Maintenance-Cost View-Selection in Large Data Warehouse Systems: Algorithms, Implementations and Evaluations

Choi Chi Hon

A Dissertation Submitted in Partial Fulfilment

of the Requirements for the Degree of

Master of Philosophy

in

Systems Engineering and Engineering Management

June 2003

Abstract

In order to efficiently support a large number of online analytical processing (OLAP) queries, a data warehouse always precomputes some of the OLAP queries and stores them as materialized views. Maintenance cost view selection problem is how to select a set of materialized views to minimize the total processing cost for OLAP queries under some constraints, such as the maintenance cost. This problem has recently been received significant attention. Several greedy/heuristic algorithms were proposed. However, the quality of the greedy/heuristic algorithms has not been well analyzed. In this thesis, the greedy/heuristic algorithms have been reexamined in various settings to provide readers with insights on the quality of these heuristic algorithms.

Besides, a new evolutionary algorithm is proposed for the maintenance cost view selection problem. Constraints are incorporated into the algorithm through a stochastic ranking procedure without penalty functions. The experimental results show that the stochastic ranking can deal with constraints effectively. The algorithm can find a near-optimal solution and scale with the problem size well.

As time passes, new queries may not be answered by the existing materialized views, a set of new views are selected to be materialized, and may replace the existing materialized views. This is known as dynamic view management problem. A dynamic predicate-based partitioning approach is proposed. It can support different kinds of OLAP queries in an existing relational database system. Encouraging results are obtained which indicate that the approach is highly feasible.

摘要

爲了可以有效而快捷地支援大量的連線分析處理查詢，數據倉庫通常都會預先計算一些這類的連線分析處理查詢並把視圖實物化，稱之爲物化視圖。其中一個最重要的問題是如何在特定的條件之下，例如：維護的費用，選取一系列的物化視圖去把處理連線分析處理查詢視圖的費用減至最低。近來這個問題已經引起高度的關注。有很多貪心／啓發式算法被提出，可是他們的素質未被很好地分析。在此，我們會在不同的設定環境下重新測試這些貪心／啓發式算法，從而令用户可以深入了解到這些貪心／啓發式算法的素質。

與此同時，針對這個問題我們還提出一個全新的有限制的演化式算法。限制的條件將透過隨機排列過程融合到我們的演化式算法中，當中並不包含懲罰函數。實驗結果顯示我們的演化式算法可以接近最佳解，另外也可以處理按比例增大的問題。

除此之外，我們亦研究動態視圖管理問題。隨著時間過去，現有的物化視圖未必可以回答新的查詢，我們必須另外再物化一些視圖和更新舊的物化視圖。我們提出一個以謂詞爲主的動態分割方案，它可以支援廣泛的連線分析處理查詢。我們的研究重點側重於關係數據庫系統中作關係連線分析處理。實驗的結果説明了我們方法是高度可行的。

Acknowledgement

I would like to give my sincerest thanks to my supervisor, Professor Jeffrey Xu Yu, not only for his valuable advice, but also for his patient teaching, encouragement and support. He understands my strengths, weaknesses and personality, and leads me to the threshold of my own mind. His enlightenment is important and useful for my future. It is really difficult for him to supervise me, an amateur researcher. Without his help, this work can hardly be finished.

Besides, special thanks must be given to Gabriel and Paul for their fruitful suggestions and comments. They brought me lots of supports when I am in adversity, and lots of joys in my life.

Moreover, I would like to thank my best friends including Holmes Su, WaiIp, KunChung, Silvia, ChingFung, XiaoLei, GouGang and LiuZheng. They have shared their valuable time to play and chat with me, and gave me valuable suggestions. I would also like to thank Phyllis for supporting me technically.

Last but not least, I would like to thank Charles Chen and my family for their enduring supports and encouragements. Their loves and supports have made my life wonderful. My hearty thank is to Charles for his physically and mentally supports. During these two years, I did not spend enough time with them. After this work, it is high time for me to pay them back. I wish I can use the knowledge and experiences gained throughout these two years to serve them.

[illegible]

The Chinese University of Hong Kong Choi Chi Hon

June 2003

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Business landscape of the financial services is quickly evolving, and markets are much more competitive and dynamic than ever. Businesses in every segment of the industry realize that their corporate and customer databases are gold mines of information that could give them a critical edge by helping them to manage investment, map market development, identify new customer prospects, anticipate demands on banking services, and predict consumer preferences and habits. Data warehouse and online analytical processing (OLAP) have been successfully deployed in many industries such as manufacturing, retailing, financial services, transportation, telecommunications and health-care. They enable executives, managers and analysts to make better and faster decisions. The amount of potential data that needs to be maintained tends to be hundreds of gigabytes or terabytes in size. As OLAP queries in such a large warehouse involve hundreds of complex aggregate queries over a huge volume of data, it is not feasible to compute these queries by scanning the data sets from scratch each time, as the processing time is too long. Therefore, minimizing OLAP query processing time becomes critical since executives, managers and analysts need to make decisions in a short time.

In this chapter, the research problem for this thesis will be identified. A brief background and description of previous research work is given in Section 1.2. The purpose and significance contribution of this research will be established in Section 1.3. Finally, the thesis organization will be described in Section 1.4.

1.1 Maintenance Cost View Selection Problem

Precomputing OLAP queries has been widely used as a common technique in data warehouses, that is the selection of a set of views to materialize under certain resource constraints for the purpose of minimizing the total query processing cost. Most research work on the view selection problem is under *disk space constraint.* Disk space constraint represents that only a limited amount of disk space can be used to store the materialized views. The *disk space view selection problem* is to select a set of interrelated views that minimizes the total query response time under a given disk space constraint. However, in real applications, the constraint is the maintenance time incurred in keeping the materialized views up-to-date at a data warehouse. The *maintenance cost view selection problem* is to select a set of views to materialize in order to minimize the query response time under a constraint of maintenance time. For the disk space constraint problem, the benefit of a view that has been chosen will remain unchanged in the subsequence view selection process. It has the *monotonic property* and therefore greedy algorithms are applicable. However, for the maintenance cost view selection problem, the monotonic property does not hold. Besides, the maintenance cost view selection problem for a large multidimensional data warehouse in a dynamic changing environment has not been well studied.

1.2 Previous Research Works

Usually, according to the query statistics on a daily-basis, the most frequently accessed OLAP queries are selected as materialized views during night to effectively utilize the computation and storage resources. The materialized views are used to accelerate the OLAP queries in the following day. This kind of view selection are known as *static view selection problem.* Recent works [4, 63, 70, 73, 78] on the static view selection problem for data warehouse provide various frameworks and heuristics for selection of views in order to optimize the sum of query response time and view maintenance time without any resource constraint. [40] provides a greedy algorithm to select views to materialize in order to minimize the total query response time under disk space constraint or under a limited number of views. [33, 34] present approximation algorithms to select a set of views that minimizes the total query response time under a given space constraint. [36] initializes the maintenance cost view selection problem in terms of time constraint. They propose two heuristic algorithms which are exponential in nature and are not scalable.

However, ad-hoc queries evolve continuously over time in daytime. Some of the new incoming queries can be answered by the existing materialized views in the data warehouse while others may not. These new queries make static materialized views quickly become outdated. In other words, the static materialized views cannot fully support the dynamic nature of the decision support analysis.

Hence, dynamic materialized views management is highly desirable to fully satisfy users' ad-hoc queries, The *dynamic view selection problem* is to modify or replace the existing materialized views to answer continuously incoming ah-hoc queries during daytime.

[74] designs a dynamic data warehouse and proposes incremental algorithm to support incoming queries. [21] proposes a chunk-based scheme to support the dynamic queries. [45] introduces a dynamic view management system, Dynamat, which stores multidimensional range fragments. More details will be given in Chapter 2.

1.3 Major Contributions

In this thesis, the maintenance cost view selection problem on the static view selection issue has been considered as well as on the dynamic view selection issue.

The static view selection has recently been received significant attention. [36] proposes two heuristic algorithms: A*-heuristic and inverted-tree greedy, in which they state that the greedy algorithm that select views base on query benefit per unit maintenance cost can deliver an arbitrarily bad solution. In contrast, Liang et al. [51] design two algorithms: two-phase greedy and integrated greedy, which are on the basis of query benefit per unit maintenance cost. Liang et al. claim that the two algorithms are able to find feasible solutions in polynomial time, however, they do not provide any analytical and performance studies. To deal with the maintenance cost view selection problem, algorithms that provide a near optimal solution in polynomial are highly desirable. However, the arguments presented in [36, 51] are not consistent. The algorithms cannot be used without any systematic study on the quality. One of the contribution of this thesis is to investigate four above-mentioned heuristic algorithms in various settings. The valuable finding provides a view on how to use the heuristic algorithms and to design new algorithms.

The maintenance cost view selection problem is a NP-hard problem. The search space for possible materialized views may grow exponentially. Through studying the heuristic algorithms, a new constrained evolutionary algorithm is proposed instead of using any penalty functions. The evolutionary algorithm use a novel stochastic ranking procedure. It is the first time to adopt the the stochastic ranking technique to this specific problem. The extensive experimental studies show that the feasible solution can be easily found by the stochastic ranking approach. In addition, the new constrained evolutionary algorithm explores the search space better than the other existing algorithms. It scales well with the problem size.

Another major contribution of this thesis is that a dynamic predicate-based partitioning approach is proposed for the dynamic view management problem. The main advantage of this approach is that it is able to fully utilize the power of existing popular relational database systems and can support different kinds of complex OLAP queries. Extensive performance studies are conducted using TPC-H benchmark data on IBM DB2 and encouraging results are obtained which indicate that the approach is highly feasible.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 reviews the concept of data warehouses, OLAP systems and previous research work related to view selection problem. Chapter 3 formalizes the research problem. Chapter 4 discusses four existing heuristic algorithms and reports their performance. The new constrained evolutionary algorithm as well as the experimental results will be presented in Chapter 5. Chapter 6 considers the dynamic maintenance cost view selection problem and proposes a new dynamic predicate-based partitioning approach. Finally, Chapter 7 concludes this thesis.

Chapter 2

Literature Review

Today's markets are much more competitive and dynamic than ever. Business enterprises prosper or fail according to the sophistication and speed of their information systems, and their ability to analyze and synthesize information using those systems. Data warehousing and OLAP provide tools for business executives to systematically organize, understand and use their data to make strategic decisions. In this chapter, basic concepts, general architecture and previous research works on data warehouses and OLAP systems will be given as the background of the research problem in this thesis. Section 2.1 introduces data warehouse and its architecture, as well as the OLAP, ROLAP, MOLAP, data cube and query optimization. Emphasis are particularly put on the materialized views and view selection, in which they will be described in Section 2.2 and 2.3. Section 2.4 concludes this chapter.

2.1 Data Warehouse and OLAP Systems

Data warehouse and OLAP systems are essential elements of decision support, and become a focus in the database industry. In this section, data warehouses and its architecture, OLAP systems, multidimensional data model and how to optimize queries will be introduced.

2.1.1 What Is Data Warehouse?

Data warehouse can be defined in many ways. Loosely speaking, a data warehouse is a copy of transaction data specifically structured for querying and reporting [44] (Page 310). Data warehouse systems allow for the integration of a variety of application systems. According to Inmon on page 31 of the Building the Data Warehouse [41]: *A data warehouse is a "subject-oriented, integrated, time-varying, non-volatile collection of data that is used primarily in support of organizational decision making"*. These data are collected from different sources, such as marketing, finance and human resources, through the rapid growth of the World Wide Web on the Internet. Let us take a closer look at these four key features: subject-oriented, integrated, time-variant, and non-volatile.

- **Subject-oriented:** A data warehouse is focus on major subjects, such as products, customers and sales. It typically provides a simple and concise view around particular subject issues to help a decision maker to model and analyze the data. This is in contrast to operational systems, which deal with processes such as customer sales transactions.

- **Integrated:** Data are stored in a consistent format even they came from multiple heterogeneous sources. Data cleaning and data integration techniques are applied to ensure consistency.

- **Time-variant:** A data warehouse stores huge and long periods of time historical data, for example, past five to ten years' data. The data implicitly or explicitly associate with a point in time to provide information from a historical perspective.

- **Non-volatile:** The data does not change once it gets into the data warehouse. A data warehouse usually requires only two operations on data accessing: initial loading of data and access of data.

The aim of a data warehouse is to support decision-making based on historical, summarized and consolidated data. The target users of data warehouse are knowledge workers, such as managers and executive analysts, who usually issue ad hoc and complex queries that require prompt response. The amount of data maintained in a data warehouse is huge in size, from hundred gigabyes to several terabytes. Upon such enormous amount of data collected from different sources, various business decisions need to be made in a few minutes, in order to cope with the rapid changes in different sectors of the market from time to time. Such timely manner requests the data warehouse to be able to answer OLAP queries efficiently, and be able to assist executives or managers to make a better and faster decisions. Typically, the data warehouse is maintained separately from the operational databases. This makes data readily accessible to the knowledge workers without interrupting the online operational systems.

2.1.2 What Is OLAP?

In order to analyze the trends and make reliable predications, a data warehouse often collects detailed data from one or multiple sources to support business analysis. Such analysis involves asking a large number of aggregate queries, and is called online analytical processing (OLAP). OLAP is computer-based technique used to analyze trends and perform business analysis using multidimensional views of business data [5]. Another good definition of the term OLAP is found in [17] as follows:

OLAP is a category of software technology that enables analysts, managers and executives to gain insight into data through fast, consistent, interactive access to a wide variety of possible views of information that has been transformed from raw data to reflect the real dimensionality of the enterprise as understood by the user.

2.1.3 Difference Between Operational Database Systems and OLAP

This section compares the difference between online transaction processing (OLTP) system and OLAP system. Han and Kamber [39] (page 43) give a comparison between OLTP and OLAP systems in Table 2.1.

The major features that distinguish between OLTP and OLAP are: (1) OLTP is a customer-oriented for detail transaction processing; OLAP is market-oriented and is used for analysis by knowledge workers. (2) OLTP typically adopts an entity-relationship data model; OLAP usually adopts either a star or snowflake model (will be discussed in the next section). (3) OLTP focuses mainly on current

Feature	**OLTP**	**OLAP**
Characteristic	operational processing	information processing
Orientation	transaction	analysis
User	clerk, DBA, database professional	knowledge workers (e.g., manager, executive analyst)
Function	day-to-day operations	long-term informational requirements, decision support
DB design	ER based, application-oriented	star/snowflake, subject-oriented
Data	current; guaranteed up-to-date	historical; accuracy maintained over time
Summarization	primitive, highly detailed	summarized, consolidated
View	detailed, flat relational	summarized, multidimensional
Unit of work	short, simple transaction	complex query
Access	read/write	mostly read
Focus	data in	information out
Operations	index/hash on primary key	lots of scans
Number of records accessed	tens	millions
Number of users	thousands	hundreds
DB size	100 MB to GB	100 GB to TB
Priority	high performance, high availability	high flexibility, end-user autonomy
Metric	transaction throughput	query throughput, response time

Table 2.1: Comparison between OLTP and OLAP system [39]

data; OLAP deals with the historical and consolidated data from heterogeneous sources.

2.1.4 Data Warehouse Architecture

A data warehouse is constructed by integrating data from multiple heterogeneous sources to support queries as well as report and help managers to make decision. [7, 8] provide an overview of data warehousing architecture and OLAP technology (Figure 2.1).

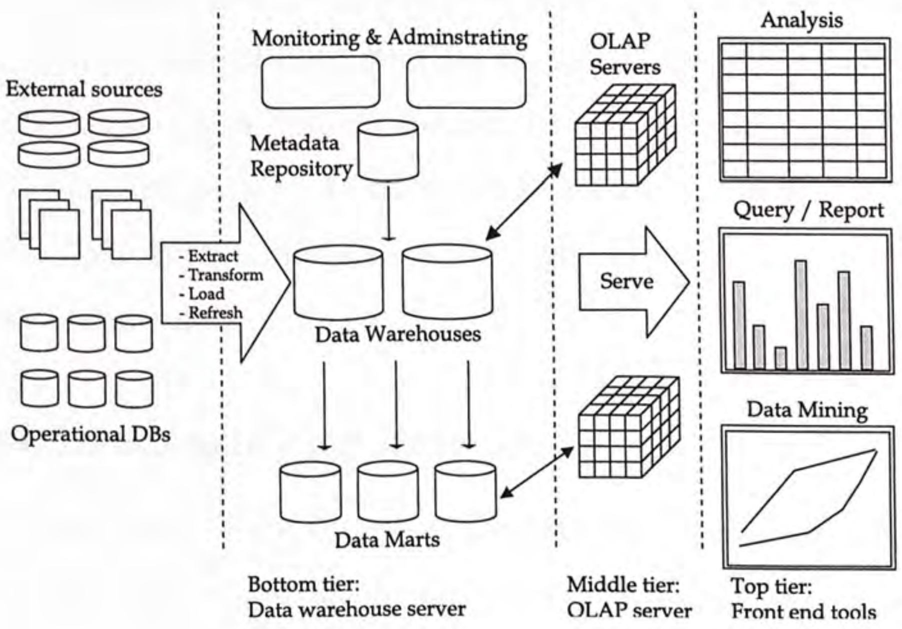


Figure 2.1: Data warehouse architecture [8]

Figure 2.1 illustrates a data warehouse architecture with three tiers: data warehouse server, OLAP server and front end tools. A data warehouse is often constructed by integrating data from multiple heterogeneous sources. Data from operational databases and external sources are extracted, transformed, loaded and refresh into data warehouse. Data are refreshed periodically to the data warehouse to reflect updates at the sources. In the data warehouse, there is a repository for storing and managing metadata, and tools for monitoring and administering the data warehousing system. Data in a data warehouse are stored and managed by an OLAP server, which is implemented either in Relational OLAP (ROLAP) or Multidimensional OLAP (MOLAP) model. An OLAP server is a high-capacity, multi-user data manipulation engine specifically designed to support and operate on multi-dimensional data structures [17]. ROLAP uses a relational or extended-relational DBMS to store and manage warehouse data. In contrast, MOLAP systems store their data as sparse arrays, which may consume more space compare with ROLAP, however, its implementation of operation is simpler and more efficient. OLAP server will be discussed in detail in the Section 2.1.8. The front end tools including query tools, report writers, analysis tools and data mining tools.

2.1.5 Multidimensional Data Model

In order to facilitate complex analyses and visualization, the data in a data warehouse are typically modeled multidimensionally. Data warehouses and OLAP tools are based on a multidimensional model which view data in the form of data cube. A multidimensional data model is defined by dimensions and facts. Figure 2.2 shows a multidimensional data. In a multidimensional data warehouse, it consists of a fact table and a collection of dimension tables [44]. Data cells are arranged by the dimensions of the data. The dimensions together are typically organized around an object of analysis which called measure. For example, in

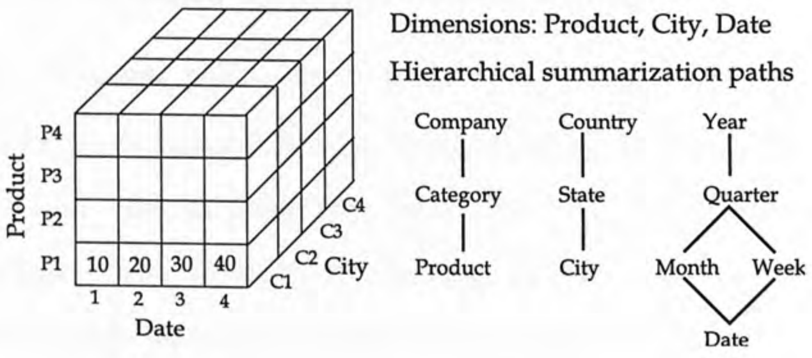


Figure 2.2: Multidimensional data [8]

Figure 2.2, three dimensions: `product` "P1", `Date` "1" and `City` "C1" determine the `measure` is 10. Each dimension is associated with a dimension table and described by a set of attributes. For example, a dimension table for `Product` may contain attributes product number, product name, category and unit price. Note that the attributes of a dimension are hierarchical. In Figure 2.2, the `Date` is organized as either a year-quarter-month-day hierarchy or a year-quarter-week-day hierarchy. Typical OLAP queries contain aggregation of measure over one or more dimensions. The hierarchy of the dimension can be summarized, that is, higher-level aggregates (e.g., group by year) may be obtained directly from lower-level aggregates (e.g., group by quarter). Based on hierarchy characteristic, OLAP tools provide some useful operations, such as: roll-up and drill-down. It provides a convenient way for users to generate summaries, aggregates, and hierarchies at each granularity level.

2.1.6 Star Schema and Snowflake Schema

The entity-relationship (ER) data model and normalization techniques are commonly used in the design of relational databases in online transaction processing environments [8]. ER database schema consists of a set of entities and relationships among them. However, ER database schema is inappropriate for data warehouses because querying efficiency is important. The most popular data model for a data warehouse is multidimensional model which can exist in the form of star schema, snowflake schema or a fact constellation schema.

Most data warehouses use a star schema to represent the multidimensional data model. A star schema consists of one central fact table and a single table for each dimension (See Figure 2.3). Each record in the fact table consists of many attributes where some of them are dimension attributes. Dimension attributes are pointers or foreign keys referencing to the dimension tables that provide its multi-dimensional coordinates. The remaining attributes in the fact table store the numeric measures for those coordinates. Sometimes a star schema is not enough to show the complicated data structure, so there is a refinement of star schema where the dimensional hierarchy is represented explicitly by normalizing the dimension tables. This representation known as snowflake schema. In a snowflake schema, tables are further added to the dimension tables to show a more detailed attributes in the data. A sample of snowflake schema is shown in Figure 2.4. Fact constellation schema is more sophisticated than star schema and snowflake schema. It requires multiple fact tables to share dimension tables.

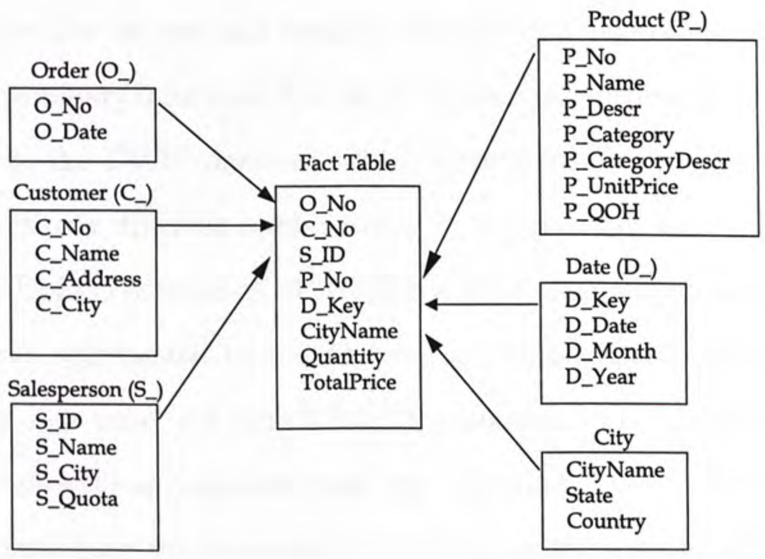


Figure 2.3: A star schema [8]

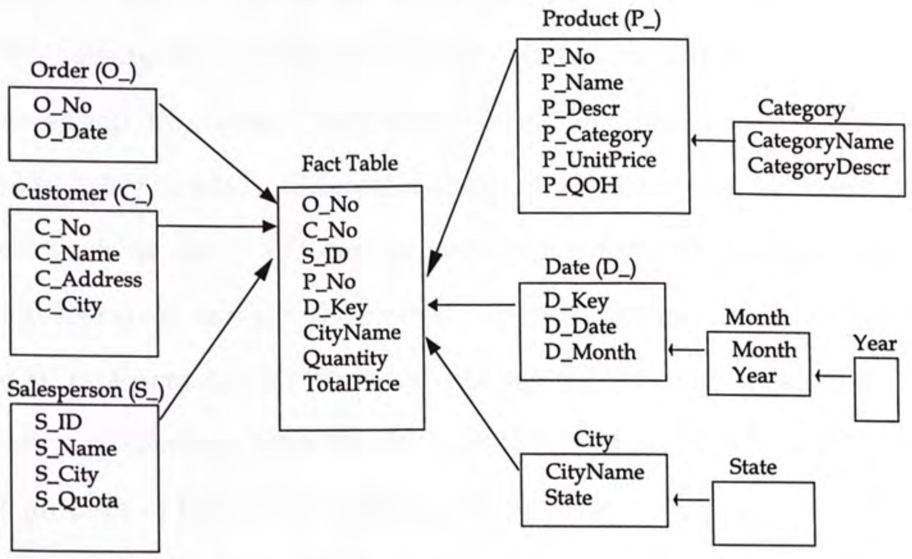


Figure 2.4: A snowflake schema [8]

2.1.7 Data Cube

Data cube provides an easy and intuitive way for data analysts to calculate various levels of summary information in the database. [27] introduces a popular data cube operator, the *CUBE* operator, which calculates all the aggregation for the detail data set over different combinations. A k-dimension attributes data cube presents 2^k SELECT-FROM-WHERE-GROUPBY aggregation queries. The aggregate queries represented by a data cube can be organized into a *lattice*. For example, the data cube in Figure 2.2 can be represented by the lattice in Figure 2.5 which denotes the aggregation: `cube by (product, city, date)`. [40] is the first one to introduce the construction of lattice corresponding to a data cube. In Figure 2.5, three dimensions (the abbreviation P for `Product`, C for `City` and D for `Date`) generate eight cuboids: PCD, PC, PD, CD, P, C, D and ALL. Each cuboid is a group by clause, that is `group by (product, city, date); group by (product, city); group by (product, date); group by (city, date); group by (product); group by (city); group by (date)` and `group by ()`, that is no `group by` clause. There is functional dependence relationship between cuboids. In other words, a data cube query can be answered by using the result of others, such as query PC can be answered by the result of query PCD. A dimension hierarchy can also be represented by a lattice. A lattice can be constructed to represent the set of views that can be obtained by grouping on each combination of elements from the set of dimension hierarchies. It turns out that a direct product of the lattice called hypercube [40].

In [50], a multidimensional data model is introduced based on relational elements. A grouping algebra is presented, cubes are modeled as functions from dimensions to the measure and are mapped to grouping relations. Furthermore,

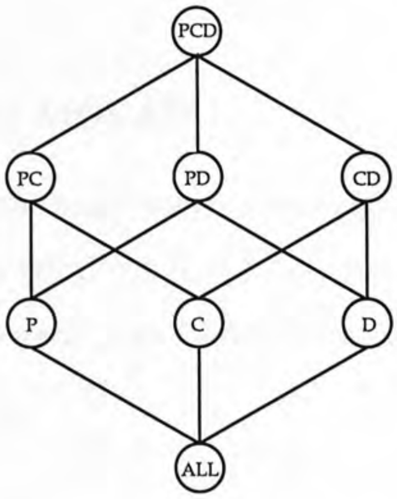


Figure 2.5: Data cube lattice

a multidimensional cube algebra is introduced in order to facilitate the data derivation. In [37], an algebra is defined with classical relational operator, the expressive power of the algebra is shown through the modeling of the data cube and monotone roll-up operators. [69] gives approximate answer to CUBE query by clustering based approaches. Sort-based algorithm and hash-based algorithm are presented in [80] for datacube computation to explore memory utilization. They extend the algorithm for single datacube computation to process multiple datacube simultaneously in [81]. [85] presents a Multi-Way Array based method for computation. It demonstrates that MOLAP approach performs much better than ROLAP algorithm. They suggest this approach could be valuable in ROLAP system. [19] presents an efficient algorithm for computing a cube by. The cost of creating a materialized view and the cost of processing a query to a data warehouse would be taken into account when selecting the set of materialized views.

2.1.8 ROLAP and MOLAP

The data storage issue is the main concern when implementing an OLAP server. In this section, Relational OLAP (ROLAP) and Multidimensional OLAP (MOLAP) approaches of implementing an OLAP server are introduced.

2.1.8.1 ROLAP

ROLAP servers are the intermediate servers that stand in between a relational back-end server and client front-end tools. ROLAP systems use relational tables as their data structure. Since ROLAP uses a relational database, it requires more processing time and/or disk space to perform some of the tasks that multidimensional databases are designed for. However, ROLAP supports larger user groups, larger amount of data and is often used when these capacities are crucial, such as in a large and complex department of an enterprise.

2.1.8.2 MOLAP

MOLAP is OLAP that indexes directly into a multidimensional database. In general, MOLAP stores data in a multidimensional array in which all possible combinations of data are reflected, each in a cell that can be accessed directly. Hence, the users are able to view different aspects of data aggregates such as sales by time, geography, and product models quickly. For this reason, MOLAP is, for most users, faster and more user-responsive than ROLAP.

Compared with ROLAP, the advantage of the MOLAP architecture is that it provides a direct multidimensional view of the data whereas ROLAP architecture is just a multidimensional interface to relational data [75]. [16] demonstrates that MOLAP has efficient storage and supports fast data retrieval. MOLAP systems always need to precompute all possible aggregations, that is why they are often more preferable than traditional ROLAP. However, MOLAP are more difficult to update and administer. The disk space consumption of MOLAP architecture is possibly much larger especially when data is sparse. In contrast, in the ROLAP architecture, relational data can be stored more efficiently than multidimensional data. Besides, ROLAP can be easily integrated into other existing relational information systems. However, ROLAP may consume larger storage space for indexing and longer Input/Output time for calculating the derived data using Structure Query Language (SQL).

In a ROLAP architecture, data are organized in a star or snowflake schema. On the other hand, MOLAP systems store data in a n-dimensional array. Each dimension of the array represents the corresponding dimension of the data cube. The contents of the array are the measures of the corresponding data cube.

2.1.9 Query Optimization

Data warehouses contain large volume of data. To answer queries efficiently and give prompt user responses, query optimization becomes a critical issue in the data warehousing environment. There are many ways to improve the query optimization such as indexing, materializing views, transforming of complex SQL queries, creating pre-aggregate summary tables, partitioning and parallel query processing technology.

Since data warehouses store historical data rather than up-to-date information, access to data warehouses are mostly read-only operations. Data warehouses are typically updated periodically in a batch fashion and during this updating process, the data warehouses are unable for querying. This is why data warehouses require to use more complex indexing structures to speed up the evaluation of queries. Bitmap indexing is a well known indexing technique [55, 56, 77]. [77] proposes static and dynamic optimization strategies for selections using bitmaps. In addition to indices on single table, the specialized nature of star schemas makes join indices especially attractive for OLAP queries. Bitmap join indices (star join index) [55] on the dimensional attributes are frequently used for efficient joins between a dimension table and the fact table. [56] presents various bitmap indexes including bit-sliced indexing and projection indexing. It also introduces a new index type, Groupset indexes, to evaluate the ad-hoc OLAP queries which involves aggregation and grouping.

The problem of rewriting a query using a set of views is a NP-hard problem. To evaluate a correlated query, [62] proposes algorithms to recognize the invariant part of the subquery and to restructure the evaluation plan to reuse the stored intermediate results. Based on syntactic characterizations of the equivalence of aggregate queries, [14] proposes an approach to rewrite aggregate queries using aggregate and non-aggregate views. [59] describes a scalable algorithm, MiniCon, for finding the maximally-contained rewriting of a conjunctive query using a set of conjunctive views.

Many researches [32, 49, 64, 68, 86] work on multiple-query optimization. Multiple-query optimization exploits that queries can share the common data and reduce the cost [68]. [86] designs three algorithms to optimize multiple related dimensional queries. Their new query evaluation primitives that allow multiple star join query plans to share portions of their evaluation. The search space of multi-query optimization is very large. [49] provides a practical method for finding and rewriting queries in a finite search space with respect to the views. [10] considers the problem of optimizing queries with aggregates. For single-block SQL queries, group-by precedes join may cause addition group by operator and increase execution space. It can be solved by a greedy conservation heuristic. For view with aggregation, it uses pull-up transformation and group-by above join. [64] provides an efficient heuristic algorithm and demonstrates that multi-query optimization is feasible and effective. Relying on a graphical representation of queries, [83] proposes a matching algorithm to rewrite a user query using materialized views instead of using base tables. [32] addresses the problem of query scheduling in multiquery optimization. In order to reduce the cost of query evaluation, several algorithms are proposed to dynamically cache a set of common sub-expression and utilize the cache space. [21] proposes a chunk-base scheme for caching queries which allows queries to partially reuse the results of previous queries with which they overlap.

2.2 Materialized View

Precomputing OLAP queries – materializing views with aggregate functions – has been widely used as a common technique in data warehouses [30]. In this section, the role of materialized views in OLAP will be introduced and some challenges in exploiting materialized views will be discussed.

2.2.1 What Is A Materialized View

A view is a derived relation defined in terms of base relations. A view thus defines a function from a set of base tables to a derived table, this function is typically recomputed every time when the view is referenced. [36]

A materialized view is a view which stores the tuples of the view in the database. Like a physically table, index structures can be built on the materialized views. Consequently, users access to the materialized views can be much faster than recomputing the query from the fact table.

2.2.2 The Role of Materialized View in OLAP

OLAP systems have been widely used for business data analysis. However, the ad-hoc OLAP queries always involve computing a lot of complex aggregation queries. It may take an enormous amount of disk input/output and CPU processing time because aggregate operations need to be performed in order to conduct statistical analysis against million of records. It becomes very critical on how to reduce OLAP query processing time due to the decision makers need to make right decisions in a short time. Precomputing OLAP queries becomes a key to achieve high performance in data warehouses. OLAP systems speed up querying and system throughput by materializing a large number of summary tables. A summary table is a materialized aggregate view. Having materialized views can significantly speed up query processing.

2.2.3 The Challenges in Exploiting Materialized View

In the chapter 4 of [30], some of the challenges in materialized views independent of the application in which they are being used have been discussed. The main issues that we are concerned are: (a) identify the views to materialize, (b) use the materialized views to answer queries, (c) how to efficiently maintain views, efficiently update the materialized views during load and refresh, and (d) the performance trade offs in using views.

The optimization criteria for selecting views to materialize can be (i) the disk space which is available for storing the views; (ii) the number of views that allowed be materialized [40]; and (iii) maintenance cost.

The problem of answering queries using views is related to a wide variety of data management problems, such as query optimization, data integration and data warehouse design [48, 76]. This problem can be stated as follows: given a query on a database schema and a set of materialized views on the same database schema, will it save the computation time if the query is answered by the views only? [38, 48] give good surveys about different approaches to this problem. The basic idea of using materialized views is to improve the query optimization. However, blind applications of materialized views may result in worst execution plan than if no views were used to answer a query. [9] presents a simple, readily implementable and comprehensive approach to enable a cost-based decision for deciding whether or not to use materialized views to answer a query. [72] provides a semantic approach to solve the problem of answering queries using views in the presence of grouping and aggregation. It describes the conditions required for a view to answer a query and a rewriting algorithm to deal with these conditions.

In [58], Park et al. propose a new method for rewriting a given OLAP query using various kinds of materialized aggregate views. They define the normal forms of OLAP queries and materialized views based on the lattice of dimension hierarchies, the semantic information in data warehouses. They present a rewriting algorithm for OLAP queries that effectively utilizes existing materialized views. Goldstein and Larson [25] present a fast and scalable algorithm for determining whether part or all of a query can be computed from materialized views and describe how it can be incorporated in a transformation-based optimizer. They consider views composed of selections, joins and a final group-by. [1] presents three cost models to generate efficient rewriting algorithms by using views to answer a query. [79] designs a framework and an efficient algorithm for materialized view design. They select a set of materialized views based on the idea of sharing the intermediate common results with the help of Multiple View Processing Plan to minimize the total query response time and the cost of maintaining the materialized views.

2.2.4 What Is View Maintenance

After a view is materialized, the materialized views are maintained by a maintenance policy. In general, view maintenance is the process of updating a materialized view in response to the changes of the underlying data. When new data come, the materialized views in the OLAP systems need to be updated. Usually, the OLAP warehouses apply the incremental maintenance approach which is to update the changes of the summary table, the warehouses can either be updated immediately as soon as a change is received (*immediate view maintenance*), or the update can be deferred until a time window (*deferred view maintenance*).

Immediate view maintenance approach keeps the materialized views updated, however, it is not scalable with respect to the number of views. Besides, immediate view maintenance approach cannot be applied in some applications due to local transactions cannot be delayed until materialized views are refreshed at the remote data warehouse. On the other hand, deferred view maintenance allows changes from several update transactions to be batched together into a single propagate and refresh operation. It imposes significant overhead on all query transactions because a query have to wait for a materialized view to be refreshed.

Compared with recomputing the view, it is cheaper to compute the changes to a view in response to the changes of the underlying database. [29, 31] present incremental view maintenance algorithms. Mumick et al. [54] propose a method for maintaining materialized aggregate views, called summary-delta table method, to efficiently maintain a materialized view. [15] presents algorithms to incrementally refresh a view during deferred maintenance and avoid the state bug. In order to minimize the batch maintenance time, the authors suggest to split the deferred maintenance work into propagate and refresh functions. [46] proposes an efficient algorithm for selecting the optimal strategy to update a set of views to minimize the down time of the data warehouses. It aims at shrinking the data warehouses update window for multiple interdependent materialized views.

[28] defines the concept of self-maintainable views, claiming these views can be maintained using only the contents of the views and the database modifications, but without accessing any of the underlying database. Self-maintainability is a desirable property for efficiently maintaining large views in applications where fast response and high availability are important. Given a materialized view, [63] presents an exhaustive approach as well as a heuristic for selecting an additional

set of views that may reduce the total maintenance cost. They suggest optimal trade off between the space and time need for view maintenance under different scenarios. [61] makes views self maintenanable by defining a set of auxiliary views to materialize at the data warehouses so that auxiliary views and materialized views can be maintained without accessing any source data.

2.3 View Selection

In order to minimize the total query processing cost for all possible OLAP queries, a set of materialized views are selected under some resource constraints. It is worth noting that it is impractical to maintain materialized views for all OLAP queries due to the huge disk space consumption and large update cost. Moreover, since the computation of the view is rather time-consuming, it may not be enough to precompute all the views within a limited time window, so the administrator need to select a set of views as materialized views in order to minimize the total query processing time, and the update time when the fact table is updated also need to take in account.

2.3.1 Selection Strategy

It is obvious that the more views are materialized, the faster queries can be answered. Due to limited amount of resources, it is usually impossible to precompute all of the views in the data warehouse. Besides, queries are complex, if we execute directly from raw data, it may take a long time to run on very large databases. Thus, it is better to materialize partial views rather than compute all of views from raw data every time. However, it is difficult to determine which are the best aggregate to be precomputed given a fixed amount of maintenance time constraint.

Most of the reported studies on materialized view selection consider a disk space constraint due to the fact that the disk consumption of OLAP queries is very large [33, 35, 40, 70]. The *disk space view selection problem* is to select a set of interrelated views that minimizes the total query processing cost under a given disk space constraint. In their paper, Harinarayan et al. [40] propose algorithms to select materialized views, in order to minimize the total query processing cost for datacube or OLAP applications. A lattice framework is used to express dependencies among views. They consider view selection problem under the disk space constraint. A linear cost model is proposed. The linear cost model states that the cost of answering a query using a view, is the number of tuples in the view. Their greedy algorithm can reach, at least, 63% of the benefit of the optimal solution to identify set of materialized views in the datacube for minimizing query processing cost. Gupta et al. [35] extend the results reported in [40] to select a set of views and indexes in datacubes. They study the precomputation of indexes and subcubes, and discuss a family of one-step near-optimal algorithms under a given disk space constraint. Gupta in [33] presents a theoretical formulation of the general view selection problem in a data warehouse, and generalizes the view selection problems as AND, OR, and AND-OR graph problems. Shukla et al. [70] introduce a heuristic algorithm called PBS which achieves the same (0.63 -f) bound as BPUS (proposed in [40])[1], and runs several orders of magnitude faster than PBUS. [2] describes how to compute a set of aggregation views. [22] defines a cost/benefit model and applies a partition algorithm to select views/indexes to

[1] f is the fraction of available space consumed by the largest aggregate.

materialize in a warehouse under some storage space constraint. All of the above works consider the disk space constraint and provide greedy algorithms using the linear cost model. In [36], Gupta and Mumick present a view selection problem under a given maintenance cost. This *maintenance cost view selection* problem is to select a set of views to materialize under a maintenance cost constraint, in order to minimize the total query processing cost. Gupta and Mumick intend to adopt a general cost model. However, in the algorithm design, they also use the linear cost model. The issues of materialized views techniques, implementations and applications are discussed in [30]. They propose two algorithms, inverted-tree greedy and A*-heuristic, to solve this problem. [78] extends [79], and proposes a method where a Multiple View Processing Plan (MVPP) is constructed and some sharing common parts are selected to be materialized in order to achieve the best combination of good query performance and low view maintenance cost. The 0-1 integer programming technique is used to obtain the optimal global processing plan. In [51], Weifa Liang et al. propose two algorithms: two-phase greedy and integrated greedy algorithm, to solve the maintenance cost view selection problem and claim that these two algorithms have polynomial time complexity. In [47], Lee and Hammer make the first attempt to solve this problem by using evolutionary algorithm.

Other directions on view selection include filtering [60] and multi-cube [71]. In [60], Qiu and Ling study two filtering methods, a functional dependency filter and a size filter. The former removes views with redundant summary information based on functional dependencies among the dimensional attributes. The latter is based on the view size to filter out any views that can be either derived from another small materialized view or has almost the same number of tuples as another materialized view from which it can be derived. Useful views are selected by these two filters. In [71], Shukla et al. analyze the view selection issues in a multi-cube environment. They study multidimensional query semantics and query benefits across multiple cubes. Like the other works, they adopt the linear cost model for simplicity. In [3], Agrawal et al. have looked at the problem of building an industry-strength tool for automated selection of materialized views and indexes for SQL workloads. Their solution is implemented as part of a tuning wizard that ships with Microsoft SQL Server 2000. In [53], Mistry et al. study how to find an efficient plan for the maintenance of a set of materialized views by exploiting common subexpressions between different view maintenance expressions. They study three inter-dependent decisions, transient materialization, permanent materialization and incremental or recomputation. In [11], Chirkova et al. consider the problem for views and workloads, and study several fundamental results concerning the view selection problem.

The work related to dynamic view management is summarized below. [74] states the dynamic data warehouse design problem: given a set of old materialized views in the data warehouse, a set of new queries to be answered by the data warehouse, and extra space allocated for materialization, select a set of new views to materialize in the data warehouse that fit in the extra space, allows a complete rewriting of the new queries over the old materialized views and minimizes the combined evaluation cost of the new queries and the maintenance cost of the new views. They formulate this problem as a state space search problem and propose generic incremental algorithms and heuristics to solve it.

Caching is a common technique for dynamic view management. [44] shows some typical characteristics of OLAP queries which are suitable for caching. Compared with the traditional database, updates in data warehouses are infrequent, and the cached data for OLAP queries may be valid for a long time. The query results present temporal and geographical locality well such that OLAP queries typically access data in a hierarchy repetitively, using operations like `group-by, aggregation, drill-down, and roll-up`. In order to speed up the query response time for the OLAP decision-making systems, different caching mechanisms have been proposed [18, 21, 42, 45, 66, 73]. They can effectively reuse the previous query results and speed up query processing time of the subsequence queries.

The semantic caching mechanism has been introduced in [18] for client/server main memory architectures. [21] performs a chunk-based scheme which divides the multidimensional query space into uniform chunks. When a query is issued, it checks whether the query can be computed directly from the previous query results which are stored in the cache or a dedicate disk storage. If some parts of the queries cannot be computed from the cache, the system will compute the missing parts using the base table. [20] extends the work on the aggregation in the cache to improve the caching performance. Dimitris et al. [43] implement a multi-tier caching system for queries. [42] extends the chunks into the peer-to-peer network to fully utilize the cache in the client side. A system called DynaMat in [45] is presented that constantly monitors incoming queries and materialized view selection and refreshes the most beneficial subset of it within a given maintenance window using different caching strategies.

2.4 Summary

In this chapter, the architecture of a data warehouse has been introduced and how the multidimensional data model is used to represent it. For decision support, data warehouses and OLAP systems collect data from various data sources, back-end tools extract and transform data to store in OLAP server. ROLAP and MOLAP are the popular approaches to implement the OLAP server which support multidimensional data model for fast data retrieval and analysis in front end tools. Star schema and snowflake schema are common in ROLAP for giving efficiency of processing queries. Many researchers have pointed out the materialized views can help achieve query optimization. The problem of answering queries using views has been received significant attention. It is to find efficient methods of answering query using a set of previously defined materialized views over the database, rather than accessing the database relations. Mainly, the materialized views selection can be under disk space or maintenance cost criteria. While some of these problems have been partially solved, it provides a number of open problems for the research community. The research problem in this thesis will be formalized in Chapter 3

Chapter 3

Problem Definition

In Chapter 2, the view selection problem has been discussed. It is to select a set of views to materialize in a data warehouse to reduce the query response time as well as warehouse maintenance cost under some constraints. In this chapter, the research problem will be formally defined. In Section 3.1 two optimization criteria for the materialized view selection problem: disk space and maintenance cost will be introduced. In Section 3.2, a derived-by relation in the lattice and the search space for this problem will be described, cost functions will also be formalized. Compared with disk space view selection problem, the difficulties of maintenance cost view selection problem will be pointed out in Section 3.3.

3.1 View Selection Under Constraint

One of the most important issues in data warehouse design is how to select a set of materialized views in order to minimize the total query processing time of OLAP queries under a certain constraint. The constraint can be either disk space constraint or maintenance cost constraint. The disk space constraint specifies the availability of the disk space in a data warehouse, whereas the maintenance cost constraint specifies how long all materialized views must be updated.

- **Disk Space Constraint Handling**: The disk space view selection problem is to select a set of interrelated views that minimizes the total query processing cost under a given disk space constraint. [33, 35, 40, 70] consider disk space constraint and provide greedy algorithms using the linear cost model. Most of the greedy algorithms start from an empty set and select the next view with the maximum benefit per unit space in turn. The benefit of the views which have been selected will be unchanged in the subsequent view selection processes, it is defined as *monotonic property*. The algorithms continue to pick views until the space limit is reached. However, as disk becomes cheap, the disk space constraint becomes less important nowadays.

- **Maintenance Cost Constraint Handling**: In real life applications, the constraint is more likely to be the maintenance cost incurred in keeping the materialized views up-to-date in a data warehouse. This maintenance cost view selection problem is to select a set of views to materialize under a maintenance cost constraint, in order to minimize the total query processing cost. This problem is more difficult than the disk space view selection problem, because the total maintenance cost for a set of views may decrease when more views are added to materialized. This is defined as *non-monotonic property*.

Maintenance cost view selection problem has been proven to be NP-hard [36]. For example, Baralis et al. describe a real store chain application that only has 4 dimensions, namely, `Product` (50 attributes), `Store` (20 attributes), `Time` (10 attributes) and `Promotion` (10 attributes) [4, 44]. However, the number of possible materialized views is over 2^{90}. The search space for possible materialized views is extremely large.

3.2 The Lattice Framework for Maintenance Cost View Selection Problem

Like [40], a lattice is denoted with a set of elements (queries and views) L and a dependence relation $\preceq$ (derived-from, be-computed-from) by $(L, \preceq)$. Given two queries q_i and q_j. q_i is dependent on q_j, $(q_i \preceq q_j)$, if q_i can be answered using the results of q_j. The lattice $(L, \preceq)$ is called a *dependent lattice.* For elements a and b of a dependent lattice $(L, \preceq)$, $a \prec b$ means $a \preceq b$ and $a \neq b$. The ancestors and descendents of an element of a lattice $(L, \preceq)$, are defined as $\mathit{ancestor}(a) = \{b \mid a \preceq b\}$ and $\mathit{descendant}(a) = \{b \mid b \preceq a\}$, respectively. A dependent lattice can be represented as a directed acyclic graph in which the lattice elements are vertices and there is an edge from a to b, if $b \prec a$ and $\nexists c(b \prec c \wedge c \prec a)$. There is a path downward from a to b if and only if $b \preceq a$.

A Multidimensional Database (MDDB) is a collection of relations, $D_1, \cdots, D_m$, F, where D_i is a dimension table and F is a fact table [4, 44]. In most real applications, an MDDB consists of multiple dimensions, and each of them in turn can be organized as hierarchies of attributes. Consider the large grocery store chain example given in [4, 44] again. The store chain has four dimensions, namely, `Product`, `Store`, `Time` and `Promotion`. The `Product` dimension has more than 50 attributes such as brand, category, diet-type, package type, weight, case size and a merchandise hierarchy. The `Store` dimension has more than 20 attributes including store address, telephone number, manager, description of store services, and sizes of different departments. It also has a geographic hierarchy. The `Time` dimension is characterized by the granularity of day; day in the month, in the quarter and in the year, holiday, special events, as many as more than 10 attributes. There are different granularity of the time hierarchy, namely, day, week, month and year. The `Promotion` dimension contains 10 attributes such as the promotion type, promotion cost and start/end data.

Suppose that an MDDB has m dimensions and the i-th dimension has n_i attributes. Assume that each dimension is characterized as a *dimensional dependent lattice.* The dependent lattice for the i-th dimension will have 2^{n_i} elements. If queries/views can be issued/made by grouping any or no member of m dimensions, the total number of elements for the MDDB will be $\prod_i^m (2^{n_i} + 1)$. As for the store chain example, the total number of OLAP queries is $(2^{50} + 1) \times (2^{20} + 1) \times (2^{10} + 1) \times (2^{10} + 1)$ and the number of elements is greater than 2^{90}. Therefore, the dependent lattice $(L, \preceq)$ in question is much more complex than a hypercube lattice. Here, let $(a_1, a_2, \cdots, a_m)$ be an m-tuple where each a_i is a point in the hierarchy of the i-th dimension, the dependence relation $\preceq$ can be defined as $(a_1, a_2, \cdots, a_m) \preceq (b_1, b_2, \cdots b_m)$ if and only if $a_i \preceq b_i$ for all i. This is called the *direct product* of the dimensional lattices [40]. A simple direct product of two dimensional lattices is shown in Figure 3.1.

A direct-product of the dimensional lattice is represented as a directed acyclic graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$. $V(G)$ is

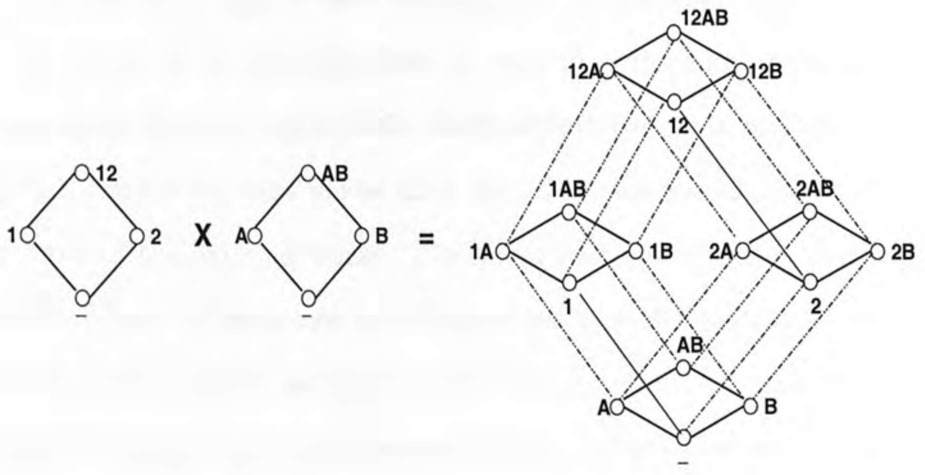


Figure 3.1: A simple direct product example.

used for the set of vertices of a graph G. The graph G has the following weights associated with vertices and edges.

- three weights on a vertex v:
  - r_v: initial data scan cost.
  - f_v: query frequency.
  - g_v: update frequency.
- two weights on an edge (v, u)
  - $w_{q_{u,v}}$: query processing cost of u using v.
  - $w_{m_{u,v}}$: updating cost of u using v.

In a general setting, given a query u and a selected materialized view v, a function $q(u, v)$ is the sum of the query processing costs associated with edges on the shortest path from v to u plus initial data scan cost of the vertex v, r_v. With

$q(u,v)$, the raw table will be used instead of v, if and only if the view v cannot answer the query u. In a similar fashion, $m(u,v)$ is the sum of the maintenance costs associated with the edges on the shortest path from v to u. Here, we attempt to adopt a more general cost model than the linear cost model [40] which has been used in most of the existing works. The linear cost model states that the cost of answering a query, u, using one of its ancestors, v, is the number of rows present in the table v (r_v). Here, as shown in the two functions $q()$ and $m()$, a general query processing cost and maintenance cost model are assumed. First, a query processing cost can be different from a maintenance cost for a pair of vertices. The maintenance costs can possibly be much lower than the query processing costs. Second, a query processing cost may involve other query processing costs (associated with edges) in addition to the initial table scan costs (associated with vertices). For example, given two dimensions, D_1 and D_2. Assume that the table for D_1D_2 is sorted on D_1. The cost of performing aggregate D_1 using D_1D_2 is different from the cost of performing aggregate D_2 on D_1D_2 due to sorting order. The cost differences need to be addressed as weights associated with edges. Third, there are multiple paths from a view to a query. The shortest path is selected as its cost in the settings.

The similar notations and definitions used in [36] is adopted to define the maintenance cost view selection problem. Given an aforementioned graph $G = (E, V)$ and a set of queries, Q ($\subseteq V(G)$).

The maintenance cost view selection problem is to select a set of views M ($\subseteq V(G)$) that minimizes $\tau(G, M)$, where

$$\tau(G, M) = \sum_{v \in V(G)} f_v \cdot q(v, M)$$

under the constraint that $U(M) \leq S$, where $U(M)$ is the total maintenance cost:

$$U(M) = \sum_{v \in M} g_v \cdot m(v, M)$$

Here, $q(v, M)$ denotes the minimum cost of answering a query v ($\in V(G)$) in the presence of the set of materialized views, M, and $m(v, M)$ is the minimum cost of maintaining a materialized view v in the presence of the set of materialized views, M.

3.3 The Difficulties of Maintenance Cost View Selection Problem

As mentioned before, maintenance cost is more likely to be the real constraint in many real applications to keep the materialized views consistent with the data in data warehouse, rather than disk space constraints. The maintenance cost view selection problem seems to be very similar to the disk space view selection problem. However, the maintenance cost view selection problem is more difficult. For the maintenance cost view selection problem, the maintenance cost of the views relies on each other. Selection of a view will affect the prior materialized views. The total maintenance cost for a set of views may decrease when more views are added to materialize while the space occupied by a set of views always increases when a new view is selected under the disk space constraint. Figure 3.2 illustrates the difference between the disk space view selection problem and maintenance cost view selection problem. Here, for a vertex, v_i, T and u are table size (r_{v_i}) (for the query processing cost) and maintenance cost ($w_{q_{u,v}}$), respectively.

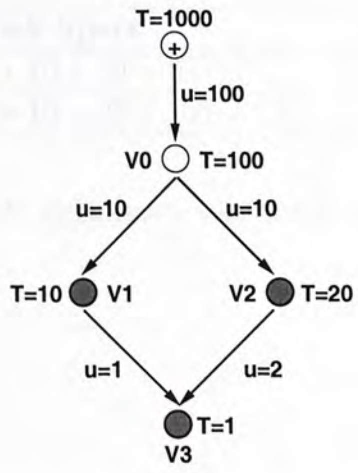


Figure 3.2: An example of view maintenance

For simplicity, the query frequency (f_v) and update frequency (g_v) are assumed to be the same for every vertex in this example. Suppose $M = \{v_3, v_1, v_2\}$ are materialized in an order of v_3 and v_1 followed by v_2. Table 3.1 shows that the total disk space used is 31 and the total maintenance cost is 221, because v_1 and v_2 need to be computed from the virtual root and v_3 is answered by v_1. Now consider materializing v_0. The total disk space used is increased to 131, and the total maintenance cost is decreased to 121, because v_1 and v_2 now can be updated by v_0. This non-monotonic property makes maintenance cost view selection very difficult.

Materialized View	Disk Space	Maintenance Cost
v_3, v_1, v_2	$1 + 10 + 20 = 31$	$1 + 110 + 110 = 221$
v_3, v_1, v_2, v_0	$1 + 10 + 20 + 100 = 131 \uparrow$	$1 + 10 + 10 + 100 = 121 \downarrow$

Table 3.1: Disk space v.s. maintenance cost

3.4 Summary

In this chapter, the disk space view selection problem has been compared with maintenance cost view selection problem, the difficulties of maintenance cost view selection has been discussed. At the mean time, the maintenance cost view selection problem is formulated as well as the cost functions is defined.

The research in this thesis focuses on view selection problem under maintenance cost constraint. To deal with this NP-hard problem, algorithms that provide a nearly optimal solution in polynomial time are highly desirable. Some heuristic algorithms have been proposed to solve this problem, however, the performance of these heuristic algorithms have not been well analyzed. The algorithms cannot be used without any systematic study on the quality. For the purpose of providing a view on how to use the heuristic algorithm and helping us to design new heuristic algorithm, four algorithms will be investigated in Chapter 4. Based on the observation, a new evolutionary algorithm is designed to solve this problem in Chapter 5. At the mean time, a partitioning algorithm is proposed for dynamic view management problem in Chapter 6. The content of this thesis is a further development of the works reported in following papers [12, 13, 26, 82]:

- Chi-Hon Choi, Jeffrey Xu Yu and Gang Gou. What Difference Heuristics Make: Maintenance-Cost View-Selection Revisited, In *Proceedings of the 3rd International Conference on Web-Age Information Management*, pages 247-258, 2002.
- Chi-Hon Choi, Jeffrey Xu Yu and Hongjun Lu, Dynamic Materialized View Management Base on Predicates, In *Proceedings of the 5th Asia Pacific Web Conference (APWEB)*, pages 583-594, 2003.
- Gang Gou, Jeffrey Xu Yu, Chi-Hon Choi and Hongjun Lu, An Efficient and Interactive A*-Algorithm with Pruning Power: Materialized View Selection Revisited, In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 231, 2003.
- Jeffrey Xu Yu, Xin Yao, Chi-Hon Choi and Gang Gou, Materialized View Selection as Constrained Evolutionary Optimization, In *IEEE Transactions on Systems, Mans, and Cybernetics on technologies promoting computational intelligence, openness and programmability in networks and Internet services*, 2003.

Chapter 4

What Difference Heuristics Make

The maintenance cost view selection problem is more difficult than the view selection problem under a disk space constraint, because a selected view may make the previously selected views less beneficial, due to the fact that the total maintenance cost for a set of views may decrease when more views are materialized while the maintenance cost always increase under disk space constraint. The problem has recently been received significant attention. Several greedy/heuristic algorithms were proposed. However, the quality of the greedy/heuristic algorithms has not been well analyzed.

In this chapter, Section 4.1 addresses the motivation. and some examples will be given in Section 4.2. Section 4.3 discusses the strength and weakness of four existing algorithms: inverted-tree greedy [36], A*-heuristic [36], two-phase greedy [51] and integrated greedy [51] for solving the maintenance cost view selection problem. Section 4.4 conducts extensive experiments studies and reports the results. A short summary is presented in Section 4.5.

4.1 Motivation

The maintenance cost view selection problem has been proven to be a NP-hard problem [36]. Gupta and Mumick [36] claim that the greedy algorithms that select views on the basis of query benefit per unit maintenance cost can deliver an arbitrarily bad solution due to the non-monotonic property of the maintenance cost view selection problem. In other words, a selected view may make the previous selected views less beneficial, because the total maintenance cost for a set of selected views may decrease when more views are materialized. Gupta and Mumick propose two algorithms, namely, inverted-tree greedy and A*-heuristic to solve this intractable problem in OR view graph and AND-OR view graph, respectively. In [51], Liang et al. propose two algorithms, two-phase greedy and integrated greedy, to solve this problem. The two algorithms are designed on the basis of query benefit per unit maintenance cost. Liang et al. claim that the two algorithms are able to find feasible solutions in polynomial time. However, they did not provide any analytical and performance studies.

To deal with the maintenance cost view selection problem, algorithms that provide a nearly optimal solution in polynomial time are highly desirable. But, the arguments presented in [36, 51] are not consistent. On the basis of query benefit per unit maintenance cost, the former indicates that greedy algorithms can generate an arbitrarily bad solution. The latter argues that greedy algorithms can possibly generate feasible solutions. The algorithms cannot be used without any systematic study on the quality.

The purpose of this chapter is to provide readers with insights on the heuristic algorithms in terms of both processing time for the algorithms to find a solution and the effectiveness of the algorithms to minimize query processing cost. Inverted-tree greedy, A*-heuristic, two-phase greedy and integrated greedy algorithms are investigated in various settings for the general case of dependence lattice. The academic significance of this work is of twofold. First, it provides a view on how to use the heuristic algorithms. Second, it assists us to design new heuristic algorithms. Extensive studies show that greedy algorithms perform well under certain conditions and the existing A*-heuristic [36] cannot always find optimal solutions.

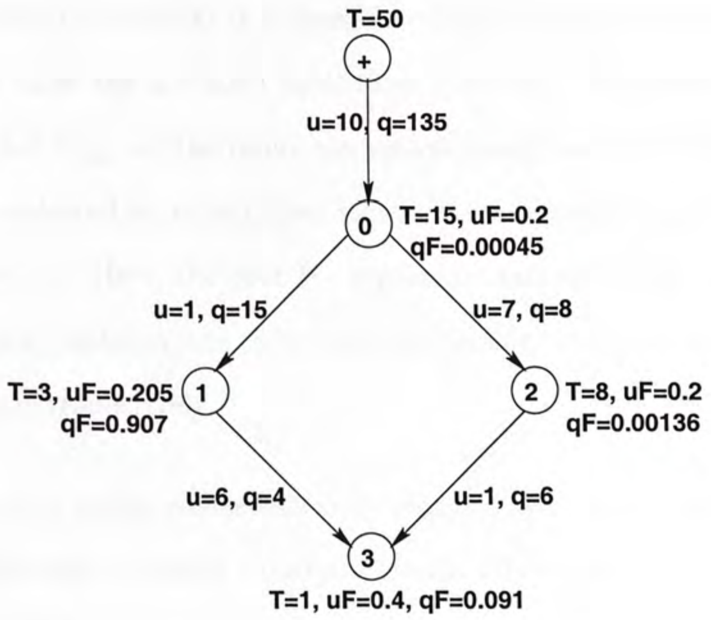


Figure 4.1: An example

4.2 Example

The main idea of the greedy heuristics proposed in [33, 35, 40, 51] is to select materialized views, in *order* of query benefit per unit space/time consumed, which is given below.

$$QBPU(v, M) = \triangle Q_v / \triangle T_v \tag{4.1}$$

Here, M is a set of selected views, v is a view to be added, $\triangle Q_v = \tau(G, M) - \tau(G, M \cup \{v\})$, and $\triangle T_v = U(M \cup \{v\}) - U(M)$.

The question is whether it is possible to use such an order for the problem that does not have the so-called monotonic property. We address the related issues below. Let C_{min} be the minimum maintenance cost constraint that allows all views to be selected as materialized views. Some examples are shown in Figure 4.1 and Figure 4.2. Here, the root (+) represents the raw table. `T`, `u`, `q`, `uF` and `qF` are, table size, maintenance cost, query processing cost, update frequency and query frequency, respectively.

Issue 1 *The total maintenance cost may decrease when a new materialized view is selected. However, a greedy algorithm always selects the greatest query benefit per time constraint.*

Consider an example in Figure 4.1. Initially, v_1 has the largest query benefit per unit time consumed. At the second stage, the greedy heuristic considers all the remaining views, $\{v_0, v_2, v_3\}$ one by one. The result is shown in Table 4.1.

It shows that $\triangle T_v$ becomes negative when v_0 is added to the set of materialized views, M. However, v_3 will be selected because it has the most query

M	v	$\triangle T_v$	$QBPU(v, M)$	Remaining Constraint
$\{v_1\}$	v_0	-0.05	-6.171	1.80
$\{v_1\}$	v_2	3.40	0.074	-1.65
$\{v_1\}$	v_3	2.40	0.227	-0.65

Table 4.1: An example for Figure 4.1

benefit per unit time consumed. But, due to the total maintenance cost will go beyond C_{min}, neither v_3 nor v_2 will be selected. The resulting set of materialized views is $\{v_1\}$. But the optimal solution is $\{v_0, v_1, v_2, v_3\}$.[1]

The quality of heuristics varies dramatically in different settings. Consider another example in Figure 4.2. Table 4.2, Table 4.3 and Table 4.4 show the qualities of the four algorithms, inverted-tree greedy, A*-heuristic, two-phase greedy and integrated greedy in three cases. In these tables, the column of `views` gives the resulting set of materialized views found by the algorithm specified in the column of `Algorithm`. `Q-cost` and `M-cost` are the total query processing cost and total maintenance cost, respectively, when the set of `views` have been materialized. These tables show that none of these four algorithms can always outperform the others.

Issue 2 *Given a large maintenance cost constraint, heuristic solutions may not be able to select all vertices as views.*

As noted in Table 4.2, when the maintenance cost constraint is C_{min}, the optimal solution is to select all views, because the maintenance cost constraint

[1]Note: most greedy heuristic algorithms use zero as a lower bound of the query benefit per unit time consumed.

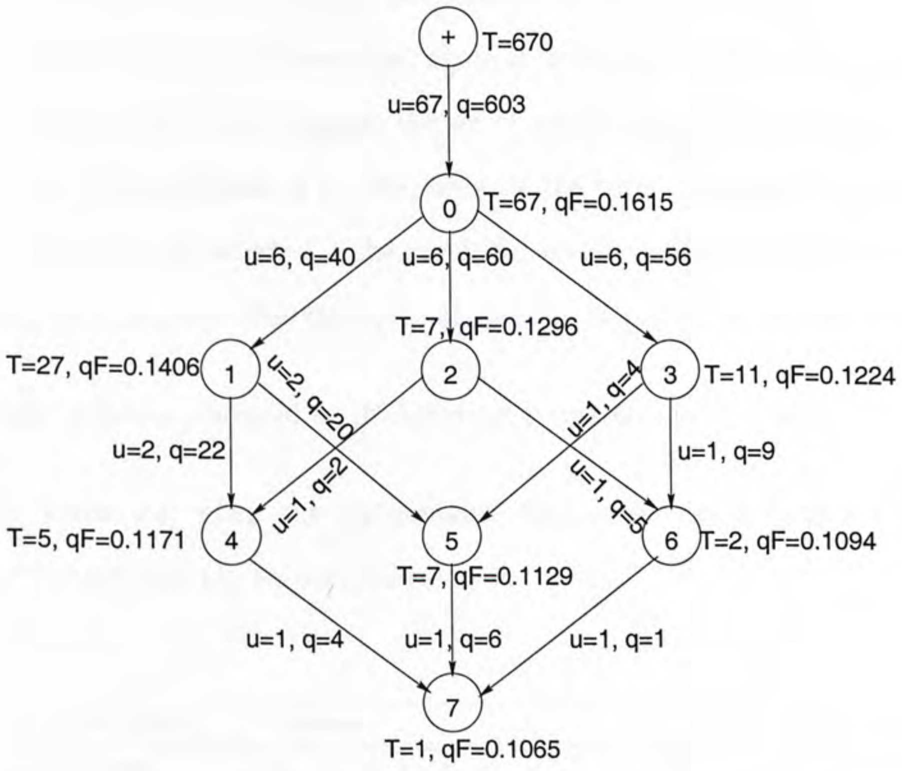


Figure 4.2: Another example (all update frequencies are 0.125.)

allows to do so. A*-heuristic and the integrated greedy are able to materialize all views. But the inverted-tree greedy and the two-phase greedy cannot achieve the optimal.

Issue 3 *A greater query benefit per unit time consumed does not necessarily lead to a minimum total query processing cost. Two sub-issues are: (a) selecting a view may make the further view selections unsuccessful, and (b) not selecting the potential best view at one stage may make it unable to be selected again.*

Table 4.3 shows an example when the maintenance cost constraint is $0.96 \times$

C_{min}. The integrated greedy might not achieve the optimal, because a potentially beneficial view might not be selected again if it cannot be selected at the stage where it must be selected. Suppose the set of materialized views is $\{v_0, v_2, v_3, v_6\}$ (Figure 4.2). The selection of v_1 fails, because the total maintenance cost will go beyond C_{min}, if v_1 is selected. The resulting set of materialized views becomes $\{v_0, v_2, v_3, v_6, v_5, v_4, v_7\}$. But the optimal solution is $\{v_0, v_1, v_2, v_3\}$ including v_1.

Issue 4 *A solution provided by A*-heuristic is not always optimal.*

Refer to Table 4.4, when the maintenance cost constraint is $0.90 \times C_{min}$, A*-heuristic cannot provide an optimal solution.

Algorithm	**Views**	**Q-cost**	**M-cost**
Optimal	$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$	18.571	11.125
Inverted-Tree	$\{v_0\}$	115.429	8.375
A*-heuristic	$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$	18.571	11.125
Two-Phase	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000
Integrated	$\{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$	18.571	11.125

Table 4.2: Performance for Figure 4.2 with constraint= C_{min}

4.3 Existing Algorithms

In this section, four algorithms are introduced: inverted-tree greedy [36], A*-heuristic [36], two-phase greedy [51], and integrated greedy [51].

Algorithm	Views	Q-cost	M-cost
Optimal	$\{v_0, v_1, v_2, v_3\}$	22.314	10.625
Inverted-Tree	$\{v_0\}$	115.429	8.375
A*-heuristic	$\{v_0, v_1, v_2, v_3\}$	22.314	10.625
Two-Phase	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000
Integrated	$\{v_0, v_2, v_3, v_4, v_5, v_6, v_7\}$	29.818	10.375

Table 4.3: Performance for Figure 4.2 with constraint= $0.96 \times C_{min}$

Algorithm	Views	Q-cost	M-cost
Optimal	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000
Inverted-Tree	$\{v_0\}$	115.423	8.375
A*-heuristic	$\{v_0, v_1, v_2, v_6\}$	37.474	10.000
Two-Phase	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000
Integrated	$\{v_0, v_2, v_3, v_6\}$	31.402	10.000

Table 4.4: Performance for Figure 4.2 with constraint= $0.9 \times C_{min}$

Algorithm 1 A*-Heuristics [36]

```
Input: A graph $G(V, E)$ and a maintenance cost constraint $S$.
Output: a set of materialized views.
 1: begin
 2: Create a tree $T_G$ having just the root A. The label associated with A is $\langle \phi, \phi \rangle$.
 3: Create a priority queue (heap) $L = \langle A \rangle$
 4: repeat
 5:    Remove $x$ from $L$, where $x$ has the lowest $g(x) + h(x)$ value in $L$
 6:    Let the label of $x$ be $\langle N_x, M_x \rangle$, where $N_x = \{v_1, v_2, \cdots, v_d\}$ for some $d \leq n$.
 7:    if $d = n$ then
 8:       return $M_x$
 9:    end if
10:    Add a successor of $x, l(x)$, with a label $\langle N_x \cup \{v_{d+1}\}, M_x \rangle$ to the list L.
11:    if $(U(M_x) < S)$ then
12:       Add to L a successor of $x, r(x)$, with a label $\langle N_x \cup \{v_{d+1}\}, M_x \cup v_{d+1} \rangle$
13:    end if
14: until (L is empty);
15: return $\phi$;
16: end
```

4.3.1 A*-Heuristic

The A*-heuristic is shown in Algorithm 1. The A*-heuristic uses an *inverse topological order* to find a set of materialized views. It defines a binary tree T_G whose leaf vertices are the candidate solutions of this problem. At each stage of searching, A*-heuristic evaluates the benefit of remaining downward branches, and selects the branch of the greatest benefit to go down. A binary search tree is shown in Figure 4.3. Each vertex in this binary search tree has a label $\langle N_x, M_x \rangle$ ($M_x \subseteq N_x$), where M_x is the set of views which have been chosen to materialize and considered to answer the set of queries N_x. The search space is $2^{|V(G)|}$, where $V(G)$ is the set of vertices of the graph G. [36] estimates the benefit of the downward branches by summing up two functions $g(x)$ and $h(x)$. $g(x)$ is the total query processing cost of the queries on N_x using the selected views in M_x. $h(x)$ is an estimated lower bound on $h^*(x)$ which is defined as the remaining query cost of an optimal solution corresponding to some descendant of x in T_G [36].

4.3.1.1 Discussions

In Table 4.2 and Table 4.3, A*-heuristic can reach an optimal solution. However, in Table 4.4, A*-heuristic can only reach a nearly-optimal solution, not the optimal solution. The reason is that the expected benefit, $h(x)$, is very difficult to estimate accurately. A*-heuristic delivers an optimal solution only when $h(x) \leq h^*(x)$. The A*-heuristic may not reach an optimal solution under some critical maintenance cost constraints. The A*-heuristic can take exponential time, in the worst case, with respect to the number of vertices in the graph [36].

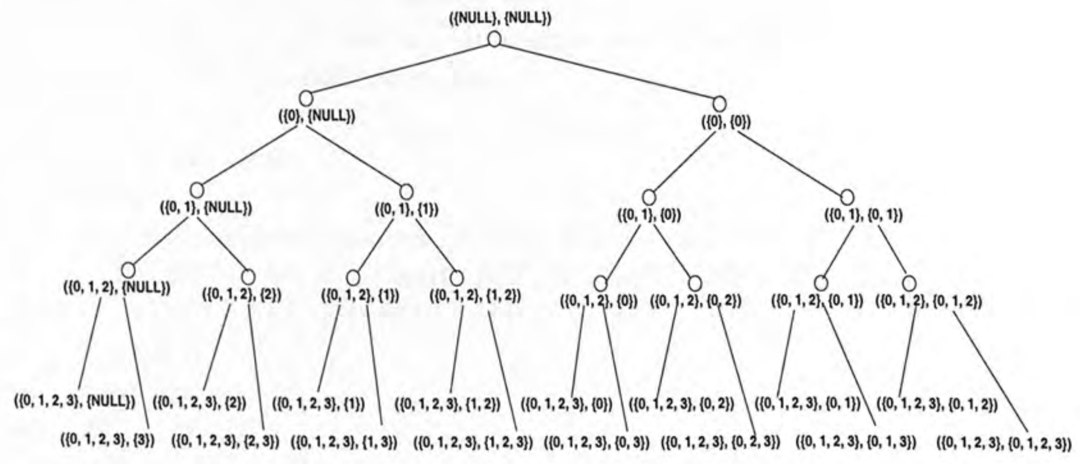


Figure 4.3: The binary search tree T_G of candidate solutions for Figure 4.1

4.3.2 Inverted-Tree Greedy

The inverted-tree greedy uses the concept of inverted tree set. Given a vertex v in a directed graph, an inverted tree set contains the vertex v and any subset of vertices reachable from v. The inverted-tree greedy is shown in Algorithm 2, where $B(C, M)$ is the query benefit associated with a set of vertices C with respect to M as $\tau(T, M) - \tau(T, M \cup C)$, and $EU(C, M)$ is the effective maintenance cost of C with respect to M as $U(M \cup C) - U(M)$. At each stage, this algorithm considers all inverted tree sets of views, T, in the given graph G, such that $T \cap M = \phi$, and selects the inverted tree set that has the most query benefit per unit effective maintenance cost.

Algorithm 2 Inverted-Tree Greedy [36]

Input: A graph $G(V, E)$ and a maintenance cost constraint S.
Output: a set of materialized views.

```
1: begin
2: $M \leftarrow \phi$; $B_C \leftarrow 0$;
3: repeat
4:    for each inverted-tree set of views $T$ in $G$ such that $T \cap M = \phi$ do
5:       if $(EU(T, M) \leq S)$ and $(B(T, M)/EU(T, M) > B_C)$ then
6:          $B_C \leftarrow B(T, M)/EU(T, M)$; $C \leftarrow T$;
7:       end if
8:    end for
9:    $M \leftarrow M \cup C$;
10: until $(U(M) \geq S)$;
11: return $M$;
12: end
```

4.3.2.1 Discussions

The steps for selecting a set of views for the example in Figure 4.2 is given below. In Table 4.2, the maintenance cost constraint is the minimum cost that allows all vertices to be selected as materialized views. In the first step, the algorithm selects v_0, because it has the maximum $B(T, M)/EU(T, M)$. In the following steps, it cannot select any more vertices. As the query benefit per unit effective maintenance cost does not increase by adding any new vertices.

Below are some observations of the inverted-tree greedy. First, the inverted-tree greedy does not guarantee a strict maintenance cost constraint, it satisfies a limit within twice the maintenance cost constraint. Second, the total time complexity of a stage of the inverted-tree greedy is $\sum_{v \in V(G)}(2^{Av})$, where $V(G)$ is the set of vertices of the graph and Av is the number of descendants of a vertex. In the worst case, it is exponential with respect to $|V(G)|$ as shown in

[36]. Third, the extensive experiments show that the inverted-tree greedy always chooses the first vertex as a part of its solution. The reason is that the algorithm calculates both the effective maintenance cost and the query benefit per unit effective maintenance cost. After selecting the first vertex, the query benefit per unit effective maintenance cost of the first vertex is highest as all other vertices can be derived from it. Therefore, the algorithm cannot effectively select any other views. Finally, by adding a new view into the set of views, the query benefit will increase. However, the query benefit per unit effective maintenance cost intends to decrease, which possibly makes the further selection of vertices fail as shown in this example.

Algorithm 3 Two-Phase Greedy [51]

```
Input: A graph $G(V, E)$ and a maintenance cost constraint $S$.
Output: a set of materialized views.
1: begin
2: Find a set of materialized views, $M_1$, to minimize the total query processing
   cost;
3: if $U(M_1) \leq S$ then
4:     return $M_1$;
5: else
6:     Find a subset of $M_1$, denoted as $M_2$, that minimizes the total query pro-
       cessing cost and satisfies $S$.
7:     return $M_2$;
8: end if
9: end
```

4.3.3 Two-Phase Greedy

The two-phase greedy [51] is illustrated in Algorithm 3, the basic idea is that it selects a subset of the materialized views, M_1, to minimize the total query processing cost without considering the maintenance cost constraint. If the total maintenance cost, $U(M_1)$, for all the views in M_1, is less than or equal to the given cost constraint, S, then, all the views in M_1 will be materialized. Otherwise, we need to further find a subset of M_1, denoted as M_2, such that (i) all the views in $M_1 - M_2$ can be derived from M_1, and (ii) $U(M_2) \leq S$ and the total query processing cost is minimized. Item (i) guarantees that all the queries can be answered using the views in M_1.

In order to make the query cost function to be consistency when comparing the two-phase greedy with other algorithms, the linear query cost function in [51] is revised as follows.

$$g(v) = \frac{\sum_{\exists v_j \in M_1 - M_2} [q(v_j, M_2) - q(v_j, M_2 \cup \{v\})] w(v_j)}{\triangle T_v} \tag{4.2}$$

The function $g(v)$ for obtaining a gain value for a vertex v takes the following factors into consideration. First, adding a new vertex v into M_2 incurs additional maintenance cost for v. However, the newly added vertex v is possible to reduce maintenance cost for the vertices that have already been selected in M_2, because those vertices may be able to use the vertex v to reduce the maintenance cost. Therefore, $\triangle T_v$ may be negative. Second, it considers effectiveness of this selection (vertex v) for all unselected views. It gives a query benefit for selecting this vertex v. Third, the weight for a vertex $v \in M_1$ is the sum of query frequencies for all the queries that choose v as its view. The weight gives a good estimation on the importance of a view $v \in M_1$, and is different from the query frequency for v itself.

In the first step, two-phase greedy heuristic do not consider maintenance

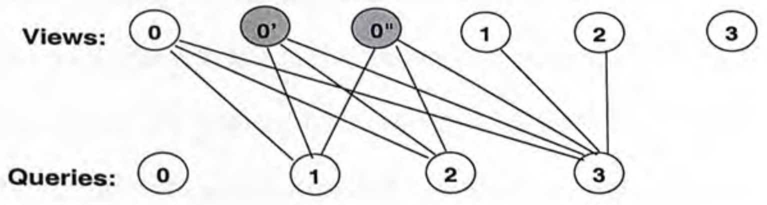


Figure 4.4: A bipartite graph example for Figure 4.1

cost. It reduces the problem to a minimum weighted maximum cardinality matching problem on a weighted bipartite graph, which can be solved in polynomial time. An example is shown in Figure 4.4. The vertex v_0 is possible to answer v_1, v_2 and v_3. The algorithm creates three copies of v_0, each of which has an edge associated with the corresponding query vertex it can be used to answer. For every edge (v_i, q_j), there is one weight assigned to it. Then, the problem of finding M_1 is reduced to a *minimum weighted maximum cardinality matching* problem on a bipartite graph G_B based on G, which can be solved in polynomial time. Apparently, minimum weighted ensures that the sum of query processing cost is minimal while the maximum cardinality ensures that the cost of all the queries is considered as optimization target. In the second step, it further selects a subset of the set of materialized views, M_1, selected in the first step.

4.3.3.1 Discussions

The two-phase greedy takes $O(m^2 + mn^{3/2})$ time, usually substantially better than the inverted-tree greedy and A*-heuristic, where m and n are the number of views and queries in the bipartite graph G_B. However, it gives neither quantitative analysis of quality of the solution nor experiment results. As the running example shows, in Table 4.2, the maintenance cost constraint is the minimum cost constraint that allows all vertices to be selected as materialized views. However, two-phase greedy delivers an approximate solution instead of a full set of materialized views. It is because during the minimum weighted maximum cardinality matching, it cannot fully select all the views. In addition, during the minimum weighted maximum cardinality matching, the view cannot match to itself. For example, in Figure 4.4, there is no edge from v_0 (a view) to v_0 (a query). It is because, if they do so, then the minimum weighted maximum cardinality matching solver will always select a view to answer itself and in result M_1 will be equal to N.

4.3.4 Integrated Greedy

The integrated greedy is summarized in Algorithm 4. When no views are selected, the total query processing cost for all the queries is very large. Then the algorithm will reduce the query processing cost by materializing views, one-by-one, as long as the total maintenance cost is bounded within the cost constraint.

Let M be the set of materialized views having been selected, and $U(M)$ be the total maintenance cost for the views in M. Recall that $\tau(G, M)$ is the total query processing cost for answering all the queries. When considering a view $v \in V(G) - M$ to be materialized, the net increase in the maintenance cost is $\triangle T_v = U(M \cup \{v\}) - U(M)$, and the amount of query processing cost reduction is $\tau(G, M) - \tau(G, M \cup \{v\})$ by spending $\triangle T_v$ unit costs. Thus, each time, it chooses a view $v \notin M$ to materialize such that the gain benefit $g(v)$ brought by

v is the maximum. The function $g(v)$ is defined as follows.

$$g(v) = \frac{\tau(G, M) - \tau(G, M \cup \{v\})}{\triangle T_v} \tag{4.3}$$

The gain benefit is similar to that used in the inverted-tree greedy in [36].

In brief, in the integrated greedy (Algorithm 4), it first selects a vertex v_0 that gives the maximum benefit when there is no view being selected. That vertex is the first vertex in the set of views, M. Next, in each iteration, it selects a vertex that will give the maximum benefit in the current iteration. Selection of a vertex in an iteration is independent from other selections. The integrated greedy can reach an optimal solution in Tables 4.2 and 4.4. However, it only reaches a near-optimal solution in Table 4.3. The reason is that it has to give up the greatest gain benefit vertex, v_1, at one stage, i, because the total maintenance cost exceeds the given cost constraint. But, in the later selections $j > i$, v_1 will never be able to be selected again.

4.3.4.1 Discussions

The integrated greedy is very similar to inverted-tree greedy. The integrated greedy is reexamined in comparison with the inverted-tree greedy by considering the following two issues: (a) the inverted-tree greedy needs to consider every inverted tree sets, and (b) the inverted-tree greedy requires the query benefit per unit effective maintenance cost for the newly selected views to be greater than the previously selected view set. The item (a) makes the inverted-tree greedy to be exponential in the number of vertices, in the worst case. As for the item (b), because the query benefit per unit effective maintenance costs intends to

Algorithm 4 A Integrated Greedy Heuristics [51]

```
Input: A graph $G(V, E)$ and a maintenance cost constraint $S$.
Output: a set of materialized views.
1: begin
2: $M \leftarrow \phi$;
3: Let $v_0$ be the first vertex with maximum $g(v_0)$;
4: $M \leftarrow \{v_0\}$;
5: $\triangle S \leftarrow S - U(M)$;
6: while $\triangle S > 0$ do
7:     $gain \leftarrow 0$;
8:     for each $v \in V(G) - M$ do
9:         $g(v) = (\tau(G, M) - \tau(G, M \cup \{v\})) / \triangle T_v$;
10:        if $g(v) > gain$ then
11:            $gain = g(v)$; $v_0 = v$;
12:        end if
13:    end for
14:    if $(\triangle S - \triangle T_{v_0}) > 0$ then
15:        $\triangle S = \triangle S - \triangle T_{v_0}$;
16:        $M \leftarrow M \cup \{v_0\}$;
17:    end if
18: end while
19: return $M$;
20: end
```

decrease when more vertices are added into the view set, the inverted-tree greedy is difficult to select more vertices. Instead, the integrated greedy uses the query benefit per unit maintenance costs. It attempts to add a vertex that will give the maximum gain into the view set. So it is weaker than the above item (b). Besides, the integrated greedy selects views one-by-one, which will significantly reduce the view selection time.

4.4 A Performance Study

Some results of the extensive performance study will be presented in this section. All the algorithms were implemented using C++ language. The maximum-weight matching solver implemented by Ed Rotherg who implemented H. Gabow's N-cube weighted matching algorithm [24] is used. It is used to find the minimum weighted matching by replacing a cost, c, on an edge with $c_{max} - c$, where c_{max} is a maximum value for all costs. All the algorithms used the same function, $q(v, M)$, to compute query processing cost and the same function, $m(v, M)$, to compute maintenance cost.

For a small dependence lattice (up to 16 elements), five different algorithms are compared: the optimal, the inverted-tree greedy, the A*-heuristic, the two-phase greedy, and the integrated greedy. Using a large dependence lattice (up to 256 elements), the scalability of the two-phase greedy and the integrated greedy is reported. These experiments were done on a Sun Blade/1000 workstation with a 750MHz UltraSPARC-III CPU running Solaris 2.8. The workstation has a total physical memory of 512M . The notations and definitions, together with the default values, for all the parameters are summarized in Table 4.5.

Given a dependent lattice $(L, \prec)$ of size N, a directed acyclic graph $G(V, E)$ is constructed. A vertex, v, has three weights, R_v, its update frequency and query frequency. An edge, from v to u, has two weights: $Q_{(u,v)}$ and $U_{(u,v)}$. These weights are assigned to the graph $G(V, E)$ as follows. First, N distinctive table sizes (R_v) are randomly generated. The N table sizes are randomly picked up and assigned to the vertices on a condition that the table sizes of ancestors of a vertex are greater than that of the vertex. Query frequencies are assumed to follow a

Notation	Definition (Default Values)
N	the number of vertices (16)
T	the cost constraint
θ_q	Zipf distribution factor for query frequency (0.2)
θ_u	Zipf distribution factor for update frequency (0.0)
R_v	table sizes for a vertex v
$Q_{(v,u)}$	query processing cost for a vertex u using v
$U_{(v,u)}$	maintenance cost for a vertex u using v

Table 4.5: System parameters.

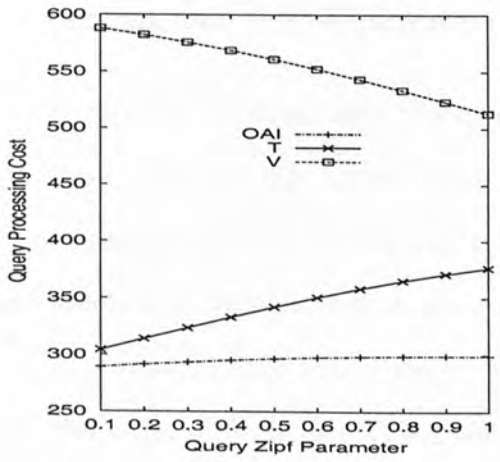
Zipf distribution. When the raw table is updated, all materialized views need to be updated. That is all vertices have the same update frequencies. Query frequencies are randomly assigned to all vertices. Given an edge from v to u, (v, u), The maintenance cost of u using v is smaller than the query processing cost of u using v. Maintenance cost is more related to the table size of u. In this set of tests, $Q_{(v,u)}$ is a number smaller than the table size of v, (R_v). $U_{(v,u)}$ is about one 10-th of the table size u, (R_u). It is important to know that the cost function $q(v, u)$ $(m(v, u))$ considers both table sizes and query processing costs (maintenance costs), associated with edges.

In order to compare the performance of the inverted-tree greedy, A*-heuristic, two-phase greedy and integrated greedy, an algorithm, called optimal algorithm, is implemented for finding the optimal solution. To find the optimal set of materialized views to precompute, the optimal algorithm enumerates all possible combinations of views, and find a set of views by which the query processing cost is minimized. Its complexity is $O(2^N)$. We abbreviate the optimal algorithm as `O`, the inverted-tree greedy as `V`, A*-heuristic as `A`, two-phase greedy as `T` and integrated greedy as I in the following figures.

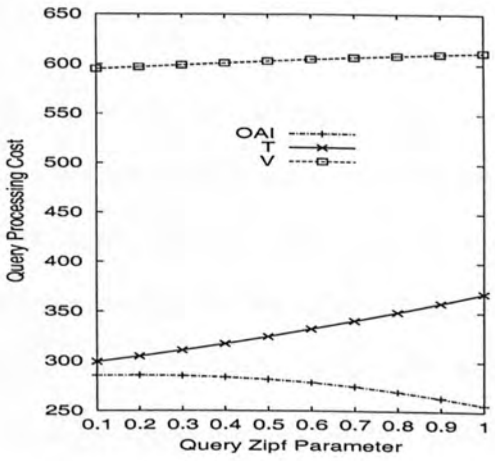
Exp-1: The impacts of query frequencies

First, the impacts of query frequencies are investigated. Query frequencies follow a Zipf distribution. In Figure 4.5, the number of vertices is 16, and the maintenance cost constraint is $0.8 \times C_{min}$, where C_{min} is the minimum maintenance cost constraint for all vertices to be selected as views. Query frequencies are varied by increasing the Zipf factor from 0.1 to 1.0. The high query frequencies are assigned to the vertices in three ways: (*i*) high level (close to the top), (*ii*) middle level, and (*iii*) low level, which are shown in Figure 4.5 (a), (b) and (c), respectively. The assignment of high query frequencies in the graph will affect the set of views to be materialized. In this testing, the A*-heuristic and integrated greedy reach an optimal solution in all cases.

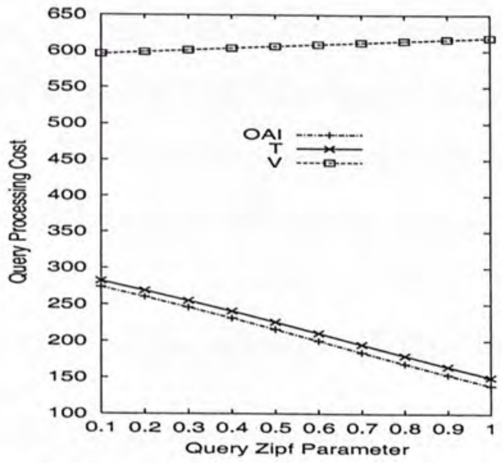
- The high query frequencies are assigned to the vertices at the high level (close to top) (Figure 4.5 (a)): The increase of query processing cost for the four algorithms is due to the fact that the high level vertices have large query cost. In contrast, the query processing cost for the inverted-tree greedy decreases. It is because that it always attempts to select high level vertices, and they are frequently retrieved.
- The high query frequencies are assigned to the vertices at the low level (Figure 4.5 (c)): The query processing cost for all algorithms is opposite to Figure 4.5 (a).
- The high query frequencies are assigned to the vertices at the middle level (Figure 4.5 (b)): The decrease of query processing cost for the three algo-



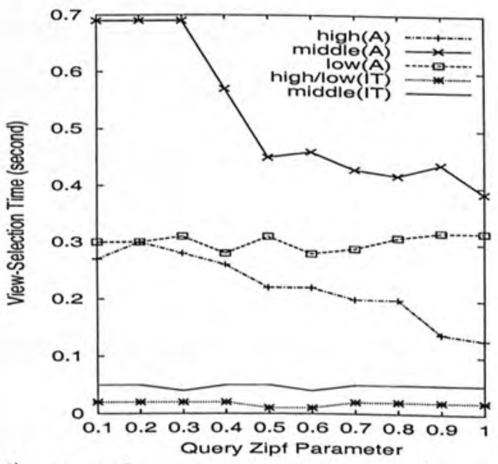
(a) Query frequency changes (high level)



(b) Query frequency changes (middle level)



(c) Query frequency changes (low level)



(d) view selection time v.s. query frequency changes (`high/low/middle(O)` is over 200. `high/low/middle(V)` is over 460.)

Figure 4.5: The impacts of query frequencies

rithms is due to the fact that the middle level vertices have lower query cost and high query frequencies. The query processing cost for the two-phase greedy and inverted-tree greedy increases, because the query frequencies of the high level vertices increase.

Figure 4.5 (d) illustrates the relationship between view selection time and Zipf factor. The five algorithms spend longer time while the high query frequencies are assigned to the vertices at the middle level. All algorithms spend the same amount of time on both high and low cases except for the A*-heuristic. It is because that, at each search stage, A*-heuristic needs to calculate the $g(x)$ and $h(x)$ of downward branches. When the high query frequencies are assigned at the top level, it is faster for A*-heuristic to reach the leaf vertices as the difference of query cost between vertices is large. However, at the middle level, the difference of query cost at each vertex becomes small, the A*-heuristic needs longer time to search other branches. Compared with the top case, A*-heuristic spends more time in the low level case. The other algorithms, the integrated greedy and two-phase greedy take nearly constant time.

Exp-2: The impact of the maintenance cost constraint

In this testing, the performance of the four algorithms are investigated by varying the maintenance cost constraint. The number of vertices is 16 and the Zipf factor is 0.2. The high query frequencies are assigned at the high level. The minimum maintenance cost constraint, denoted C_{min}, allows all vertices to be selected as materialized views.

The results are shown in Figure 4.6 (a), (b) and (c), where the maintenance cost constraint used is $p \times C_{min}$ where p varies from 0.7 to 1.0. (Note: when $p < 0.7$, none of the algorithms can select any views.) A larger p implies that it is likely to select more views. When $p = 1$, it means that all vertices are possibly selected. In Figure 4.6 (a) and (b), the optimal is chosen as the denominator to compare. The inverted-tree greedy did not include (V) in these figures, because it makes the other differences less visible. For reference, the maintenance costs for the inverted-tree greedy are, as pairs of (p, maintenance cost), (0.70, 1), (0.80, 0.88), (0.90, 0.77) and (1.0, 0.69). The query processing costs for the inverted-tree greedy are, as pairs of (p, query-processing-cost), (0.70, 1), (0.80, 1.99), (0.90, 6.42), (1.0, 11.25). The inverted-tree greedy is inferior to all others. The query processing cost is the reciprocal of the maintenance cost (Figure 4.6 (a) v.s. Figure 4.6 (b)).

The performance study shows that the maintenance cost constraint is the most critical factor that affects the quality of the heuristic algorithms. Some observations are given below.

- **Issue 1**: As Figure 4.6 (a) and (b) suggested, in a multidimensional data warehouse environment, Issue 1 has less impacts on greedy algorithms. Explanation is shown as below. When selecting a view v, the total maintenance cost, $U(M \cup \{v\})$, depends on two factors: update cost, $m(v, M)$, and update frequency of the vertex v, g_v. Recall $m(u, v)$ is the sum of the maintenance costs associated with the edges on the shortest path from v to u, so the total maintenance cost will be greater than zero, when a new vertex is added. On the other hand, since u is derived from v, the update frequency of v should be greater than or equal to u's update frequency in a multidimensional data warehouse environment. As a result, $\triangle T_v > 0$.

Therefore, Issue 1 is not a real issue in a multidimensional environment.

- **Issue 2**: The two-phase greedy and the inverted-tree greedy cannot select all views to materialize, even though the maintenance cost constraint allows it. Two-phase greedy only gives an approximate solution.
- **Issue 3**: The greedy algorithms become unstable when the maintenance cost constraint is over $0.9 \times C_{min}$. The integrated greedy is impossible to select proper views. The reasons are given in Section 4.1.
- **Issue 4**: The A*-heuristic cannot always find optimal solutions, in particular, when it is over $0.9 \times C_{min}$. For instance, when $p = 0.95$, A*-heuristic selected $\{v_0, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{12}\}$ of a 16 vertex graph. Its maintenance cost is 27.25, and its query processing cost is 93.18. But, the optimal solution included $\{v_0, v_1, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}\}$. The maintenance cost and query processing cost for the optimal solution are, 26.75 and 75.54, respectively. It is because A*-heuristic estimates the expected benefit, $h(x)$, which might not be accurate. It points out a very important fact for a greedy algorithm, if it misses selecting a vertex, (v_2 in this case), it will affect the other selections.

Figure 4.6 (c) shows that the view selection time for the five algorithms. Because the number of vertices is fixed ($N = 16$), all the view selection time for all the cases are the same. The inverted-tree greedy consumes more view selection time than the optimal (the middle). It is because that, for computing the optimal solution, only all 2^{16} subsets are checked once. Recall that the inverted-tree greedy needs to check the maximum query benefit per unit effective maintenance costs at every stage, and check powersets repeatedly. All the other three algorithms: the

A*-heuristic, the integrated greedy and the two-phase greedy can be efficiently processed.

Exp-3: Scalability

In this experimental study, all the parameters are fixed except for the number of vertices. Two sets of results are shown. Figures 4.7 (a), (b) and (c) show a comparison of the five algorithms: the optimal, the integrated greedy, the two-phase greedy, the inverted-tree greedy and the A*-heuristic by varying the number of vertices, N, from 4 to 16. The maintenance cost constraint, C_{min}, is the minimum maintenance cost constraint that allows all vertices to be selected. In Figure 4.7 (d), (e) and (f), the integrated greedy is compared with the two-phase greedy by varying the number of vertices, N, from 4 to 256. The maintenance cost constraint is $0.8 \times C_{min}$.

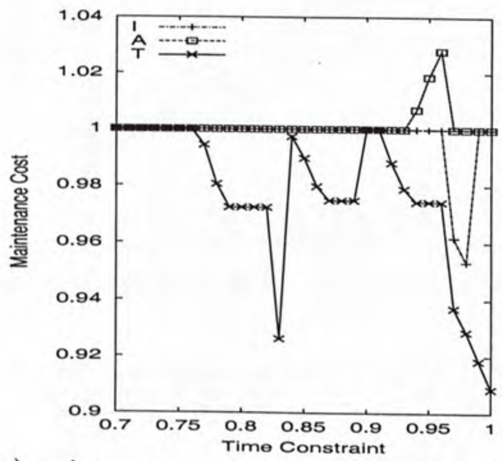
Figure 4.7 (b) shows the query processing costs. Due to the number of views to be selected, as shown in the previous testings, the A*-heuristic and integrated greedy always give an optimal solution. The two-phase greedy gives a feasible and good approximation. The A*-heuristic, integrated greedy and two-phase greedy outperform the inverted-tree greedy significantly. Figure 4.7 (c) shows the view selection time of these algorithms. The optimal algorithm is exponential to the number of N. The inverted-tree greedy is also exponential to N, and takes longer time to reach the solution than the optimal algorithm. The A*-heuristic is exponential to N in the worst case. When the number of vertices is over 120, the view selection time for the integrated greedy increases exponentially. On the other hand, the view selection time for the two-phase greedy is small. In addition, the query processing time for the two-phase greedy is acceptable when the number of vertices is large. In conclusion, when $N \leq 120$, the integrated greedy is recommended to use. When $N > 120$, the two-phase greedy is a reasonable choice in practice.

4.5 Summary

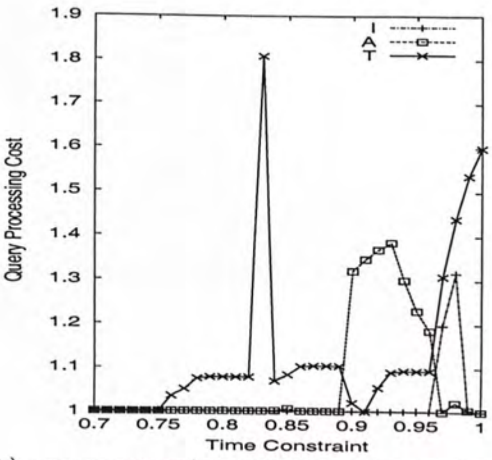
The selection of views to materialize is one of the most important issues in designing a data warehouse. The maintenance cost view selection problem has been re-examined under a general cost model. Heuristic algorithms can provide optimal or near optimal solutions in a multidimensional data warehouse environment under certain conditions: the update cost and update frequency of any ancestor of a vertex is greater than or equal to the update cost and update frequency of that vertex, respectively.

In the extensive performance studies, the A*-heuristic, integrated greedy and two-phase greedy significantly outperformed the inverted-tree greedy. The greedy algorithms are not stable when the maintenance cost constraint is over 90% of the minimum maintenance cost constraint that allows all views to be selected.

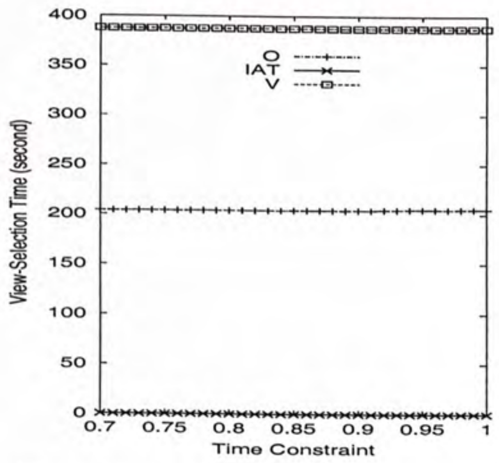
The two-phase greedy and the integrated greedy are scalable. When the number of vertices in a graph is less than or equal to 120, the integrated greedy can compute fast and give an optimal solution. When the number of vertices is greater than 120, the two-phase greedy is recommended to use due to the efficiency. The two-phase greedy gives a good approximate solution, which is close to the optimal solution, in the testing for a small number of vertices (16).



(a) maintenance cost v.s. cost constraint (0.7-1)

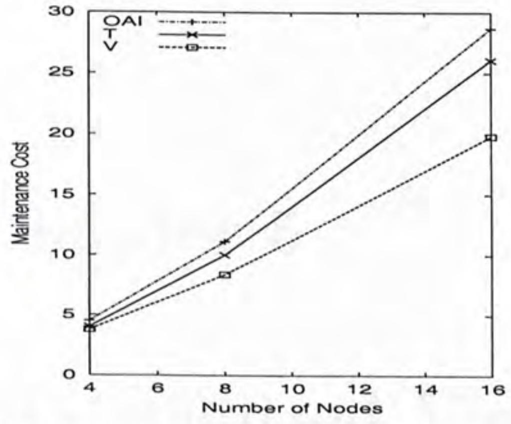


(b) query processing cost v.s. cost constraint (0.7-1)

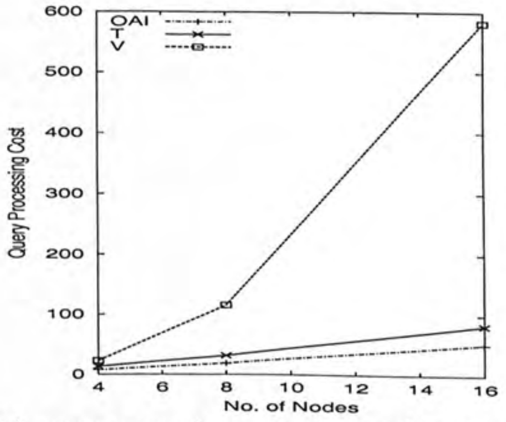


(c) view-selection time v.s. cost constraint

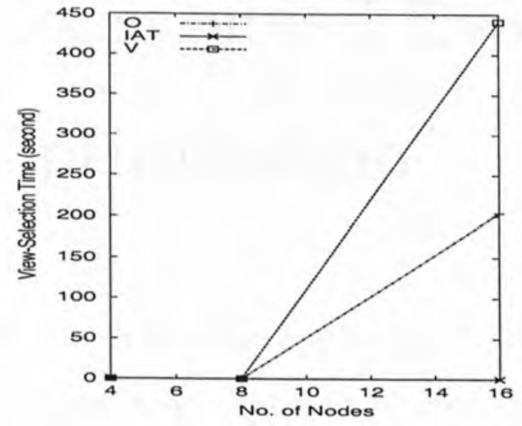
Figure 4.6: The impacts of the maintenance cost constraint



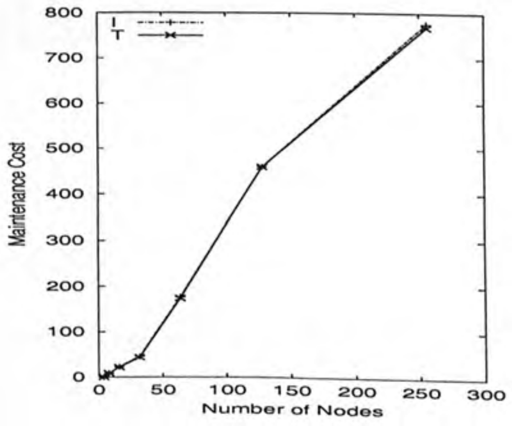
(a) maintenance cost v.s. number of vertices



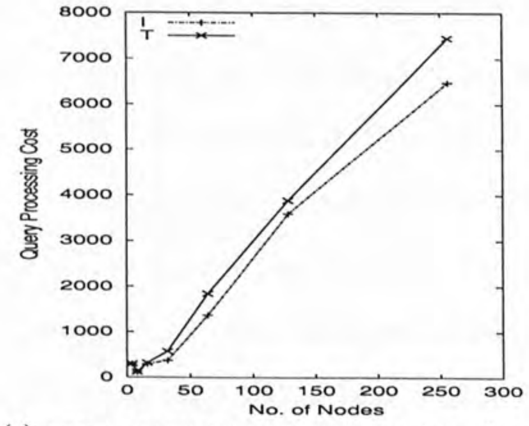
(b) query processing cost v.s. number of vertices



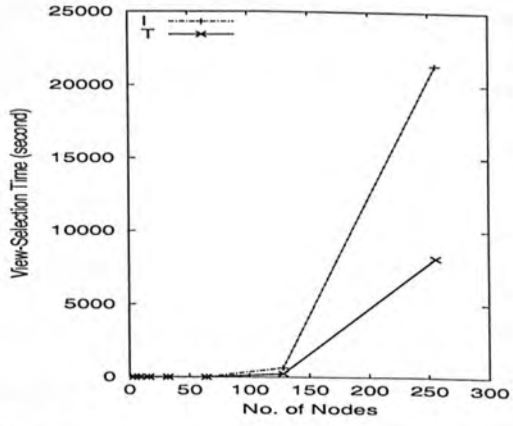
(c) view-selection time v.s. number of vertices



(d) maintenance cost v.s. number of vertices



(e) query processing cost v.s. number of vertices



(f) view-selection time v.s. number of vertices

Figure 4.7: Scalability

Chapter 5

Materialized View Selection as Constrained Evolutionary Optimization

The search space for possible materialized views may be exponentially large for the maintenance cost view selection problem. A heuristic algorithm often has to be used to find a near optimal solution. In this chapter, a new constrained evolutionary algorithm is proposed for the maintenance cost view selection problem. Constraints are incorporated into the algorithm through a stochastic ranking procedure. No penalty functions are used.

In this chapter, motivation will be given in Section 5.1. Constraint handling will be described in Section 5.2.1. The evolutionary algorithm will be introduced in Section 5.2.2. Extensive performance studies will be reported in Section 5.3. A short summary will be presented in Section 5.4.

5.1 Motivation

Computational intelligence plays a significant role in supporting the design of intelligent systems [6, 87]. Hence, computational intelligence is highly desirable to assist the design of a data warehouse system as a network service that collects data from different remote data sources and disseminates high-quality data analysis to decision makers locally and remotely in an efficient way. Zhang et al. [84] proposed an evolutionary approach to materialized view selection, but they did not consider any constraints. Lee and Hammer [47] made the first attempt to solve the maintenance cost view selection problem using evolutionary algorithms. They tested nine different ways to add a penalty function to the original objective function. However, their results were less satisfactory. They did not show any results for problems larger than 20 views, even though they mentioned that they did try to tackle large problems.

In this chapter, a new constrained evolutionary algorithm is proposed for the maintenance cost view selection problem. The algorithm does not use any penalty functions. Instead, a novel stochastic ranking procedure is used. It is the first time to adopt the technique to this specific problem. The extensive experimental studies show that feasible solutions can be easily obtained by the stochastic ranking approach. In addition, the new constrained evolutionary algorithm explores the search space better than the other existing algorithms. It can scale well with the problem size.

5.2 Evolutionary Algorithms

Evolutionary computation techniques have been received a great attention [52]. Some evolutionary algorithms were proposed to solve the maintenance cost view selection problem, because of its robustness. [84] first proposed an evolutionary approach for materialized views selection problem without considering any constraints. [47] made the first attempt to solve the maintenance cost view selection problem by evolutionary algorithms, but did not show any experimental results for problems larger than 20 views.

Here, a new evolutionary algorithm is proposed and it fits the maintenance cost view selection problem well. First, a pool of bit string genomes are generated randomly. This is the initial population. Each genome represents a candidate solution to the problem to be solved. The length of this genome is the total number of vertices in the lattice. 1 and 0 mean that the vertices need to be materialized or not, respectively. A genome can be formulated as: genome = $(x_1$ x_2 x_3 $\cdots$ $x_N)$, where N is the total number of vertices in the lattice. Here, $x_i = 1$ if view v_i is selected for materialization and $x_i = 0$ if view v_i is not selected for materialization. For example, in Figure 5.1 (which has been introduced in Chapter 3), N is 4. Genome = (0 1 1 1) means that three views, v_1, v_2, and v_3, are materialized. During the crossover and mutation processes, good candidates will survive and poor candidates will die. In the following, penalty methods, stochastic ranking and the new evolutionary algorithm will be introduced.

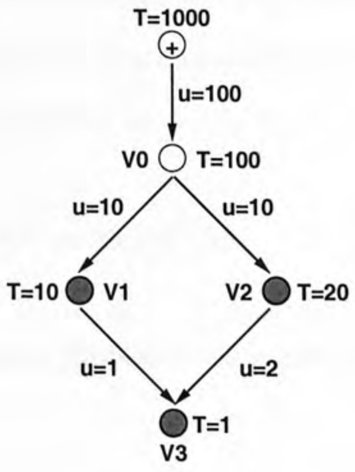


Figure 5.1: An example of view maintenance

5.2.1 Constraint Handling: Penalty v.s. Stochastic Ranking

Lee and Hammer in [47] used a genetic algorithm with the penalty method to set a static penalty coefficient, r_g, to find a near-optimal solution to the maintenance cost view selection problem. (Hereafter it is called as LEE algorithm.) In brief, we introduce their penalty-based approaches, and express our concerns.

Let $x = (x_1, x_2, \cdots, x_N)$, for $x_i = 0$ or 1, and $M_x = \{v_i | x_i = 1, i = 1, 2, \cdots, N\}$, the original maintenance cost view selection problem can be formulated as follows:

$$\begin{aligned} &Maximize\ f(x) = B(G, M_x) = \tau(G, \phi) - \tau(G, M_x) \\ &subject\ to\ :\ U(M_x) \leq S \end{aligned}$$

This is a combinatorial optimization problem. The common method for dealing with constrained optimization problems is to introduce a penalty function to the objective function to penalize the solutions which violate the constraint. Usually, the penalty function can be defined as

$$\phi(x) = max\{U(M_x) - S, 0\}.$$

Then, the original optimization problem with constraints can be transformed into an unconstrained one:

$$Maximize\ f(x) = B(G, M_x) - r_g \cdot \phi(x),$$

where r_g is the penalty coefficient. The choice of the penalty coefficient r_g is very important [65]. A too small r_g will result in under-penalization, i.e. infeasible solutions not being penalized enough. So many infeasible solutions may be found. A too large r_g will result in over-penalization, some "beneficial" infeasible solutions being penalized too much during the course of evolutionary optimization. The reason why some infeasible solutions may be beneficial during the course of evolution is that, when feasible regions in the whole search space are disjoint, some infeasible regions may act as bridges among feasible regions. If a too large r_g makes such infeasible solutions inaccessible, then it is difficult for an evolutionary algorithm to jump from one feasible region to another one which may have better fitness values. Thus, an overly large r_g may prevent a good feasible solution from being found. Because of the importance of r_g, there has been much research work done on it. However, the setting of r_g is a difficult problem. It is difficult to find a precise value to realize the right balance between the original objective function and the penalty function. Even most dynamic setting methods, which start with a low r_g value and end with a high r_g, are not likely to work well for problems for which the unconstrained global optimum is far away from its constrained one [65]. In [47], the fitness function $f(x)$ has three forms:

Subtract mode(S)

$$\begin{aligned} f(x) &= B(G, M_x) - Pen(x), & if\ B(G, M_x) - Pen(x) \geq 0, \\ &= 0, & if\ B(G, M_x) - Pen(x) < 0. \end{aligned}$$

Divide mode(D)

$$\begin{aligned} f(x) &= B(G, M_x)/Pen(x), & if\ Pen(x) > 1, \\ &= B(G, M_x), & if\ Pen(x) \leq 1. \end{aligned}$$

Subtract and Divide mode(SD)

$$\begin{aligned} f(x) &= B(G, M_x) - Pen(x), & if\ B(G, M_x) > Pen(x), \\ &= B(G, M_x)/Pen(x), & if\ B(G, M_x) \leq Pen(x)\ and\ Pen(x) > 1, \\ &= B(G, M_x), & if\ B(G, M_x) \leq Pen(x)\ and\ Pen(x) \leq 1. \end{aligned}$$

Their penalty functions[1] also have three forms:

$$\begin{aligned} &Logarithmic\ penalty\ (LG): & Pen(x) = log_2(1 + \rho \cdot (U(M_x) - S)) \\ &Linear\ penalty\ (LN): & Pen(x) = 1 + \rho \cdot (U(M_x) - S) \\ &Exponential\ penalty\ (EX): & Pen(x) = (1 + \rho \cdot (U(M_x) - S))^2 \end{aligned}$$

Whichever of the three fitness function forms above is used in practice, this can be considered as a penalty method of a static r_g value. We note that in the *subtract*

[1]EX should really be called "polynomial", but let us use what the authors used.

mode, in fact, $r_g = 1$. In the other two modes, D and SD, although no explicit r_g values are shown, they are fixed. The unconstrained fitness function $f(x)$ is composed of $B(G, M_x)$ and $Pen(x)$, and this relation does not change in the whole evolutionary process. As it does in numerical function optimization problems, such a penalty method does not work very well in combinatorial optimization problems either. Its experiment results will be reported in Section 5.3.

Since finding an optimal r_g value is difficult and the penalty methods setting a static or dynamic r_g value do not work well for the optimization problem with constraints, [65] put forward a new constraint handling technique, named stochastic ranking, to balance the dominance of the objective and penalty functions for constrained *numerical* optimization. The novel idea of this technique is the introduction of a probability P_f for rank-based selection. During the course of ranking, pairs of two adjacent individuals are compared. If they are both feasible solutions, naturally, they will be compared according to the objective function. However, when either of them is infeasible, the probability of comparing them according to the objective function is P_f, while the probability of comparing them according to the penalty function will be $1 - P_f$. Since P_f is a probability, it gives an opportunity for both the objective and penalty functions to rank a pair. When $P_f > 1/2$, the ranking is biased towards the objective function. When $P_f < 1/2$, the ranking is biased towards the penalty function. So P_f can balance the objective and penalty functions more directly, explicitly and conveniently. By adjusting P_f, we can adjust the balance between the objective function and the penalty function easily. Moreover, it does not spend any extra computing cost for setting r_g values since it does not use any penalty terms. In practice, it usually sets $P_f < 1/2$ to reduce the ratio of infeasible solutions to the whole in the final generation. For different optimization problems, we will conduct experiments using different P_f values.

5.2.2 The New Stochastic Ranking Evolutionary Algorithm

Based on the analysis in the last section, we observe that the stochastic ranking approach will have better performance for this problem than the LEE method. Although stochastic ranking has been used for constrained numerical optimization problems and shown good performance using (μ, λ) evolution strategy [65, 67], it is unclear whether it is effective for combinatorial optimization problems. This chapter presents the first attempt towards generalizing this approach to combinatorial optimization problems, using a operator sequence, crossover-mutation-selection, as used in generic algorithms.

The basic framework of the evolutionary algorithm is shown in Algorithm 5. Similar to most evolutionary algorithms, both crossover and mutation are used. The crossover operator is *uniform crossover*, as shown in Algorithm 6. It exchanges the information of two chromosomes to generate two new chromosomes. The mutation operator is similar to the most usually used one, as shown in Algorithm 7. The probability that every bit of every gene will be flipped is P_m. The key difference from most evolutionary algorithms is the stochastic ranking procedure used. The stochastic ranking algorithm is based on [65], but modified in some places for the specific problem of materialized views selection, which is shown in Algorithm 8. It is used for ranking the union of new and old individuals. The ranking procedure is similar to bubble-sort. In every sweep of N, every two adjacent individuals are compared. If there is no any change of individual's rank after a sweep, then this bubble-sort-like procedure can be terminated. Note: N is the number of vertices in the lattice.

It is worth noting that, for dealing with numerical optimization problems, [65] uses a (μ, λ) evolution strategy, and set the truncation level as $\mu/\lambda \approx 1/7$, where μ and λ are the number of parents and the number of children, respectively. In this chapter, the materialized view selection problem is treated as a combinatorial optimization problem using a typical operator-sequence as used in genetic algorithms. Like most genetic algorithms, λ offsprings are generated from μ parents, where $\lambda = \mu$.

Algorithm 5 The Basic Framework of the New Evolutionary Algorithm (denoted `EA`)

```
Parameter: population size $P$
 1: begin
 2: Generate the initial population $G(0)$;
 3: repeat
 4:     $t = t + 1$;
 5:     $G_1(t) \leftarrow$ UniformCrossover($G(t-1)$); {refer to Algorithm 6.}
 6:     $G_2(t) \leftarrow$ Mutation($G_1(t)$); {refer to Algorithm 7.}
 7:     $S \leftarrow$ StochasticRanking($G(t-1) \cup G_2(t)$), which sorts $G(t-1) \cup G_2(t)$ to
        an ordered individuals sequence $S$ of size $2 \times P$; {refer to Algorithm 8.}
 8:     $G(t) \leftarrow$ the anterior $P$ individuals of $S$;
 9: until (termination condition is satisfied)
10: end
```

Algorithm 6 UniformCrossover

```
Input: Generation G
Parameter: crossover probability $P_c$
 1: begin
 2: Select a pair of individuals of G randomly : $g_1 = (b_1, b_2, \cdots, b_N)$ , $g_2 = (c_1, c_2, \cdots, c_N)$;
 3: sample $u \in U(0,1)$;
 4: if $(u < P_c)$ then
 5:    for every bit $i$ of individual do
 6:       sample $r$, either 0 or 1;
 7:       if $(r = 1)$ then
 8:          the bit $i$ of $g_1$'=$b_i$;
 9:          the bit $i$ of $g_2$'=$c_i$;
10:       else
11:          the bit $i$ of $g_1$'=$c_i$;
12:          the bit $i$ of $g_2$'=$b_i$;
13:       end if
14:    end for
15: else
16:    $g_1' = g_1$;
17:    $g_2' = g_2$;
18: end if
19: Repeat the above procedure $P/2$ times, which will generate $P$ new individuals;
20: end
```

Algorithm 7 Mutation

```
Input: Generation G
Parameter: mutation probability $P_m$
1: begin
2: for every individual in $G$ do
3:    for every bit in the individual do
4:       Mutate the bit with the probability of $P_m$;
5:    end for
6: end for
7: end
```

Algorithm 8 Stochastic Ranking

Input: $\lambda = 2 \times P$ individuals $\{I_j | j = 1, ..., \lambda\}$
Parameter: balance parameter P_f
Note: the fitness function: $f(x) = B(G, M_x)$, the penalty function: $\phi(x) = max\{U(M_x) - S, 0\}$. The N is set to be λ as analyzed in [65].

```
1: for $i = 1$ to $N$ do
2:    for $j = 1$ to $\lambda - 1$ do
3:       sample $u \in U(0,1)$;
4:       if $(\phi(I_j) = \phi(I_{j+1}) = 0)$ or $(u < P_f)$ then
5:          if $f(I_j) < f(I_{j+1})$ then
6:             $swap(I_j, I_{j+1})$;
7:          end if
8:       else
9:          if $\phi(I_j) > \phi(I_{j+1})$ then
10:            $swap(I_j, I_{j+1})$;
11:         end if
12:      end if
13:   end for
14:   if no $swap$ done then
15:      break;
16:   end if
17: end for
```

5.3 Experimental Studies

In this section, some results of experimental studies are presented. All the algorithms were implemented using C++ language. These experiments were done on a Sun Blade/1000 workstation with a 750MHz UltraSPARC-III CPU running Solaris 2.8. The workstation has a total physical memory of 512M.

5.3.1 Experimental Setup

In order to evaluate the performance of the new stochastic ranking evolutionary algorithm (EA) and the best result of the penalty-based algorithm (LEE), an algorithm, optimal algorithm, is implemented for finding the optimal solution. To find the optimal set of materialized views to precompute, all possible combinations of views are enumerated to find a set of views by which the query processing cost is minimized. Its complexity is $O(2^N)$, where N is the number of vertices. For a small number of lattice (16 vertices), we compare among EA, LEE, A*-heuristic [36] and the optimal algorithm. For a large number of lattice up to 256 vertices, the scalability of the EA algorithm will be reported. In the following, LEE is the best result given in [47].

Table 5.1 summarizes all the parameters together with the default values used in the following experiments. The maintenance cost constraint is a crucial condition for the maintenance cost view selection problem. In the experiments, the minimum maintenance cost constraint, C_{min}, is the minimum value which allows all views to be selected as materialized views.

5.3.2 Experimental Results

5.3.2.1 Feasibility of the Solutions

First, we investigate the feasibility of the solutions of EA by varying the P_f value. In Figures 5.2 (a) to (d), the number of vertices is 32. The results were averaged over 30 independent runs of EA algorithm. In Figure 5.2 (a), the y-axis indicates the percentage of feasible solutions in the final generation. Recall that maintenance cost constraint has a big effect on the results. In this testing, we try to use

Notation	Definition (Default Values)
N	the number of vertices (16)
θ_q	Zipf distribution factor for query frequency (0.2)
θ_u	Zipf distribution factor for update frequency (0.0625)
R_v	table size for a vertex v
$Q_{(v,u)}$	query processing cost for a vertex u using v
$U_{(v,u)}$	maintenance cost for a vertex u using v
C_{min}	the minimum maintenance cost constraint that allows all vertices to be selected as materialized views
P_f	probability for stochastic ranking function (0.4)
P_m	mutation probability (0.001)

Table 5.1: Notations and definitions of the system parameters used in experiments.

different maintenance cost constraint to see how P_f deals with the maintenance cost constraint. In Figure 5.2 (a), `Cmin(1)` and `Cmin(0.8)` represent the maintenance cost constraint $1 \times C_{min}$ and $0.8 \times C_{min}$, respectively. When the maintenance cost constraint is `Cmin(1)`, the percentage of feasible solution is always one hundred. It shows that `EA` can always find a feasible solution if the maintenance cost constraint is large enough. When the maintenance cost constraint is `Cmin(0.8)`, it shows that P_f can alter the percentage of feasible solutions very easily. When $P_f = 0.5$, the percentage of feasible solutions drops sharply from 100% to 0%. If maintenance cost constraint is reduced to $0.5 \times C_{min}$, the `EA` gets all 0s solutions in the final generation since the maintenance cost constraint is too low to select any vertices.

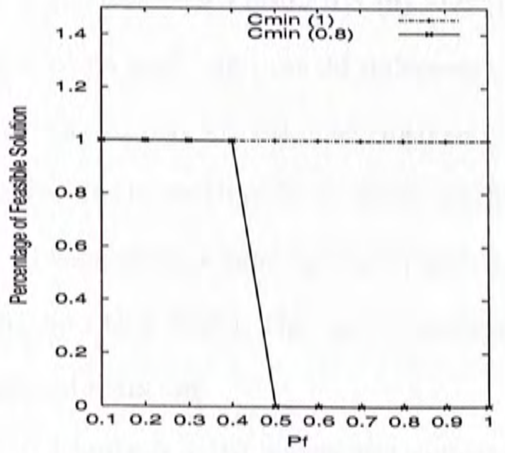
In Figure 5.2 (b), (c) and (d), the optimal solution produced by A*-heuristic is chosen as the denominator to evaluate `EA`. In these three figures, the maintenance cost constraint is $0.8 \times C_{min}$. Figure 5.2 (b) shows the quality of the feasible solutions. The y-axis represents a ratio of the average query processing cost of the feasible solutions over the optimal query processing cost. As expected, when P_f is less than or equal to 0.4, the average query processing cost of feasible solutions is greater than 1, because the query processing cost of the optimal solution is the lowest among all the feasible solutions. In contrast, when $P_f > 0.4$, the average query processing cost of feasible solutions is equal to 0 as there are no feasible solutions found. (Figure 5.2 (a) shows that the percentage of feasible solution is equal to 0 when $P_f > 0.4$.)

Figure 5.2 (c) shows the quality of the infeasible solutions in the final generation. Since the infeasible solutions trade off the maintenance cost with a lower and better overall query processing cost, the average query processing cost of infeasible solutions is less than 1. Figure 5.2 (d) shows the maintenance cost of the infeasible solutions from the optimal maintenance cost. It shows that in the worst case, the average maintenance cost of infeasible solutions will not greater than 1.3 times of the maintenance cost of the optimal solution.

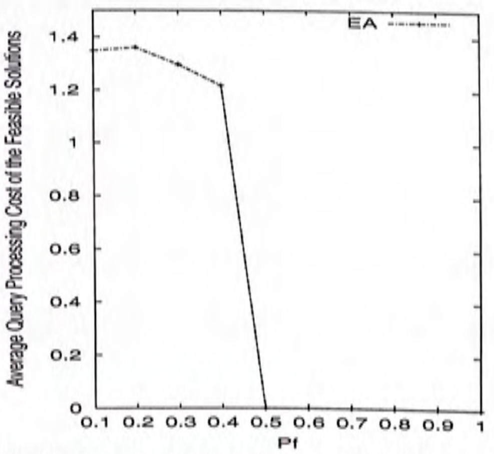
The above testings demonstrate that P_f gives a convenient way to fine-tune the algorithm. By varying the P_f value, `EA` can deal with the maintenance cost constraint well. As a result, we will choose $P_f = 0.4$ as the default P_f value in the subsequent experiments.

5.3.2.2 Optimality of Solutions

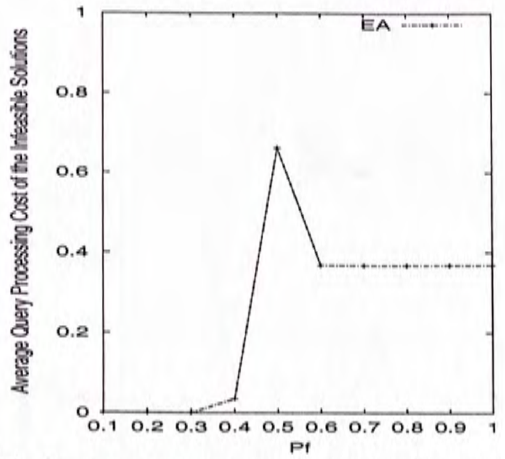
In this experimental study, the performance of `EA`, `LEE`, A*-heuristic and the optimal algorithm are investigated under different maintenance cost constraints. Let the maintenance cost constraint be $p \times C_{min}$. In Figure 5.3 (a) and (b), p



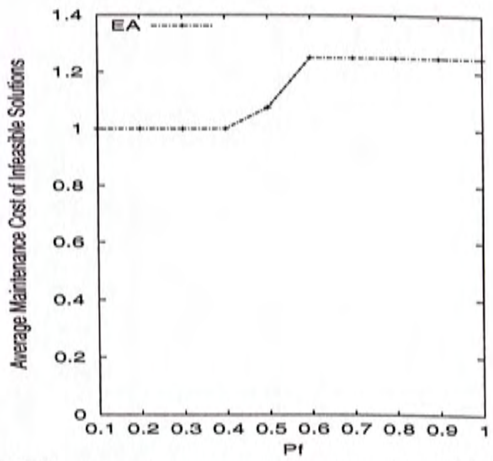
(a) P_f v.s. percentage of feasible solutions



(b) P_f v.s. average query processing cost of feasible solutions



(c) P_f v.s. average query processing cost of infeasible solutions



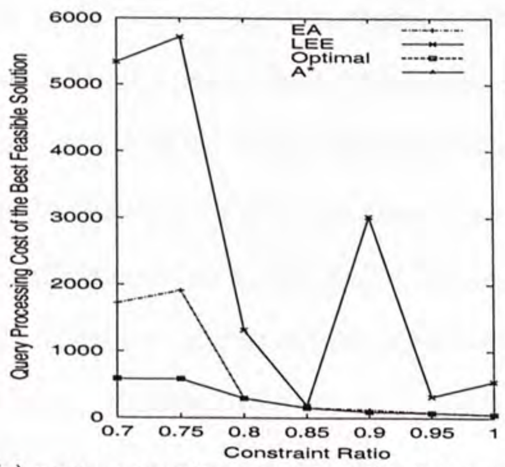
(d) P_f v.s. average maintenance cost of infeasible solutions

Figure 5.2: Feasibility of the solutions by varying the P_f value

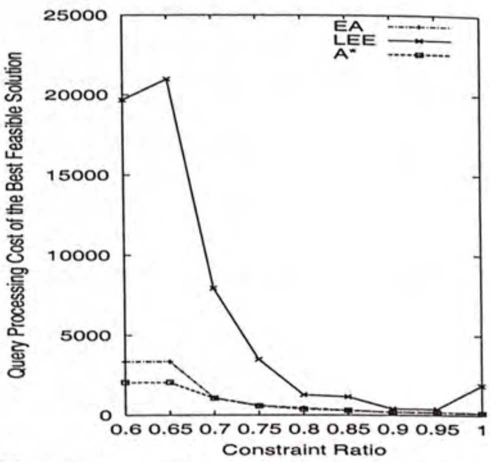
varies from 0.7 to 1. (Note that when $p < 0.7$, none of the algorithms can select any views.) A larger p value implies that it is likely to select more views. When $p = 1$, it means that all vertices may be selected. The number of vertices is 16 and 32 respectively in Figure 5.3 (a) and (b). We took the average query processing costs of EA and LEE over 30 independent runs.

In Figure 5.3 (a), the optimal solution is computed by using exhaustive search. It shows that A*-heuristic performs in the same way as the optimal. The EA always gives a near optimal feasible solution that is very close to the optimal. On the other hand, the query processing cost of LEE is much higher than the optimal solution.

Figure 5.3 (b) shows the comparison among EA, LEE and A*-heuristic. It shows that EA can find near optimal feasible solutions that are very closed to A*-heuristic. EA outperforms the LEE algorithm significantly.



(a) query processing cost v.s. maintenance cost constraint (16 vertices)



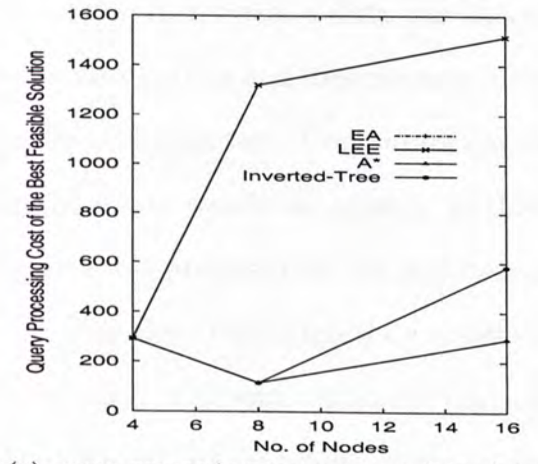
(b) query processing cost v.s. maintenance cost constraint (32 vertices)

Figure 5.3: Optimality of solutions with different maintenance cost constraint

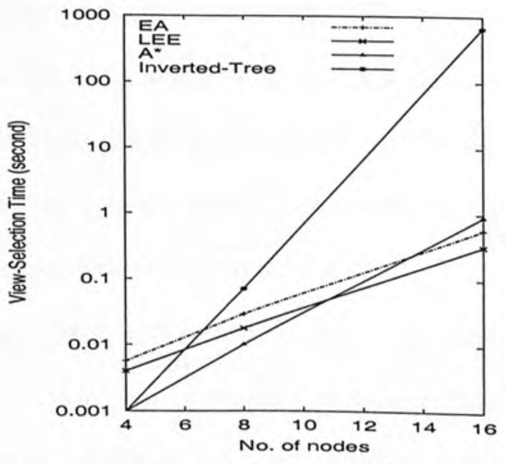
5.3.2.3 Scalability of the Algorithms

There are several existing algorithms for solving the maintenance cost view selection problem. Figure 5.4 shows the performance of four algorithms, namely, `LEE`, `EA`, A*-heuristic and `inverted-tree` [36], when the maintenance cost constraint is $0.8 \times C_{min}$. Figure 5.4 shows a small-scale problem with the number of vertices varies from 4 to 16. As shown in Figure 5.4 (a), `EA` and A*-heuristic performs the best (the same as the optimal). The `inverted-tree` greedy is inferior to `EA` and A*-heuristic but is superior to `LEE`. Figure 5.4 (b) shows the view selection time. The `inverted-tree` greedy cannot deal with large-scale problems, due to its view selection time.

The existing algorithms do not perform well when computing a large dependent lattice. Evolutionary algorithms can explore this search space better. Since A*-heuristic and `inverted-tree` greedy cannot deal with the lattice up to 256, `EA` is compared with `LEE` by varying the number of vertices, N, from 4 to 256. The maintenance cost constraint is $0.8 \times C_{min}$. For the number of vertices from 4 to 64, the average query processing cost for both algorithms is taken over 30 independent runs. When the number of vertices is greater than 128, it ran once due to the longer execution time. In Figure 5.5 (a), `EA` significantly outperforms the `LEE` algorithm in terms of minimizing the query processing cost. However, `EA` took a longer time than `LEE` to find better solutions, according to Figure 5.5 (b). It is worth noting that `EA` is much more likely to find feasible solutions as well while `LEE` tends to get stuck at a poor solution fairly early.

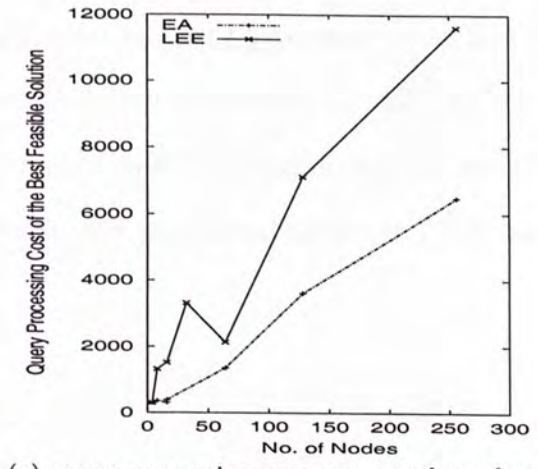


(a) query processing cost v.s. number of vertices

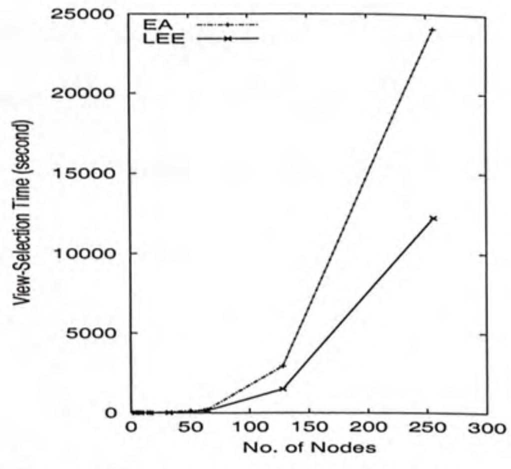


(b) view selection time v.s. number of vertices

Figure 5.4: Four algorithms



(a) query processing cost v.s. number of vertices



(b) view selection time v.s. number of vertices

Figure 5.5: Scalability of algorithm by varying the number of vertices

5.4 Summary

As a network service, a data warehouse system collects data from different remote data sources and disseminates high-quality data analysis to decision makers locally and remotely. Computational intelligence plays a significant role in design of a data warehouse system. In this chapter, a new constrained evolutionary algorithm is proposed for the maintenance cost view selection problem.

The algorithm is based on a novel constraint handling technique — stochastic ranking. Although stochastic ranking has been used in numerical constrained optimization, its suitability for combinatorial optimization was unclear. This chapter demonstrates that a revised stochastic ranking scheme can be applied to constrained combinatorial optimization problems successfully.

The new evolutionary algorithm has been evaluated against both heuristic and other evolutionary algorithms. The experiments results show that `EA` can provide significantly better solutions than previous algorithms in terms of minimization of query processing cost and feasibility. In comparison with the latest evolutionary algorithm, i.e. the `LEE` algorithm [47], `EA` can avoid premature convergence and keep improving the solution, while the `LEE` algorithm tends to get stuck at a poor local optimum fairly early.

Chapter 6

Dynamic Materialized View Management Based On Predicates

For the purpose of satisfying different users' profiles and accelerating the subsequence OLAP queries in a large data warehouse, dynamic materialized OLAP view management is highly desirable. Previous work caches data as either chunk or multidimensional range fragments. The chunk-size or fragment-size need to be determined beforehand statically, and a prepartitioning technique is used. The efficiency of these approaches is at the expense of either space consumption or the restrictions on query types. In this chapter, we focus on ROLAP in an existing relational database system. A dynamic predicate-based partitioning approach is proposed, which can support a wide range of OLAP queries. Extensive performance studies using TPC-H benchmark data on IBM DB2 is conducted. Encouraging results are obtained which indicate that the approach is highly feasible.

In this chapter, Section 6.1 gives the motivation and the brief background of the dynamic view management. Section 6.2 gives some examples. Section 6.3 outlines two static prepartitioning-based view management approaches. In Section 6.4, the new dynamic predicate-based partitioning approach is introduced. The performance results are presented in Section 6.5. Section 6.6 concludes this chapter.

6.1 Motivation

As discussed in Chapter 2, precomputing OLAP queries becomes a key to achieve high performance in data warehouses. Many works [4, 30, 35, 36, 40, 51, 70] study the static view management problem. However, as different users may have different preferences, they may be interested in similar but different portions of data from time to time. Therefore, Their query patterns are difficult to predict. Furthermore, ad-hoc queries, which are not known in advance, make the static materialized views quickly become outdated. Hence, static materialized views cannot fully support the dynamic nature of the decision support analysis. In order to fully satisfy users' ad-hoc queries, dynamic materialized views management is highly desirable.

The main difference between dynamic view management and static view management is the view tuning. Static view management precomputes the views based on historical data, and intends to use those views for certain time intervals, such as one day. As a result, it cannot handle the case that the major access patterns drift from the previous access patterns. On the other hand, dynamic view management system selects the beneficial views at the current moment whenever new queries come in, and fine-tunes the materialized views as much as possible to serve the future queries.

For dynamic view management, [45] introduces a dynamic view management system, which stores multidimensional range fragments. They intend to answer incoming queries by either using a single cached query or using the base tables. They restrict the incoming queries as a multidimensional range query, because there may be up to a large number of combinations that need to be checked for finding the most efficient way of answering the queries. Consequently, data across multiple fragments cannot be used. [21] performs a chunk-based scheme which divides the multidimensional query space into uniform chunks. When a query is issued, it checks whether the query can be computed directly from the previous query results which are stored in the cache or a dedicate disk storage. If some parts of the queries cannot be computed from cache, the system will compute the missing parts using the base table. Both [21, 45] use a static prepartitioning approach.

As most of the existing popular database applications are built on top of relational database systems, like IBM DB2, in this chapter, a dynamic view management system is built on top of relational data warehouse. The main advantage of this approach is that it is able to fully utilize the power of relational database systems. We attempt to release the restrictions imposed on the multidimensional fragments [45], and intend to answer more general OLAP queries. Different from [21, 45], views/tables are partitioned based on user predicates dynamically. Ezeife in [23] presents horizontal fragmentation ideas and schema for selecting and materializing views to reduce query response time and maintenance cost. However,

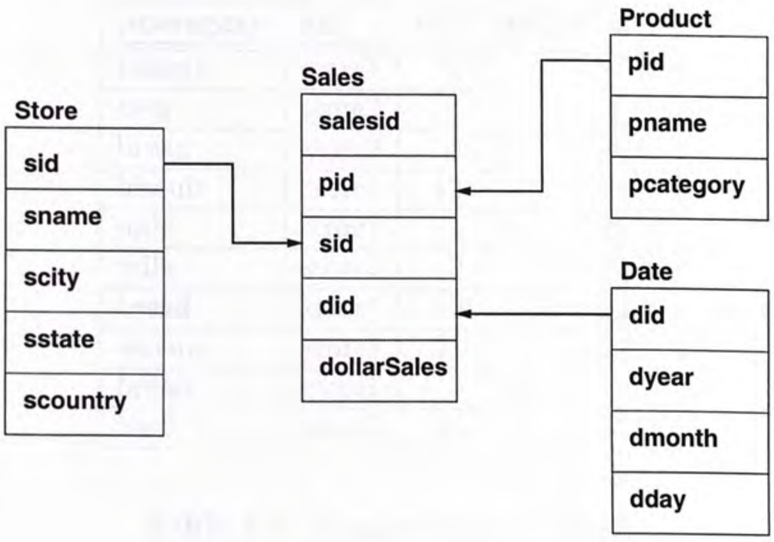


Figure 6.1: A star schema

Ezeife only considers the static view selection problem. For the dynamic materialized view management, we further study three issues: (1) predicate selection for partition, (2) repartitioning and (3) view replacement policies.

6.2 Examples

Figure 6.1 shows a simple 3-dimensional MDDB with a fact table, `Sales`, and three dimension tables, `Product`, `Store` and `Date`. In the `Sales` table, `pid`, `sid` and `did` are the foreign keys of the corresponding dimension tables. The measure in the `Sales` table is `dollarSales`. Consider a query (Query 1) that requests to report the detailed measure `dollarSales` for every combination of product category (`pcategory`), store (`sid`) and date (`did`) below.

pcategory	**sid**	**did**	**dollarSales**
biscuit	store1	3	2093
beer	store2	4	2011
bread	store2	5	2010
biscuit	store1	12	1359
milk	store1	1	2356
milk	store1	2	3526
bread	store1	15	2121
biscuit	store1	11	2101
bread	store1	5	1021
beer	store2	16	3216

Table 6.1: Assumed Query 1 result

Query 1 *Find* `dollarSales` *for every combination of* `pcategory`, `sid` *and* `did`.

select *pcategory, sid, did, dollarSales*

from *Sales s, Product p*

where $s.pid = p.pid$

Assume Table 6.1 shows the result of Query 1 which forms a temporal 3-dimensional MDDB with `dollarSales` as measure. Here, the three dimensions are `pcategory`, `sid` and `did`. Thus, a datacube with 8 vertices can be constructed on top of Table 6.1 to support the 8 group-by OLAP queries: group by (`pcategory`, `sid`, `did`), (`pcategory`, `sid`), (`pcategory`, `did`), (`sid`, `did`), (`pcategory`), (`sid`), (`did`), and ().

OLAP queries are those queries for decision making such as answering the average, minimum and maximum measures at a certain granularity of partitioning. With a star-schema, an OLAP query may involve selections based on some dimension values and/or joins between the fact table and the dimension tables. For example, three OLAP queries (Query 2, Query 3 and Query 4) are shown below. Like Query 1, the subsequence three OLAP queries need to join the fact table `Sales` and the dimension table `Product`. However, they can be computed using Table 6.1. In particular, Query 2 uses three attributes, `pcategory`, `sid` and `did` in its group-by clause. Query 3 and Query 4 use two out of three attributes involved in Query 2 in their group-by clause. Note: Query 3 and Query 4 use the same selection clause but different selection conditions.

Query 2 *Find the total* `dollarSales` *for every* `pcategory`, `sid` *and* `did` *where* `pcategory` *is* `biscuit` *and* `did` *is less than* 10.

> **select** *pcategory, sid, did, sum(dollarSales)*
>
> **from** *Sales s, Product p*
>
> **where** *s.pid = p.pid and pcategory = 'biscuit' and did < 10*
>
> **group by** *pcategory, sid, did*

Query 3 *Find the total dollarSales for every* `pcategory` *and* `did` *where* `pcategory` *is* `biscuit` *and* `did` *is less than 8.*

> **select** *pcategory, did, sum(dollarSales)*
>
> **from** *Sales s, Product p*
>
> **where** *s.pid = p.pid and pcategory = 'biscuit' and did < 8*
>
> **group by** *pcategory, did*

Query 4 *Find the total dollarSales for every* `pcategory` *and* `did` *where* `pcategory` *is* `biscuit` *and* `did` *is greater than or equal to 10.*

> **select** *pcategory, did, sum(dollarSales)*
>
> **from** *Sales s, Product p*

where *s.pid* = *p.pid and pcategory* = *'biscuit' and did* $\geq$ *10*

group by *pcategory, did*

It is worth noting that, in this chapter, the derived-from or be-computed-from relationship is more restrictive than the same relationship defined in [40]. It is because [40] does not consider the possible conditions in the where clause, but in this chapter, such conditions are taken into consideration. For example, based on the group-by attributes only, Query 4 can be computed from the result of Query 2, because the group-by attributes used in Query 4 is a subset of the group-by attributes used in Query 2. However, the differences in the selection conditions imply that Query 4 cannot be computed from the result of Query 2. The derived-from relationships for the four queries are listed below: Query 2 $\preceq$ Query 1, Query 3 $\preceq$ Query 1, Query 4 $\preceq$ Query 1, and Query 3 $\preceq$ Query 2.

The *dynamic materialized view management* is defined as how to maintain some results of OLAP queries (known as materialized views) in a limited space, in order to maximize the possibility to answer other OLAP queries in runtime.

6.3 Related Work: Static Prepartitioning-Based Materialized View Management

In this section, Two static prepartitioning-based materialized view management approaches [21, 45] are outlined. Both approaches cache the granularity of data as either chunk or fragment to support OLAP queries.

In [21], `chunked-file` is proposed to support OLAP queries. The `chunked-file` uses multi-dimensional arrays to store chunks, where a chunk, ranging at any level

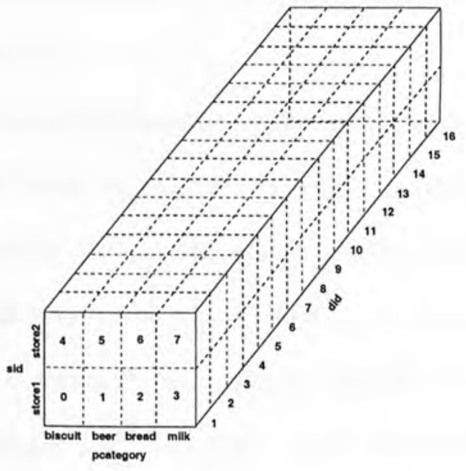


Figure 6.2: The chunked file for Table 6.1.

in the hierarchy, is proportional to the number of distinct values in the corresponding dimension at that level. For example, assume the result of Query 1 is shown in Table 6.1. Here, pcategory has 4 possible categorical values, sid has two categorical values (storel/store2), and did has 9 distinctive numerical values in the domain [1, 16]. Accordingly, a chunked file is created with 128 ($= 4 \times 2 \times 16$) chunks, as illustrated in Figure 6.2. For instance, the chunk [0][0][0] stores a measure for pcategory = 'biscuit', sid = 'store1' and did = 1. In total, there are 72 ($= 4 \times 2 \times 9$) non-empty chunks, and 56 chunks without a value. For instance, there is no measure for pcategory = 'biscuit', sid = 'store1' and did = 1 (the chunk [0][0][0]). The space consumption of a chunked file can be very large, in particular, when data is sparse. The chunked-file approach can support all the four queries, Query 1-4, using the chunked file shown in Figure 6.2. For example, for processing Query 2, chunked-file needs to scan the leftmost chunks (Figure

6.2) where `pcategory` = '`biscuit`' and `did` < *10. It is worth noting that many* of the chunks are empty.

In [45], a system, called `DynaMat`, is proposed that dynamically materializes information at multiple levels of granularity in the form of fragments. However, `DynaMat` puts some restrictions on the query patterns. `DynaMat` can only efficiently support the following three types of multidimensional range queries: (i) select a full range[1] of a dimension, for instance, for all `did` between 1 and 16; (ii) a single value, like `pcategory` = '`biscuit`'; and (iii) an empty range[2].

Assume that the result of Query 1 is shown in Table 6.1. `DynaMat` can partition data in three ways, along one of the three dimensions: `pcategory`, `sid` and `did`, respectively. For example, suppose `DynaMat` partitions data along the `did` dimension, there are totally 16 fragments [1,16]. Each fragment keeps data with a distinctive `did` value. `DynaMat` cannot support those queries that do not satisfy any of the three types efficiently. For example, `DynaMat` cannot answer Query 2 efficiently using any materialized views, even thought they are available, because one of the selection conditions is `did` < 10. The condition, `did` < 10, makes Query 2 unsatisfied any of the three above-mentioned types, (i), (ii) and (iii). In a similar fashion, `DynaMat` cannot support Query 3 and Query 4 using any materialized views. When `Dynamic` cannot support OLAP queries using the materialized views, it has to process the queries using fact table and dimension tables.

Despite the efficiency of these two approaches, namely `chunked-file` and `DynaMat`, as reported in [21, 45]. Some observations can be made below.

[1]A full range means the value is between the minimum value and maximum value of this dimension.

[2]An empty range means the dimension is not in the present in the query

- The majority of data warehouses are built on top of existing relational database systems. The chunk-based caching approach, `chunked-file`, is not directly applicable, and cannot be widely used. The efficiency of `chunked-file` is at the expense of space consumption. `Chunked-file` may result in a huge number of empty chunks, in particular, when data is sparse. Consequently, the query processing cost using a chunked file can be high, because it needs to access a large number of empty chunks. The high space consumption is somehow against its goal to maximize the possibility of supporting more queries using a limited space. In addition, it is difficult for `chunked-file` to determine the optimal chunk sizes,
- `DynaMat` partitions data into fragments. Only some types of OLAP queries can be supported efficiently. The reason that `DynaMat` cannot support a wide range of OLAP query types is due to the fact that the cost of finding a set of fragments to answer an OLAP query can be high itself. In addition, it is difficult for `DynaMat` to select a dimension(s) as the basis to partition data.

6.4 A New Dynamic Predicate-based Partitioning Approach

Here, ROLAP-based materialized view management approach is proposed, which can be easily developed on top of relational database management systems. Unlike `chunked-file` that uses arrays to store data, relations are used to support materialized views. The space consumption of relations are much smaller than that of multidimensional arrays. Unlike `DynaMat` that selects a dimension to partition data, *predicates* used in OLAP queries are used to partition. This approach can support a wide range of OLAP queries, which are attempt to minimize the space consumption in a relational database system.

The dynamic predicate-based partitioning approach is illustrated with an example here. Suppose Query 1 is processed and its result is shown in Table 6.1. By using two selected predicates from Query 2: `pcategory` = '`biscuit`' (denoted p_1) and `did` < 10 (denoted p_2), Table 6.1 can be horizontally partitioned into four materialized views: $p_1 \wedge p_2$, $\neg p_1 \wedge p_2$, $p_1 \wedge \neg p_2$, and $\neg p_1 \wedge \neg p_2$. They are shown in Table 6.2. Note that each partition will be stored in a table in a commercial multidimensional database.

Partition	**pcategory**	**sid**	**did**	**dollarSales**
R_1	biscuit	store1	3	2093
R_2	beer	store2	4	2011
	bread	store1	5	1021
	bread	store2	5	2010
	milk	store1	1	2356
	milk	store1	2	3526
R_3	biscuit	store1	12	1359
	biscuit	store1	11	2101
R_4	bread	store2	15	2121
	beer	store2	16	3216

Table 6.2: A materialized view by partition Table 6.1 using the two predicates p_1 and p_2

Obviously, there is no overlapping between any pair of partitions in Table 6.2. Suppose a user issues Query 4. Since Query 4 can be directly answered by the third partition in Table 6.2, the query processing cost is to scan 2 tuples. On the other hand, `chunked-file` needs to scan 14 chunks along the two leftmost rows (Figure 6.2). `DynaMat` cannot efficiently process Query 4 because Query 4 does not satisfy any of the three query types: (i), (ii) or (iii) stated in Section 6.3. Therefore, to process this query, `DynaMat` needs to scan the fact table `Sales` and the dimension table `Product` in which the overhead is very large.

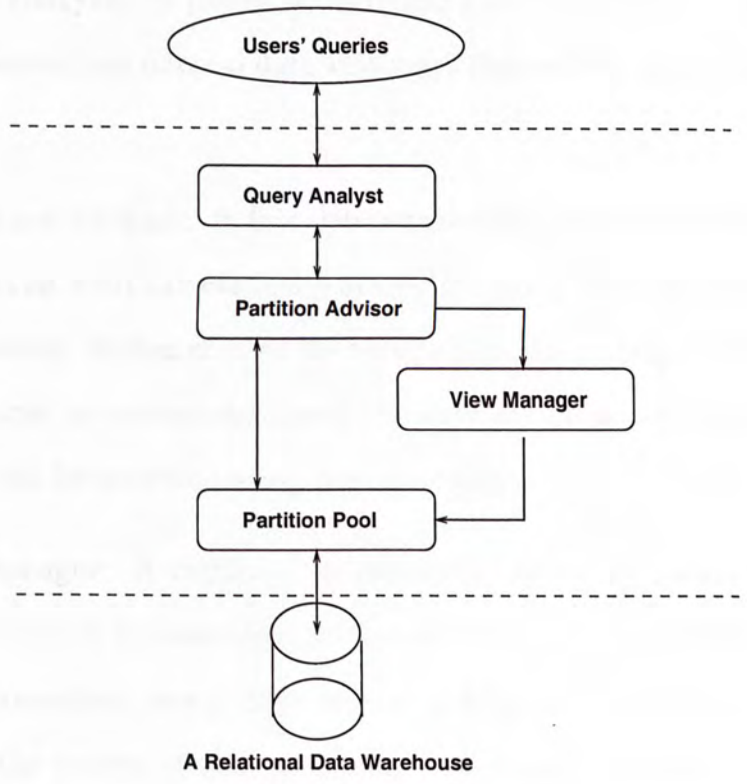


Figure 6.3: A system overview

6.4.1 System Overview

The system overview is depicted in Figure 6.3. It is built on a relational database system (IBM DB2), which consists of four main components: `Query Analyst`, `Partition Advisor`, `View Manager` and `Partition Pool`. These components are described as follows.

- `Query Analyst`: It parses an incoming query and converts the necessary information into internal data structures that will be used in other components.

- `Partition Advisor`: It first determines which partition candidates in the `Partition Pool` can efficiently answer the query based on the derived-from relationship. It then chooses the best partition(s) among the set of partition candidates to answer the query. If there are no qualified partitions, the query will be answered using the base tables.

- `View Manager`: It monitors the incoming queries and performs two main tasks. First, it decides which predicates are the most beneficial to partition the materialized views. This decision is based on a cost model which estimates the benefit of each predicate. If the predicates used in partitioning are changed, `View Manager` will repartition the materialized views. Second, when the disk space reaches the limit, `View Manager` uses a replacement policy to replace partitions in `Partition Pool`.

- `Partition Pool`: It is the information repository that stores the materialized views which are horizontally partitioned based on the incoming query predicates.

In the following, `Partition Advisor` and `View Manager` will be discussed in detail as they are more complicated.

6.4.2 Partition Advisor

A typical OLAP query involves selections which are based on some dimension values and/or joining the fact table with one or more dimension tables followed by a group-by operation. Predicate-based partitioning is to horizontally partition a view into a set of disjoint sub-views such that there are no overlapping between any sub-views. The predicates for horizontal partition are those predicates appearing in the where clause. A simple predicate is of the form, $A\theta v$, where A is an attribute, θ is one of the six operations ($=$, $<$, $>$, $\neq$, $\leq$, $\geq$), and v is a constant in the domain of A. For example, in Query 2 here are two simple predicates: $\texttt{pcategory} = \text{`}\texttt{biscuit}\text{'}$ (denoted p_1) and $\texttt{did} < 10$ (denoted p_2). The conjunction of simple predicates is called *minterm predicate* [57]. Each simple predicate can occur in a minterm predicate either in its natural form or its negated form. Recall that it is always possible to transform a Boolean expression into conjunction normal form. For Query 2, there are four minterm predicates, M_1, M_2, M_3 and M_4, as follows:

$$\begin{aligned}
M_1 &= p_1 \wedge p_2 \Longleftrightarrow \texttt{pcategory} = \text{`}\texttt{biscuit}' \wedge \texttt{did} < 10 \\
M_2 &= \neg p_1 \wedge p_2 \Longleftrightarrow \texttt{pcategory} \neq \text{`}\texttt{biscuit}' \wedge \texttt{did} < 10 \\
M_3 &= p_1 \wedge \neg p_2 \Longleftrightarrow \texttt{pcategory} = \text{`}\texttt{biscuit}' \wedge \texttt{did} \geq 10 \\
M_4 &= \neg p_1 \wedge \neg p_2 \Longleftrightarrow \texttt{pcategory} \neq \text{`}\texttt{biscuit}' \wedge \texttt{did} \geq 10
\end{aligned}$$

In the system, a materialized view, V_i, is associated with an OLAP query,

q_{V_i}. For example, the whole Table 6.2 is a materialized view represented by Query 1. This materialized view has four partitions: R_1, R_2, R_3 and R_4, for the corresponding four minterms: M_1, M_2, M_3 and M_4 mentioned above. Accordingly, each partition R_i can be represented using a query q_{R_i}.

For an incoming query q, `Partition Advisor` will first determine the set of materialized views that can answer q by checking if $q \preceq q_{V_i}$. In other words, all attributes in q must also be in q_{V_i}, the selection condition used in q implies that it is a subset of q_{V_i}, and the aggregate functions used in the two queries are the same. Second, `Partition Advisor` needs to determine a partition R_j in a materialized view V_i to answer the query q. The partition selection algorithm is shown in Algorithm 9. In order to eliminate the overhead, `Partition Advisor` attempts to select one partition, R_i, to answer a given query. It is because, in general, the cost of selecting partitions is exponential in terms of the number of partitions available. When there is no single partition that can answer the query, there are two ways to solve it as shown in Algorithm 9: (1) use a whole materialized view to answer the query; (2) use the base tables to answer the query. As discussed later, when there is a materialized view available for answering the query, `View Manager` considers whether need to repartition the materialized view after processing the query.

6.4.3 View Manager

`View Manager` maintains the top m predicates that give the highest predicate benefits for a materialized view, where m is predefined by a data warehouse administrator. Note that the number of predicates influences how a materialized view is partitioned. In general, m predicates can create a maximum of 2^m minterm

Algorithm 9 Partition Selection

Input: q: an incoming user query;
Output: a partition to answer the user query.

```
1:  begin
2:  Let bestPartition be the base table(s) that can answer $q$;
3:  for each materialized view $V_i$ in Partition Pool do
4:      Let $q_{V_i}$ be the corresponding query that represents $V_i$;
5:      if $q \preceq q_{V_i}$ then
6:          if $|bestPartition| > |V_i|$ then
7:              bestPartition $\leftarrow V_i$;
8:          end if
9:          for each partition $R_j$ of $V_i$ do
10:             if $q \preceq q_{R_j}$ and $|bestPartition| > |R_j|$ then
11:                 bestPartition $\leftarrow R_j$;
12:             end if
13:         end for
14:     end if
15: end for
16: return bestPartition;
```

Algorithm 10 Selecting top m predicates for a materialized view V

Input: a query q with k simple predicates $\{p_1, p_2, \cdots, p_k\}$, a materialized view V with n simple predicates $\{p_1, p_2, \cdots, p_n\}$.
Output: m highest predicate benefit ($PB(p_i)$) predicates.

```
1:  begin
2:  for $i = 1$ to $k$ do
3:      for $j = 1$ to $n$ do
4:          if the predicate $p_i$ in the query $q$ matches a predicate $p_j$ maintained with
            the view $V$ then
5:              $PB(p_j) \leftarrow f_i \times |R_i| + PB(p_j)$;
6:          else
7:              create a new predicate benefit $PB(p_i)$;
8:              $PB(p_i) \leftarrow f_i \times |R_i|$;
9:          end if
10:     end for
11: end for
12: return the $m$ highest predicates;
```

fragments during the horizontal partitioning process. To reduce the cost of such huge overhead, the most frequent predicates are selected for horizontal partitioning when the number of predicates is large. The process of selecting a predicate, p_i, depends on two factors: its relative access frequency, f_i, and its corresponding partition size, $|R_i|$. The predicate benefit of a predicate, p_i, denoted $PB(p_i)$, is estimated as follows:

$$PB(p_i) = |R_i| \times f_i \tag{6.1}$$

The top m highest benefit predicates are selected using Algorithm 10. Here, suppose the relative query frequency of an incoming query q is f_q. The access frequency of p_i used q is $f_i = f_q$.

When a user query is issued, and it cannot be answered using a partition but a materialized view, V, `View Manager` will calculate the predicate benefits to see whether there is any change in the top m predicates associated with V, by taking both of the predicates used in q and V into account. If there is any change in the top m predicates, `View Manager` will repartition V using the new m predicates. Otherwise, `View Manager` will not repartition V. The repartitioning algorithm is shown in Algorithm 11.

The query results are stored in `Partition Pool` as materialized views, when there is free space. When `Partition Pool` is full, a replacement policy is adopted to store the beneficial partitions. In other words, an incoming query result is not stored in the pool by default, and is stored only if it is beneficial. The commonly used caching techniques like `LRU` and `FIFO` are not suitable for handling OLAP

Algorithm 11 Repartitioning

```
Input: a query ($q_V$) and its corresponding materialized view ($V$);

1: begin
2: Let $L$ hold the $m$ highest predicates for $V$;
3: Let $L'$ the $m$ highest predicates used in $V$, (selected by Algorithm 10);
4: if $L \neq L'$ then
5:     remove all existing partitions for $V$;
6:     generate new minterm predicates;
7:     delete infeasible minterms;
8:     repartition $V$ using the new minterm predicates.
9: end if
```

queries, because OLAP queries are group-by queries in nature, and the derived-from relationships cannot be easily managed by `LRU` and `FIFO`. Thus, strategies similar to [45] are used in the system. Two goodness measure, `SFF` and `SPF`, are defined to evaluate which partition can be stored in `Partition Pool`.

- `Small Partition First (SFF)`: The intuition behind this approach is that larger partitions are more likely to be hit by a future query. A larger partition implies fewer number of partitions stored in `Partition Pool`. Fewer number of partitions will reduce the overheads used in `Partition Advisor` and `View Manager`. Query frequency does not need to take into account in this strategy. However, if a larger partition is not a hot access region, it may waste the space and slow down the whole query response time as other hot queries have to access the base table. Let R_i denote a partition. The goodness of R_i is measured by its size below.

$$Goodness(R_i) = |R_i|$$

- `Small Penalty First (SPF)`: In this strategy, query frequency, query processing time and partition size are taken into consideration. Let R_i denote a partition. The goodness of R_i is defined as:

$$Goodness(R_i) = \frac{f_{R_i} \times qcost(R_i)}{|R_i|}$$

  where f_{R_i} is the query frequency of accessing R_i, $qcost(R_i)$ is the cost of recomputing R_i if it is removed, and $|R_i|$ is the partition size. Note the recomputing cost, $qcost(R_i)$ is simply estimated using Algorithm 9 when R_i is absent.

6.5 A Performance Study

All of the experiments are conducted on a Sun Blade/1000 workstation with a 750MHz UltraSPARC-III CPU running Solaris 2.8. The workstation has a total physical memory of 512M. We employ the TPC-H[3] benchmark dataset, and conduct the testing using IBM DB2[4] version 7.1.

The TPC-H benchmark is a decision support benchmark for ad-hoc queries. It consists of eight separate and individual tables. The tables and relationships between columns of these tables are shown in Figure 6.4. Two parts of TPC-H schema are used in the testing.

- **Small-Schema**: For testing feasibility, query locality and the effectiveness of disk size, 9 attributes in the `part` table are used. The total number of

[3]http://www.tpc.org

[4]http://www-3.ibm.com/software/data/iminer/fordata/

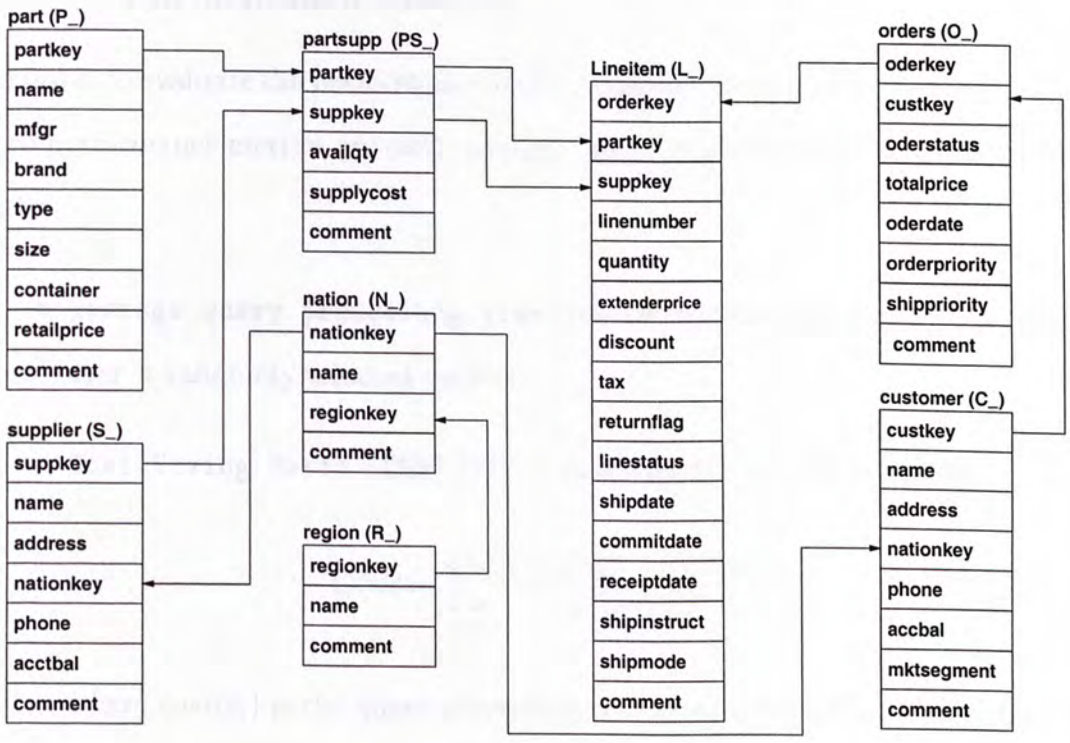


Figure 6.4: The TPC-H schema.

tuples in the `part` table is 200,000 and its size is about 35MB, as designed in TPC-H benchmark. The `part` table describe the parts made by a particular manufacturer, its size, brand, type and retailprice.

- **Large-Schema**: For testing scalability, all the 8 tables are joined with all attributes in the TPC-H schema. The total number of tuples is 6,000,000 and its size is about 3,350MB.

6.5.1 Performance Metrics

In order to evaluate the performance of the predicate-based partitioning approach, two performance metrics are used: average query processing time and cost saving ratio.

- `average query processing time`: it is the average query processing costs over n randomly selected queries.
- `Cost Saving Ratio (CSR)` [21]: it measures the results as follows.

$$CSR = \sum_{i=0}^{n} \frac{wcost(q_i) - cost(q_i)}{wcost(q_i)} \times \frac{1}{n}$$

where $cost(q_i)$ is the query processing cost using partitions and $wcost(q_i)$ is the query processing cost using non-partitions, that is using the base tables in the data warehouse.

6.5.2 Feasibility Studies

In this experiment, the feasibility of the predicate-based partitioning approach is investigated by using the small-schema of 8 dimensions and a measure (`retailprice`). Seven different queries templates[5] are designed with respect to each vertex in the datacube. For each query template, at least 10 queries are randomly generated using the TPC-H `qgen` program. As a result, 70 different OLAP queries are issued with randomly generated predicates as well as randomly query frequency. The sequence of these 70 queries are randomly determined.

[5]No queries access the empty group-by clause vertex in the datacube.

The predicate-based partitioning approach is compared with an implementation of `chunked-file` [21] in the relational database system, IBM DB2. `Partition Pool` is assumed to be large enough to store the root vertex (the largest materialized view) in the datacube. In fact, it is about 21% of the base table. The assumption is made for the following three reasons. First, all queries can be answered using the largest materialized view. Second, we can focus on feasibility analysis of the partitioning technique, and ignore the effectiveness of replacement policies in testing. Third, we can minimize the workload in `Partition Advisor`, because there are only a few partitions. When testing the 70 queries one by one, two highest beneficial predicates are selected to generate minterms ($m = 2$). The materialized view will be dynamically divided into 4 partitions based on the two predicates. The materialized view would be repartitioned dynamically, if necessary.

The `chunked-file` approach uses a predetermined chunk size statically. In this testing, the materialized views are pre-partitioned into k even partitions to simulate the chunk-based [21] in ROLAP environments. Three k values are tested: 4-chunk, 9-chunk and 25-chunk. The reason why there is no comparison with `DynaMat` is that most OLAP queries cannot be efficiently answered by `DynaMat` due to the restrictions on the query types.

Figure 6.5 shows that dynamic predicate-based partitioning approach (PP) outperforms the static pre-partitioning significantly in terms of average query processing time in a relational database. It is totally not surprised. The reason is that the dynamic partitioning learns from predicates and attempts to repartition the materialized views in an eager manner. The incoming queries are most likely to be answered by a predicate-based partition. Consequently, the query processing cost must be reduced.

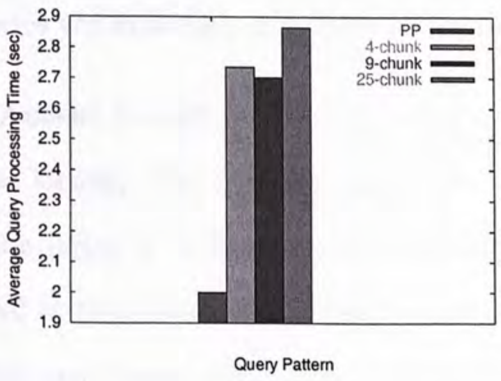


Figure 6.5: Static prepartitioning v.s. dynamic predicate-based partitioning

6.5.3 Query Locality

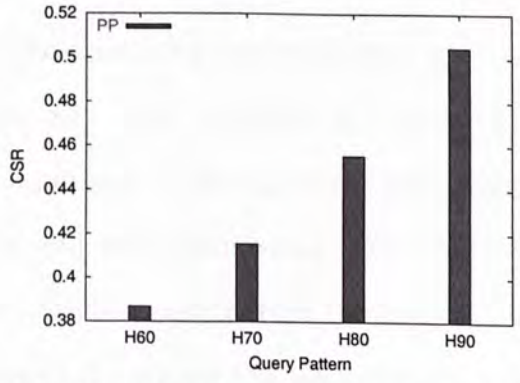
To study the query locality, two sets of experiments are conducted based on *data access locality* and *hierarchical access locality* using the small-schema:

- **Data Access Locality**: Most users have their own preferences which may last for a while. That is, they may be interested in one part of data. For instance, a Hong Kong stock analyst is most likely and often to query the Hong Kong stocks rather than all stocks in the world. To simulate the data access locality, a certain percentage of the database is designed as a hot region such that the queries are most likely to access the designated part of the database.
  - `H60`: 60% of the queries access 20% of the datacube.
  - `H70`: 70% of the queries access 20% of the datacube.

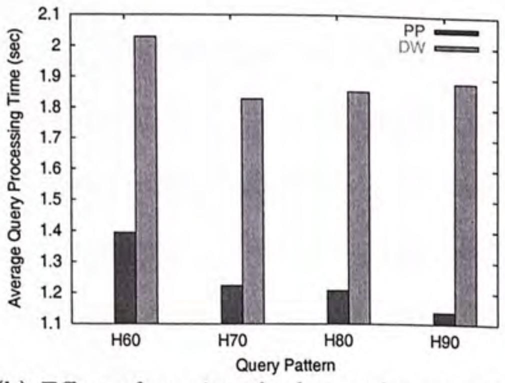
- H80: 80% of the queries access 20% of the datacube.
- H90: 90% of the queries access 20% of the datacube.

The rest of queries are uniformly distributed over the database.

- **Hierarchical Access Locality**: Proximity queries are used to model hierarchical access locality. For instance, users may be primarily interested in the Hang Seng Index in early morning. Afterwards, they may be interested in its trend in this week, this month, or this year, based on the time hierarchy. In this experiment, the degree of hierarchical access locality can be tuned by varying the mix of random queries and proximity queries.
  - Q60: 60% queries are proximity queries and 40% are random generated.
  - Q70: 70% queries are proximity queries and 30% are random generated.
  - Q80: 80% queries are proximity queries and 20% are random generated.
  - Q90: 90% queries are proximity queries and 10% are random generated.

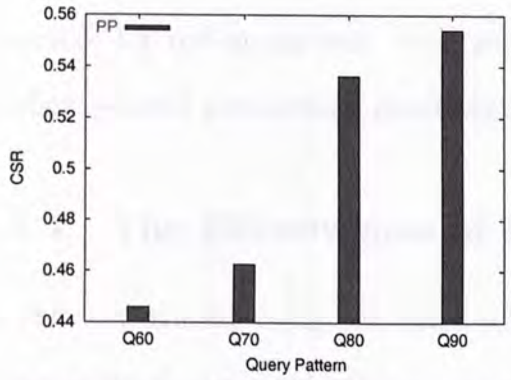


(a) Effect of varying the hot region on CSR

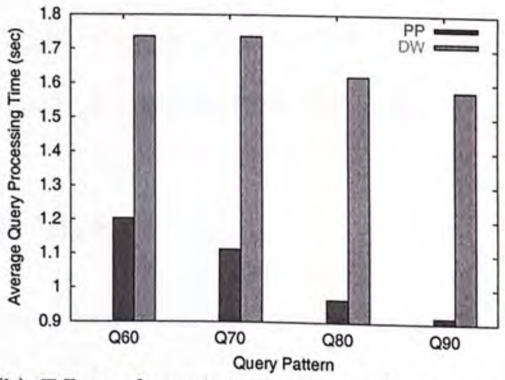


(b) Effect of varying the hot region on query response time

Figure 6.6: Testing different data access locality patterns



(a) Effect of varying the proximity on CSR



(b) Effect of varying the proximity on query response time

Figure 6.7: Testing different hierarchical access locality patterns

Assume that the space available is to hold 10% of the base tables. For each of the query patterns above, 100 queries are issued, the average query processing time and CSR are calculated. In the following figures, PP and DW represent the dynamic predicate-based partitioning approach and the non-partitioning approach, respectively. Figure 6.6 and 6.7 show that the dynamic predicate-based partitioning approach exploit the locality very good. Figure 6.6 (a) shows the performance for query pattern with a designated hot region. Note that CSR increases with a larger hot region of the database. Figure 6.6 (b) shows the comparison of average query processing time between predicate-based partitioning and non-partitioning. The dynamic predicate-based partitioning approach can dramatically reduce the average query processing time compared with DW. Figure 6.7 shows the performance for proximity query pattern. In Figure 6.7 (a), CSR increases sharply as the proximity percentage increases. This is because more incoming queries can be derived from the partitions. Note that `Q90` reaches the highest CSR, which denotes that the predicate-based partitioning is favorable for roll-up queries. In Figure 6.7 (b), compared with DW, dynamic predicate-based partitioning diminishes the average query processing time.

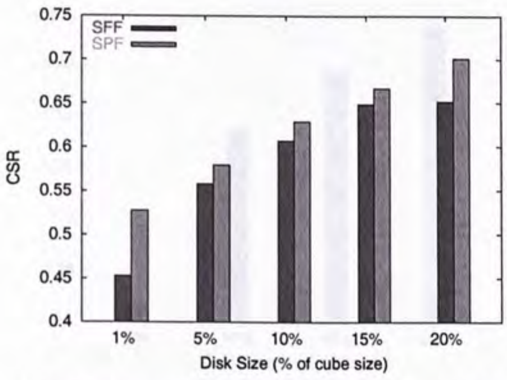
6.5.4 The Effectiveness of Disk Size

In this experiment using the small-schema, the disk size is varied for testing dynamic predicate-based partitioning approach. The query pattern to be tested is `Q80`. By adjusting the disk space used in `Partition Pool` to be 1%, 5%, 10%, 15%, and 20% of the whole datacube, 100 queries with SFF and SPF replacement policies are tested.

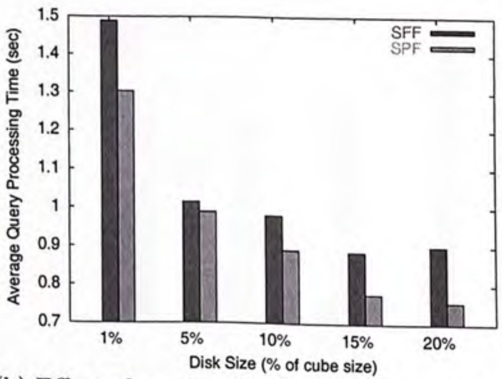
Figure 6.8 (a) shows that CSR increases while the cache size increases. This is because more partitions can reside on disk. If the disk space is too small to store the previous results, partitions would not be effectively used to answer the incoming queries. Most queries have to access the base tables. Therefore, CSR becomes lower. When the disk space is large, `Partition Pool` can store more partitions. Thus, CSR becomes higher with a higher hit ratio. As expected, the average query processing cost is reduced. Both Figure 6.8 (a) and (b) show that the replacement policy `SPF` outperforms `SFF`.

6.5.5 Scalability

In this experiment using the large-schema, `Q80` query pattern is used and the disk size is varied from 5% to 20% of the data cube, using the large schema (the schema by joining all tables in the TPC-H schema). Recall Figure 6.8 which shows that SPF outperforms SFF. Therefore, 100 queries are issued with SPF replacement



(a) Effect of varying the disk size on CSR



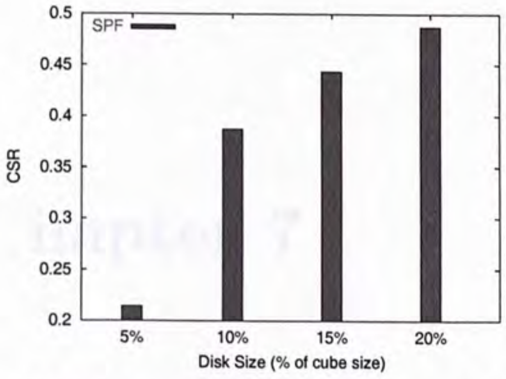
(b) Effect of varying the disk size on query processing time

Figure 6.8: A comparison of SFF and SPF on varying the disk size

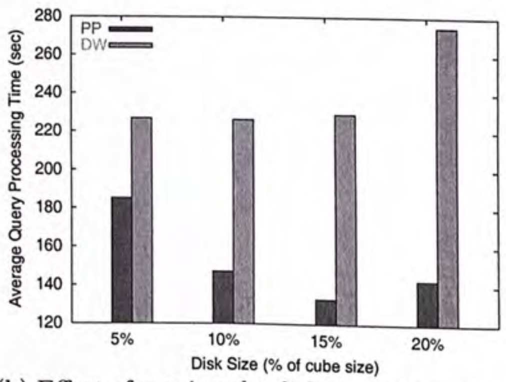
policy. Figure 6.9 (a) shows that CSR increases as disk size increases due to the fact that a larger disk size can reside more partitions. Figure 6.9 (b) shows the difference of the average query processing time between using the predicated-based partitioning approach and the non-partitioning approach. One could see that the dynamic predicated-based partitioning approach can dramatically reduce the average query processing cost.

6.6 Summary

This chapter focuses on ROLAP, and a new dynamic predicate-based partition materialized view management approach is proposed for caching OLAP queries in a relational multidimensional database. User predicates is used to partition views instead of using a predetermined threshold to pre-partition views. Based on the user predicates, the materialized views are partitioned into horizontal fragments,



(a) Effect of varying the disk size on CSR



(b) Effect of varying the disk size on query processing time

Figure 6.9: Effect of varying the disk size

which allows fine granularity caching as well as coarse caching. The approach can dynamically materialize the incoming query results and exploit them for future reuse. The experimental results show that predicate-based partition exhibit high query locality, and outperform the pre-partitioning approach in terms of ROLAP.

The system can effectively monitor the incoming query predicates and decide whether or not to repartition the materialized view, so as to serve the future user queries. The replacement policy is a critical factor for utilizing the disk space during the materialized view management. Moreover, when dynamic predicate-based partitioning approach is used, SPF is recommended as an replacement policy rather than SFF, because the former one always outperforms the latest one in such situation.

Chapter 7

Conclusions and Future Work

In this thesis, a research problem of maintenance cost view selection problem have been identified and formalized. First, four existing heuristic algorithms: A*-heuristic, inverted-tree greedy, two-phase greedy and integrated greedy have been re-examined to provide readers with insights on the qualities of these heuristic algorithms. Experimental results show that heuristic algorithms can provide optimal or near optimal solution in a multidimensional data warehouse environment where the update cost and update frequency of any ancestor of a vertex must be greater than or equal to the update cost and update frequency of that vertex, respectively. Compared with the other three greedy algorithms, inverted-tree greedy reached the highest query processing cost and view selection time. A*-heuristic cannot guarantee to achieve an optimal solution always and is not scalable. However, the two-phase greedy and the integrated greedy are scalable.

At the mean time, a new evolutionary algorithm is designed for the maintenance cost view selection problem. A revised stochastic ranking technique is the first time to be adopted to solve this specific problem and it works very successful. The new evolutionary algorithm has been evaluated against both heuristic and the other evolutionary algorithms. The experimental results show that the algorithm provides a significantly better solutions than the existing algorithms in terms of minimization of query processing cost and feasibility.

To solve dynamic view management problem, a new dynamic predicate-based partition materialized view management approach is proposed for caching OLAP queries in a relational multidimensional database (ROLAP) environment. Focus on ROLAP, users' predicates are used to partition materialized views into horizontal fragments which allows fine granularity caching as well as coarse caching. The approach can dynamically materialize the incoming query results and exploit them for future reuse. The experimental results show that predicate-based partition exhibits high query locality.

Since the replacement policy is a critical factor for utilizing the caching space for dynamic view selection problem. In the future, we would like to design a feasible replacement policy to improve the performance of the dynamic view selection problem.

Bibliography

[1] F. N. Afrati, C. Li, and J. D. Ullman. Generating efficient plans for queries using views. In *Proceedings of the 27th ACM SIGMOD international conference on Management of data*, 2001.

[2] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 506–521, 1996.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Base*, pages 496 - 505, 2000.

[4] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 156–165, 1997.

[5] A. Berson and S. J. Smith. *Data warehousing, data mining, and OLAP*. McGraw-Hill, 1997.

[6] J. C. Bezdek. What is computational intelligence? *Computational Intelligence Imitating Life (appearing in [87])*, pages 1–12, 1994.

[7] S. Chaudhuri and U. Dayal. Data warehousing and olap for decision support. In *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*, 1997.

[8] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM SIGMOD International Conference on Management of Data*, 26(1), 1997.

[9] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the 11th Int. Conference on Data Engineering*, pages 190–200, 1995.

[10] S. Chaudhuri and K. Shim. An overview of cost-based optimization of queries with aggregates. *Data Engineering Bulletin*, 18(3):3–9, 1995.

[11] R. Chirkova, A. Y. Halevy, and D. Suciu. A formal perspective on the view selection problem. In *Proceedings of the 27th International Conference on Very Large Data Base*, pages 59–68, 2001.

[12] C.-H. Choi, J. X. Yu, and G. Gou. What difference heuristics make: Maintenance-cost view-selection revisited. In *Proceedings of the 3rd International Conference on Web-Age Information Management*, pages 247–258, 2002.

[13] C.-H. Choi, J. X. Yu, and H. Lu. Dynamic materialized view management base on predicates. In *Proceedings of the 5th Asia Pacific Web Conference (APWEB)*, pages 583–594, 2003.

[14] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the 13th International Conference on Principles of Database Systems*, pages 155–166, 1999.

[15] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. *ACM SIGMOD Record*, 25(2):469–480, 1996.

[16] G. Colliat. Olap, relational, and multidimensional database systems. *ACM SIGMOD Record*, 25(3):64–69, 1996.

[17] O. Council. Olap and olap server definitions. In *http://www.olapcouncil.org/research/glossaryly.htm*, 1997.

[18] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 330–341, 1996.

[19] M. F. de Souza and M. C. Sampaio. Efficient materialization and use of views in data warehouses. *ACM SIGMOD Record*, 28(1):78–83, 1999.

[20] P. Deshpande and J. F. Naughton. Aggregate aware caching for multi-dimensional queries. In *Processings of the 7th International Conference on Extending Database Technology*, pages 167–182, 2000.

[21] P. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1998.

[22] C. I. Ezeife. A uniform approach for selecting views and indexes in a data warehouse. In *Proceedings of the 2nd International Database Engineering and Applications Symposium (IDEAS '97)*, page 110, 1997.

[23] C. I. Ezeife. Selecting and materializing horizontally partitioned warehouse views. *Data and Knowledge Engineering*, 36(2), January 2001.

[24] H. Gabow. *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs.* PhD thesis, Stanford University, 1973.

[25] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the 27th ACM SIGMOD International Conference on Management of Data*, 2001.

[26] G. Gou, J. X. Yu, C.-H. Choi, and H. Lu. An efficient and interactive a*-algorithm with pruning power: Materialized view selection revisited. In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA)*, page 231, 2003.

[27] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 12th International Conference on Data Engineering*, pages 152–159, 1996.

[28] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proceedings of the 5th Extending Database Technology*, pages 140–144, 1996.

[29] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bulletin, Special Issue on Materialized Views and Data Warehouse*, 18(2):3–18, 1995.

[30] A. Gupta and I. S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. The MIT Press, 1999.

[31] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993.

[32] A. Gupta, S. Sudarshan, and S. Vishwanathan. Query scheduling in multi query optimization. In *Proceedings of the 5th International Database Engineering and Applications Symposium (IDEAS '01)*, 2001.

[33] H. Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of the 6th International Conference on Database Theory*, pages 98–112, 1997.

[34] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for OLAP. In *Proceedings of the 13th International Conference on Data Engineering*, pages 208–219, 1997.

[35] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proceedings of the 13th International Conference on Data Engineering*, pages 208–219, 1997.

[36] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proceedings of the 7th International Conference on Database Theory*, pages 453–470, 1999.

[37] M. Gyssens and L. V. S. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 106–115, 1997.

[38] A. Y. Halevy. Theory of answering queries using views. *ACM SIGMOD Record*, 29(4):40–47, 2000.

[39] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann, September 2000.

[40] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, pages 205–216, 1996.

[41] W. H. Inmon. *Building the Data Warehouse.* John Wiley, 2002.

[42] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K.-L. Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *Proceedings of the 28th ACM SIGMOD International Conference on Management of Data*, pages 25–36, 2002.

[43] P. Kalnis and D. Papadias. Proxy-server architectures for OLAP. *ACM SIGMOD Record*, 30(2):367–378, 2001.

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

[45] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *Proceedings of the 25th ACM SIGMOD International Conference on Management of Data*, pages 371–382, 1999.

[46] W. J. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the warehouse update window. *ACM SIGMOD Record*, 28(2):383–394, 1999.

[47] M. Lee and J. Hammer. Speeding up materialized view selection in data warehouses using a randomized algorithm. *International Journal of Cooperative Information Systems*, 10(3):327–353, 2001.

[48] A. Levy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270 - 294, 2001.

[49] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 95–104, 1995.

[50] C. Li and X. S. Wang. A data model for supporting on-line analytical processing. In *Processings the 5th International Conference on Information and Knowledge Management*, pages 81–88, 1996.

[51] W. Liang, H. Wang, and M. E. Orlowska. Materialized view selection under the maintenance time constraint. *Data and Knowledge Engineering*, 37(2), May 2001.

[52] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[53] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 27th ACM SIGMOD International Conference on Management of Data*, pages 249–260, 2001.

[54] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*, pages 100 – 111, 1997.

[55] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, 1995.

[56] P. O'Neil and D. Quass. Improved query performance with variant indexes. *ACM SIGMOD Record*, 26(2):38–49, 1997.

[57] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice Hall, 1999.

[58] C.-S. Park, M. H. Kim, and Y.-J. Lee. Rewriting OLAP queries using materialized views and dimension hierarchies in data warehouses. In *Proceedings of the 17th International Conference on Data Engineering*, pages 515–523, 2001.

[59] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *The VLDB Journal*, pages 484–495, 2000.

[60] S. G. Qiu and T. W. Ling. View selection in OLAP environment. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 447–456, 2000.

[61] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, pages 158–169, 1996.

[62] J. Rao and K. A. Ross. Reusing invariants: a new strategy for correlated queries. In *Proceedings of the 24th ACM SIGMOD international conference on Management of data*, pages 37–48, 1998.

[63] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: trading space for time. *ACM SIGMOD Record*, 25(2):447–458, 1996.

[64] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *ACM SIGMOD Record*, 29(2):249–260, 2000.

[65] T. P. Runarsson and X. Yao. Stochastic ranking for constrained evolutionary optimization. *IEEE Transactions on Evolutionary Computation*, 4(3):284–294, September Sept. 2000.

[66] P. Scheuermann, J. Shim, and R. Vingralek. Watchman : A data warehouse intelligent cache manager. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 51–62, 1996.

[67] H.-P. Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, Inc., 1995.

[68] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[69] J. Shanmugasundaram, U. M. Fayyad, and P. S. Bradley. Compressed data cubes for OLAP aggregate query approximation on continuous dimensions. In *Proceedings of the 5th Knowledge Discovery and Data Mining*, pages 223–232, 1999.

[70] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 488–499, 1998.

[71] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized view selection for multi-cube data models. In *Proceedings of the 7th International Conference on Extending Database Technology*, pages 269–284, 2000.

[72] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 318–329, 1996.

[73] D. Theodoratos and T. Sellis. Data warehouse configuration. In *Processings of the 23rd International Conference on Very Large Data Bases*, pages 318–329, 1997.

[74] D. Theodoratos and T. Sellis. Dynamic data warehouse design. In *Proceedings of the 1st International Conference on Data Warehousing and Knowledge Discovery, (DaWaK'99)*, pages 1 – 10, 1999.

[75] P. Vassiliadis and T. Sellis. A survey of logical models for olap databases. *ACM SIGMOD Record*, 28(4):64–69, 1999.

[76] J. Widom. Research problems in data warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management*, pages 25–30, 1995.

[77] M.-C. Wu. Query optimization for selections using bitmaps. *ACM SIGMOD Record*, 28(2):227–238, 1999.

[78] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *The VLDB Journal*, pages 136–145, 1997.

[79] J. Yang, K. Karlapalem, and Q. Li. Tackling the challenges of materialized view design in data warehousing environment. In *Proceedings of the 7th International Workshop on Research Issues in Data Engineering*, pages 0–, 1997.

[80] J. X. Yu and H. Lu. Hash in place with memory shifting: Datacube computation revisited. In *Proceedings of the 15th International Conference on Data Engineering*, page 254, 1999.

[81] J. X. Yu and H. Lu. Multi-cube computation. In *Proceedings of the 7th International Conference on Database Systems for Advanced Applications*, pages 126–133, 2001.

[82] J. X. Yu, X. Yao, C.-H. Choi, and G. Gou. Materialized view selection as constrained evolutionary optimization. *IEEE Transactions on Systems, Mans, and Cybernetics on technologies promoting computational intelligence, openness and programmability in networks and Internet services*, 2003.

[83] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *Proceedings of the 26th ACM SIGMOD international conference on Management of data*, pages 105–116, 2000.

[84] C. Zhang, X. Yao, and J. Yang. An evolutionary approach to materialized view selection in a data warehouse environment. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, 31(3):282–294, Aug. 2001.

[85] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorith for simultaneous multidimensional aggregations. In *Proceedings of the 23rd ACM-SIGMOD International Conference on Management of Data*, pages 159–170, 1997.

[86] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proceedings of the 24th ACM SIGMOD international conference on Management of data*, pages 271–282, 1998.

[87] J. M. Zurada, R. J. M. II, and C. J. Robinson. *Computational Intelligence Imitating Life*. IEEE Press, 1994.