Design Issues in a Data Warehouse

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engginering

by

Anurag Srivastava

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560012

March 1999

To

“My Parents”

Contents

List of Figures

Acknowledgements

I would like to thank my project guide *Dr. Jayant Haritsa* for his excellent guidance throughout the course of this project. I am grateful to him for his constant support and encouragement, without which this project would not have materialized. His attitude towards perfection is admirable.

I express my special thanks to my batch mates *Garg, Kansal, Tomer* and others who helped me throughout my stay in IISc and made it a pleasant and memorable one. It is just because of them that i never felt i am thousand of miles away from home. I would always remember the time I had spent with them.

I am thankful to all my DSL friends for providing me a wonderful working environment. The pain I have given to all of you will help you remember me.

Finally, I thank my parents for their help and understanding without which I would have never made into this prestigious institute. Their love, blessings and encouragement were a constant source of inspiration to me. Whatever I am now, is because of them. Many thanks to *Babloo* and *Monu bhaiya, Bhabhi, Guriya, Tinku, Tonu* and *Rinku* for their love and support.

Abstract

Traditional **Online Transaction Processing** (**OLTP**) systems are not suitable for decision support applications because their emphasis is on providing **ACID** (Atomicity, Consistency, Isolation, Durability) semantics. Hence the performance of these systems is poor for decision support applications.

Decision support applications use **Online Analytical Processing (OLAP)** which is a recent and important application of database systems. OLAP queries are complex and can take many hours if executed directly on raw data. The most common method of reducing the execution time is to precompute some of the queries in the **data warehouses** and build indexes over them. Data warehouses typically maintain historical information as well as recent information and therefore they tend to be very large and grow over time.

In this thesis, we investigate various design issues for data warehouses. In the first part, we focus on the cube query which is the N dimensional generalization of SQL *group by* query. We have implemented an efficient algorithm for computing cube queries which uses various kind of optimization techniques (e.g minimized sorting, smart caching etc.) to reduce the number of disk accesses. We show that our algorithm implementation performs better than a commercial database implementation by 15-20%. In the second part, we look into various schema alternatives for data warehouses. We have done a performance analysis of the **star** and **snowflake** schemas for various levels of data redundancy and shown that the star schema is a reasonable choice for data warehouses in most environments.

Chapter 1

Introduction

1.1 What is a DBMS?

A **database management system (DBMS)** is a collection of programs that enables users to create and maintain database. The **DBMS** is a general-purpose software system that facilitates the process of defining, constructing and manipulating databases for various applications. **Defining** a database involves specifying the data types, structure and constraints for the data to be stored in the database. **Constructing** the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. **Manipulating** a database includes such functions as querying the database, updating the database and generating reports from the data.

1.2 What is a Data Warehouse?

"A *Data Warehouse is a subject-oriented, integrated, time variant and non volatile collection of data in support of management's decision-making process"* [IWG96].

It is a decision support database that is maintained separately from an organization's operational database. This database is designed based on what kind of information is important in company's decision making process (e.g., sales marketing, health care membership and providers, etc.) and it adopts a specialized schema design for maximum

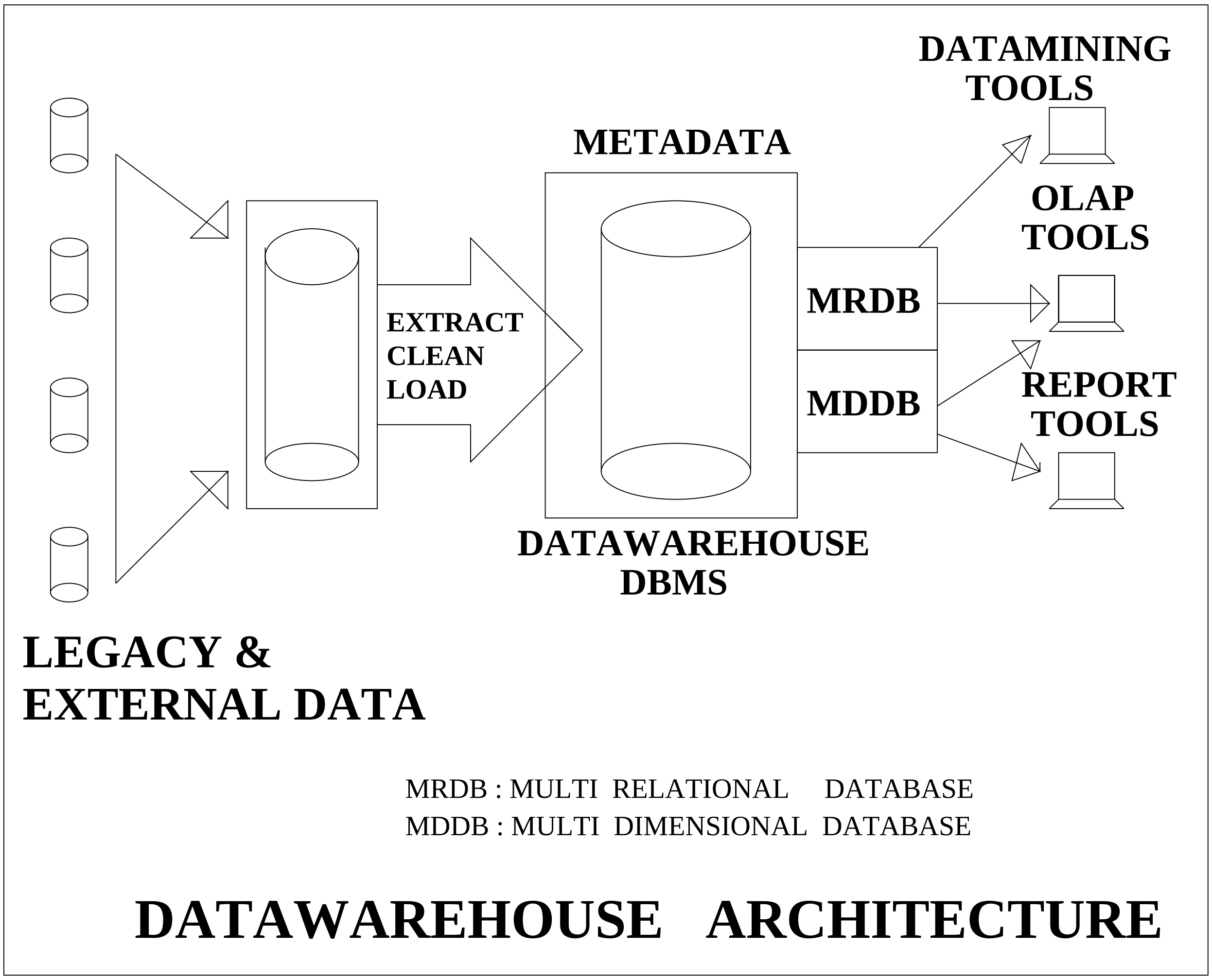


Figure 1.1: Data Warehouse Architecture

efficiency in performance. Generally a data warehouse is populated by taking data from various legacy sources. Extracting appropriate data from existing operational databases, cleansing or scrubbing the data, denormalizing the data and then loading the data into the database are the typical steps involved in creating a data warehouse. (This data population process is also known as the data transformation process.) The data warehouse is shown in Figure 1.1.

This database can then be used by top executives, managers, analysts and other end-users as a rich source of company's information. They can ask business questions and can make key and timely business decisions from their desktops using GUI Online Analytical Processing **(OLAP)** tools.

1.3 Attributes of a Data Warehouse

The fundamental attributes of a data warehouse are as follows [IWG96]:

1.3.1 Subject Oriented

A data warehouse is organized around subjects. Subject orientation presents the data in a format that is consistent and much clearer for end users to understand. In contrast, operational data such as order processing and manufacturing databases are organized around business activities or functional areas. They are typically optimized to serve a single static application. The functional separation of applications causes companies to store identical information in multiple locations. The duplicated information's format is usually inconsistent. For data warehouse subjects could be "Product", "Customers" and "Orders" as opposed to "Purchasing" and "Payroll".

1.3.2 Integrated

Integration of data within a warehouse is accomplished by dictating consistency in format, naming, etc. Operational databases, for historic reasons, often have major inconsistencies in data representation. For example, a set of operational databases may represent "male" and "female" by "m" and "f", by "1" and "2", by "x" and "y". Frequently the inconsistencies are more complex and subtle. Hence cleansing operations convert various representations of data into one format so that they can be accommodated into a single schema. By definition, data is always maintained in a consistent fashion in a data warehouse.

1.3.3 Time Variant

Data warehouses are time variant in the sense that they maintain both historical and (nearly) current data. Operational databases, in contrast, contain only the most current, up-to-date data values. Furthermore, they generally maintain this information for no more than a year (and often much less). By comparison, data warehouses contain data that is generally loaded from the operational databases daily, weekly or monthly and then typically maintained for a period of 3 to 5 years. This aspect marks a major difference between the two types of environments. Historical information is of high importance to decision-makers. They often want to understand trends and relationships between data. For example, the product manager for a soft drink maker may want to see the relationship between coupon promotions and sales. This type of information is typically impossible to determine with an operational database that contains only current data.

1.3.4 Non-volatile

Non-volatility, another primary aspect of data warehouses, means that after the informational data is loaded into the data warehouse, changes, inserts or deletes are rarely performed. Data is read-only in most of the cases. The loaded data is transformed data that originated from the operational databases. The data warehouse is subsequently reloaded or, more likely, appended on a periodic basis with new, transformed or summarized data from the operational databases. Apart from this loading process, the information contained in the data warehouse generally remains static. The property of non-volatility permits a data warehouse to be heavily optimized for query processing.

1.4 Issues in Data Warehouse Design

There are various issues involved in data warehouse design.The most important issues involved are as follows:

1.4.1 Choosing a Schema

In an operational database design, the data is highly normalized to support consistent updates and to maintain referential integrity. In a data warehouse design, the data is highly denormalized to provide instant access without having to perform a large number of joins. The data warehouse adopts a star or snowflake schema to maximize performance. A star or snowflake schema design is very different from that of an operational database schema design as it represents data as a logical array in which each dimension is a subject around which analysis is performed.

1.4.2 ROLAP vs. MOLAP

Currently there are two dominant OLAP architectures. Relational OLAP (ROLAP) and multidimensional OLAP (MOLAP). ROLAP techniques store data in tables in a traditional RDBMS and provide sophisticated front end to support the complex functions needed by OLAP users. MOLAP techniques, on the other hand, provide custom OLAP servers which typically store data as multidimensional arrays or custom storage structures specifically designed for efficient storage and retrieval of sparse multidimensional arrays.

There is ongoing debate over which architecture more effectively supports OLAP functionality. ROLAP approach is scalable but MOLAP approach is more efficient when data warehouse size and dimensions are known in advance [HSB98].

1.4.3 Building Cubes

While operational databases maintain state information, data warehouses typically maintain historical information. As a result, data warehouses tend to be very large and to grow over time. Users of **Decision support systems (DSS)** are typically interested in identifying trends rather than looking at individual records in isolation. DSS involve complex queries on very large databases and make heavy use of aggregations. The size of data warehouse and the complexity of queries can cause queries to take very long time to complete. This delay is unacceptable in most DSS environments as it severely limits productivity. The usual requirement is query execution times of a few seconds or a few minutes at the most.

There are many ways to achieve such performance goals. Query optimizers and query evaluation techniques can be enhanced to handle aggregations better [CS94]. Since response times should be small, query optimization is critical. Users typically view the data as multidimensional "data cubes". Each cell of the data cube is a view consisting of an aggregation of interest. The value of many of these cells are dependent on the values of other cells in the data cube. A common and powerful query optimization technique is to materialize (precompute) some of the frequently-asked queries rather then computing them from raw data each time. [HRU96] discusses a close to optimal solution for the problem of which queries (view) to materialize when it is too expensive to materialize all of them.

1.5 Advantages of a Data Warehouse

Data warehouses are rapidly becoming the key to gaining competitive advantage for businesses. It allows businesses to get data that is merged away in operational databases and turn that data into useful information. Data warehouses are becoming integral part of the company's information system. It can be used as a stand alone database for **Data Mining** operations for discovering interesting patterns in the database. Many corporations have built or are building unified decision-support databases (data warehouses) on which users can carry out their analysis.

1.6 Contribution of the Thesis

In this thesis, we have focussed on the back end of the data warehouse. It has been assumed that the input data is already in clean format. Based on these assumptions, efficient algorithm for computing cube queries has been implemented. We have shown that our implementation performs better than the commercial database system DB2's implementation. Performance evaluation of various data warehouse schemas, e.g. star and snowflake, is also done and it has been shown that star schema is a reasonable choice in most of the cases.

1.7 Organization of the Report

The organization for the rest of the report is as follows: Chapter 2 highlights some of the problems with the SQL GROUP BY operator and gives an introduction of the

CUBE operator. Chapter 3 discusses efficient algorithms for computing cubes. Chapter 4 discusses the various schema alternatives for data warehouses. Chapter 5 gives an overview of IBM's relational database system DB2, which has been used as back end for the software development and performance studies. Chapter 6 compares the performance of cube algorithms and data warehouse schemas. Finally, Chapter 7 summarizes our conclusions and gives future research avenues.

Chapter 2

Data Cube: A Relational Aggregation Operator

2.1 Motivation

Data analysis applications typically aggregate data across many dimensions looking for interesting patterns. They categorize data values and trends, extract statistical information, and then contrast one category with another. There are four steps involved in such data analysis:

formulating a query that extracts relevant data from a large database,

extracting the aggregated data from the database into a file or table,

visualizing the results in a graphical way, and

analyzing the results and formulating a new query.

Thus, visualization as well as data analysis tools do "dimensionality reduction", often by summarizing the data along the dimensions that are important. For example, in trying to analyze car sales, we might focus on the role of model, year and color of the cars in sale. Thus we ignore the differences between two sales along the date of sale or dealership dimension but analyze the total sale for cars by model, by year and by color only. Along with summarization and dimensionality reduction, data analysis applications extensively use constructs such as histogram, cross-tabulation, subtotals, roll up and drill down.

2.2 Relational and SQL Data Extraction

How do traditional relational databases fit into this multi-dimensional data analysis picture? How can 2D flat files (SQL tables) model an N-dimensional problem? Furthermore, how do the relational system support operations over N-dimensional representations that are central to visualization and data analysis programs [GBLP96]?

The answer to these questions is that relational systems model N-dimensional data as a relation with N-attribute domains. For example, 4-dimensional (4D) earth temperature and pressure data is typically represented by a Weather Table 2.1. The first four columns represent four dimensions: time, latitude, longitude and altitude. Additional columns represent measurements at 4D points such as temperature and pressure. Each individual weather measurement is recorded as a new row of this table and these measured values are aggregated over time (the hour) or space (measurement point).

Table 2.1: Weather Table

Weather					
Time(UCT)	Latitude	Longitude	Altitude(m)	Temp(c)	Pres(mb)
27/02/99:1530	37:58:33N	122:45:28W	102	21	1009
Many more rows like the ones above and below					
28/02/99:1530	34:16:18N	27:05:55W	102	23	1024

As mentioned before, visualization and data analysis tools extensively use dimensionality reduction (aggregation) for better comprehensibility. Often data along the other dimensions that are not included in a "2D" representation are summarized via aggregation in the form of histograms, cross-tabulation, subtotals, etc. In the SQL standard [MS93], we depend on aggregation functions and the GROUP BY operator to support aggregation.

The SQL standard (SQL 92) provides five functions to aggregate the values in a table: COUNT(), SUM(), MIN(), MAX() and AVG(). For example, the average of all measured temperatures is expressed as:

SELECT AVG(Temp)

From Weather;

In addition, SQL allows aggregation over distinct values. The following query counts distinct number of reporting times in the Weather table:

SELECT COUNT(DISTINCT Time)
From Weather;

Aggregate functions return a single value. Using the GROUP BY construct, SQL can also a create table of many aggregate values indexed by a set of attributes. For example, the following query reports the average temperature for each reporting time and altitude.

SELECT Time, Altitude, **AVG**(Temp)
From Weather
GROUP BY Time, Altitude;

GROUP BY is a relational operator which partitions the relation into disjoint tuple sets and then aggregates over each set.

SQL's aggregation functions are widely used in database applications. This popularity is reflected in the presence of aggregates in a large number of queries in the decision-support benchmark [TPCD93]. Beyond the five standard aggregate functions defined so far, many SQL systems add statistical functions (median, standard deviation, variance, etc.), physical function (center of mass, angular momentum, etc.) and some systems even allow users to add aggregate functions called **user defined functions (UDF)**, e.g. Redbrick systems, DB2 , etc..

2.3 Problems with the GROUP BY Operator

The SQL aggregate functions and GROUP BY operator produce zero-dimensional or one-dimensional aggregates. However, often applications need N-dimensional generalization of these operators. Certain common forms of data analysis are difficult with these SQL aggregation constructs. As explained next, three common problems are [GBLP96]: (1) Histograms, (2) Roll-up Totals and Sub-Totals for drill-downs, and (3) Cross Tabulations.

The standard SQL GROUP BY operator does not allow a direct construction of histograms (aggregation over computed categories). For queries based on the Weather table, it would be nice to be able to group times into days, weeks, or months, and to group locations into areas (e.g. US, Canada, Europe,..). If a Nation() function maps latitude and longitude into the name of the country containing that location, then the following query would give the daily maximum reported temperature for each nation.

SELECT day, nation, **MAX**(Temp)
From Weather
GROUP BY Day(Time) **AS** day, Nation(Latitude, Longitude) **AS** nation;

A more serious problem relates to roll-ups using totals, and sub-totals for drill-down reports. Reports commonly aggregate data at a coarse level and then successively finer levels. The car sales report in Table 2.2 shows this idea. Data is aggregated by Model, then by Year and then by Color. The report shows data aggregated at three levels. Going up the levels is called **rolling-up** the data. Going down is called **drilling-down** the data. Data aggregated at each distinct level produces a sub-total.

Table 2.2 suggests creating 2^N aggregation columns for a roll up of N elements. The representation of Table 2.2 is not relational because the empty cells (presumably null values) cannot form a key. Representation Table 2.3 is an elegant solution to this problem, but number of columns grows as the power set of the number of aggregated attributes, creating difficult naming problems and very long names.

Rather than extending the result table to have many new columns, a more conservative approach prevents the exponential growth of columns by *overloading* column values. The idea is to introduce an “ALL” value. Table 2.4 demonstrates this relational and more convenient representation. The dummy value “ALL” has been added to fill in the super-aggregation items.

Table 2.2: Sales Roll Up by Model by Year by Color

Model	Year	Colour	Sales by Model by Year by Colour	Sales by Model by Year	Sales by Model
Maruti	1990	Blue	12		
		White	35		
				47	
	1991	Blue	32		
		White	39		
				71	
					118

Table 2.3: Sales Roll Up by Model by Year by Color (Alternative representation)

Model	Year	Colour	Sales by Model by Year by Colour	Sales by Model by Year	Sales by Model
Maruti	1990	Blue	12	47	118
Maruti	1990	White	35	47	118
Maruti	1991	Blue	32	71	118
Maruti	1991	White	39	71	118

Table 2.4 is not really a completely new representation or operation. Since Table 2.4 is a relation, it is not surprising that it can be built using standard SQL. The SQL statement to build this **Sales Summary** table from the raw **SALES** data is:

```
SELECT 'ALL', 'ALL', 'ALL', SUM(Sales)
    FROM SALES
    WHERE Model='Maruti'
UNION
SELECT Model, 'ALL','ALL', SUM(Sales)
    FROM SALES
    WHERE Model='Maruti'
    GROUP BY Model
```

Table 2.4: Sales Summary

Model	Year	Colour	Sales
Maruti	1990	Blue	12
Maruti	1990	White	35
Maruti	1990	ALL	47
Maruti	1991	Blue	32
Maruti	1991	White	39
Maruti	1991	ALL	71
Maruti	ALL	ALL	118

```
UNION
SELECT Model, Year, 'ALL', SUM(Sales)
    FROM SALES
    WHERE Model='Maruti'
    GROUP BY Model, Year
UNION
SELECT Model, Year, Color, SUM(Sales)
    FROM SALES
    WHERE Model='Maruti'
    GROUP BY Model, Year, Color
```

This is a simple 3-dimensional roll-up. Aggregating over N-dimensions requires N such unions. Roll-up is asymmetric as Table 2.4 aggregates sales by year but not by color. These missing rows are shown in Table 2.5.

Table 2.5: Sales summary rows missing form Table 2.4 to convert roll-up into a cube

Model	Year	Colour	Sales
Maruti	ALL	Blue	44
Maruti	ALL	White	74

These additional rows could be captured by adding the following clause to the SQL statement above:

```
UNION
SELECT Model, 'ALL', Color, SUM(Sales)
    FROM SALES
    WHERE Model='Maruti'
    GROUP BY Model, Color;
```

The symmetric aggregation result is a table called a **cross-tabulation** or **cross tab** for short.

The representation suggested by Table 2.4 and unioned GROUP BY's solve the problem of representing aggregate data in a relational data model. The problem remains that expressing roll up and cross-tab queries with conventional SQL is daunting. A six dimension cross-tab requires a 64-way union of different GROUP BY operators to build the underlying representation.

There is another very important reason why it is inadequate to use GROUP BY's. The resulting representation of aggregation is too complex to analyze for optimization. On most SQL systems this will result of 64 scans of the data and 64 sorts or hashes [GBLP96].

2.4 CUBE and ROLLUP Operators

Recently, a new operator, called the *data cube* or simply *cube* was introduced in [GBLP96] for conveniently supporting multiple aggregates in OLAP databases. The CUBE operator is the *N*-dimensional generalization of the GROUP BY operator. Creating a data cube requires generating the power set (set of all subsets) of the aggregation columns. Since the CUBE is an aggregation operation, it can be externalized by overloading the SQL GROUP BY operator. For example, in SALES tables with attributes Model(M), Year(Y), Color(C) and Sales, the typical CUBE query will look like:

```
SELECT M, Y, C, SUM(Sales)
FROM SALES
GROUP BY CUBE M, Y, C
```

This query will result in computation of $2^3 = 8$ group-bys: MYC, MY, MC, YC, M, Y, C and ALL, where ALL denotes the empty group-by.

The semantics of the CUBE operator is that it first aggregates over all the attributes in the GROUP BY clause as in standard GROUP BY. Then, it UNIONs each super-aggregate of the global cube–substituting ALL for the aggregating columns. If there are N attributes in the GROUP BY clause, there will be $2^N - 1$ super-aggregate values. If the cardinality of the N attributes are $C_1, C_2, ..C_N$ then the cardinality of the resulting cube relation is $\Pi(C_i + 1)$. The extra value in each domain is ALL. For example, the SALES table has 2 x 2 x 2 = 8 rows, while derived data cube has 3 x 3 x 3 = 27 rows. Figure 2.1 shows the example of cube syntax. If the application wants only a roll-up or drill-down report, similar to the data in Table 2.4, the computing full cube is unnecessary. Indeed, some part of the cube may be meaningless.

The solution is to offer ROLLUP in addition to CUBE as it just produces the super-aggregates:

$(v_1, v_2, ..., v_n,$ f()),

$(v_1, v_2, ..., ALL,$ f()),

...

$(v_1, ALL, ..., ALL,$f()),

(ALL, ALL,...,ALL, f()),

2.5 The GROUP BY, CUBE, ROLLUP Algebra

The GROUP BY, ROLLUP and CUBE operators have an interesting algebra [GBLP96]. The CUBE of a ROLLUP or GROUP BY is a CUBE. The ROLLUP of a GROUP BY is a ROLLUP. Algebraically, this operator algebra can be stated as:

CUBE(ROLLUP) = CUBE

ROLLUP(GROUP BY) = ROLLUP

So it makes sense to arrange the aggregation operators in the compound order where the "most powerful" cube operator at the core, then a roll-up of the cubes and then a group-by of the roll-ups. One can use any subset of the three operators and the operators

```
SELECT Model, Year, Color, SUM(Sales)
FROM    SALES
WHERE Model  IN {'Maruti', 'Fiat'}
      AND Year  BETWEEN 1990 and 1991
GROUP BY CUBE Model, Year, Color
```

SALES			
Model	Year	Color	Sales
Maruti	1990	Blue	12
Maruti	1990	White	35
Maruti	1991	Blue	32
Maruti	1991	White	39
Fiat	1990	Blue	18
Fiat	1990	White	15
Fiat	1991	Blue	22
Fiat	1991	White	17

CUBE

SALES DATA CUBE			
Model	Year	Color	Sales
Maruti	1990	Blue	12
Maruti	1990	White	35
Maruti	1990	ALL	47
Maruti	1991	Blue	32
Maruti	1991	White	39
Maruti	1991	ALL	71
Maruti	ALL	Blue	44
Maruti	ALL	White	74
Maruti	ALL	ALL	118
Fiat	1990	Blue	18
Fiat	1990	White	15
Fiat	1990	ALL	33
Fiat	1991	Blue	22
Fiat	1991	White	17
Fiat	1991	ALL	39
Fiat	ALL	Blue	30
Fiat	ALL	White	32
Fiat	ALL	ALL	62
ALL	1990	Blue	30
ALL	1990	White	50
ALL	1990	ALL	80
ALL	1991	Blue	54
ALL	1991	White	56
ALL	1991	ALL	110
ALL	ALL	Blue	80
ALL	ALL	White	106
ALL	ALL	ALL	190

Figure 2.1: A 3D data cube built from the table at left by the CUBE statement at the top of figure

are evaluated in bottom up order:

GROUP BY select list

ROLLUP select list

CUBE select list

2.5.1 A discussion of the ALL value

Each ALL value really represents a set, the set over which the aggregate was computed. For SALES data cube, the respective sets are:

ALL(Model) = Maruti, Fiat

ALL(Year) = 1990, 1991

ALL(Color) = White, Blue

ALL value can be interpreted as a context sensitive token representing the set. The introduction of ALL creates substantial complexity and it touches many aspects of the SQL language. To name a few [GBLP96]:

- ALL becomes a new keyword denoting a set value.
- ALL [NOT] ALLOWED is added to column definition syntax and to the column attributes in the system catalogs. For example:
  In SALES database if a value ALL appears in the Model column, there is no way to find out whether it actually refers to a car model ALL or it is the aggregated column.
- The set interpretation guides the meaning of the relational operators{=, IN}.

It is convenient to know when a column value is an aggregate. Hence a special function ALL() should be implemented, which can test for a non-NULL value.

2.5.2 Avoiding the ALL value

ALL value can be avoided by using following procedure:

- Using NULL value in place of ALL value

- Implement a Boolean function GROUPING() that, given a select list element, returns TRUE if the element is an ALL value and false otherwise.

In this design, users can simulate the ALL value in the following manner:

```
SELECT Model, Year, Color, SUM(Sales),
                GROUPING(Model),
                GROUPING(Year),
                GROUPING(Color),
FROM SALES
GROUP BY CUBE Model, Year, Color;
```

Wherever the ALL value appeared before, now the corresponding value will be NULL in the data field and TRUE in the corresponding grouping field. For example, the global sum of Figure 2.1 will be the tuple:

(NULL, NULL, NULL, 190, TRUE, TRUE, TRUE),

rather than the tuple one would get with the "real" cube operator:

(ALL, ALL, ALL, 190)

Using the limited interpretation of ALL makes it possible to express results of CUBE as a single relation in the current framework of SQL.

Chapter 3

Computing Cubes Efficiently

3.1 Introduction

As mentioned in the previous chapter, the CUBE operator is the N-dimensional generalization of the GROUP BY operator. It computes 2^N group-bys for a N attribute relation. The straightforward way to support the above query is to rewrite it as a collection of 2^N group-by queries and execute them separately. This brute-force approach is considerably slow and there are several ways in which this simple solution can be improved. In this chapter we will discuss some fast algorithms for computing the data cube. The aggregation functions are assumed to be *distributive* [GBLP96], that is, they allow input set to be partitioned into disjoint sets that can be aggregated separately and later combined. Examples of distributive functions include *max, min, count and sum.* The algorithms are also applicable to the *algebraic* functions [GBLP96], such as *average*, that can be expressed in terms of other distributive functions (*sum* and *count* for *average*). However, as pointed out in [GBLP96], there are some aggregate functions (*holistic* functions) e.g. *median*, that cannot be computed in parts and combined.

3.2 Possible Optimizations

There are two basic methods for computing a group-by: (1) the sort based method and (2) the hash-based method [GRA93]. These methods can be used to compute multiple group-bys by incorporating the following optimizations [SAG96]:

3.2.1 Smallest-parent

This optimization aims at computing a group-by from the smallest previously computed group-by. In general, each group-by can be computed from a number of other group-bys. Figure 3.1 shows a four attribute cube ($ABCD$) and the options for computing a group-by from a group-by having one more attribute, called its *parent*. For instance, AB can be computed from ABC, ABD or $ABCD$. ABC or ABD are clearly better choices for computing AB, as the number of tuples in them will be less than $ABCD$. In addition, even between ABC and ABD, there can often be big difference in size making it critical to consider size in selecting a parent for computing AB.

3.2.2 Cache-results

This optimization aims at caching (in memory) the results of a group-by from which other group-bys are computed to reduce disk I/O. For instance, for the cube of Figure 3.1, having computed ABC, it is possible to compute AB from it while ABC is still in memory.

3.2.3 Amortize-scans

This optimization aims at amortizing the disk reads by computing as many group-bys as possible, together in memory. For instance, if the group-by $ABCD$ is stored on disk, it is possible to reduce disk read costs if all of ABC, ABD and BCD were computed in one scan of $ABCD$.

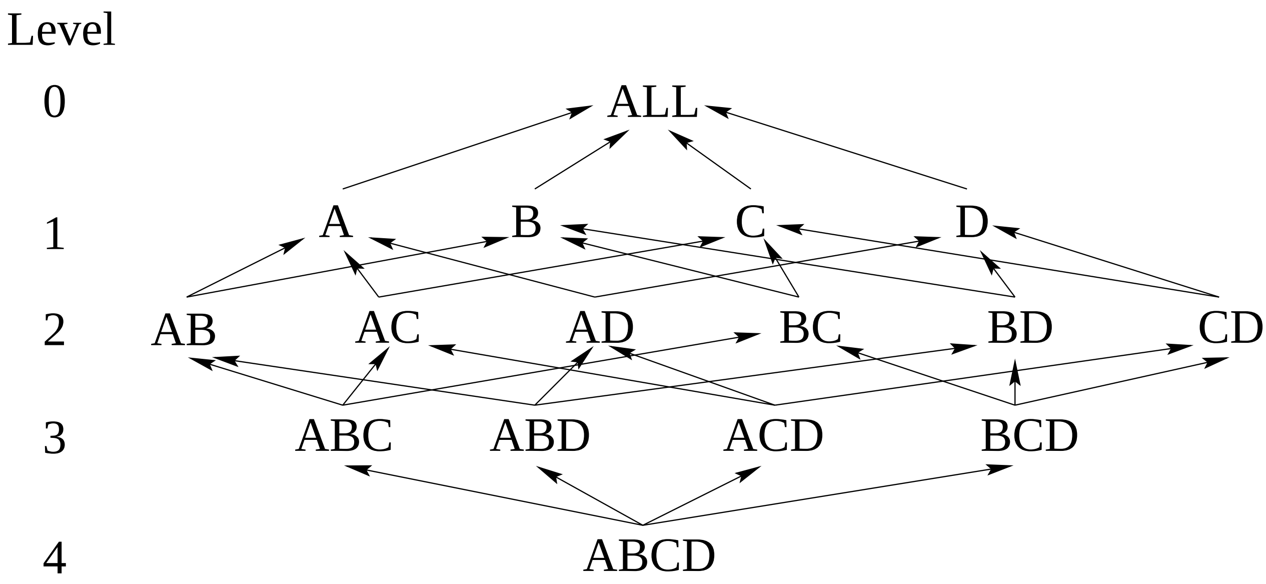


Figure 3.1: A search lattice for the cube operator

3.2.4 Share-sorts

This optimization is specific to the sort-based algorithms and aims at sharing sorting cost across multiple group-bys.

3.2.5 Share-partitions

This optimization is specific to hash-based algorithms. When the hash table is too large to fit in memory, data is partitioned and aggregation is done for each partition that fits in memory. We can save on partitioning cost by sharing this cost across multiple group-bys.

For OLAP databases, the size of the data to be aggregated is usually much larger than the available main memory. Under such constraints, the above optimizations are often contradictory. For computing B, for instance, the **smallest-parent** optimization will favor BC over AB if BC is smaller, but the **cache-results** optimization will favor AB, if AB is in memory and BC is on disk.

3.3 Sort-based Method

In this section, a sort based algorithm **PipeSort** [SAG96] is presented that incorporates the optimizations mentioned in section 3.2. As mentioned before, there can be conflict about which optimization to choose. It is necessary, therefore, to do global planning to decide what group-by is computed from what and the attribute order in which it is computed. PipeSort combines the optimizations share-sorts and smallest-parent to get the minimum total cost.

PipeSort includes the optimization **share-sort** by using data sorted in a particular order to compute all group-bys that are prefixes of that order. For instance, if the raw data is sorted on attribute $ABCD$, then it is possible to compute group-bys $ABCD$, ABC, AB and A without additional sorts. The algorithm also aims at choosing *smallest-parent* for a group-by. PipeSort also includes the optimizations *cache-results* and *amortize-scans* to reduce disk scan cost by executing multiple group-bys in a *pipelined* fashion. For instance, consider the previous example of using data sorted in the order $ABCD$ to compute prefixes $ABCD$, ABC, AB and$A$. Instead of computing each of these group-bys separately, it is possible to compute them in a pipelined fashion as follows. After sorting the raw data in the attribute order $ABCD$, scan the sorted data to compute group-by $ABCD$. Every time a tuple $ABCD$ is computed, it is propagated in the pipeline to compute ABC; every time a tuple of ABC is computed, it is propagated up to compute AB and so on. Thus, each pipeline is a list of group-bys all of which are computed in a single scan of the sort input stream. During the course of execution of a pipeline, it is only needed to keep one tuple per group-by in the pipeline in memory.

3.3.1 Algorithm PipeSort

Assume that for each group-by an estimate of the number of distinct values is available. A number of statistical procedures (e.g. [HNSS95]) can be used for this purpose. The input to the algorithm is the search lattice defined as follows:

Search Lattice

A search lattice for the data cube [HRU96] is a graph where a vertex represents a group-by of the cube. A direct edge connects group-by i to group-by j whenever j can be generated from i, and j has exactly one attribute less than i (i called the parent of j). Thus, the out-degree of any node with k attributes is k. Figure 3.1 is an example of a search lattice. Level k of the search lattice denotes all group-bys that contain exactly k attributes. The keyword ALL is used to denote empty group-by (Level 0). Each edge e_{ij} in the search lattice 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.

The output, O of the algorithm is a subgraph of the search lattice where each group-by is connected to a single parent group-by from which it will be computed and is associated with an attribute order in which it will be sorted. If the attribute order of a group-by j is a prefix of the order of its parent i, then j can computed from i with out sorting i and in O, edge e_{ij} is marked A and incurs cost $A(e_{ij})$. Otherwise, i has to be sorted to compute j and in O, e_{ij} is marked S and incurs cost $S(e_{ij})$. Clearly, for any output O, there can be at most one out-edge marked A from any group-by i, since there can be only one prefix of i in the adjacent level. However, there can be multiple out-edges marked S from i. The objective of the algorithm is to find an output O that has minimum sum of edge costs.

Algorithm:

The algorithm proceeds level-by-level, starting 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 weighted [1] bipartite matching problem [PS82] as follows:

Level $k + 1$ of the original search lattice is first transformed by making k additional

[1] The weighted bipartite matching problem is defined as follows: We are given graph with two disjoint sets of vertices V_1 and V_2 and a set of edges E that connect vertices in set V_1 to vertices in set V_2. Each edge is associated with a fixed weight. The weighted matching problem selects the maximum weight subset of edges from E such that in the selected subgraph each vertex in V_1 is connected to at most one vertex in V_2 and vice versa.

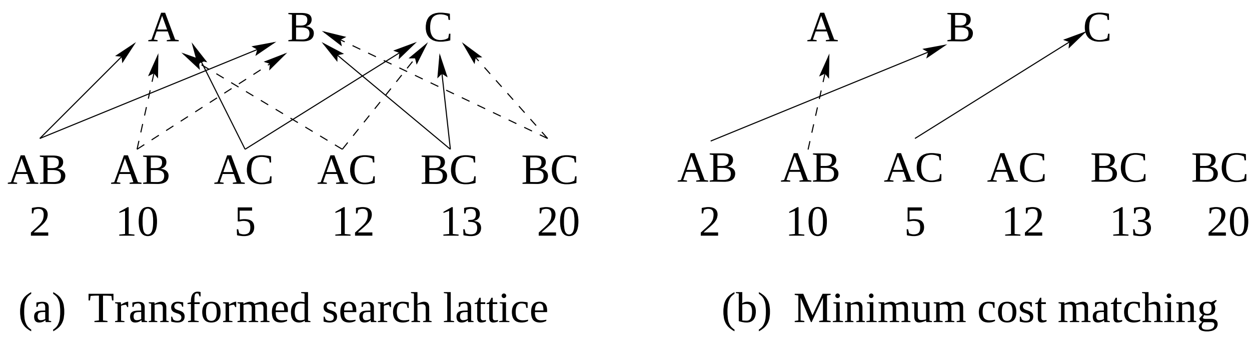


Figure 3.2: Computing level 1 group-bys from level 2 group-bys in a 3 attribute cube

copies of each group-by in that level. Thus each level $k+1$ group-by has $k+1$ vertices which is the same as the number of children or out-edges of that group-by. Each replicated vertex is connected to the same set of vertices as the original vertex in the search lattice. The cost on an edge e_{ij} from the original vertex i to a level k vertex j, is set to $A(e_{ij})$ whereas all replicated vertices of i have edge cost set to $S(e_{ij})$. After that find the minimum [2] cost matching in the bipartite graph induced by this transformed graph. In the matching so found, each vertex h in level k will be matched to some vertex g in level $k+1$. If h is connected to g by an $A()$ edge, then h determines the attribute order in which g will be sorted during its computation. On the other hand, if h is connected by an $S()$ edge, g will be re-sorted for computing h.

For illustration, we show how level 1 group-bys are generated from level 2 group-bys for a three attribute search lattice. As shown in Figure 3.2, we first make one additional copy of each level 2 group-by. Solid edges represent the $A()$ edges whereas dashed edges indicate the $S()$ edges. The number underneath each vertex is the cost of all out-edges from this vertex. In the minimum cost matching (Figure 3.2(b)), A is connected to AB with an $S()$ edge and B by an $A()$ edge. Thus at level 2, group-by AB will be computed in the attribute order BA so that B is generated from it without sorting and A is generated by re-sorting BA. Similarly, since C is connected to AC by an $A()$ edge, AC will be generated in the attribute order CA. Since, BC is not matched to any level 1 group-by, BC can be computed in any order.

[2] It is possible to convert minimum weight matching to maximum weight matching defined earlier by replacing each edge w by $max(w) - w$ where $max(w)$ is the maximum edge cost.

PipeSort:

(Input: search lattice with A and S edges costs)

For level $k = 0$ to $N - 1$ /* N is the total number of attributes */

/* determine how to generate level k from level $k + 1$ */

Generate-Plan($k + 1$ —> k);

For each group-by g in level $k + 1$

Fix the sort order of g as the order of the level k group-by

that is connected to g by an $A()$ edge;

Generate-Plan($k + 1$ —> k)

Create k additional copies of each level $k + 1$ vertex;

Connect each copy vertex to the same set of level k vertices as the original vertex;

Assign cost $A(e_{ij})$ to edge e_{ij}, from the original vertex and cost $S(e_{ij})$ to edge

from the copy vertex;

Find the minimum cost matching on the transformed level $k + 1$ with level k;

3.4 Hash-based Method

In this section, we will give an overview of a hash-based algorithm **PipeHash** [SAG96] for computing a data cube. For hash-based methods, it is important to carefully allocate memory for multiple hash-tables for incorporating optimizations **cache-results** and **amortize-scans**. For instance, if the hash table for AB and AC fit in memory then the two group-bys could be computed in one scan of ABC. After ABC is computed one could compute A and B while AB is still in memory and thus avoid the disk scan of AB. If memory were not a limitation, we could include all optimization stated in section 3.2 as follows:

For $k = N$ to 0

For each $k + 1$ attribute group-by, g

Compute in one scan of g all k attribute group-by for

which g is the smallest parent;

Save g to disk and destroy hash table of g ;

However, the data to be aggregated is usually too large for the hash-tables to fit in memory. The standard way to deal with limited memory when constructing hash-tables is to partition the data on one or more attributes. When data is partitioned on some attribute, say A, then all group-bys that contain A can be computed by independently grouping on each partition–the results across multiple partitions need not be combined. We can share the cost of data partitioning across all group-bys that contain the partitioning attribute, leading to the optimization **share-partitions**. The algorithm PipeHash incorporates this optimization and also includes the optimizations **cache-results**, **amortize-scans** and **smallest-parent**. This algorithm is not used in our work and details of the algorithm can be found in [SAG96].

Chapter 4

Schema Alternatives

4.1 Introduction

The description of a database is called the **database schema** (or the **meta data**). A database schema is specified during database design and it describes how the database is organized on disk. As mentioned before, OLTP database schemas are usually optimized for the recording of business transactions and uses various normalization forms (e.g First normal form, Second normal form, Boyce-Codd normal form etc.). **Normalization of data** can be looked on as a process during which unsatisfactory relation schema are decomposed by breaking up their attributes into smaller relation schemas that possess desirable properties. Normalization has several advantages for data integrity (e.g preventing update anomalies) in an OLTP environment, but works strongly against a DSS environment, as follows:

- Normalization process [HSB98] take descriptions of entities from the business domain and breaks them apart into a number of small tables. Breaking up entities into small tables also reduces the amount of redundant data stored in the database since these each table can be joined with other tables to form more than one business entity. Reducing data redundancy means that same piece of data will only be stored in the database once. By having the data stored only once, the problem of multiple copies disappears when data changes. For DSS, the main problem with

OLTP schema is columns that describe entities will be distributed among a number of tables which will then need to be joined together in order to get the required information for doing meaningful analysis. These joins that have to be performed can each require a significant amount of processing by the database system.

- Because of normalization, answering simple business questions will require joining large number of tables. Multiple join paths might exist between any set of tables as tables share multiple keys and can be joined in several different ways. Hence business users must determine how to join the tables.

- Cryptic naming conventions [MD94] and data formats. One of the primary reasons why business users cannot read normalized schema is that it encourages cryptic, shorthand naming conventions for columns and tables.

Hence the data warehouse uses highly denormalized schema to provide instant access without having to perform a large number of joins which results in poor performance. There are mainly two types of schema used for data warehouse: (1) star schema, and (2) snowflake schema. The star or snowflake schema design represents data as an array where each dimension is a subject around which analysis is performed.

4.2 Star Schema

The name star schema [SP94] comes from the pattern formed by the entities and relationships when represented as an entity-relationship diagram (ERD). Typically, it has a single object in the middle radially connected to other surrounding objects like a star as shown in Figure 4.1. The table in the middle is called "fact table", which contains compound primary key, with one segment for each "dimension", and additional columns of additive, numeric facts. There will be a single "dimension" table (for each dimension) with a generated key and a level indicator that describes the attribute level of each record. For example in Figure 4.1, for STORE dimension, record might refer to stores, districts or regions. Hence it is necessary to assign a level number to records (such as 1=store,

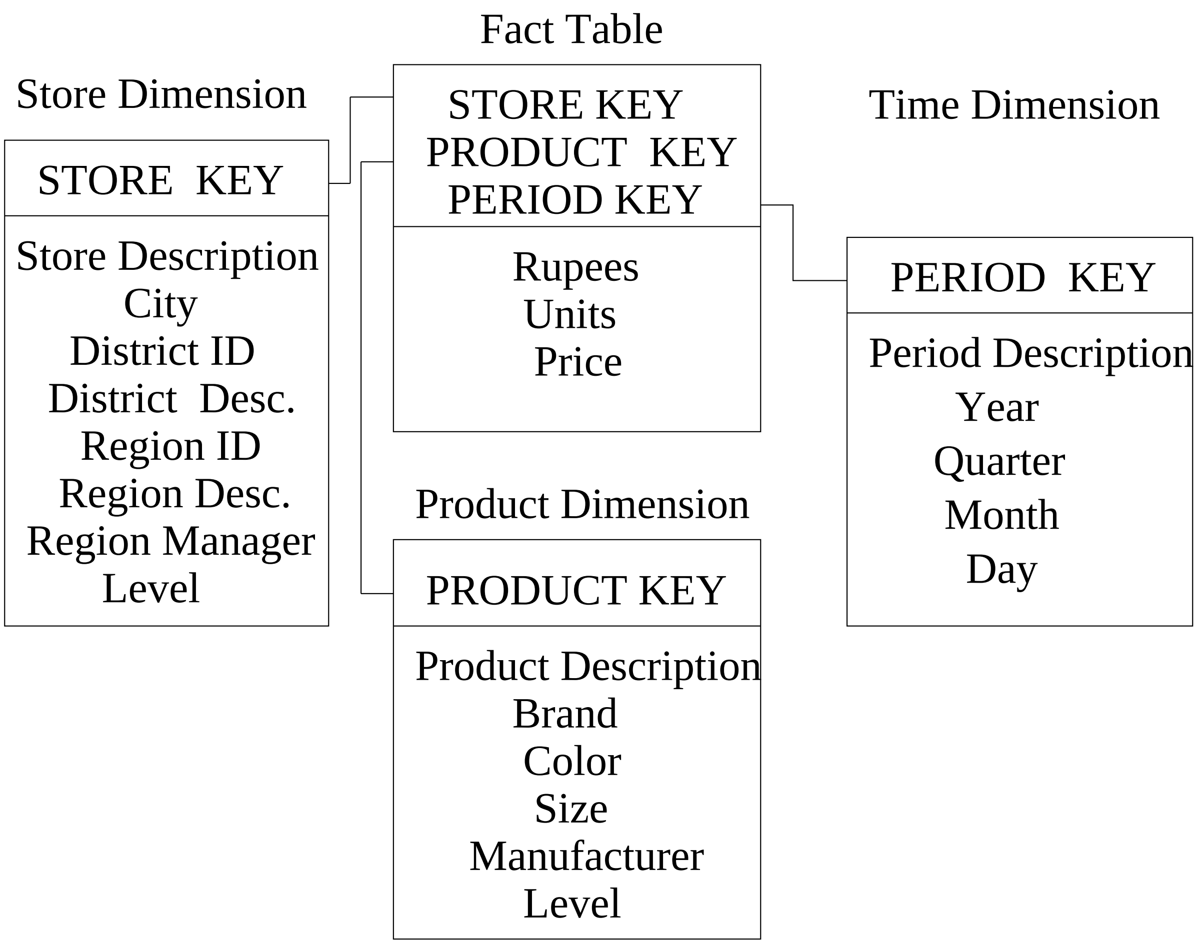
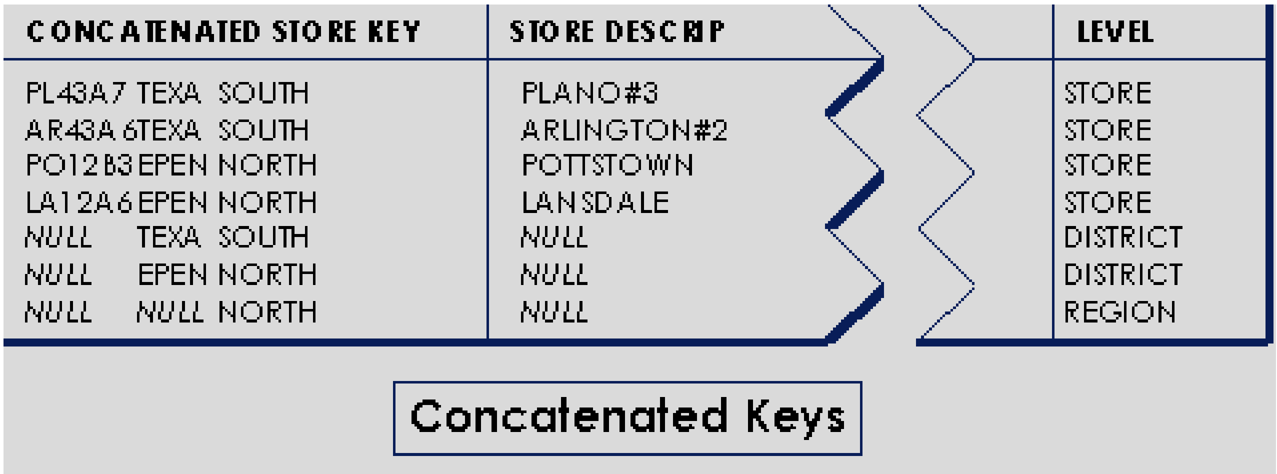


Figure 4.1: A typical star schema for sales database

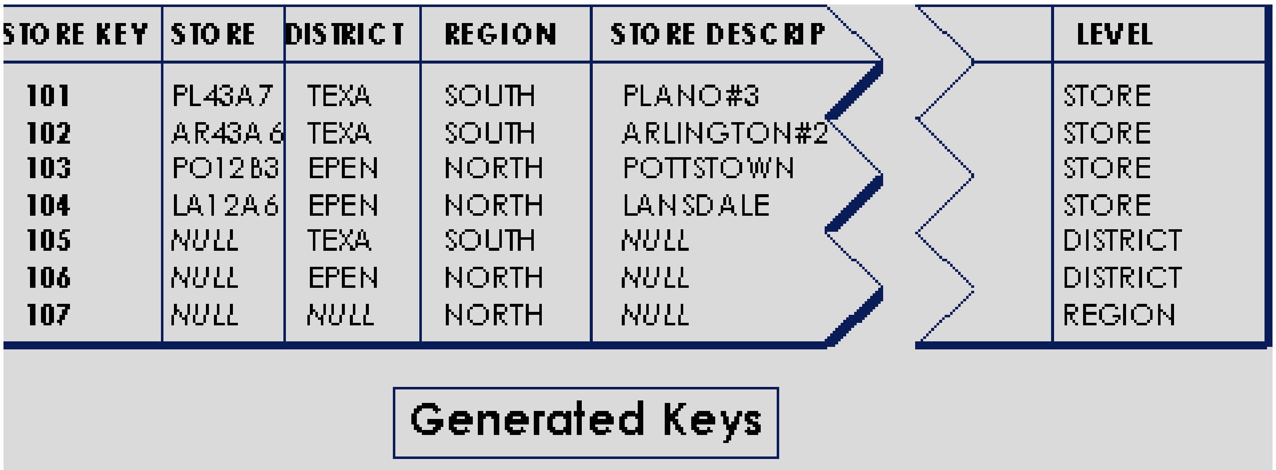
2=district, 3=region) by putting values in the "level" column. The single fact table will contain detail data, such as sales dollars, for a given store, for a given product in a given time period.

Since star schema will have only one key from each dimension, generating a key for a dimension is necessary. One approach is to form a compound key by concatenating all the keys into a single key but the size of the key will be large and nulls can be confusing. As shown in Figure 4.2, compound key for the STORE dimension is generated by concatenating three key fields Store Description, District Description and Region Description. Hence the best solution is to drop the use of "meaningful" keys altogether and instead generate the smallest possible key that will insure the uniqueness of each record. Integers



CONCATENATED STORE KEY	STORE DESCRIP	LEVEL
PL43A7 TEXA SOUTH	PLANO#3	STORE
AR43A6 TEXA SOUTH	ARLINGTON#2	STORE
PO12B3 EPEN NORTH	POTTSTOWN	STORE
LA12A6 EPEN NORTH	LANSDALE	STORE
NULL TEXA SOUTH	NULL	DISTRICT
NULL EPEN NORTH	NULL	DISTRICT
NULL NULL NORTH	NULL	REGION

Figure 4.2: Compound key generation for dimension table by concatenating the keys



STORE KEY	STORE	DISTRICT	REGION	STORE DESCRIP	LEVEL
101	PL43A7	TEXA	SOUTH	PLANO#3	STORE
102	AR43A6	TEXA	SOUTH	ARLINGTON#2	STORE
103	PO12B3	EPEN	NORTH	POTTSTOWN	STORE
104	LA12A6	EPEN	NORTH	LANSDALE	STORE
105	NULL	TEXA	SOUTH	NULL	DISTRICT
106	NULL	EPEN	NORTH	NULL	DISTRICT
107	NULL	NULL	NORTH	NULL	REGION

Figure 4.3: Compound key generation for dimension table using integers

are most efficient for generating keys. This method for generating key gives highest possible performance as the size of the integer is small and also it requires very low maintenance as the data warehouse matures.

Advantages:

The advantages for star schema are as follows: It is easy to understand as data is organized around subjects. Since data is in highly denormalized format, browsing through data is easy. Defining hierarchies is easy. Few joins are needed to get the information about all the columns. Maintenance required for the data warehouse is low because of use of integers for compound key generation.

Disadvantages:

Since data is not normalized, there is lot of redundancy. Hence representation requires more space as compared to the snowflake schema. Also, because of redundancy, various consistency problem may occur if the data is updated.

4.3 Snowflake Schema

The snowflake schema [HSB98], shown in Figure 4.4, is an extension of the star schema where each point of the star explodes into more points. In this schema, the dimension tables are more normalized as compared to the star schema. The advantages provided by the snowflake schema are improvements in query performance due to minimized disk storage for the data and improved performance by joining smaller normalized tables rather than large denormalized ones. Many consistency problems (due to updates) are also solved because of normalization. The snowflake schema also increases the flexibility of the application because normalization lowers the granularity of the dimensions. However, since the snowflake schema has more tables, even browsing through the data requires joining the tables to get the complete information, hence performance is poor. It can also give poor performance for the aggregation queries if the data storage has very low redundancy.

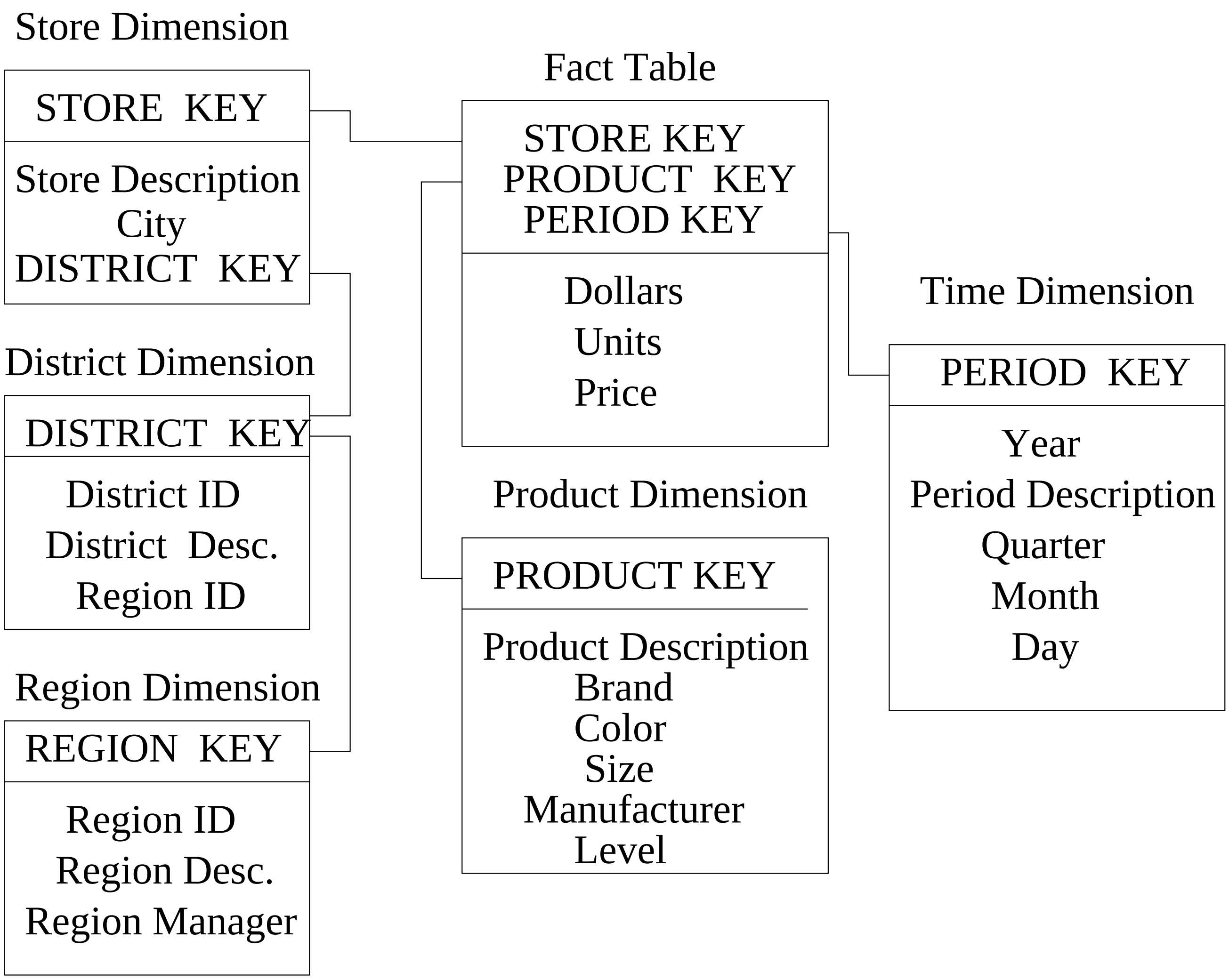


Figure 4.4: A typical snowflake schema for sales database

Chapter 5

IBM DB2

5.1 Introduction

In this chapter, we will give an overview of IBM's DB2 database system which has been used as a back end for our software development and experiments. We will present the storage structure of DB2 and its call level interface. DB2 is an abbreviation for Database 2. It is a relational database management system which runs on a variety of operating systems and has SQL as its query language. It has the following features [CD94] [EN94]:

- Transaction processing.
- Concurrency control.
- Buffer management.
- Security and authorization.
- Remote access facility.
- Query management facilities (QMF).
- Embedded and dynamic SQL.
- Application programming interface (API).

- Call level interface (CLI).
- Distributed database services.
- Web enabled with java support.
- Multimedia capable with image, audio, video and text support.

We have used DB2's CLI for software development.

5.2 Storage Structure

In this section, we will examine the way DB2 physically represents the data in storage. A database in DB2 is a collection of logically related objects. Figure 5.1 is a schematic representation of DB2's principal storage objects and their interrelationships. It can be interpreted as follows:

- The total collection of stored data is divided up into a number of disjoint databases – several users databases and several system databases. System databases contain various catalog and configuration information.
- Each database is divided up into a number of disjoint *spaces* – several **table spaces** and several **index spaces**, in general. A "space" corresponds to page sets (collection of fixed size pages).
- Each table space contains one or more **stored tables**. A stored table is the physical representation of a base table; it consists of a collection of **stored records**, one for each row of corresponding base table. Each stored table corresponds to a file, though very large tables might be partitioned across several stored files.
- Each index space contains exactly one **index** (a B-tree). A given index must be wholly contained within a single index space. A given stored table and all of its associated indexes must be wholly contained within a single database.

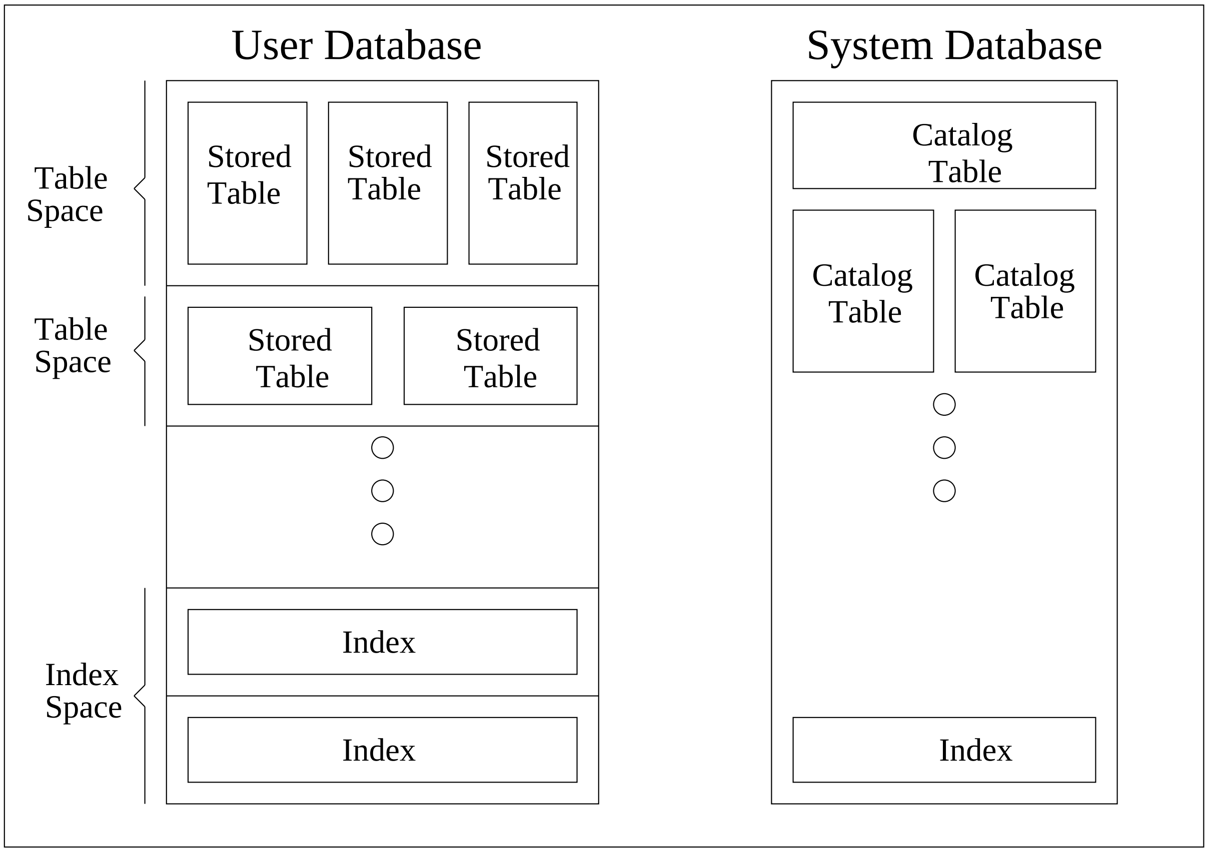


Figure 5.1: The major storage objects of DB2

- Each stored record must be fully contained within a single page; however, one stored table can be spread over several pages and one page can contain stored records from several stored tables.

5.3 The System Catalogs

In DB2 the tables called *catalog* are accessible via SQL for the database administrator or other authorized users. The DB2 system catalog contains a variety of information including the definitions of base tables, views, indexes, applications, users access privileges and application plans. These descriptions are referred to by the system in order to perform certain tasks; for example, during query optimization, the bind component accesses the catalog to obtain index information. DB2 takes a uniform approach to store user data and the catalog: both are stored as tables. Some of the important tables are:

1. **SYSCAT.TABLES:** This has one entry for every base table in the system. The information for each table contains, among other things, its name, the name of its creator and the total number of columns it contain.

2. **SYSCAT.COLUMNS:** This has one entry for every column defined in the system. For every column name, the name of the table to which it belongs, its type and other information are stored. The same column name may appear in multiple tables.

3. **SYSCAT.INDEXES:** This contains for each index the name of the index, the name of the indexed table, the name of the user who created the index and so on.

5.4 DB2 Call Level Interface

DB2 **CLI** is a programming interface that C or C++ applications can use to access DB2 databases [RM97]. DB2 CLI is based on the Microsoft **Open Database Connectivity (ODBC)** specification and the ISO CLI standard. When application uses DB2 CLI, dynamic SQL statements are passed as function arguments to the database manager for processing. DB2 CLI is an alternative to embedded dynamic SQL, providing another way to access DB2 databases using dynamic SQL. The application will usually contain the following parts:

Initialization:

Initialization allocates environment and connection handles and connects application to the database. A handle is a variable that refers to a data object controlled by the DB2 CLI. There are environment, connection, statement and descriptor handles. Using handles frees the application from having to allocate and manage global variables or data structures such as the SQLCA used in embedded SQL applications.

Transaction Processing:

Transaction processing consists of the five steps:

1. Allocate statement handle before any SQL statements can be executed. A statement handle refers to the data object that contains information about an SQL statement managed by DB2 CLI. This includes information such as dynamic parameters, cursor information, result values and status information.

2. Prepare and execute SQL statements using one of the two methods. The transaction starts implicitly with the first access to the databases.

   - Prepare then execute, which splits the preparation of the statement from the execution. This method is used when a statement will be executed repeatedly, usually with different parameter values.
   - Execute directly, which combines the prepare and execute steps into one. This method is used when a statement will be executed only once.

3. Process the results of the SQL statements.

   - Receive Query Result: If the statement is a query, fetch each row of the result set using **SQLFetch**.
   - Update Data: If the statement is modifying data, application needs only to check for diagnostic messages. Applications can use **SQLRowCount()** to get the number of rows affected by the SQL statement.
   - Free the statement handle ending processing for that statement.
   - Commit (complete) or roll back (undo) the transaction. The transaction ends at this point.

Termination:

Termination disconnects application from the database and frees the handles.

5.5 Advantages of DB2 CLI

1. **Portability:** DB2 CLI applications use a standard set of functions to pass SQL statements to the database. Hence only compiling and linking DB2 CLI applications

is needed before running them.

2. **No Binding:** There is no to need bind individual DB2 CLI applications to each database they access. Applications only need to bind the bind files that are shipped with DB2 CLI once for all.

3. **Extending fetch and input:** DB2 CLI functions allow to retrieve multiple rows in the database into an array with a single call. They can also execute a SQL statement many times using an array of input variables.

4. **Extended data conversion:** DB2 CLI automatically converts data between SQL and C data types. For example, fetching any SQL data type into a C char data type converts it into a character-string representation.

5. **Consistent interface to catalog:** The form of database catalogs can vary among systems. DB2 CLI provides consistent interface to query catalog information about tables, columns, foreign keys, primary keys and user privileges. This shields the application from catalog changes across releases of database servers and from difference among database servers. Writing catalog queries that are specific to a particular server or product version is not needed.

Chapter 6

Performance Evaluation

6.1 Introduction

In this chapter, we will analyze the performance of cube algorithms and will compare the performance of various data warehouse schemas. In the first part, we will compare the performance of our implementation of cube algorithm with that used in DB2. In the second part, we will compare the performance of the star and snowflake schemas for various values of data repetitions.

6.2 Performance of the Cube Algorithms

In this section we compare the performance of our cube query with the DB2's implementation. Our cube query uses the cube algorithm PipeSort mentioned in Chapter 3. The relational table used for these experiments has six columns and the size of a tuple is 167 bytes. A six dimensional cube query on a Student relation is used for experiments. The query is as follows:

SELECT Student_key, Course_key, Time_key, Advisor_key,
Mess_key, Hostel_key, **AVG**(Grade)

FROM Student

GROUP BY CUBE (Student_key, Course_key, Time_key, Advisor_key, Mess_key, Hostel_key)

The results of the query are shown in Table 6.1. Table 6.1 shows the time taken by the two implementations as well as the temporary space needed for query with respect to database size. As we can see from Table 6.1 that our implementation is performing better than DB2's implementation by 10-20%. For small databases, the improvement is around 10-15% but for large databases it is in the range of 15-20%. For small database, improvement is less because large number of tuples can be kept in the main memory and aggregation can be done in the memory itself and only few disk accesses are required. As the database gets bigger the cost of sorting the tuples becomes significant which

Table 6.1: Comparison of cube query

Index	No. of tuples (Thousand)	DB2 time (Min : Second)	Our time (Min : Second)	DB2 temporary space (MB)	Our temporary space (MB)
1	10000	0:40	0:35	2	0.5
2	25000	1:20	1:09	13	0.75
3	100000	7:15	4:25	50	1.5
4	200000	9:57	8:33	110	4
5	500000	21:48	17:28	242	7
6	1000000	44:45	35:18	459	19

results in increased running time of queries. Our implementation performs better because for computing smaller aggregates, it reuses the sorting done by previously computed aggregates and avoids additional sorting. Another thing worth noting is the temporary space needed for computing queries. As mentioned in Chapter 2, if the cardinality of the N attributes are $C_1, C_2, ..C_N$ then the cardinality of the resulting cube relation is $\Pi(C_i + 1)$. Hence for large databases having high cardinality attributes total number of tuples in the result of cube query is very high. Hence the temporary space required for sorting the tuples and storing the result is very high. As it can be seen in row 6 of Table 6.1 that DB2 is using very large amount of temporary space (459 MB) for processing cube query. Out of 459 MB, some space is needed to store the result of query (roughly

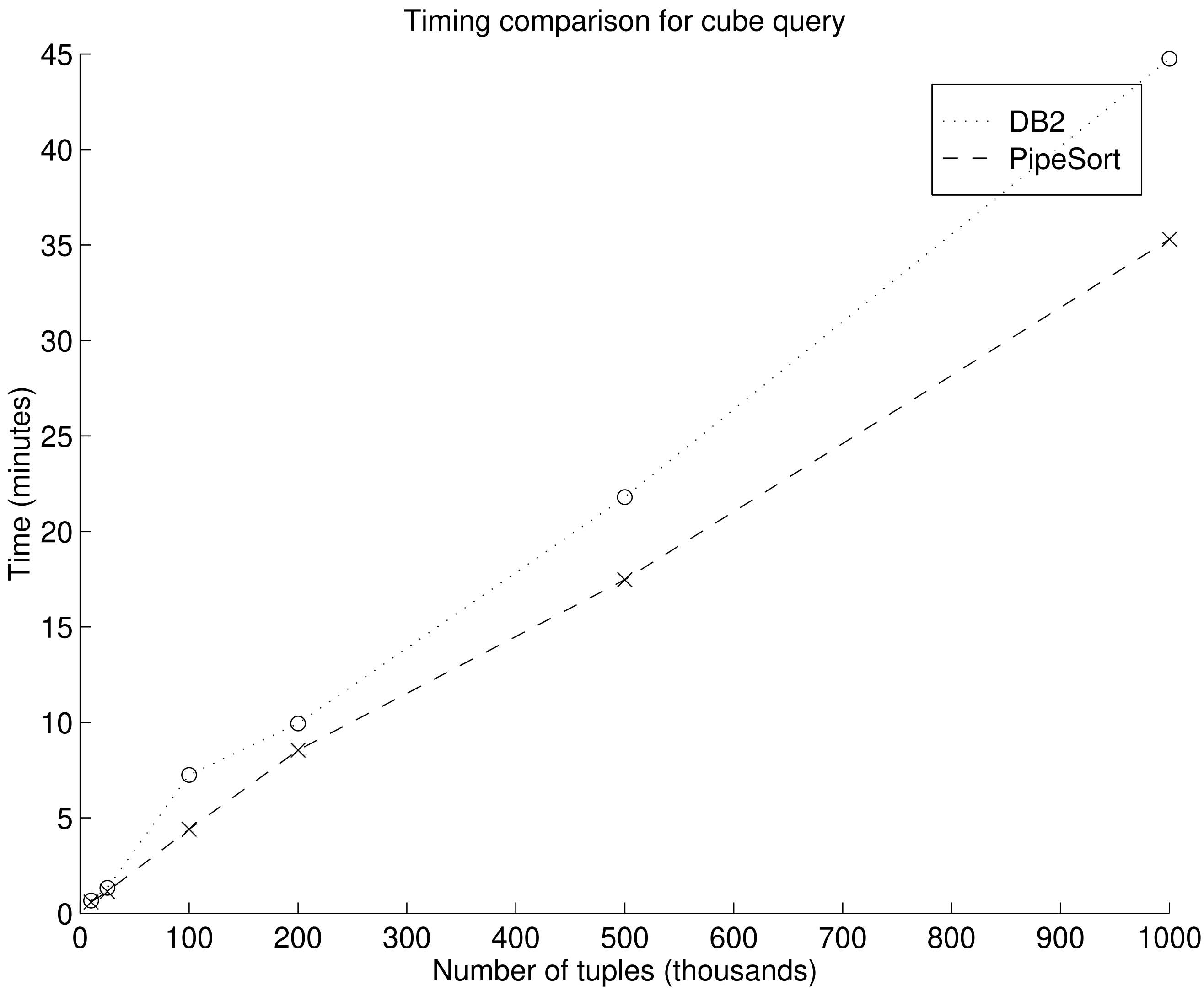


Figure 6.1: Timing comparison of DB2 and our implementation for cube query

around 95 MB) so that the application program can later fetch the result records. But apart from that, rest of space is used for processing the query (e.g. sorting the tuples). Our implementation does not need this amount of temporary space because (1) we are not storing the result of cube query on any temporary space and we output the tuple as soon as it is computed., (2) PipeSort makes several optimizations (smallest parent, share sorts, etc.) therefore when an aggregate is computed all its subsets (which require the same sort order) are computed in the same sort. It avoids additional sorting which results in lesser number of disk accesses and less temporary space requirement. Though DB2 is using large temporary space for executing query but the timing difference is still not large, which essentially means that some kind of optimizations have been made into it at the expense of space. We are not aware of the techniques used in DB2 as there is no public documentation anywhere. Figure 6.1 shows the timing for DB2 and our implementation which clearly shows improvement for large database sizes.

6.3 Performance Comparison of Star and Snowflake Schema

In this section we will compare the performance of the star schema with the snowflake schema for various values of data repetitions. The database used for experiments has an organization similar to a typical sales database. It has a centralized fact table and various dimension tables. The star schema for this database is shown in Figure 6.2.

The centralized fact table has keys from various dimensions and other attributes of interest (e.g price, dollars). The dimension tables are Store, Product and Time. Store dimension attributes are Store Name, City Name, Region Name etc.. Various attributes (Dim1, Dim2, etc.) are used to create a long tuple so that the snowflake representation of Store dimension has a long chain of attributes. The snowflake schema for the database is shown in Figure 6.3. In creating snowflake schema, only attributes of Store dimension have been broken down into smaller dimension tables. The number of tuples in the database are around 200000 [1] and it is kept constant for the experiments. The value of the attributes

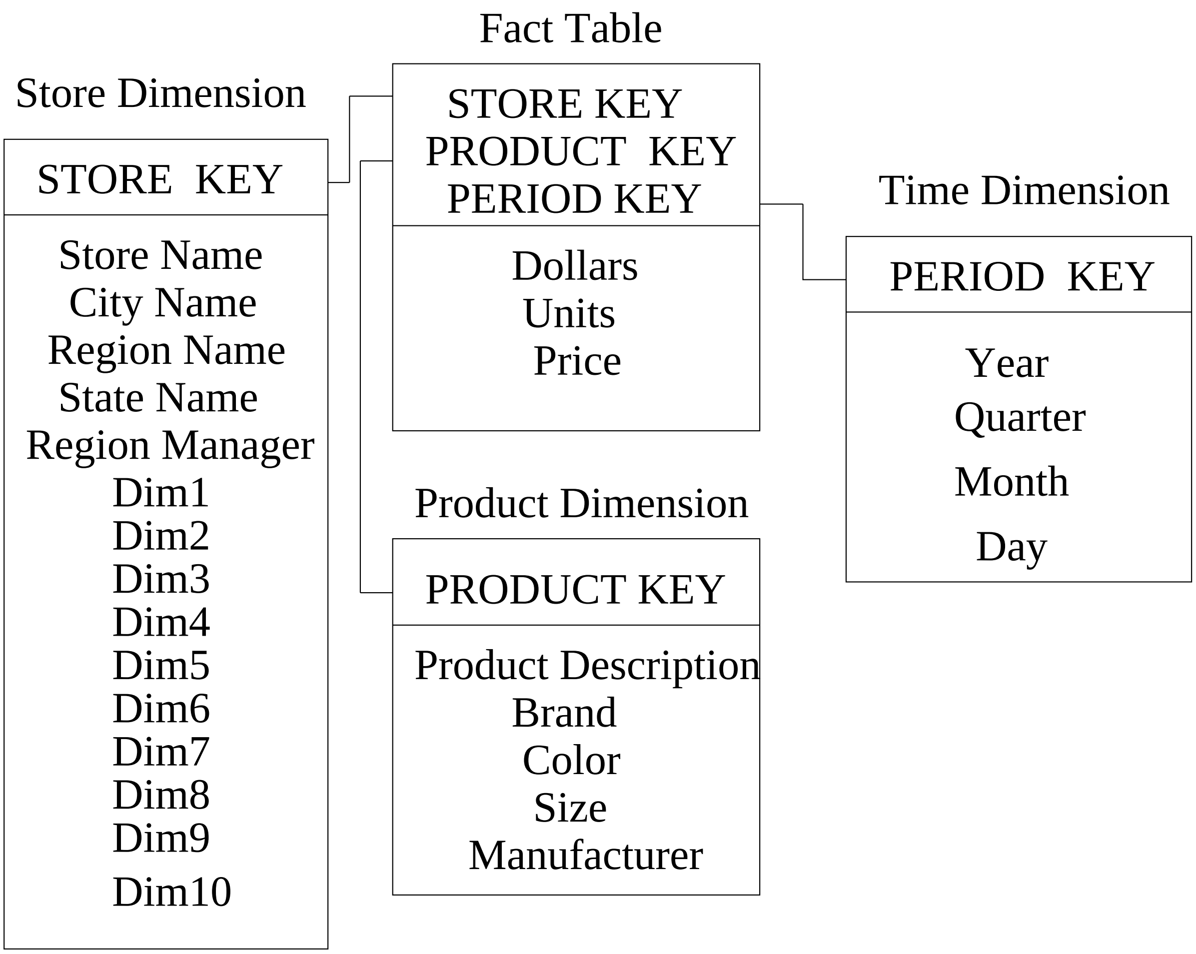


Figure 6.2: Star schema used for experiments

is changed during the experiments to vary the cardinality (repetitions). The query used for experiments involves group by over all the attributes of Store dimension. The results of the experiments are summarized in Table 6.2. As seen in the Table 6.2, star schema performs far better than snowflake schema in most of the cases. In general, the snowflake schema takes longer time than star schema because it takes more number of joins of dimension tables to get the same information. For non-repeating data, snowflake schema takes exceptionally long time whereas for highly repeating data it takes same amount of time or slightly less amount of time than star schema. This can be explained as follows:

[1] For snowflake schema, computing a query requires joining all the tables, which takes lot of temporary disk space hence we could not increase the database size to more than 200000 tuples.

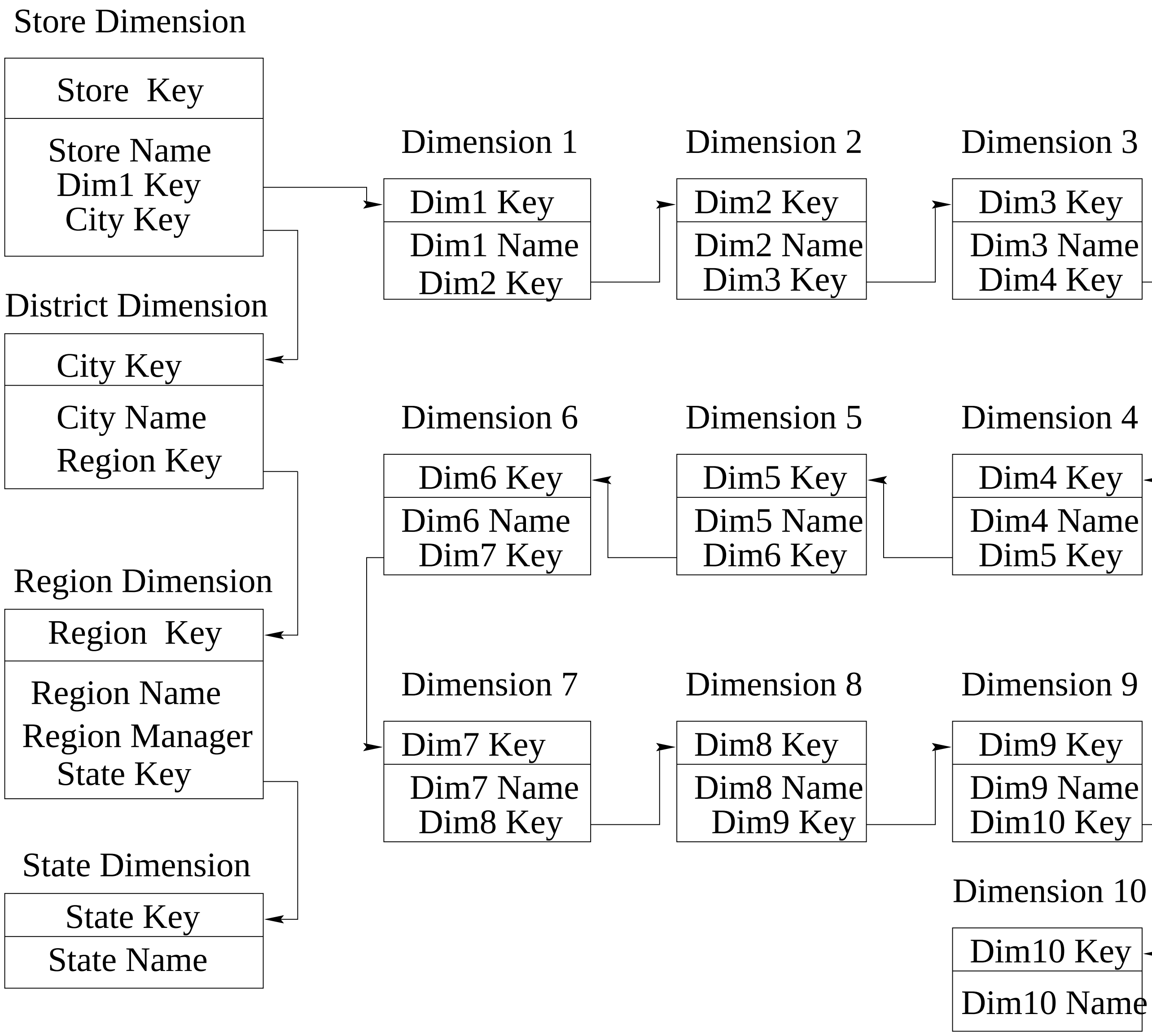


Figure 6.3: Snowflake schema used for experiments

Table 6.2: Comparison of star and snowflake schema

Index	Data repetitions percentage	Star schema time (Hour : Min : Second)	Snowflake schema time (Hour : Min : Second)
1	0	0:2:31	4:23:41
2	20	0:2:15	2:20:37
2	40	0:1:55	1:05:24
3	60	0:1:32	0:39:22
4	80	0:1:16	0:15:04
5	90	0:1:10	0:6:47
6	100	0:1:02	0:1:00

Figure 6.5 shows the star and snowflake schema representation for one set of data. This data doesn't have any repetitions hence representing it in snowflake requires additional space for storing 6 keys. Hence snowflake representation for this kind of data requires more space and queries take longer time for computing the joins of various tables. Whereas for star schema only browsing through the whole table is needed. Figure 6.6 shows the star and snowflake schema representation for another set of data. This set of data has repetitions therefore snowflake representation is more compact as it avoids duplicating long character attributes in all records. But as mentioned before, retrieving information in snowflake representation requires additional join and time. As Figure 6.4 shows that star schema doesn't perform badly for very highly repeating data set and its performance is very close to snowflake which essentially means that cost of joins involved is significant as compared to browsing large repeating attributes. The performance of snowflake is better than star schema when attributes length are very high (say more than 500 bytes) because snowflake schema encourages compact representation. We have done some experiments with tuples size over 1000 bytes and observed that the snowflake schema performs better than the star schema but this improvement is not more that 5-10%.

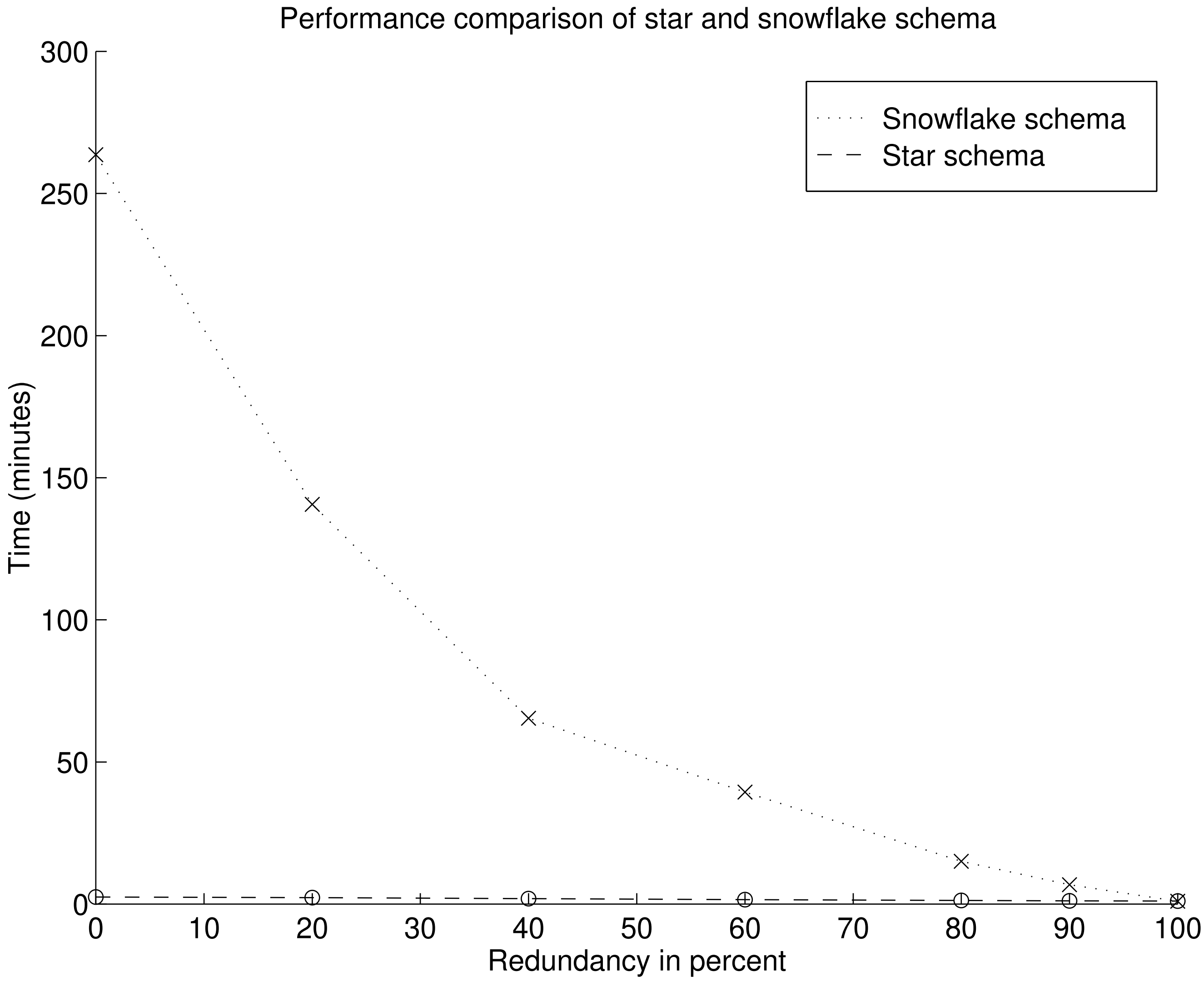


Figure 6.4: Plot of time vs. data repetition for star and snowflake schema

6.4 Timing Measurements

Timing measurements for the queries are done using *time* command available on **ULTRA-SPARC1**. It uses *time()* system call. *time()* system call returns the value of time in seconds since 00:00:00 UTC, January 1, 1970. *time* command uses *time()* system call at the beginning and at the end of program and calculates the difference of values which is the time elapsed.

Brand	Manufacturer
Lux	Hindustan Lever Ltd.
Liril	Procter & Gamble
Vicco	Dabur India Ltd.

(a) Star Schema

Brand	Key
Lux	1
Liril	2
Vicco	3

Key	Manufacturer
1	Hindustan Lever Ltd.
2	Procter & Gamble
3	Dabur India Ltd.

(a) Snowflake Schema

Figure 6.5: Representation of typical non repeating data in star and snowflake schema

Brand	Manufacturer
Lux	Hindustan Lever Ltd.
Pears	Hindustan Lever Ltd.
Dove	Hindustan Lever Ltd.

(a) Star Schema

Brand	Key
Lux	1
Pears	1
Dove	1

Key	Manufacturer
1	Hindustan Lever Ltd.

(a) Snowflake Schema

Figure 6.6: Representation of typical repeating data in star and snowflake schema

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we have investigated some of the issues involved in the design of a data warehouse. Since data warehouses are used for OLAP and OLAP queries heavily use aggregation, it is important that the response time for queries should be low for interactive data analysis applications. We have discussed efficient ways of computing cube query which is a N dimensional generalization of group-by query. As computing the cube involves computing 2^N group-bys, the straightforward approach of computing union of 2^N group-bys is very inefficient and takes very long time to complete. This delay is unacceptable in DSS environments, as it severely limits productivity. We have shown that the sort-based algorithm PipeSort, which makes several optimizations (using caching, sorting etc.), require few disk accesses and runs faster. We have compared the performance of our implementation of the algorithm with the commercial database DB2's implementation of cube query and showed that our implementation is 15-20% faster than DB2. It is also observed that our implementation requires very small amount of temporary space needed for executing query as compare to DB2. This is because our implementation attempts to compute a group-by from its smallest parent group-by or from the parent group-by which is in still in memory hence avoiding the additional disk accesses.

We have also looked into the various schema alternatives for a data warehouse and showed that for most of the cases the performance of star schema is good. Since all the data in star schema is organized in single dimension table, it is easy to browse through data which might be of interest to many business users. Snowflake schema is highly normalized and can be used when attributes are very long and redundancy is very high but at the expense of poor browsing performance as the data is distributed across many dimension tables.

7.2 Future Work

There are various ways in which work presented in this thesis can be extended. We have only focussed on a class of algorithm called sort-based algorithms for computing the cube query. We haven't considered the hash-based algorithms for computing cube query which can be implemented and compared with sort-based algorithms for various database sizes, redundancy and schema. This will help in deciding what algorithm to use for a particular kind of data distribution and database size. As shown in Figure 1.1 a complete data warehouse has various components. In this thesis, we have only focussed on the back end of the data warehouse. We have assumed that the data is already in the clean format but operational databases often have major inconsistencies in data representation. Hence additional work is needed before a data warehouse can be populated. **Data extraction** component extracts the necessary data from various legacy sources which is to be stored in the data warehouse. **Data cleaning** component dictates consistency in the format of data which is collected from various sources so that it can be merged under a unified schema. These components are the integral part of a data warehouse system hence future work can also focus on these parts. We have also not focussed on the end user tools which actually work as front end for the data warehouse. End user tools can be visualization softwares which can present the data in visual format where a user can actually see the data in form of a multidimensional data cube. User should also be able to perform operations like "**drill down**", "**roll up**" and "**slicing**" on the cube. This kind of operations give users freedom to analyze the the data at the desired granularity. **Data mining** softwares can also be developed which can be used as front end to data warehouse where a user can actually discover patterns in the data and analyze the trends.

Bibliography

[BS97] A. Berson, S.J. Smith, "Data Warehousing, Data Mining & OLAP". McGraw-Hill, 1997.

[CD94] C.J. Date, "An Introduction to Database Systems". Sixth Edition, Addition Wesley, 1994.

[CS94] S. Chaudhuri, K. Shim, "Including Group By in Query Optimization". *In Proc. of Twentieth International Conference on Very Large Databases (VLDB),* pages 354-366, Santiago, Chile, September 1994.

[EN94] R. Elmasri, S.B. Navathe, "Fundamentals of Database Systems". Second Edition, The Benjamin Cummings Publishing Company, Inc., 1994.

[GBLP96] Jim Gray, Adam Bosworth , Andrew Layman, Hamid Pirahesh, "Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab and Sub-Totals". *In Proc. of the 12th International Conference on Data Engineering,* pages 152-159, 1996.

[GRA93] G. Graefe, "Query Evaluation Techniques for Large Databases". ACM Computing Surveys, pages 73-170, June 1993

[HNSS95] P.J. Haas, J.F. Naughton, S. Seshadri, L. Stokes, "Sampling-based Estimation of the number of distinct values of an attribute". *In Proc. of the Eighth International Conference on Very Large Databases (VLDB)*, pages 311-22, Zurich, Switzerland, September 1995.

[HRU96] Venky Harinarayan, Anand Rajaraman, Jeffrey Ullman, "Implementing Data Cubes Efficiently". *In Proc. of the ACM-SIGMOD Conference,* 1996.

[HSB98] Jennie Hou, Cailean Sherman, Terry O'Brien, "Data Warehousing On HP3000 Using IMAGE/SQL - A New Alternative!". Hewlett Packard WhitePaper, 1998.

[IWG96] W. H. Immon, J. D. Welch, Katherine L. Glassey, "Managing the Data Warehouse". John Wiley and Sons, 1996.

[KS91] Henry F. Korth, Abraham Silberschatz, "Database System Concepts". Second Edition, McGraw Hill, 1991.

[MD94] Marc Demarest, "Improving Data Legibility in Decision Support System". DBMS Magazine, 1994. Available from *http://www.hevanet.com/demarest/marc/schema.html.*

[MS93] J. Melton, A.R. Simon, "Understanding The New SQL, A Complete Guide". Morgan Kaufmann Publishers, 1993.

[PS82] C.H. Papadimitriou and K. Steiglitz, "Combinatorial Optimization: Algorithms and Complexity". Chapter 11, pages 247-254, Prentice Hall, Eaglewood Cliffs, N.J., 1982.

[RK96] Ralph Kimball, "The Data Warehouse Toolkit". John Wiley and Sons, 1996.

[RM97] Reference Manual, "Road Map to DB2 Programming". IBM DB2 Universal Database, Version 5, IBM Corporation, 1997.

[SAG96] Sunita Sarawagi, Rakesh Agrawal, Ashish Gupta, "On Computing the Data Cube" . Research Report RJ 100026, IBM Almaden Research Center, San Jose, California, 1996. Available from *http://www.almaden.ibm.com/cs/quest.*

[SB97] Stroustrup, B, "C++ Programming Language". Third Edition, Addision Wesley, 1997.

[SP94] Stephen Peterson, "Stars: A Pattern Language for Query Optimized Schema". Sequent Computer Systems WhitePaper, 1994.

[TPCD93] J. Gray, "The Benchmark Handbook for Database and Transaction Processing Systems". Second Edition, Morgan Kaufmann Publishers, 1993.