 1 requires $|D_1|$ cells (note that $|D_2| < |D_1|$) to compute node ALL.

3.3.3 R-cubing with MNST

The aggregation computation is designed to be disk-based such that memory will be allocated dynamically to the required cube cells and they will be saved onto the disk as soon as they are not needed. After they are saved, their allocated memory will be freed. To illustrate the process step by step, we will use the MNST in figure 3.4(b) and cube C in figure 3.5.

(1) Since the cell at $V(0,0,0)$ is a core cell, we traverse the MNST from node 012 at level 0 (the root level). We know that from node 012 we can compute nodes 01, 02 and 12 along dimensions D_2, D_1 and D_0 respectively. Therefore, the cell $V(0,0,0)$ is summed to the cuboid cells at $V(0,0,3)$ along dimension D_2, $V(0,4,0)$ along dimension D_1, and $V(5,0,0)$ along D_0. In total, we have three cuboid cells in the memory at this point. (2) Repeat the process for the next core cell at $V(0,0,1)$. This node is summed to $V(0,0,3)$, $V(0,4,1)$ and $V(5,0,1)$ respectively. Thus, two more cuboid cells are loaded into memory. (3) Similarly, the next core cell at $V(0,0,2)$ will be summed to $V(0,0,3)$, $V(0,4,2)$ and $V(5,0,2)$ respectively (two more cuboid cells are loaded into memory). (4) Before we proceed further, we found that the cuboid cell at $V(0,0,3)$ has got all the sums it should need. Now we should check with the MNST whether this cuboid cell is useful as a subcuboid cell for other cuboids. From the MNST, this groupby corresponds to node 01 and no tree nodes are dependent upon it. Thus, $V(0,0,3)$ is saved and its memory freed. (5) The core cells at $V(0,1,0), V(0,1,1), V(0,1,2), V(0,2,0), \ldots, V(0,3,1), V(0,3,2)$ will be examined in turn, and they will be summed to the corresponding cuboid cells. Again, the cuboid cells at $V(0,1,3)$, $V(0,2,3)$ and $V(0,3,3)$ will have got what they need and be saved and freed. (6) Note that when reading in the core cell $V(0,3,0)$, the cell will be summed to the cuboid cell at $V(0,4,0)$. At this moment, this cuboid cell has got what it wants, so it in turn will check the MNST whether it can be useful for other cuboids. In the MNST, this cuboid cell refers to the node 02 and node 0 at level 2 is dependent on it. Therefore, the cell $V(0,4,0)$ is summed to the cell $V(0,4,3)$ (load this cell into memory) along dimension D_2. This process will repeat for each core cell and cuboid cell until all core cells have been traversed and all cuboid cells have been

materialized.

It is clear that this process recursively traverse the nodes in the MNST to ensure that all cuboid cells will be computed from some subcuboid cells such that the minimum number of subcuboid cells are scanned.

We can see that two requirements must be met before a cuboid cell is saved and freed. First, the cuboid cell must have been fully materialized such that it has got the sums from all its corresponding subcuboid cells. For example, take the cuboid cell c at $V_c(0,0,3)$ as an example. Let cell b at $V_b(0,0,2)$ be one of its subcuboid cells. From the MMST, cell b is summed to cell c along dimension D_2, where its cardinality is $|D_2| = 3$ (as shown in figure 3.5). At this point, we know that cell b is already the last core cell to keep cell c in the memory because $v_{b_2} = 2 = |D_2| - 1$. The first requirement is thus as follows. Let the dimension along which the summation occurs be D_d. Check if the d-th coordinate of the subcuboid vector equals $|D_d| - 1$. If not, the cuboid cell is not yet materialized. The second requirement ensures that when a cuboid cell is fully materialized, it is not freed until it has been summed to all the other cuboids that are dependent upon it. For example, when the cuboid cell at $V(0,4,0)$ has been materialized, we cannot free it yet until it is summed to the cuboid cell at $V(0,4,3)$.

3.3.4 R-cubing without MNST

The MNST is very essential to the aggregation process because it coordinates when a cuboid is loaded into memory or when it is freed. Nevertheless, it is very expensive to construct the MNST data structure, especially when the number of dimensions is large. Eventually, to build such a data structure will consume lots of memory.

Fortunately, we do not have to use the MNST data structure at all to achieve the same aggregation result. By examining the 3 dimensional MNST in figure 3.4(b) carefully, we have observed some interesting facts as follows.

First, the number of dimensions involved in level i must be one dimension less than the number of dimensions involved in level $i-1$.

Secondly, when a groupby node is computed along dimension D_i, where $i < 2$, it is further needed to compute other cuboids along dimensions D_{i+1}, D_{i+2}, etc. For example, the node 12 is computed along dimension D_0 from the root node 012. It is needed further to compute the nodes 2 and 1 along dimensions D_1 (D_{0+1}) and D_2 (D_{0+2}) respectively. On the other hand, node 02 is computed along dimension D_1, so it computes other cuboids along dimension D_2 (D_{1+1}) only. Thus, it is needed for node 0 only. Therefore, given the dimension ID as the starting *index* along which this groupby node was computed, to compute other groupbys using this one will require dimensions ID from $index + 1, index + 2, \ldots, N - 1$, where N is the number of dimensions in the cube.

Finally, based on the second observation, for each level, a groupby node will not be further used for other groupby nodes if it is computed along dimension D_2. Therefore, any cuboid corresponding to this groupby node can be saved and freed when the two requirements are met.

Level	Index	Along Dim	From Node	To Node	From Cell	To Cell
0	-1	0	012	12	$V(0,0,0)$	$V(5,0,0)$
0	-1	1	012	02	$V(0,0,0)$	$V(0,4,0)$
0	-1	2	012	01	$V(0,0,0)$	$V(0,0,3)$
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	1	2	02	0	$V(0,4,0)$	$V(0,4,3)$
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	0	1	12	2	$V(5,0,0)$	$V(5,4,0)$
1	0	2	12	1	$V(5,0,0)$	$V(5,0,3)$
⋮	⋮	⋮	⋮	⋮	⋮	⋮
2	1	2	2	ALL	$V(5,4,0)$	$V(5,4,3)$
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 3.5: The computation of cuboid cells with respect to cell $V(0,0,0)$.

As a result, the good news is that we do not have to use the MNST data structure to aggregate efficiently. Each different groupby node can be recursively traversed from level 0 to the bottom level. Table 3.5 shows the relationship between the cuboids and their subcuboids with respect to the core cell $V(0,0,0)$ using `R-cubing`.

3.3.5 MMST vs. MNST

The MNST is simpler to construct than the MMST introduced in [ZDN97] mainly because of three reasons. First, for any node M at level i, there may be more than one node at level $i-1$ from which it can be computed. The MMST must choose the node that makes the node M require the minimum memory, but the MNST approach simply examines the candidates such that the one which can compute M by the smallest dimension will win. Secondly, with this simpler selecting method, we do not need to calculate the memory requirement needed for each node. Finally, the logic behind the MNST allows us to coordinate the overlapped computation of cuboids without the real MNST data structure.

In addition, with MNST, we will not consume more memory than the MMST. In fact, the memory requirement could be about the same, if not less. For example, the MMST for cube C in figure 3.5 (shown in figure 3.4(a)) tells us that the cuboid cell $V(5,4,3)$ is computed by the following subcuboids: $V(5,0,3)$, $V(5,1,3)$, $V(5,2,3)$ and $V(5,3,3)$ along dimension D_1. Our approach with MNST (shown in figure 3.4(b)) uses a different set of subcuboids: $V(5,4,0)$, $V(5,4,1)$, and $V(5,4,2)$ along dimension D_2.

The advantage of the MMST approach over the MNST approach is that the cuboid cell $V(5,4,3)$ can be materialized earlier. With the optimal dimension order, we know that once the core cell at $V(4,0,0)$ is read, the subcuboid cell $V(5,0,0)$ can be materialized, and it can compute the subcuboid cell $V(5,0,3)$ right away along dimension D_2. This goes on until the subcuboid cell $V(5,0,3)$ is materialized and according to the MMST, it can compute the cuboid cell at $V(5,4,3)$ right away.

With the MNST approach however, when the subcuboid cell $V(5,0,0)$ is materialized, it computes both subcuboid cells $V(5,4,0)$ and $V(5,0,3)$ along dimension D_1 and D_2 respectively. Note that the subcuboid cell $V(5,0,3)$ will be materialized much earlier than the subcuboid cell $V(5,4,0)$. With the MNST approach, the subcuboid cell $V(5,0,3)$ will not be used further once it is materialized, so it will be saved and freed. The cuboid cell $V(5,4,3)$ will not be materialized until $V(5,4,2)$ is. Nonetheless, the number of subcuboids required in memory for both approaches are more or less the same. Furthermore, the cuboid cell $V(5,4,3)$ will not be in memory until needed later and it will be in memory much shorter than the MMST approach. Most importantly, we use fewer number of subcuboids to compute it than the MMST approach.

3.3.6 When some cells are empty ...

`R-cubing` assumes the dimensions are ordered according to the optimal dimension order. It also assumes the core cells are scanned in the order described in the following. First, the core cells have to be lined up such that along any dimension D_d, the i-th value has to be read before the j-th value if $i < j$. That is, the cell at $V(\ldots, v_{D_{d_i}}, \ldots)$ has to be read before $V(\ldots, v_{D_{d_j}}, \ldots)$, for $0 \leq d < N$. Moreover, all cells in the cube, including the empty ones, must be traversed in order to ensure that all cuboid cells will be materialized.

For example, the cuboid cell $V_c(0,0,3)$ is materialized by feeding the core cells in the order of $V_b(0,0,0)$, $V_b(0,0,1)$ and $V_b(0,0,2)$ along dimension D_2. If this order is not maintained, say $V_b(0,0,2)$ is read before $V_b(0,0,1)$, the cuboid cell $V_c(0,0,3)$ would have been materialized prematurely. If however $V_b(0,0,2)$ is not read, the cuboid cell $V_c(0,0,3)$ would have been kept in the memory waiting to be materialized.

Consequently, even the empty core cell has to be scanned in the correct order such that all cuboids will be materialized appropriately.

3.3.7 Chunk-based Cubes vs. Cell-based Cubes

As stated in section 3.2.4, we have implemented a chunk-based compressed datacube, which adds a subcuboid chunk to another cuboid chunk instead of adding a cell to another. According to `R-cubing`, `chunk 0` is added to `chunk 12` (along D_0), `chunk 4` (along D_1) and `chunk 1` (along D_2) as shown in figure 3.6, which also shows how the cells in those chunks are numbered according to the optimal dimension order. We see that cells with offsets {0,9,18} of `chunk 0` ($LV(0,0,0)$, $LV(1,0,0)$, and $LV(2,0,0)$) are summed to cell 0 of `chunk 12` ($LV(0,0,0)$) along dimension D_0. Similarly, cells at offsets {0,3,6} of `chunk 0` ($LV(0,0,0)$, $LV(0,1,0)$, and $LV(0,2,0)$) are summed

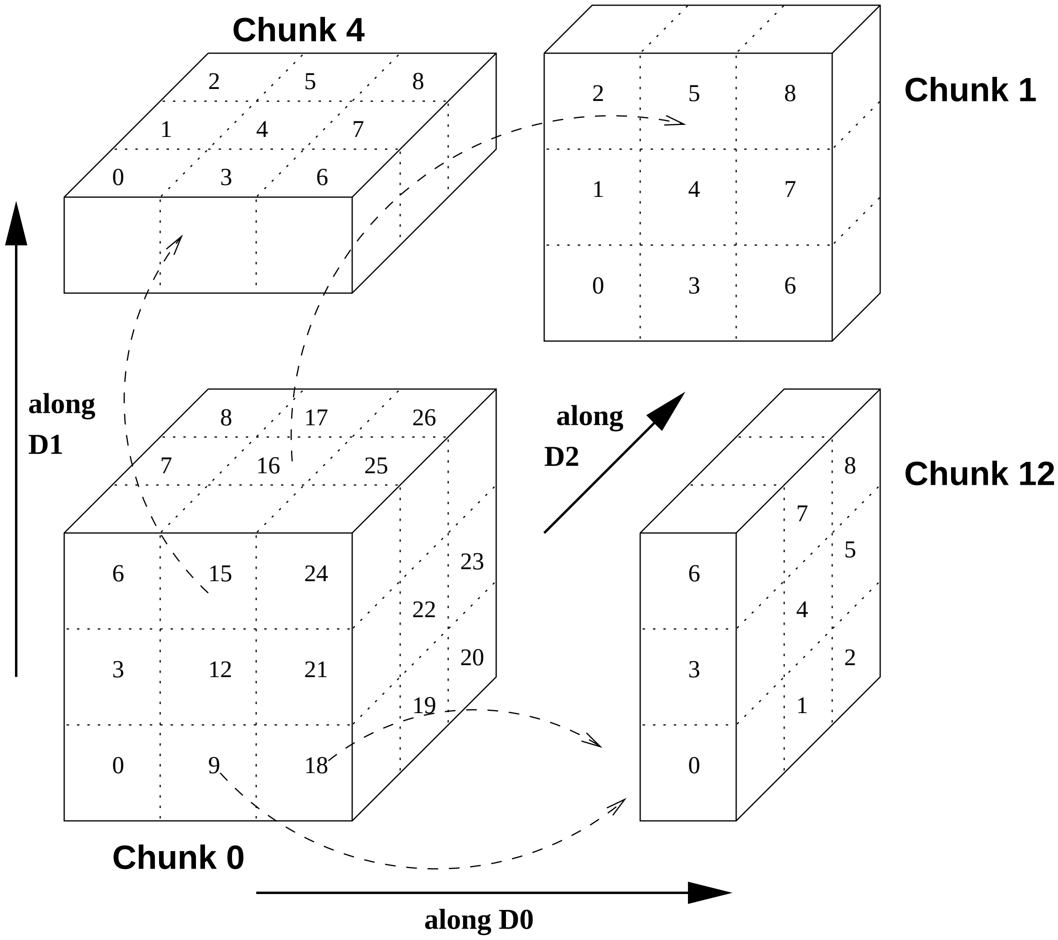


Figure 3.6: The Overview of adding Chunk 0 to its related cuboid chunks.

to cell 0 of chunk 4 along dimension D_1, and cells at offsets {15,16,17} of chunk 0 ($LV(1,2,0)$, $LV(1,2,1)$, and $LV(1,2,2)$) are summed to cell 5 of chunk 1 ($LV(1,2,0)$) along dimension D_2.

In general, given the local vector LV for a subcuboid chunk cell, and this subcuboid chunk is summed to another along dimension D_a, then we set $lv_a = 0$ while leave the rest of the address components intact. The process of summing a subcuboid chunk to another is like pressing that subcuboid along a dimension. Algorithm Press-Chunk in table 3.6 shows how a subcuboid chunk is pressed to its related cuboid chunks. As discussed in section 3.3.6, all core chunks (including the empty ones) have to be traversed in the right order such that all cuboid chunks will be materialized

```
PROCEDURE Press-Chunk(int numCuboids, Chunk aChunk, Chunk cuboid[],
                      int alongDim[]) {
    // INPUT:     number of cuboids, subcuboid chunk, cuboid chunks,
                  and along dimensions
    // OUTPUT:    none
    // N is the number of dimensions in the cube
    // ncells is the number of non-empty cells in the subcuboid
    for (i = 0; i < ncells; i++) {
        // cell[i] is the i-th non-empty cell in subcuboid aChunk
        offset = cell[i].GetOffset();        // get cell offset
        // from this offset get the local vector in subcuboid aChunk
        V = aChunk.Index-To-Vector(offset);
        U = V;                               // create the vector copy
        for (j = 0; j < numCuboids; j++) {
            u_alongDim[j] = 0;               // from vector U get the offset in cuboid
            c_offset = cuboid[j].Vector-To-Index(U);
            cuboid[j].AddCell(cell[i], c_offset);
            u_alongDim[j] = v_alongDim[j];   // reset
        } // end for
    } // end for
} // end PROCEDURE
```

Table 3.6: The Algorithm Press-Chunk.

appropriately.

Figure 3.7 shows the aggregation paths from the following core chunks: chunk 0, chunk 2, chunk 6, and chunk 8 to the rest of the cuboid chunks. Note that some chunk IDs are surrounded by rectangular boxes. For example, after chunk 0 is summed to chunk 1, according to `R-cubing`, this cuboid chunk can be saved and freed. At this point, chunk 0 has finished its job and will be saved and freed too. Thus, all chunks in similar situation will be surrounded by rectangular boxes.

Sometimes we might not be able to keep all the intermediate cuboid chunks in memory, and what we can do instead is to save the intermediate results on disk. Therefore, we might end up with several copies of the intermediate results of the same chunk. We need to read them back to memory and merge the results after the

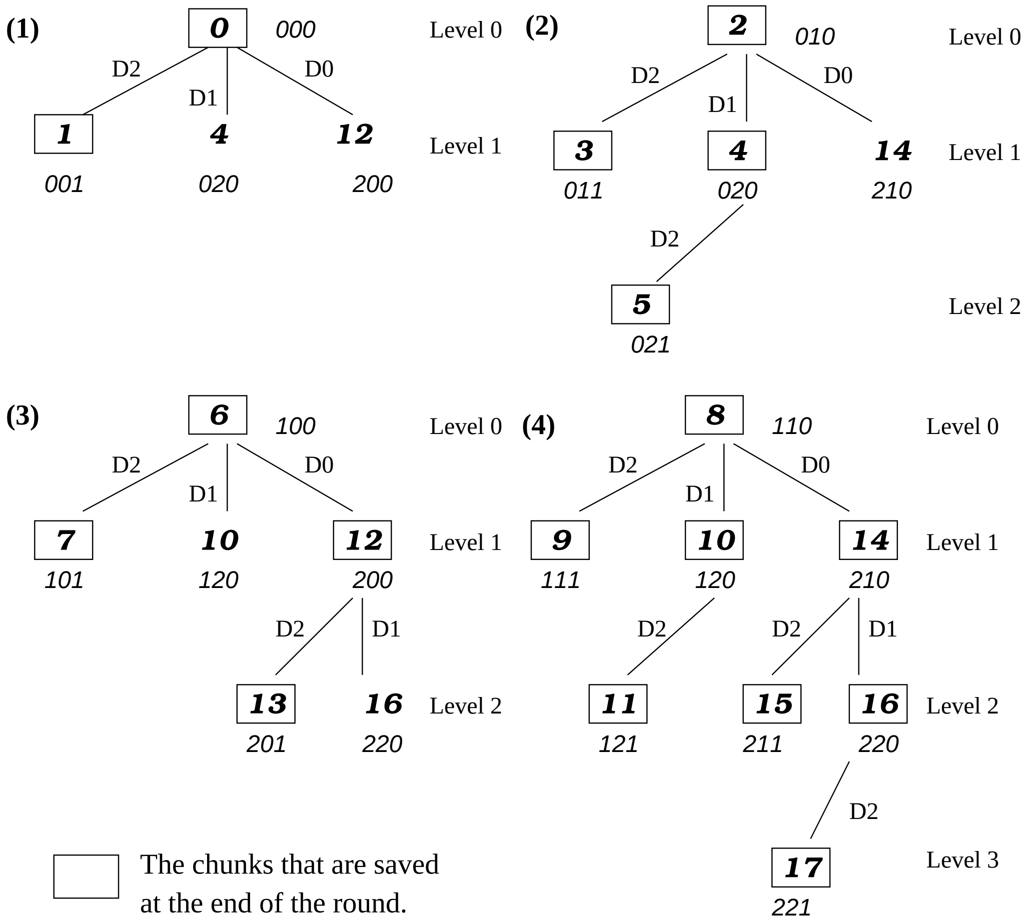


Figure 3.7: Another view of the aggregation paths from the core chunks to their related cuboid chunks.

aggregation process is over. However, with medium number of dimensions (we have tried up to 20), we do not need to use this alternative method at all.

3.4 Combination of both ROLAP and MOLAP approach

Though our datacube uses the MOLAP approach, it actually has some nice ROLAP features. We know that ROLAP has its advantages mainly because it uses the relational tables to store and manage the data, so the data is stored compactly and the scalability and transactional features of the relational database can be exploited.

Our chunk-based datacube is compressed such that empty chunks are not materialized and non-empty chunks are compressed. A datacube is stored as a relational table and its chunks are stored as tuples. For each chunk, the cells are mapped to a bunch of bits such that they can be stored as a field in the tuple. Hence, we do not need to create any customized indexing scheme to address any chunk in the cube. Since a chunk should be small enough to fit in the main memory, we always retrieve the cube one chunk at a time. Once a chunk is in memory, it has the advantages of the MOLAP approach. Therefore, our chunk-based compressed datacube has the nice features of both ROLAP and MOLAP, without having any redundancies.

Chapter 4

OLAP-based Data Mining

The datacube is constructed mainly to provide users the flexibility to view data from different perspectives as long as some dimensions of the cube contain multiple levels of abstraction. "The main goals for OLAP are fast access, fast calculations, ..., user-friendly interfaces, and flexible viewing, given the challenges of lots of data with many levels of detail and multiple factors in a decentralized multi-user environment." [Tho97]

OLAP should support ad hoc analytical queries, some of which may require computations performed on the fly. A typical ad hoc analysis example is querying how overall product profitability was in California last quarter. Seeing that profitability was lower than expected, the manager might navigate down into individual counties while still looking at overall product profitability. If some counties are significantly worse than the others, the manager might navigate further down into the product groups for these counties, always looking for some evidence that explains the higher-level anomalies. Each step in this train of thought constituted a query, and each query follows the result of the previous one.

4.1 Concept Hierarchies

One or more data columns in a table can be grouped to form schema hierarchies. Figure 4.1 shows two typical dimension hierarchies where the region hierarchy

(a) The REGION hierarchy

- REGION
  - Asia
    - Malaysia
    - China
    - India
    - Japan
    - Singapore
  - Europe
    - Belgium
    - France
    - Germany
    - Norway
    - Spain
    - United Kingdom
  - North America
    - Canada
    - Mexico
    - United States

(b) The CATEGORY hierarchy

- CATEGORY
  - Food Line
    - Baked Goods
    - Canned Food
    - Drink
    - Snack
  - Outdoor Line
    - Camping Equipment
    - Carry-Bags
    - Sport Wear
    - Sports Equipment
    - Sunblock

Figure 4.1: The Concept Hierarchies of (a) REGION and (b) CATEGORY

is formed by grouping the `continent`(level 0) and the `country`(level 1) columns and the `category` hierarchy is formed by grouping the `product line`(level 0) and `product type`(level 1) columns.

Previous works [Fis87, CC94, HF94] have described how to automatically generate concept hierarchies for numerical columns in the database table based on data semantics and/or data distribution statistics. Thus, a numerical column `revenue` in the database table can actually form a concept hierarchy as illustrated in the following example.

Given the `revenue` column, say there are not that many tuples that have revenue larger than \$4000. Then column `revenue` can be divided into five groups: (1)

0.00~1000.00, (2) 1000.00~2000.00, (3) 2000.00~3000.00, (4) 3000.00~4000.00, and (5) 4000.00+. These intervals can be further divided such that the first group contains the following subgroups: (a) 0.00~200.00, (b) 200.00~400.00, (c) 400.00~600.00, (d) 600.00~800.00, and (e) 800.00~1000.00 and so on.

The boundaries in the numerical hierarchies are defined as $[low, high)$ such that any tuples with revenue $0 \leq revenue < 200$ will belong to the first subgroup of the first group. For more details please refer to [Lu98].

4.2 OLAP Operations

To support OLAP, the datacube should provide the following capabilities: **roll-up** (increasing the level of abstraction), **drill-down** (decreasing the level of abstraction or increasing detail), **slice and dice** (selection and projection), and **pivot** (re-orienting the multidimensional view of data)

	Food Line	Outdoor Line	CATEGORY_total
Asia	59,728	151,174	210,902
Europe	97,580.5	213,304	310,884.5
North America	144,421.5	326,273	470,694.5
REGION_total	301,730	690,751	992,481

Table 4.1: A Crosstab view of Sales in different regions and product categories .

We will use table 4.1 to describe how our datacube supports the above operations. This table displays a crosstab of sales by dimension region against dimension category. The left most column of table 4.1 shows the values of level 1 of dimension region, which are {Asia, Europe, North America}. Also, the first row shows the values of level 1 of dimension category, which are {Food Line, Outdoor Line}. The rightmost column displays the totals and so does the bottommost row. For example, at row Asia and column CATEGORY_total, the crosstab entry shows the total of sales in Asia for all categories. Similarly, at column Food Line and row REGION_total, the crosstab entry shows the total of sales in Food Line for all regions. Therefore, at row REGION_total and column CATEGORY_total, the entry contains the total for all sales.

The totals shown in the last column and the last row can be obtained from the cuboid cells in the cube. If this cube has only two dimensions, the number of groupby combinations is $2^2 = 4$. One of them is the empty groupby (the core cells). The other two are shown in the last row and the last column. The cell that intersects both last row and last column corresponds to the empty groupby, which is composed of one cell only and contains the total of the measurements kept in all the cube cells.

4.2.1 Drill-down and Roll-up

	Food Line	Outdoor Line	CATEGORY_total
Malaysia	618	9,418	10,036
China	33,198.5	74,165	107,363.5
India	6,918	0	6,918
Japan	13,871.5	34,965	48,836.5
Singapore	5,122	32,626	37,748
Belgium	7797.5	21,125	28,922.5
France	16,809.5	46,899	63,708.5
Germany	17,603	39,365	56,968
Norway	11,440	35,980	47,420
Spain	18,989	30,454	49,443
United Kingdom	24941.5	39,481	64,422.5
Canada	29,116.5	69,310	98,426.5
Mexico	12,743.5	24,284	37,027.5
United States	102,561.5	232,679	335,240.5
REGION_total	301,730	690,751	992,481

Table 4.2: Drill down on dimension region.

Table 4.2 shows the resulting crosstab when we drill down into dimension region. The continents are now broken down into countries. Note that the last row is exactly the same as the last row in table 4.1 because it shows the total of sales of Food Line, Outdoor Line and all categories regardless of what region it is. Thus, the drill down operation has not affected this row of totals at all.

In table 4.1, North America and Food Line have sales 144,421.5. This number is actually the sum of Canada, Mexico, and United States against Food Line, which is

29,116.5 + 12,743.5 + 102,561.5 = 144,421.5. Clearly it is impossible to know the exact break-down on sales from the higher level data. Based on the method *Attribute-Oriented Induction*, introduced by Cai, et al [CCH91] and Han, et al [HCC93], we build our datacube as follows.

The datacube is actually constructed with the values of the leaf levels of all the involved dimensions. If the cube has only dimensions region (number of distinct values in the leaf level is 14) and category (number of distinct values in the leaf level is 9), this cube will have $(14+1) \times (9+1) = 150$ cells (remember we have to add one more cell to each dimension for the cuboid cells) in total and will be displayed like table 4.2 except that dimension category will have been drilled down to the leaf level values as well. We call this cube the **base cube**.

Then for the OLAP output, we roll up both dimensions in the base cube to the levels where there are not many distinct values. For a user, s/he would normally like to see no more than 7 distinct values in the beginning. Of course, this threshold can be changed by the user. Let us set it at 7, so level 0 for both dimensions are shown (as in table 4.1).

To roll up from the base cube is very straightforward. We traverse each non-empty cell in the cube and from its position in the multidimensional space we determine its corresponding position in the higher level of the multidimensional space. For example, if we are at the cell China and Camping Equipment, then according to the dimension hierarchies in figure 4.1, we know that the cell now belongs to Asia and Outdoor Line. We will construct a **working cube** with dimensions of sizes 3 and 2 respectively. Then the measurements stored in this cell will be summed to the corresponding cell in the working cube. Again, each dimension will have one extra cell for the aggregates such that this working cube will have $(3+1) \times (2+1) = 12$ cells in total. Instead of applying `R-cubing` to compute the aggregates, we simply map the cuboid cells in the base cube to those in the working cube. For example, the cells denoted by Malaysia/CATEGORY_total, China/CATEGORY_total, etc. will be mapped to Asia/CATEGORY_total. Note that the cell that corresponds to the empty groupby, REGION_total/CATEGORY_total, will still be mapped to that cell in the working cube.

Therefore, when the user decides to drill down into region, we actually roll up from the base cube to create a working cube such that only the values of dimension category are mapped to one level higher. Therefore, the result is shown in table 4.2.

When the user decides to roll up from table 4.2 from region, instead of rolling up from the base cube, we can roll up from the current working cube and create yet another working cube. Once the new working cube is created, the old one could be destroyed. Alternatively, this cube can be saved as one of the steps in the user OLAP history.

4.2.2 Slice and Dice

	Food Line	Outdoor Line	CATEGORY_total
Canada	29,116.5	69,310	98,426.5
Mexico	12,743.5	24,284	37,027.5
United States	102,561.5	232,679	335,240.5
North America	144,421.5	326,273	470,694.5

Table 4.3: Slice on value North America of dimension region.

Slicing into one dimension is very much like drilling one level down into that dimension. Table 4.3 shows the result of slicing into the value North America of dimension region in table 4.1. Notice that the last row of table 4.3 is now exactly the same as the row denoted by North America in table 4.1. In other words, table 4.3 shows the details of North America only while keeping the values of dimension category the same. Therefore, the three core rows in table 4.3 are exactly the same as the last three core rows in table 4.2.

To implement this slice operation, we actually drill down one level into dimension region to create a new working cube. Then we check dimension region for the direct descendants of North America, which are Canada, Mexico and United States. Therefore, the working cube is essentially the same as the one after a one-level drill down operation, but we limit the number of entries displayed on the crosstab according to the slice command.

A dice operation is like a slice on more than one dimension. On a crosstab display, dicing means slicing on the row and column dimensions. Table 4.4 shows the result of

	Camping Equipment	Carry-Bags	Sport Wear	Sports Equipment	Sunblock	Outdoor Line
Canada	39,056	15,008	2,512	7,282	5,452	69,310
Mexico	15,788	5,224	640	1,392	1,240	24,284
United States	168,238	27,044	5,904	16,752	14,741	232,679
North America	223,082	47,276	9,056	25,426	21,433	326,273

Table 4.4: Dice on value North America of dimension region and value Outdoor Line of dimension category.

dicing into North America of dimension region and Outdoor Line of dimension category of table 4.1. In table 4.1, the entry denoted by the values North America and Outdoor Line has revenue 326,273, which is exactly the total shown in table 4.4. Also, the last column in table 4.4 is exactly the same as the second column of table 4.3. Therefore, to achieve a dice operation, one could always slice into North America of table 4.1 first to obtain table 4.3, and then slice again into Outdoor Line of table 4.3.

To implement the dice operation, we drill down on both dimensions region and category to create a new working cube. Notice that this will be the same as the base cube since we have reached the leaf levels of both dimensions. Then again we find the direct descendants of both North America and Outdoor Line in the dimension hierarchies and limit the crosstab display accordingly.

4.2.3 Pivoting

	Canada	Mexico	United States	North America
Food Line	29,116.5	12,743.5	102,561.5	144,421.5
Outdoor Line	69,310	24,284	232,679	326,273
CATEGORY_total	98,426.5	37,027.5	335,240.5	470,694.5

Table 4.5: Pivoting operation.

Table 4.5 shows the result of pivoting of table 4.3. Essentially the row and column dimensions are switched. The cells in the working cube are re-mapped to the correct crosstab entries.

4.2.4 Display Raw Data

We support the option of displaying the original tuples of the source data table from which the cube was constructed. When the user keeps slicing or dicing, we might reach the point where both dimensions are at the leaf levels, like table 4.4. At that point, the user might find that United States are doing particularly well with Camping Equipment. If the user wants to dice further into this crosstab entry, we will submit the appropriate query to the database system where the source table resides to fetch the right set of tuples for display.

4.2.5 More than Two Dimensions

When there are more than two dimensions, we can still use the base cube and the working cube to display the crosstab output. Let the cube have one more dimension, revenue, and only dimensions region and category be selected. The result is still the same as shown in table 4.1 because of the followings. First, using the preset threshold for the number of distinct values for each dimension, we create the first working cube. From this working cube, we examine only those cuboid cells whose address components corresponding to dimension revenue are ALL.

For example, if we treat the cube in figure 3.3 as if it was the working cube and dimension D_0 was dimension revenue, then only the cells in chunks 12, 13, 14, 15, 16, 17 will be examined. In chunks 12 and 14 we have "non-cuboid (core)" cells because neither D_1 nor D_2 has the value ALL. The measurements in those cells will be displayed as the core entries of the crosstab. The cells in chunks 13, 15, 16 and 17 will be displayed in the crosstab as the row and column totals.

4.3 OLAP + Data Mining → On-Line Analytical Mining

Motivated by the popularity of the OLAP technology, which facilitates powerful multidimensional analysis for data warehouses, we have developed an **On-Line Analytical Mining** (OLAM) mechanism for multidimensional data mining in large databases and data warehouses.

4.3.1 Motivation

Data mining tools often require the raw data to be first integrated and cleaned such that it is consistent for mining. This requires costly preprocessing steps such as data cleaning, data transformation, and data integration [FPSSU96]. Since a data warehouse normally goes through these preprocessing steps for OLAP operations, it serves as a valuable data source for data mining.

Secondly, OLAP provides such facilities as drilling, pivoting, filtering, dicing and slicing such that the user can traverse the data flexibly, define the set of relevant data, analyze data at different granularities, and visualize the results in different forms. These operations can also be applied to data mining [HCC98] to make it an exploratory and effective process.

Finally, data mining is mainly "hypothesis-driven" in the sense that user usually does not predict what kinds of knowledge to be mined beforehand. Together with OLAP, data mining functions can provide an overview of the discovered knowledge such that the user can investigate further on any interesting patterns or anomalies. "Sometimes one may even like to trade mining accuracy for fast response since interactive mining may progressively lead miners to focus the search space and find more and more important patterns. Once the user can identify a small search space, more sophisticated but slower mining algorithms can be called up for careful examination." [HCC98] OLAP mining does insure faster response than mining in the raw data directly because with OLAP operations, the size of the data set is relatively more compact. Moreover, during the OLAP mining process, the user can choose to see the raw data at any moment, so the mining can be more interactive and allow more user involvement.

4.3.2 Architecture

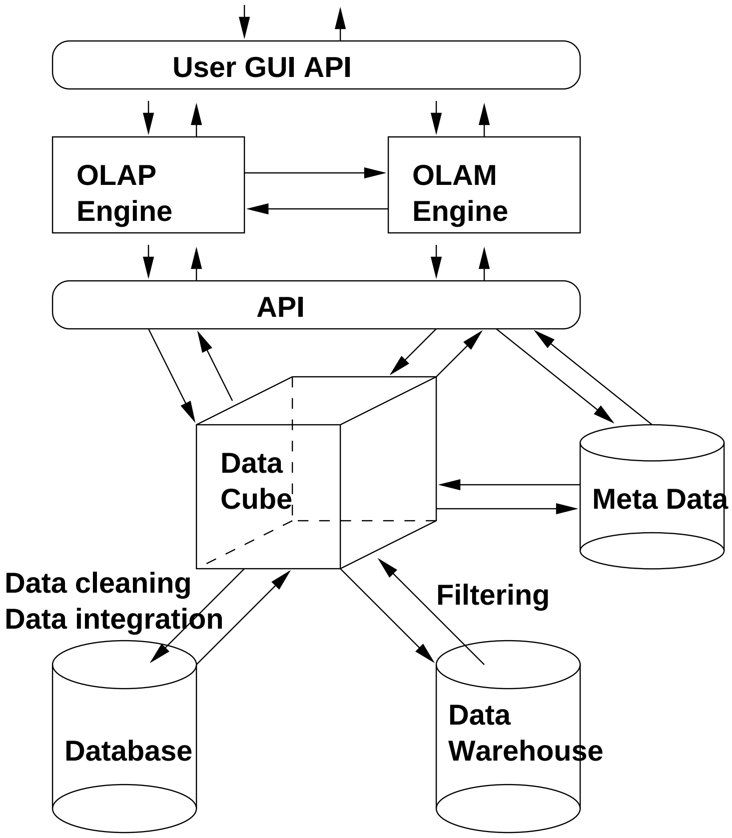


Figure 4.2: An integrated OLAM and OLAP architecture

An OLAM engine performs analytical mining in datacubes in a similar manner as an OLAP engine performs on-line analytical processing. Therefore, we can combine both of them as shown in figure 4.2, where the OLAM and OLAP engines both accept user's on-line queries and work with the datacube in the analysis. Moreover, an OLAM engine can consist of multiple data mining modules, such as concept description, association, classification, prediction, clustering, time-series analysis, etc., which may interact with each other for effective mining. Thus, an OLAM engine can be more sophisticated than an OLAP engine.

We have constructed the datacube engine and on top of it we build the mining modules. There is essentially no difference between the datacube required for OLAP and that for OLAM except that OLAM analysis often involves a larger number of dimensions with finer granularities than what an OLAP analysis needs. Therefore, OLAM analysis requires powerful datacube construction and accessing tools. Even so, the mining modules should be able to work with any OLAP products available on the market as long as the appropriate APIs exist.

4.3.3 Implementation

The current OLAP mining system DBMiner has been developed based on the previous works [HFH+94, HF95, FH95, HF96, HFW+96, HCC+97]. The system supports mining of association rules [KHC97], classification rules [KWG+97], and prediction [Che98] at multiple concept levels.

DBMiner evolves from a system that used attribute-oriented induction [HCC93] technique to generalize attribute concepts to their corresponding more abstract level concepts. Such generalization led to a new and compressed generalized relation. This is actually very similar to the relational OLAP (ROLAP) implementation of the roll-up operation.

For faster response in OLAP and data mining, our later implementation has adopted the MOLAP approach using the datacube technology. In fact, generalizing from a base cube to a working cube costs less than generalizing from a base relation to a working relation because a relation requires quite a bit of searching and elimination of duplicate tuples for a roll-up operation.

With DBMiner, the user has the freedom to define a datacube from a database table or view by indicating the number of dimensions and specifying their corresponding hierarchies. Then the base cube will be constructed and stored on the disk. Since each chunk (core or cuboid) is stored on the disk as a single unit, we can easily identify which of these chunks will be needed when the user only wants a subset of the stored datacube with the same idea as stated in section 4.2.5. For this particular mining session, a temporary base cube will be formed based on the chunks loaded. The initial working cube will then be created based on this temporary base cube and the preset threshold.

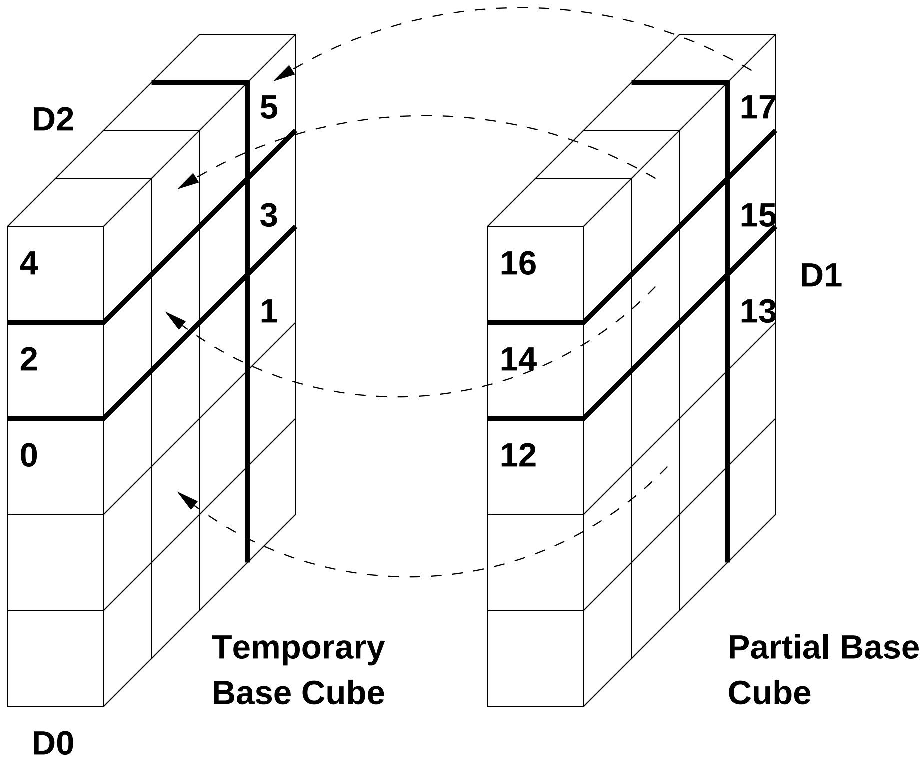


Figure 4.3: Mapping of chunks from Partial Base to Temp Base

The temporary base cube will not duplicate the base cube chunks stored on the disk. In fact, this temporary cube only need to set up the chunk mappings as shown in figure 4.3. Let us use cube C in figure 3.3 again to illustrate how to form these mappings. Say the user wants two dimensions D_1 and D_2 only. According to our convention, we want all cells with the address vector like ($\texttt{ALL}, -, -$). That is, only chunks 12, 13, 14, 15, 16, 17 in figure 3.3 are needed. The temporary base cube will be set up as if it had these chunks only and the meta-data of these chunks will be copied. Of course the temporary base cube will reference these chunks using a different set of chunk IDs as shown in figure 4.3.

Different mining modules can be performed on the initial working cube. When the user is interested in a particular dimension, the user can either drill down into or roll up from that dimension for more or less details. In addition, the user can use different mining modules to discover different patterns. Let us now examine datacube-based association analysis module in detail.

Datacube-based Association Analysis

Association analysis is an important data mining function. There have been many studies on mining association rules in transaction databases [AS94, SA95, HF95]. Typical association rules look like the following:

$$P(x) \wedge Q(x) \Rightarrow R(x)[support\%][confidence\%],$$

where

$$support = \frac{count(P, Q, R)}{count(all)},$$

and

$$confidence = \frac{count(P, Q, R)}{count(P, Q)}.$$

The datacube offers additional flexibility and efficiency in association rule mining. There are two kinds of association rules: **inter-dimension association** and **intra-dimension association**. Let us use the "grading" table from a university database to distinguish between these two types of associations. Suppose it contains four attributes as shown below.

$$grading = \langle student, course, semester, grade \rangle.$$

Inter-dimension association is the association among a set of *distinct* dimensions of a data cube. For example, the association between student and grade, such as "the students in Fall semester tend to have good grades".

Intra-dimension association is the association with regard to one or a set of dimensions into a transaction like set. For example, the associations between each course and the overall course performance is an intra-dimension association. Let us take course as the reference dimension and the course_id as the reference level, the remaining set of dimensions, {student, semester, grade}, are grouped into a transaction-like set. A typical association rule is "*a course offered in the fall semester is likely to be offered again in the spring semester.*".

The datacube serves as an efficient data structure for mining both kinds of associations. First, it is easy to group data according to one or a set of dimensions using the cube structure. Second, *count* and other aggregate measurement have been pre-computed in the cube which facilitates the association testing and filtering. Furthermore, multi-level association can be mined by drilling down or rolling up along any dimension in the cube.

Let us examine mining multi-level, inter-dimension association rule further. With the cube structure, the *support* and *confidence* measures of association rules can be easily computed based on the measurements carried in each cell of the working cube. For example, to test whether "senior students in Fall semester tend to have good grades" has both support and confidence greater than the user defined thresholds, we check the *count* field of the following cells: the cell at $V_1(senior, \texttt{ALL}, FallSem, Goodgrade)$, the cell at $V_2(\texttt{ALL}, \texttt{ALL}, \texttt{ALL}, \texttt{ALL})$, and the cell at $V_3(senior, \texttt{ALL}, FallSem, \texttt{ALL})$. The support is therefore $\frac{count(V_1)}{count(V_2)}$ and the confidence is $\frac{count(V_1)}{count(V_3)}$. If both the support and confidence are greater than the corresponding thresholds, then this is a strong association rule.

However, it is a bit trickier to mine intra-dimension association rule. The support and confidence cannot use the *count* field in the cube cell because it is in a completely different context. Once we are looking for this kind of transaction-based association rules, the count is the number of "transactions" that the corresponding "items" are purchased together. To illustrate, let us look at how we came up with the rule "*a course offered in the fall semester is likely to be offered again in the spring semester.*" by using table 4.6, which is a crosstab view of the cube where students and grades are both set with the value ALL.

Since course is the reference dimension and course_id as the reference level(transactions), the semesters Spring, Summer, Fall are then treated as items. Therefore, the *count* field

	Spring	Summer	Fall	SEMESTER_total
Course_A	123	155	134	412
Course_B	200	0	298	498
⋮	⋮	⋮	⋮	⋮
Course_K	0	0	214	214
⋮	⋮	⋮	⋮	⋮
COURSE_total	...	...	...	...

Table 4.6: A Crosstab View of the Cube: Course vs. Semester.

in the cube cells are no longer relevant in calculating support or confidence. What we want to know is whether the *count* is 0 or not. If $count = 0$, then the corresponding "item" is not purchased in that transaction. For example, for Course_B transaction, item Summer is not in. Therefore, to calculate the support for an item, we count the number of transactions that carry this item. Let us now look at Spring item. From table 4.6, we know the support is at least 2. For Fall item, the support is at least 3. If the support threshold is 2, then both these items are large.

Therefore, to check whether "*a course offered in the fall semester is likely to be offered again in the spring semester.*" can be a strong intra-dimensions association rule, we checked the support for both Fall and Spring items. From table 4.6, we know the support for both is at least 2. Divide this support by the total number of transactions give us the support for this rule. If it exceeds the support threshold, then we compute the confidence as $\frac{support(\mathsf{Spring} \wedge \mathsf{Fall})}{support(\mathsf{Fall})}$. In this case, if $support(\mathsf{Spring} \wedge \mathsf{Fall})$ is 2 and $support(\mathsf{Fall})$ is 3, then the confidence is $\frac{2}{3}$.

Again, the efficiency in cell count retrieval of the datacube ensures fast computation of mining of intra-dimension association rules.

4.3.4 Contribution

Let us use OLAP-based association rule mining, in particular we choose inter-dimension association, as our example to illustrate why OLAP mining is helpful. As stated above, OLAP mining will first generalize the cube such that the number of distinct values of each dimension will be less than a certain number adjustable by the user. That is, the user should be able to handle these distinct values comfortably.

In this case, with smaller number of distinct values of each dimension, the support of each distinct value will be relatively high. Therefore, the user will obtain association rules in high conceptual level. These rules are sometimes interesting enough. If the user is not satisfied, the user can drill down to obtain rules in more details. At this point, the user is already more focused before s/he starts.

On the other hand, the user could find the results too detailed and decide to roll up instead of drilling down. In this case, we will have even fewer number of distinct values and each "item" will then have even higher support count. That is, some items which did not have large support before might now be large enough (because several of them might have been grouped to a single item). Thus, some new interesting association rules might be shown.

Obviously, direct mining in the raw data do not provide this flexibility or power to the user. This phenomenon can be applied to other data mining modules too.

Chapter 5

Performance Analysis and Discussions

We will analyze the performance of our approach by comparing it with our implementation of a previously published method and comparing DBMiner with an existing OLAP product. We will also discuss the strengths and limitations of our approach after the performance analysis.

5.1 Performance Results and Analysis

In this section, we will present the performance results of comparing our implementation of a previously published MOLAP algorithm — `Multi-way Array Cubing` [ZDN97] with our MOLAP algorithm — `R-cubing`. Then we will compare the time taken for cube construction in DBMiner and that of another existing product. All experiments were run on a Pentium 200 MHz PC running Windows NT v4.0. The PC has a 128 MB memory and a 2 GB local disk.

5.1.1 Multi-way Array cubing compared with R-cubing

We used two synthetic data sets to study the algorithms' performance. We are particularly interested in how the increase of the number of dimensions affects `R-cubing` and `Multi-way Array cubing` [ZDN97]. Since both algorithms are exponential with respect to the number of dimensions, we only investigated their difference in time to compute all the aggregations. The limit to the chunks is set to be a bit less than two giga cells (2,147,483,600), yet most chunks will not have this size because they will likely be sparse.

The first data set contains 1000 tuples and we have constructed cubes from 3 dimensions to 10 dimensions. Notice that the density of non-empty cells in the multidimensional space varies from cubes to cubes. The dimension sizes range from 2 to 82, but most of them have more than 40 distinct values. The second data set contains only 60 tuples. Most dimensions are of sizes 1, 2 or 3, and some dimensions have more values (from 7 to 45). From this one we have built cubes of 4, 6, 8, 10, 12, 14, and 15 dimensions. Notice that when the number of dimensions increase, the sparsity of the cubes also increases at a rapid rate.

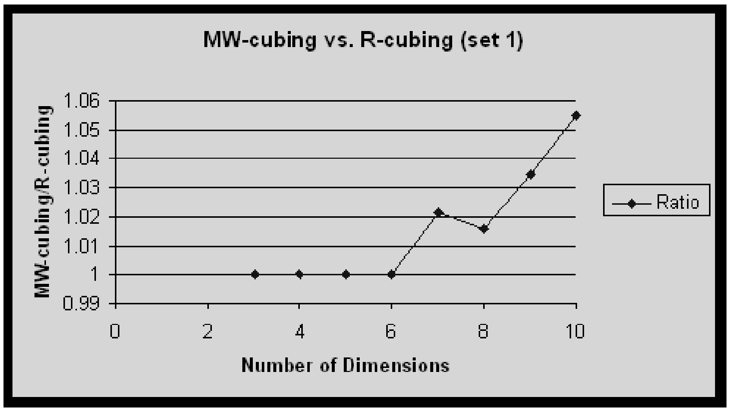


Figure 5.1: Data set 1(1000 tuples).

In figure 5.1, we show the ratio of time (in seconds) taken to compute all aggregations of `Multi-way Array cubing` to that of `R-cubing`. We can see that when the number of dimensions is small (from 3 to 5), their performance are the same. However, when the number of dimensions increases, their difference starts to increase. The main difference is that `Multi-way Array cubing` needs to construct the costly Minimum Memory Spanning Tree data structure. It not only takes time to build but also occupies part of the memory to coordinate the aggregation computation. With `R-cubing`, this data structure is completely eliminated and more memory can be used for aggregation computation.

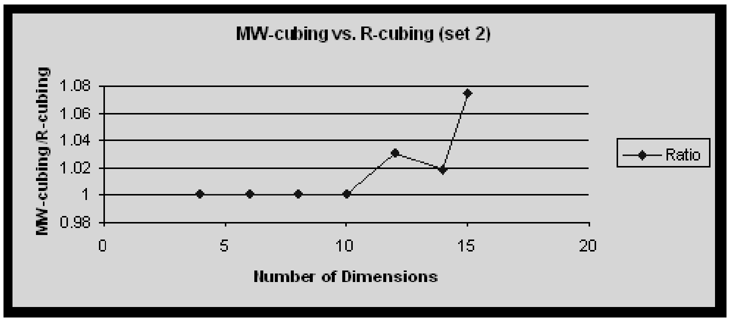


Figure 5.2: Data set 2(60 tuples).

Figure 5.2 again shows the ratio of time taken to compute all aggregations of `Multi-way Array cubing` to that of `R-cubing`. The results are similar to that of the first data set.

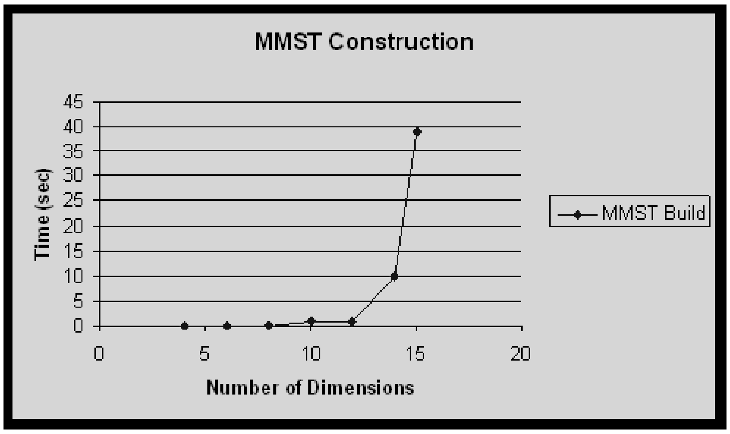


Figure 5.3: Construction Time of MMST with respect to Number of Dimensions.

Figure 5.3 shows the time taken to build the MMST data structure when building cubes from 4 dimensions to 15 dimensions from the second data set. When the number of dimensions is small, the time taken is trivial. However, when the number of dimensions increases, the time taken grows rapidly. Sooner or later, the memory will not be enough to hold such a data structure.

5.1.2 DBMiner compared with another OLAP Product

We used two synthetic data sets to compare DBMiner with Cognos PowerPlay[1]. The first data set contains 65,536 tuples and we have constructed cubes of 7, 10, 14, and 21 dimensions. The results are shown in figure 5.4. Another set of data consists of approximately 240,000 rows, and we have constructed cubes from 2 to 6 dimensions as shown in figure 5.5.

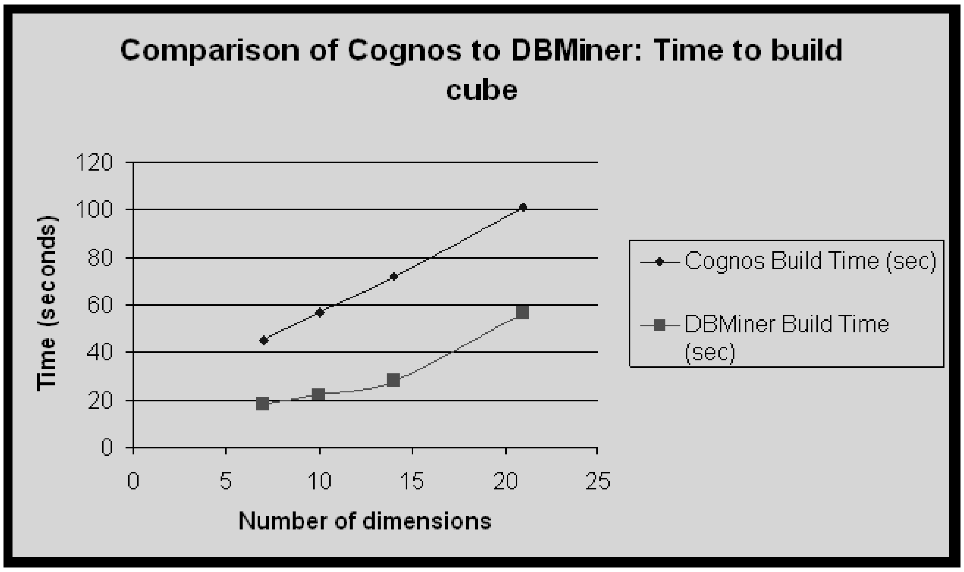


Figure 5.4: Comparison of Cognos to DBMiner (65,536 tuples).

The difference between two data sets are as follows. The first data set consists of more replicated data such that the number of distinct values for each dimension is quite small even though the number of tuples is about 65k. The second data set was extracted from real customer support data, and each dimension has a lot of distinct

[1] http://www.cognos.com/busintell/products/powerplay_overview.html

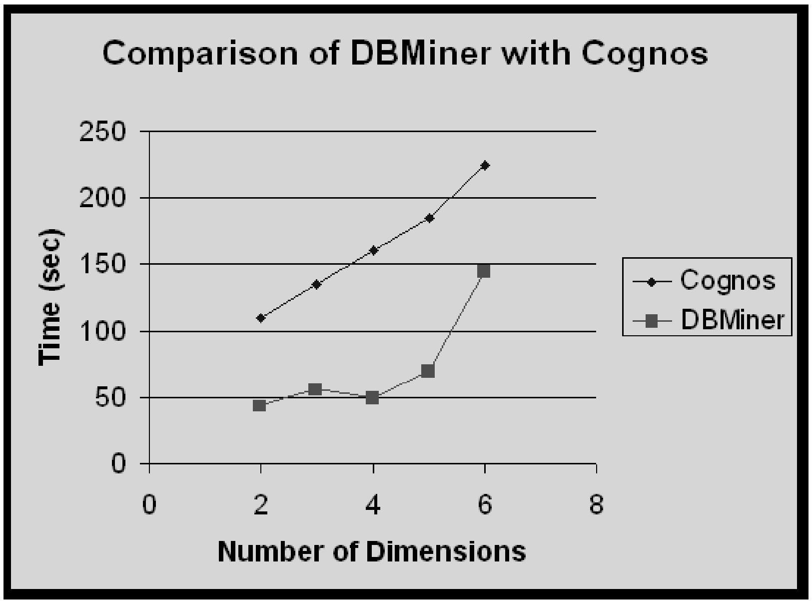


Figure 5.5: Comparison of Cognos to DBMiner (240,000 tuples).

values. Obviously, DBMiner performs better than Cognos PowerPlay for both data sets.

5.1.3 Other Testing Data Sets

We have tested DBMiner on some real data sets such as CITYDATA[2], NSERC95[3], and some other data sets provided by some companies. Also, we currently support an active customer, who has been able to get interesting OLAP mining results using DBMiner.

5.2 Discussion

We are going to discuss the strengths and limitations of our datacube and how we can improve it even further.

[2]purchased from US Department of Commerce
[3]got from SFU University Research Office

5.2.1 Cube Chunking

Currently, we divide the cube into chunks by simply setting the limit to the chunk sizes. During the design, we have thought of chunking the cube according to the concept hierarchies for the dimensions. That is, the dimension values in the same segment should belong to the same parent concept in the dimension hierarchy. However, this approach is not only cumbersome, but also not as useful as it first seemed. Therefore, we finally dropped this idea.

5.2.2 Modification of Concept Hierarchies

We have talked about how the cubes are constructed and saved as base cubes on the disk. When the user requests to change the concept hierarchies of any dimension of those stored cubes, we do not have to reconstruct the base cubes. If the user simply rearranges the higher level concepts in the dimension hierarchy while keeping the concepts in the leaf level intact, then the base cube does not need to be reconstructed. On the other hand, if the concepts in the leaf level are deleted or changed, then the stored base cube has to be discarded and reconstructed from scratch.

5.2.3 Sparse Data Problem

Our datacube works very well with small to medium sized datacubes. From the performance analysis above, we know that it computes the aggregations, which often is the bottleneck of cube construction, faster than an existing popular product in the market. However, when the data sets grow significantly bigger, our cube will start to have address space problems.

We have designed our cube to be chunk-based such that it has both global and local multidimensional space. Both address space should hold up to 2^{31} distinct address values, so combined together we have $2^{31} \times 2^{31}$ address space. Even though this seems large, with extremely sparse data, the number of cube cells could exceed this limit. A relation is sparse with respect to a set of dimensions if its cardinality is a small fraction of the size of the cross-product of the dimension domains [RS97].

An alternative is to change the way we address the cube cells. For example, we do not need to use long integers. We can probably investigate other approaches to index the cube cells. Say we can use more than one long integer instead of just a single one for a coordinate space. To make things more complicated, we can keep chunking the cube so that we have more than two multidimensional space. Nonetheless, this approach will eventually fail on some extremely sparse data relations.

Currently, our approach has already combined some nice features of both ROLAP and MOLAP. We can probably modify it to be more of a ROLAP approach than a MOLAP approach (which might penalize our performance with small to medium sized data sets). We call this approach the **Multidimensional Database Analysis** (MDDB). We set up a metadata structure to "cube" data for handling. The data will be kept in its raw level instead of being generalized to the leaf levels of its corresponding dimension hierarchies. Its dimensions and the dimension values are mapped to dimension IDs and value IDs and so on. Therefore, the dimension IDs will not be mapped to the array position indexes, and the cells are represented as tuples. The measurement values will be stored together with the dimension encoded IDs.

In addition, instead of materializing all cuboid cells for a cube (if the cube is huge, then the extra storage needed could be as big as the cube itself, if not bigger), we probably do a subset of them. It is not easy to select which ones to be materialized, but we may involve user's input in this case to let the user decide. Alternatively, we may simply choose not to materialize them. When the user is doing OLAP or OLAP mining, it is unlikely that the number of dimensions interested will be large. We can construct a temporary cube with respect to these dimensions and compute all its aggregates on-the-fly. Our MOLAP cube works the best with small to medium sized data, so there are probably not much difference between loading a subset of a huge cube from disk and building a small temporary cube in memory. Therefore, with MDDB, we again combine the power of ROLAP and MOLAP together.

5.2.4 Discovery-Driven Exploration

A business analyst who explores the OLAP data often looks for regions of anomalies, which may lead to identification of problem areas or new opportunities. However, this kind of "hypothesis-driven" exploration for anomalies has a few shortcomings. First, simply looking at cube data aggregated at various levels of details to hunt down an anomaly is a tedious task. Moreover, higher level aggregations may not be affected by an anomaly occurring underneath because of the large amount of data aggregated. Finally, it might be hard for an analyst to spot the exception even if the anomaly does occur because there are many values. Sarawagi, et al [SAM98] proposed a new "discovery-driven" method of data exploration where an analyst's search for anomalies is guided by precomputed indicators of exceptions at various levels of detail in the cube. They claimed that this approach increases the chances of user noticing abnormal patterns in the data at any level of aggregation. They considered a value in a datacube cell to be an exception if it is significantly different from the value anticipated based on a statistical model.

We can extend our datacube such that it is able to compute the anticipated value of a cell efficiently in context of its position in the cube and the trends along different dimensions that the cell belongs to using some statistical model.

5.2.5 Multi-Feature Cubes

We can also extend our datacube to support complex aggregation at multiple granularities. That is, it can support a decision support query in which multiple dependent aggregations are computed within each group. An example is "For each item, find its minimum price in 1997, and the total sales among all minimum price tuples". This kind of queries are called **Multi-Feature Cubes** [RSC98]. Such queries are quite practical for decision support systems, but may require sophisticated change to the current datacube technology.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The high demand of sophisticated decision support tools have made data warehousing systems popular and successful. Data warehouses are built based on the multidimensional data model (datacube), which support a number of typical datacube operations: roll-up, drill-down, slice and dice, and pivot that are known as On-Line Analytic Processing (OLAP). Another distinctive feature of the multidimensional data model is its stress on precomputation of aggregation of numerical measures by one or more dimensions in order to speed up query processing time.

There are two popular models for the OLAP servers for data warehouses: Relational OLAP (ROLAP) and Multidimensional OLAP (MOLAP). The former manages and stores the data in the relational databases and support extensions to SQL and special access to efficiently implement the multidimensional data model and operations. It is known to be scalable but slow with small data sets. The latter one directly manages and stores the data over some special data structures (arrays). It is known to have sparse matrix problem but has excellent performance with small to medium sized data.

Knowledge Discovery (KDD) refers to the overall process of discovering useful knowledge from data while data mining refers to the application of algorithms for extracting patterns from data. There are a number of interesting and important applications of KDD in the business world, the medical world, the modern science, etc. There are a wide variety of data mining algorithms, and the important point is that each technique suits some problems better than others. Thus, there is no 'universal' data mining method.

OLAP-based Mining or OLAM, which combines the OLAP technology together with data mining, has more advantages when compared with mining directly from raw data. There is not much need for data preprocessing because this is done for the data warehouse already. Mining with OLAP capabilities let users traverse data at different granularities and visualize the results in different forms. Decision makers often do not know exactly what they are looking for. The results obtained from mining in the raw data often overwhelm the users with too much information. With OLAP mining however, the results are more or less shown in an overview where the users can focus or investigate further on any unusual patterns or anomalies. Finally, OLAP mining is more interactive and friendly because it has faster response than plain mining.

We have designed a chunk-based compressed multidimensional datacube integrated with some nice features of the ROLAP approach. We have also designed an efficient disk-based algorithm for the computation of aggregation (which often is the bottleneck of datacube construction) — `R-cubing`, which shows to have good performance. In addition, we have been motivated to develop an OLAP mining system, DBMiner, by integration of the datacube computation, relational database technology, and data mining technology with high performance.

Yet there are still considerable amount of work that can still be done to the current datacube technology. We will discuss this further in the next section.

6.2 Future Work

6.2.1 Scalability problems

The number of groupbys grows exponentially with the number of dimensions. Therefore, when the number of dimensions is more than a certain number (say 50), it is infeasible to handle that data with our current technology. As discussed before, we can set up a metadata structure to build the Multidimensional Database (MDDB) to handle sparse data. It combines the scalability features of the ROLAP approach and the excellent indexing and accessing features of the MOLAP approach. We only aggregate a very small subset of the dimensions or probably do not aggregate at all. We can use our MOLAP approach to build cubes on-the-fly for OLAP and OLAP mining. Therefore, with MDDB, we can cope with sparse data comfortably.

6.2.2 OLE DB for OLAP

Recently, Microsoft has proposed "OLE DB for OLAP" as a standard multidimensional interface for databases, which defines *Multi-Dimensional Expressions*(MDX). This interface contains some interesting and challenging features that allows clients to ask several related dimensional queries in a single MDX expression. We can adjust our datacube accordingly to support these MDX expressions as our APIs to clients.

6.2.3 Discovery-Driven Exploration

We can precompute the indicators of exceptions at various levels of detail in the cube as described in [SAM98]. Abnormal patterns in the data at any level of aggregation could be easily noticed. We can extend our datacube to compute the anticipated value of a cell in context of its position in the cube and the trends along different dimensions to which the cell belongs using some statistical model. This can be further extended to be used by different data mining modules.

6.2.4 Multi-Feature Cubes

We can extend our datacube to have **Multi-Features** [RSC98] to support a decision support query in which multiple dependent aggregations are computed within each group. This, however, may require sophisticated change to the current datacube technology. Nonetheless, these multiple dependent aggregates are very useful measurements for some data mining modules.

6.2.5 OLAM on Complex Types of Data

It is challenging to extend the current OLAM method to mine on complex types of data, such as complex data objects (for object-oriented database or object-relational database), spatial data, text and multimedia data and Web-based data. The research and development has just started and there is still a long way to go. Nevertheless, a spatial OLAP mining prototype called GeoMiner was implemented on top of cube engine of DBMiner. It has been demoed in different conferences in 1997 and it has received various positive feedback.

Bibliography

[AAD$^+$96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 Int. Conf. Very Large Data Bases*, pages 506–521, Bombay, India, Sept. 1996.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data*, pages 207–216, Washington, D.C., May 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.

[CC94] W. W. Chu and K. Chiang. Abstraction of high level concepts from numerical values in databases. In *Proc. AAAI'94 Workshop Knowledge Discovery in Databases (KDD'94)*, pages 133–144, Seattle, WA, July 1994.

[CCH91] Y. Cai, N. Cercone, and J. Han. Attribute-oriented induction in relational databases. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 213–228. AAAI/MIT Press, 1991.

[CCS93] E. F Codd, S. B. Codd, and C. T. Salley. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. In *E. F. Codd & Associates available at http://www.arborsoft.com/OLAP.html*, 1993.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26:65–74, 1997.

[Che98] S. Cheng. *Statistical Approaches to Predictive Modeling in Large Databases*. M.Sc. Thesis, Simon Fraser University, Canada, January 1998.

[CHY96] M. S. Chen, J. Han, and P. S. Yu. Data mining: An overview from a database perspective. *IEEE Trans. Knowledge and Data Engineering*, 8:866–883, 1996.

[Col96] G. Colliat. OLAP, relational and multidimensional database systems. *SIGMOD Record*, 25:64–69, 1996.

[Cor] Arbor Software Corporation. Key Essbase advantages for OLAP. In *http://www.arborsoft.com/olap/advantag.html*.

[CS96] P. Cheeseman and J. Stutz. Bayesian classification (AutoClass): Theory and results. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 153–180. AAAI/MIT Press, 1996.

[DNR+97] P. Deshpande, J. Naughton, K. Ramasamy, A. Shukla, K. Tufte, and Y. Zhao. Cubing algorithms, storage estimation, and storage and processing alternatives for olap. *Data Engineering Bulletin*, 20(1):3–11, 1997.

[EKX95] M. Ester, H.-P. Kriegel, and X. Xu. Knowledge discovery in large spatial databases: Focusing techniques for efficient class identification. In *Proc. 4th Int. Symp. Large Spatial Databases (SSD'95)*, pages 67–82, Portland, Maine, August 1995.

[FH95] Y. Fu and J. Han. Meta-rule-guided mining of association rules in relational databases. In *Proc. 1st Int'l Workshop Integration of Knowledge Discovery with Deductive and Object-Oriented Databases (KDOOD'95)*, pages 39–46, Singapore, Dec. 1995.

[Fis87] D. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172, 1987.

[FPSS96a] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 1–35. AAAI/MIT Press, 1996.

[FPSS96b] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. Knowledge discovery and data mining: Towards a unifying framework. In *Proc. 2nd Int. Conf. Knowledge Discovery and Data Mining (KDD'96)*, pages 82–88, Portland, Oregon, Aug. 1996.

[FPSSU96] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational operator generalizing group-by, cross-tab and sub-totals. In *Proc. 1996 Int'l Conf. Data Engineering*, pages 152–159, New Orleans, Louisiana, Feb. 1996.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environment. In *Proc. 21st Int. Conf. Very Large Data Bases*, pages 358–369, Zurich, Switzerland, Sept. 1995.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25:73–170, 1993.

[HCC93] J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Trans. Knowledge and Data Engineering*, 5:29–40, 1993.

[HCC+97] J. Han, J. Chiang, S. Chee, J. Chen, Q. Chen, S. Cheng, W. Gong, M. Kamber, G. Liu, K. Koperski, Y. Lu, N. Stefanovic, L. Winstone,

B. Xia, O. R. Zaïane, S. Zhang, and H. Zhu. DBMiner: A system for data mining in relational databases and data warehouses. In *Proc. CASCON'97: Meeting of Minds*, pages 249–260, Toronto, Canada, November 1997.

[HCC98] J. Han, S. Chee, and J. Y. Chiang. Issues for on-line analytical mining of data warehouses. In *Proc. of 1998 SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'98)*, pages 2:1–2:5, Seattle, Washington, June 1998.

[HF94] J. Han and Y. Fu. Dynamic generation and refinement of concept hierarchies for knowledge discovery in databases. In *Proc. AAAI'94 Workshop Knowledge Discovery in Databases (KDD'94)*, pages 157–168, Seattle, WA, July 1994.

[HF95] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases*, pages 420–431, Zurich, Switzerland, Sept. 1995.

[HF96] J. Han and Y. Fu. Exploration of the power of attribute-oriented induction in data mining. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 399–421. AAAI/MIT Press, 1996.

[HFH+94] J. Han, Y. Fu, Y. Huang, Y. Cai, and N. Cercone. DBLearn: A system prototype for knowledge discovery in relational databases. In *Proc. 1994 ACM-SIGMOD Conf. Management of Data*, page 516, Minneapolis, MN, May 1994.

[HFW+96] J. Han, Y. Fu, W. Wang, J. Chiang, W. Gong, K. Koperski, D. Li, Y. Lu, A. Rajan, N. Stefanovic, B. Xia, and O. R. Zaïane. DBMiner: A system for mining knowledge in large relational databases. In *Proc. 1996 Int'l Conf. Data Mining and Knowledge Discovery (KDD'96)*, pages 250–255, Portland, Oregon, August 1996.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[Inca] Stanford Technology Group Inc. INFORMIX MetaCube. In *http://www.informix.com/informix/press/1998/rolapjan.htm.*

[Incb] MicroStrategy Incorporated. The case for relational OLAP. In *http://www.strategy.com/dw_forum/WhitePapers/Case4Rolap/execsumm.htm.*

[Inm92] W. H. Inmon. *Building the Data Warehouse.* QED Technical Publishing Group, Wellesley, Massachusetts, 1992.

[IP96] J. Elder IV and D. Pregibon. A statistical perspective on knowledge discovery in databases. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 83–115. AAAI/MIT Press, 1996.

[KHC97] M. Kamber, J. Han, and J. Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proc. 3rd Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 207–210, Newport Beach, California, August 1997.

[Kim96] R. Kimball. *The Data Warehouse Toolkit.* John Wiley & Sons, New York, 1996.

[KS91] H. F. Korth and A. Silberschatz. *Database System Concepts, 2ed.* McGraw-Hill, 1991.

[KWG+97] M. Kamber, L. Winstone, W. Gong, S. Cheng, and J. Han. Generalization and decision tree induction: Efficient classification in data mining. In *Proc. of 1997 Int. Workshop Research Issues on Data Engineering (RIDE'97)*, pages 111–120, Birmingham, England, April 1997.

[Lu98] Y. Lu. *Concept Hierarchies in Data Mining: Specification, Generation and Application.* M.Sc. Thesis, Simon Fraser University, Canada, January 1998.

[NH94] R. Ng and J. Han. Efficient and effective clustering method for spatial data mining. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 144–155, Santiago, Chile, September 1994.

[PCY95] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. 1995 ACM-SIGMOD Int. Conf. Management of Data*, pages 175–186, San Jose, CA, May 1995.

[PS91] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 229–238. AAAI/MIT Press, 1991.

[PSF91] G. Piatetsky-Shapiro and W. J. Frawley. *Knowledge Discovery in Databases.* AAAI/MIT Press, 1991.

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

[RS97] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 23rd Int. Conf. Very Large Data Bases*, pages 116–125, Athens, Greece, August 1997.

[RSC98] K. A. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. In *Proc. Int. Conf. of Extending Database Technology (EDBT'98)*, pages 263–277, Valencia, Spain, March 1998.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 1995 Int. Conf. Very Large Data Bases*, pages 407–419, Zurich, Switzerland, Sept. 1995.

[SAM98] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In *Proc. Int. Conf. of Extending Database Technology (EDBT'98)*, pages 168–182, Valencia, Spain, March 1998.

[Sar97] S. Sarawagi. Indexing OLAP data. *Bulletin of the Technical Committee on Data Engineering*, 20:36–43, 1997.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases*, pages 432–443, Zurich, Switzerland, Sept. 1995.

[SS94] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proc. 1994 Int. Conf. Data Engineering*, pages 328–336, Feb. 1994.

[SSU96] A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database research: Achievements and opportunities into the 21st century. *ACM SIGMOD Record*, 25:52–63, March 1996.

[Tho97] E. Thomsen. *OLAP Solutions: Building Multidimensional Information Systems*. John Wiley & Sons, 1997.

[Wid95] J. Widom. Research problems in data warehousing. In *Proc. 4th Int. Conf. Information and Knowledge Management*, pages 25–30, Baltimore, Maryland, Nov. 1995.

[YL95] W. P. Yan and P. Larson. Eager aggregation and lazy aggregation. In *Proc. 21st Int. Conf. Very Large Data Bases*, pages 345–357, Zurich, Switzerland, Sept. 1995.

[ZDN97] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data*, pages 159–170, Tucson, Arizona, May 1997.

[Zia94] W. Ziarko. *Rough Sets, Fuzzy Sets and Knowledge Discovery.* Springer-Verlag, 1994.

[ZRL96] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data*, pages 103–114, Montreal, Canada, June 1996.