

# Some Issues in Design of Data Warehousing Systems

Ladjel Bellatreche

Kamalakar Karlapalem

Mukesh Mohania

Department of Computer Science  
University of Science & Technology  
Clear Water Bay Kowloon  
Hong Kong  
{ladjel, kamal}@cs.ust.hk

Department of Computer Science  
Western Michigan University  
Kalamazoo, MI 49008-5021  
U.S.A.  
mohania@cs.wmich.edu

## 1 Introduction

Information is one of the most valuable assets of an organization and when used properly can assist intelligent decision making that can significantly improve the functioning of an organization. Data warehousing is a recent technology that allows information to be easily and efficiently accessed for decision making activities. On-Line Analytical Processing (OLAP) tools are well-studied for complex data analysis. A data warehouse is *a set of subject-oriented, integrated, time varying, and non-volatile databases used to support the decision-making activities* [42].

The conceptual architecture of a data warehousing system is shown in Figure 1. The data

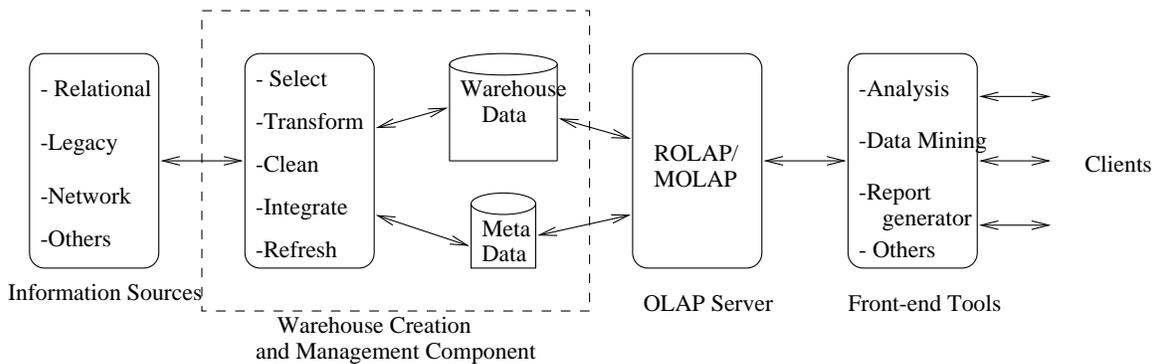


Figure 1: A conceptual data warehousing architecture

warehouse creation and management component includes software tools for selecting data from information sources (which could be operational, legacy, external, etc. and may be distributed, autonomous and heterogeneous), cleaning, transforming, integrating, and propagating data into the data warehouse. It also refreshes the warehouse data and meta-data when source data is updated. This component is also responsible for managing the warehouse data, creating indices on data tables, data partitioning and updating meta-data. The warehouse data contains the detail data, summary data, consolidated data and/or multidimensional data.

The meta-data is generally held in a separate repository. The meta-data contains the informational data about the creation, management, and usage of the data warehouse. It serves as a bridge between the users of the warehouse and the data contained in it. The warehouse data is also accessed by the OLAP server to present the data in a multidimensional way to the front end tools (such as analytical tools, report writers, spreadsheets and data mining tools) for analysis and informational purposes. Basically, the OLAP server interprets client queries (the client interacts with front end tools and pass these queries to the OLAP server) and converts them into complex SQL queries required to access the warehouse data. It might also access the data from the primary sources if the client's queries need operational data. Finally, the OLAP

server passes the multidimensional views of data to the front end tools, and these tools format the data according to the client's requirements.

There are two approaches to creating the warehouse data - bottom-up and top-down. In a bottom-up approach, the data is obtained from the primary sources based on the data warehouse applications and a profile of the likely queries which is typically known in advance. The data is then selected, transformed, and integrated by data acquisition tools. In a top-down approach, the data is obtained from the primary sources whenever a query is posed. In this case, the warehouse system determines the primary data sources in order to answer the query. These two approaches are similar to *eager* and *lazy* approaches discussed in [70]. The bottom-up approach is used in data warehousing because user queries can be answered immediately and data analysis can be done efficiently since data will always be available in the warehouse. Hence, this approach is feasible and improves the performance of the system. Another approach is a hybrid approach, which combines aspects of the bottom-up and top-down approaches. In this approach, some data is stored in a warehouse, and other data can be obtained from the primary sources on demand [37].

The warehouse data is typically modeled *multi-dimensionally*. The multidimensional data model [3, 52] has been proved to be the most suitable for OLAP applications. OLAP tools provide an environment for decision making and business modeling activities by supporting ad-hoc queries. There are two ways to implement multidimensional data model:

- by using the underlying relational architecture to project a pseudo-multidimensional model and
- by using true multidimensional data structures like arrays.

We discuss the multidimensional model and the implementation schemes in Section 2.

The data fragmentation is a very well known method used in the relational databases and aims to reduce the cost of processing queries. This technique can be adapted to the data warehouse environments, where the size of the fact tables is very large and there are many join operations. In section 3, we describe the fragmentation technique and show how it can be applied to data warehouse star/snowflake schemas.

Warehouse data can be seen as a set of materialized views which are derived from the source data. OLAP queries can be executed efficiently over materialized views but the number of views that should be materialized at the warehouse needs to be controlled, else this can result to materialize all possible queries (this is known as *data explosion*). In Section 4 we discuss the design issues related to warehouse views.

The technique of view materialization is hampered by the fact that one needs to anticipate the queries to materialize at the warehouse. The queries issued at the data warehouse are mostly *ad-hoc* and cannot be effectively anticipated at all times. Thus, answering these queries require effective indexing methods since queries involve joins on multiple tables. The traditional indexing methods in relational databases do not work well in the data warehousing environment. New access structures have been proposed for data warehousing environments. We investigate different types of indexing schemes in Section 5.

The view and index selection problems are two problems mostly studied independently in data warehouses. This causes an inefficient distribution of resources (space, computation time, maintenance time etc.) between views and indexes. In section 6, we discuss the problem of distributing space between views and indices in order to select two sets of materialized views and indexes to guarantee a performance.

## 2 Data Models for a Data Warehouse

The data models for designing traditional OLTP systems are not well-suited for modeling complex queries in data warehousing environment. The transactions in OLTP systems are made up of simple, pre-defined queries. In the data warehousing environments, the queries tend to use joins on more tables, have a larger computation time and are ad-hoc in nature. This kind of processing environment warrants a new perspective to data modeling. The *multidimensional* data model i.e., the *data cube* turned out to be an adequate model that provides a way to aggregate facts along multiple attributes, called *dimensions*. Data is stored as *facts* and *dimensions* instead of rows and columns as in relational data model. Facts are numeric or factual data that represents a specific business activity and the dimension represents a single perspective on the data. Each dimension is described by a set of attributes.

A multidimensional data model (MDDM) supports complex decision queries on huge amounts of enterprise and temporal data. It provides us with an integrated environment for summarizing (using aggregate functions or by applying some formulae) information across multiple dimensions. MDDM has now become the preferred choice of many vendors as the platform for building new on-line analytical processing (OLAP) tools. The user has the leverage to *slice* and *dice* the dimensions, thereby, allowing him/her to use different dimensions during an interactive query session. The data cube allows the user to visualize aggregated facts multi-dimensionally. The level of detail retrieved depends on the number of dimensions used in the data cube. When the data cube has got more than 3 dimensions, then it is called the *hyper cube*. The dimensions form the axes of the hypercube and the solution space represents the facts as aggregates on measure attributes (see Figure 2).

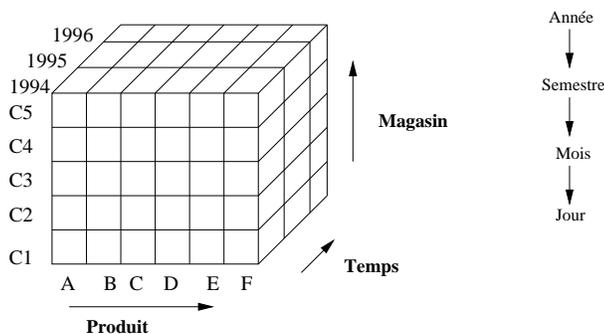


Figure 2: A Data Cube

### 2.1 Implementation Schemes

The conceptual multidimensional data model can be physically realized in two ways, (1) by using traditional relational databases, called *ROLAP architecture* (Relational On-Line Analytical Processing) (example includes Informix Red Brick Warehouse [41]) or (2) by making use of specialized multidimensional databases, called *MOLAP architecture* (Multidimensional On-Line Analytical Processing) (example includes Hyperion Essbase OLAP Server [40]). The advantage of MOLAP architecture is that it provides a direct multidimensional view of the data whereas the ROLAP architecture is just a multidimensional interface to relational data. On the other hand, the ROLAP architecture has two major advantages: (i) it can be used and easily integrated into other existing relational database systems, and (ii) relational data can be stored more efficiently

than multidimensional data [69].

We will briefly describe in details each approach.

### 2.1.1 Relational Scheme

This scheme stores the data in specialized relational tables, called fact and dimension tables. It provides a multidimensional view of the data by using relational technology as an underlying data model. Facts are stored in the fact table and dimensions are stored in the dimension table. Facts in the fact table are linked through their dimensions. The attributes that are stored in the dimension table may exhibit attribute hierarchy.

**Example 1** *Let us consider a star schema from [22]. It models the sales activities for a given company. The schema consists of three dimension tables *CUSTOMER*, *PRODUCT*, and *TIME*, and one fact table *SALES*. The tables and attributes of the schema are shown in Figure 3.*

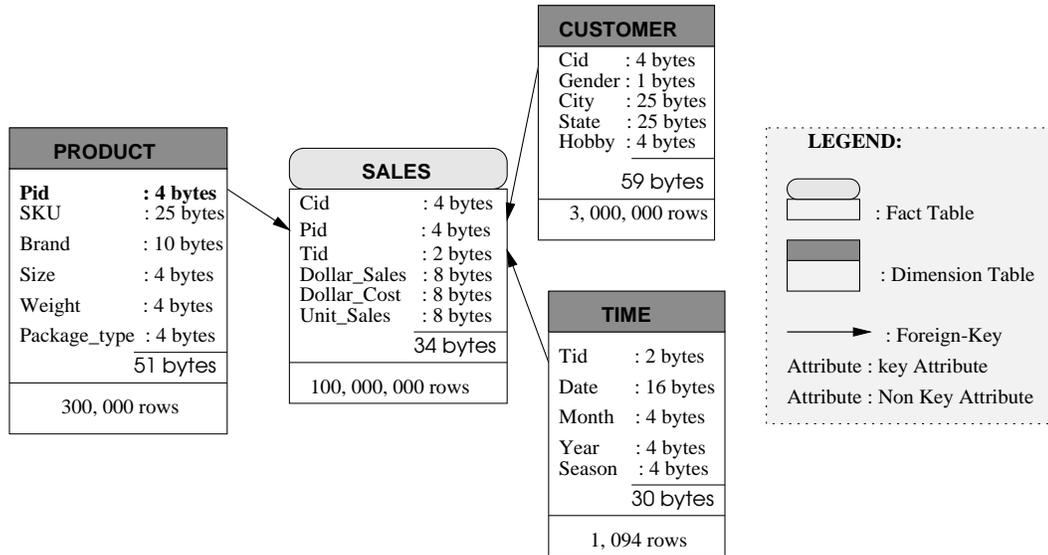


Figure 3: An Example of a Star Schema

*Star schema/snowflake schema* is used to support multidimensional data representation. It offers flexibility, but often at the cost of performance because of more joins for each query required. A star/snowflake schema models a consistent set of facts (aggregated) in a fact table and the descriptive attributes about the facts are stored in multiple dimension tables. This schema makes heavy use of de-normalization to optimize complex aggregate query processing. In a star schema, a single fact table is related to each dimension table in a many-to-one (M:1) relationship. Each dimension tuple is pointed to many fact tuples. Dimension tables are joined to fact table through foreign key reference; there is a referential integrity constraints between fact table and dimension table. The primary key of the fact table is a combination of the primary keys of dimension tables. Note that multiple fact tables can be related to the same dimension table and the size of dimension table is very small as compared to the fact table.

As we can see in Figure 3, the dimension table *TIME* is de-normalized and therefore, the star schema does not capture hierarchies (i.e. dependencies among attributes) directly. This is captured in *snowflake schema*. Here, the dimension tables are normalized for simplifying the data selecting operations related to the dimensions, and thereby, capture attribute hierarchies. In this schema, the multiple fact tables are created for different aggregate levels by pre-computing

aggregate values. This schema projects better semantic representation of business dimensions. The Figure 4 shows an example of snowflake schema after *TIME* dimension table in Figure 3 has been normalized.

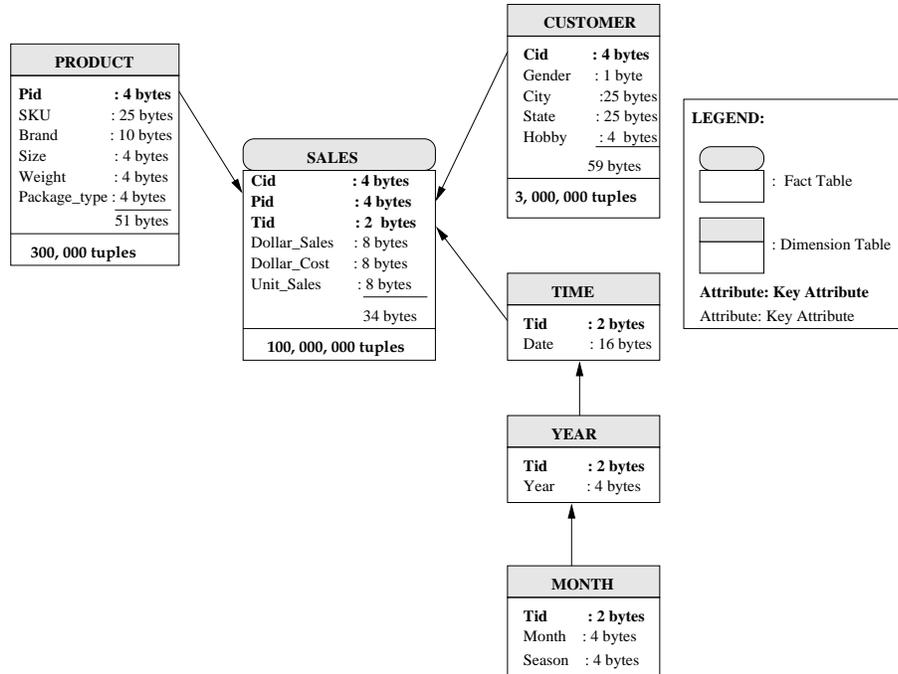


Figure 4: Snowflake Schema Example

A star schema/snowflake schema is usually a query-centric design as opposed to a conventional update-centric schema design employed in OLTP applications. The typical queries on the star schema are commonly referred to as star-join queries, and exhibit the following characteristics:

1. There is a multi-table join among the large fact table and multiple smaller dimension tables, and
2. Each of the dimension tables involved in the join has multiple selection predicates on its descriptive attributes.

### 2.1.2 Multidimensional Scheme

This scheme stores data in a matrix using array-based storage structure. Each cell in the array is formed by the intersection of all the dimensions, therefore, not all cells have a value. The multi-dimensional data set requires smaller data storage since the data is clustered compactly in the multidimensional array. The values of the dimensions need not be explicitly stored. The  $n$ -dimensional table schema is used to support multidimensional data representation which is described next.

#### $n$ -dimensional Table Schema

An  $n$ -dimensional table schema is the fundamental structure of a multidimensional database which draws the terminology of the statistical databases. The attribute set associated with

this schema is of two kinds: *parameters* and *measures*. An  $n$ -dimensional table has a set of attributes  $R$  and a set of dimensions  $D$  associated with it. Each dimension is characterized by a distinct subset of attributes from  $R$ , called the parameters of that dimension. The attributes in  $R$  which are not parameters of any dimension are called the measure attributes. This approach is a very unique way of *flattening* the data cube since the table structure is inherently multidimensional. The actual contents of the table are essentially orthogonal to the associated structure. Each *cell* of the data cube can be represented in an  $n$ -dimensional table as table entries. These table entries have to be extended by certain dimensions to interpret their meaning. The current literature on an  $n$ -dimensional table however does not give an implementation of the MDDB which is different from the implementation suggested by the already existing schemas. This implementation breaks up the  $n$ -dimensional table into dimension tables and fact tables which snowballs into snowflake schema and traditional ROLAP. The challenge with the research community is to find mechanisms that translate this multidimensional table into a true multidimensional implementation. This would require us to look at new data structures for the implementation of multiple dimensions in one table. The relation in relational data model is a classic example of 0-dimensional table.

## 2.2 Constraints on the Cube Model

In a relational schema, we can define a number of integrity constraints in the conceptual design. These constraints can be broadly classified as key constraints, referential integrity constraints, not null constraint, relation-based check constraints, attribute-based check constraints and general assertions (business rules). These constraints can be easily translated into triggers that keep the relational database consistent at all times. This concept of defining constraints based on dependencies can be mapped to a multidimensional scenario.

The current literature on modeling multidimensional databases has not discussed the constraints on the data cube. In a relational model, the integrity and business constraints that are defined in the conceptual schema provide for efficient design, implementation and maintenance of the database. Taking a cue from the relational model, we need to identify and enumerate the constraints that exist in the multidimensional model. An exploratory research area would be to categorize the cube constraints into classes and compare them with the relational constraints. The constraints can be broadly classified into two categories: *intra-cube* constraints and *inter-cube* constraints.. The intra-cube constraints define constraints within a cube by exploiting the relationships that exist between the various attributes of a cube. The relationship between the various dimensions in a cube, the relationships between the dimensions and measure attributes in a cube, dimension attribute hierarchy and other cell characteristics are some of the key cube features that need to be formalized as a set of intra-cube constraints. The inter-cube constraints define relationships between two or more cubes. There are various considerations in defining inter-cube constraints. Such constraints can be defined by considering the relationships between dimensions in different cubes, the relationships between measures in different cubes, the relationships between measures in one cube and dimensions in the other cube and the overall relationship between two cubes, i.e., two cubes might *merge* into one, one cube might be a *subset* of the other cube, etc.

## 2.3 Operations in Multidimensional Data Models

Data warehousing query operations include standard SQL operations, such as selection, projection and join. In addition, it supports various extensions to aggregate functions, for example, percentile functions (e.g. top 20 percentile of all products), rank functions (e.g. top 10 products), mean, mode, and median. One of the important extensions to the existing query language

is to support multiple ‘group by’ by defining *roll-up*, *drill-down*, and *cube* operators. Roll-up corresponds to doing further group-by on the same data object. Note that roll-up operator is order sensitive, that is, when it is defined in the extended SQL, the order of columns (attributes) matters. The function of drill-down operation is the opposite of roll-up.

The hypercube which involves joining of multiple tables to represent facts needs a new set of algebraic operations. A new algebra needs to be proposed for the multidimensional environment. The idea of *faster* query processing requires an extension to existing SQL in the existing environment. New operators like *cube*, *push*, *pull*, *restrict*, *star join* and *merge* have been proposed in literature but all these operators are very specific to the schema for which they are designed [3, 48, 50, 6].

### 3 Data Partitioning in Data Warehouses

The data partitioning concept in the context of distributed databases aims to reduce query execution time and facilitates the parallel execution of queries. In this chapter, we use partitioning and fragmentation interchangeably. Partitioning is an important technique for implementing very large tables in data warehouse environments [21]. The idea is to make a large table more manageable by dividing it in multiple tables. Oracle first introduced a limited form of partitioning with partition views in Oracle7 (Release 7.3). Fully functional table partitioning is available in Oracle8 [21].

#### 3.1 Motivation

The main reasons that motivate us to use the partitioning in data warehouse environments are:

1. building indices like join indices on the whole data warehouse schema can cause problem of maintaining them, because whenever we need to execute a query, the whole indices should be loaded from the disk to the main memory. The sizes of this type of indices can be very huge [10]. But if we have a partitioned data warehouse with  $N$  sub-star schemas, we can build for each sub-star schema join indices that can be easier to maintain and to load. On the other hand, even though indexing can help in providing good access support at the physical level, the number of irrelevant data retrieved during the query processing can still be very high. The partitioning aims to reduce irrelevant data accesses [7, 59].
2. since the OLAP queries use joins of multiple dimension tables and a fact table, the derived horizontal partitioning developed in the relational databases can be used to efficiently process joins across multiple relations [16].
3. designing a warehouse database for an integrated enterprise is a very complicated and iterative process since it involves collection of data from many departments/units and data cleaning, and requires extensive business modeling. Therefore, some organizations have preferred to develop a *data mart* to meet requirements specific to a departmental or restricted community of users. Of course, the development of data marts entail the lower cost and shorter implementation time. The role of the data mart is to present convenient **subsets of a data warehouse** to consumers having specific functional needs. There can be two approaches for developing the data mart, either it can be designed integrating data from source data (called bottom-up approach) or it can be designed deriving the data from warehouse (called, top-down approach) [28] (see Figure 5). By advocating the top down design for data mart and without partitioning, we need to assign to each data mart the whole data warehouse; even this data mart accesses only a subset of this data warehouse.

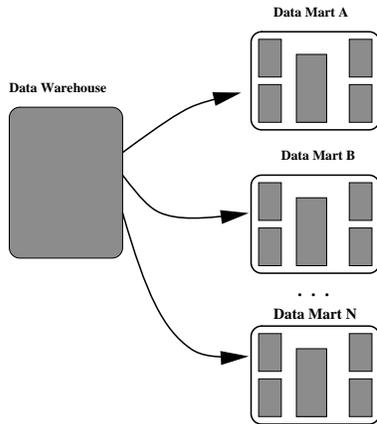


Figure 5: The Top Down Flow from DWs to Data Marts

By analyzing the needs of each data mart, we can partition the centralized data warehouse into several fragments that can be used as allocation units for data marts.

4. parallelism is a good technique to speed up the OLAP query execution. With a partitioned sub-star schema, we can associate each star schema to one machine and execute queries in parallel. For example, MCI Telecommunications' IT data center in Colorado Springs is running a massive 2TB decision support data warehouse called *warehouseMCI* on a 104 node IBM RS/6000 SP massively parallel processing system. The database is growing at the rate of 100GB to 200GB per month [23].
5. the main disadvantage of the horizontal partitioning in databases, is that when update operations occur some tuples may migrate from one fragment to another. The migration operation is sometimes costly. In the data warehouse environments, updates are not common [45] compare to the append operations. Even if there are some update operations [38], the data warehouses are only periodically updated in a batch fashion [58], during the time where the data warehouses are unavailable for querying. This situation makes the utilization of horizontal partitioning feasible, as the reorganization of its fragments can be done off-line.

Partitioning plays an important role in the design of a distributed database system [59]. It enables the definition of appropriate units of distribution which enhance query performance by enabling concurrent and parallel execution of queries. Data partitioning involves using the application/query processing characteristics to fragment the database into a set of fragments. Each fragment contains data that is most relevant to one or more applications. Therefore, an application accesses only a subset of fragments (not the complete database) thus reducing the amount of irrelevant data accessed [59].

### 3.2 Fragmentation Alternatives

Two types of partitioning are possible: vertical and horizontal

Given a relation  $R(K, A_1, A_2, \dots, A_n)$ , where  $A_1, A_2, \dots, A_n$  are the attributes and  $K$  is the key. Each attribute  $A_i$  has a domain of values  $dom(A_i)$ . The vertical partitioning (VP) of  $R$  is given by  $V^1(K, A_1^1, A_2^1, \dots, A_{k_1}^1), V^2(K, A_1^2, A_2^2, \dots, A_{k_2}^2), \dots, V^p(K, A_1^p, A_2^p, \dots, A_{k_p}^p)$  where in each  $A_j^i \in \{A_1, A_2, \dots, A_n\}$ . The set of vertical fragments  $V_1, V_2, \dots, V_p$  is disjoint if each attribute  $A_i$  ( $1 \leq i \leq n$ ) of relation  $R$  belongs to one and only one vertical fragment. The key  $K$  is repeated in each fragment to facilitate the reconstruction of relation  $R$  from the vertical

fragments. The original relation  $R$  is reconstructed by joining the  $p$  vertical fragments. The main advantage of vertical partitioning is that it reduces irrelevant attribute accesses of the user queries. The main disadvantage is that it requires the costly join operation to access two or more vertical fragments.

Each horizontal fragment of relation  $R$  is a subset of the tuples of  $R$ . The tuples of each fragment must satisfy a predicate clause. The horizontal partitioning (HP) of  $R$  is given by:  $H_1 = \sigma_{cl_1}(R), H_2 = \sigma_{cl_2}(R), \dots, H_q = \sigma_{cl_q}(R)$ , where  $cl_i$  is a predicate clause. The reconstruction of the relation  $R$  is obtained by uniting the original fragments with the union operation.

Two versions of HP are cited by the researchers [16]: primary HP and derived HP. Primary HP of a relation is performed using predicates that are defined on that relation. On the other hand, derived HP is the partitioning of a relation that results from predicates defined on another relation.

A lot of work has been done on the partitioning in the relational models [59] and object models [15, 9, 27, 7] compared to the data warehouses. In this chapter, we will concentrate on HP because it is well adapted to data warehouses. Also, the TPC-D benchmark [60] in its implementation allows the utilization of HP, but it discards the utilization of the vertical partitioning.

### 3.3 Horizontal Partitioning Algorithms in Databases

Several algorithms were proposed in performing HP [17, 54, 56, 55, 59, 15, 27, 26]. These algorithms can be classified into two main categories: (1) query-driven algorithms and (2) data-driven algorithms.

#### 3.3.1 Query-Driven Algorithms

These algorithms are performed based on a set of most frequently asked queries and their access frequencies for a specific application [55]. These queries respect the 80/20 rule which considers that 20% of user queries account for 80% of the total data access in the database system. This category of algorithms is divided into two types:

- **Affinity based algorithms:** The most proposed algorithms for HP in the relational and object databases are affinity-based [17, 59, 56, 15, 27]. The affinities are used between predicates. Predicates having a high affinity are grouped together to form a horizontal fragment. The algorithm of Navathe et al. [56] starts by performing an analysis on the predicates defined by a set of queries accessing a relation to be horizontally partitioned. From these queries, a *predicate usage matrix* is built. The rows and the columns of this matrix represent the queries and predicates, respectively. The value of this matrix  $use(Q_i, p_j)$  is equal to 1, if the predicate  $p_l$  is used by the query  $Q_j$ , 0 otherwise. This matrix is used to generate another matrix called *predicate affinity matrix*, where each value  $(p_l, p_{l'})$  represents the sum of the frequencies of queries which access predicates  $p_l$  and  $p_{l'}$ , simultaneously. After that, the authors apply the graph-based algorithm defined in [55] to group these predicates into a disjoint subsets of predicates (each subset of predicates can be a potential horizontal fragment).

In each group, the authors optimize (if it is possible) by using implications between the predicates defined in queries, then they generate the horizontal fragments. Each fragment is defined by a boolean combination of predicates using the logical connectives ( $\wedge, \vee$ ).

Özsu et al. [59] developed a primary algorithm for HP. This algorithm has an input, a set of queries and their frequencies. Its steps are as follows:

1. determine the set of *complete and minimal* predicates  $P = \{p_1, p_2, \dots, p_n\}$  [16]. A complete set of predicates is *minimal* if, and only if, all its elements are relevant.
2. determine the set of *minterm predicates*  $M$  of  $P$  which is defined as follow:  

$$M = \{m_i \mid m_i = \bigwedge_{p_j \in P} p_j^*\}, \text{ where } p_j^* = p_j \text{ or } p_j^* = \neg p_j \text{ (} 1 \leq j \leq n), (1 \leq i \leq 2^n)$$
3. eliminate the contradictory minterm predicates using the predicate implications.

This approach can generate  $2^n$  horizontal fragments for  $n$  simple predicates.

- **Cost-based algorithms:** The main disadvantage of the affinity based algorithms is that they use the affinity as a measure of groupment of predicates. This measure can *only express the affinity between pairs of attributes*, and cannot express the affinity among several (more than two) predicates [9]. On the other hand, all these algorithms ignore the physical costs corresponding to the savings in the amount of irrelevant data accessed. The utility of the HP can be measured by the amount of savings in disk accesses required for query execution. It is often argued that the advantage of the HP lies in the reduction in the amount of savings in disk accesses required for query execution [44, 36]. Although this argument is understandable from an intuitive point of view, not much work has been done to evaluate the impact of this type of partitioning on query evaluation on a quantitative basis. According to Karlapalem et al. [44], two factors, input/output (IO) operations and data transfer, are the most important for the performance of the applications in distributed database systems. As the goal of partitioning of the classes is to obtain the minimum cost for processing a set of queries, we develop an algorithm based on a cost-driven approach for executing a set of queries that respects the 80/20 rule. In [8, 9], we have proposed an algorithm for the horizontal partitioning in the object model based on a cost model, but it can be applied to the relational model. The basic idea of this algorithm is that it starts with a set of the most frequently asked queries, and other physical factors (like predicate selectivity, class size, page size, instance length, etc.).

Let  $C$  be the class to be horizontally partitioned, and let  $P = \{p_1, p_2, \dots, p_N\}$  be the set of simple predicates defined on this class using a set of queries. From these simple predicates, we generate all minterm predicates [16]  $\{m_1, m_2, \dots, m_z\}$ . After that, we exhaustively enumerate all possible schemes. A scheme can be represented by a minterm or a combination of several minterms using the logical connector OR. For each scheme, we calculate the cost of executing all these queries. The objective of this cost model is to calculate the cost of executing these queries, each of which accesses a set of objects. The cost of a query is directly proportional to the number of pages it accesses. The total cost is estimated in terms of disk page accesses. Finally, the scheme with minimal cost gives the best HP of the class  $C$ .

We note that this algorithm needs an exhaustive enumeration strategy to generate all schemes. For small values of minterms, this procedure is not computationally expensive. However, for large values of number of minterms, the computation is very expensive, for example, for 15 minterms, the number of schemes is 1382958545. For further details about the complexity of enumerating all schemes, see [8].

To reduce the complexity of the cost-driven algorithm, we propose another algorithm called *approximate algorithm* which is based on a hill-climbing technique [43]. The approximate algorithm starts with the set of fragments generated by an algorithm that has a lower complexity [15, 56]. This algorithm gives rise to a set of fragments. Based on the queries accessing these fragments, the approximate algorithm tries to shrink or expand some fragments *in order to reduce the query processing cost*.

### 3.3.2 Data-Driven Algorithms

This kind of HP was studied in parallel databases and formulated as follows: suppose that we have  $N$  disks available, for any given row of a relation  $R$ , we must decide on which of the  $N$  disks it has to reside [53]. Three standard approaches are developed: range partitioning, round robin, and hashing. With the range partitioning each of the  $N$  disks is associated with a range of key values. The main advantage of range partitioning technique is that it acts as a kind of built-in-index if tuples are retrieved based on the key values. But its main disadvantage is the data skew. In second technique, the tuples are partitioned in a round robin fashion. This strategy is used as the default strategy in Gamma machine [26]. This technique avoids the load imbalance. Finally, the hash partitioning is performed using a hash function, where identical key values will be hashed to the same disk. It also avoids data skew.

### 3.4 Partitioning Issues in Data Warehouses

All work done on vertical partitioning in the data warehouse context has been applied on the physical design level (index selection problem).

The vertical partitioning has been introduced on the definition of projection index proposed by O’Neil et al. [58]. Chaudhuri et al. [20] developed a technique called *index merging* to reduce storage and maintenance of index. Their method is somehow an extension of the vertical partitioning. Recently, Datta et al. [25] developed a new indexing technique called “*Curio*” in data Warehouses modeled by a star schema. This index speeds up the query processing and it does not require a lot of storage space.

Concerning the HP, a little work has been done. Noaman et al. [57] proposed an architecture for a distributed data warehouse. It is based on the ANSI/SPARC architecture [67] that has three levels of schemas: internal, conceptual and external. The distributed data warehouse design proposed by these authors is based on top-down approach. There are two fundamental issues in this approach: fragmentation and allocation of the fragments to various sites. Authors proposed a horizontal fragmentation algorithm for a fact table of a data warehouse. This algorithm is an adaptation of the work done by [59].

In this chapter, we present a methodology for partitioning a data warehouse modeled by a star schema and we will show the issues and problems related to this problem.

### 3.5 Problems

Partitioning in data warehouses is more challenging compared to that in relational and object databases. This challenge is due to the several choice of partitioning schema of a star or snow flake schemas:

1. Partition only the dimension tables using the primary partitioning. This scenario is not suitable for OLAP queries, because the sizes of dimension tables are generally small compare to the fact table. Most of OLAP queries access the fact table which is very huge. Therefore, any partitioning that do not take into account the fact table is discarded.
2. Partition only the fact table using the primary partitioning. In a data warehouse modeled by a star schema, most of OLAP queries access dimension tables first and after that to the fact table (see 2.1.1).
3. Partition both fact and dimension tables, but independently. This means that we use the primary partitioning to partition the fact and dimension tables without taking into account the relationship between these tables. Based on the previous choices, this partitioning is not benefit for OLAP queries.

4. Partition *some/all* dimension tables using the primary partitioning, and use them to derived partitioned fact table. This approach is best in applying partitioning in data warehouses. Because it takes into consideration the queries requirements and the relationship between the fact table and dimension tables. In our study, we opt for last solution.

### 3.6 Partitioning Algorithm for a Star Schema

In [12], we have proposed a methodology for fragmenting a star schema with  $d$  dimension tables  $\{D_1, D_2, \dots, D_d\}$  and one fact table  $F$ . This methodology can be easily adapted to snowflake schemas. In this section, we will review the basic ideas of this approach.

Note that any fragmentation algorithm needs application informations defined on the tables that have to be partitioned. The information is divided into two categories [59]: quantitative and qualitative. Quantitative information gives the selectivity factors of selection predicates and the frequencies of queries accessing these tables. Qualitative information gives the selection predicates defined on dimension tables.

A simple predicate  $p$  is defined by:

$$p : A_i \theta \text{ Value}$$

where  $A_i$  is an attribute,  $\theta \in \{=, <, \leq, >, \geq, \neq\}$ ,  $\text{Value} \in \text{Dom}(A_i)$ .

The algorithm we proposed in [12] has as input a set of most frequently asked OLAP queries  $Q = \{Q_1, Q_2, \dots, Q_n\}$  with their frequencies. The main steps of this algorithm are:

1. Enumeration of all simple predicates used by the  $n$  queries.
2. Attribution to each dimension table  $D_i$  ( $1 \leq i \leq d$ ) its set of simple predicates ( $SSP^{D_i}$ ).
3. Each dimension table  $D_i$  having  $SSP^{D_i} = \phi$  cannot be fragmented. Let  $D_{candidate}$  be the set of all dimension tables having a non empty  $SSP^{D_i}$ . Let  $g$  be the cardinality of  $D_{candidate}$ .
4. Application of COM\_MIN algorithm [59] to each dimension table  $D_i$  of  $D_{candidate}$ . This algorithm takes a set of simple predicates and then generates a set of complete and minimal.
5. For fragmenting a dimension table  $D_i$ , it is possible to use one of the algorithms proposed by [16, 59] in the relational model. These algorithms generate a set of disjoint fragments, but their complexities are exponential of the number of used simple predicates. As result, we use our algorithm proposed in the object model which has a polynomial complexity [7].  
Each dimension table  $D_i$  has  $m_i$  fragments  $\{D_{i1}, D_{i2}, \dots, D_{im_i}\}$ , where each fragment  $D_{ij}$  is defined as follows:  
 $D_{ij} = \sigma_{cl_j^i}(D_i)$  where  $cl_j^i$  ( $1 \leq i \leq g, 1 \leq j \leq m_i$ ) represents a clause of simple predicates.
6. Partition the fact table using the fragmentation schema of the dimension tables.

The number of fragments of the fact table ( $N$ ) is equal to:  $N = \prod_{i=1}^g m_i$  (for more details see [7]). Therefore, the star schema  $S$  is decomposed into  $N$  sub-schemas  $\{S_1, S_2, \dots, S_N\}$ , where each one satisfies a clause of predicates.

**Example 2** To show how this algorithm works, let us consider the star schema in Figure 3 and the six OLAP queries obtained from [12]. From these queries, we enumerate all selection predicates:

$p_1 : C.Gender = 'M'$ ,  $p_2 : C.Gender = 'F'$ ,  $p_3 : P.Package\_type = "Box"$ ,  $p_4 : P.Package\_type = "Paper"$ ,  $p_5 : T.Season = "Summer"$  and  $p_6 : T.Season = "Winter"$ .

The set of simple predicates for each table are (step 2):  $SSP^{CUSTOMER} = \{p_1, p_2\}$ ,  $SSP^{PRODUCT} = \{p_3, p_4\}$  and  $SSP^{TIME} = \{p_5, p_6\}$ .

For each set, we generate the set of complete and minimal simple predicates (step 4): We obtain the following:

- $SSP_{Min-Com}^{CUSTOMER} = \{p_1, p_2\}$ ,
- $SSP_{Min-Com}^{PRODUCT} = \{p_3, p_4\}$  and
- $SSP_{Min-Com}^{TIME} = \{p_5, p_6\}$

By applying the fragmentation algorithm [7] for each dimension table, we obtain the following fragments:

- $CUSTOMER : Cust\_1 = \sigma_{Gender='M'}(CUSTOMER)$  and  $Cust\_2 = \sigma_{Gender='F'}(CUSTOMER)$ ,
- $PRODUCT : Prod\_1 = \sigma_{Package\_type='Box'}(PRODUCT)$  and  $Prod\_2 = \sigma_{Package\_type='Paper'}(PRODUCT)$ ,
- $TIME : Time\_1 = \sigma_{Season='Winter'}(TIME)$  and  $Time\_2 = \sigma_{Season='Summer'}(TIME)$

Finally, the fact table can be horizontally partitioned into 8 ( $N = 2 \times 2 \times 2$ ) fragments.

Our algorithm generates a large number of fragments of the fact table. For example, suppose that the dimension tables are fragmented as follows:

- $CUSTOMER$  into 50 fragments using the State attribute <sup>1</sup>,
- $TIME$  into 12 fragments using the Month attribute, and
- $PRODUCT$  into 2 fragments using the Package\_type

Therefore, the fact table is fragmented into 1200 ( $= 50 \times 12 \times 2$ ) fragments.

Consequently, it will be very hard for the data warehouse administrator (DWA) to maintain these fragments. Therefore it is important to reduce the number of fragments of the fact table. We focus on this problem in the next sections.

### 3.6.1 Horizontal Partitioning and OLAP Queries

When the derived HP is used to fragment a relation  $R$  based on the fragmentation schema of a relation  $S$ , two potential cases of join exist: simple join and partitioned join.

In the data warehouse context, when the fact table is horizontally partitioned based on the dimension tables, we will never have a partitioned join (i.e., the case wherein a horizontal fragment of the fact table has to be joined with more than one fragment of the dimension table will not occur). As a result, we will have only simple joins as given by the following theorem:

**Theorem 1** *Let  $F$  and  $D_i$  be a fact table and a dimension table of a given star schema, respectively. If the dimension table  $D_i$  is horizontally partitioned into set of disjoint horizontal fragments let say,  $\{D_{i1}, D_{i2}, \dots, D_{im_i}\}$ , where each fragment is defined by clause predicate:  $D_{ij} = \sigma_{cl_j}(D_i)$  and the fact table  $F$  is derived partitioned based on the HF's of  $D_i$ , then, the distributed join between  $F$  and  $D_i$  is always represented only by simple join graph.*

**Proof 1** *We prove it by contradiction. Let  $F_p$  be a fragment of the fact table  $F$  defined by:  $F_p = F \ltimes D_{ij}$ . Suppose that  $F_p$  can be joined with two fragments  $D_{ij}$  et  $D_{il}$  ( $l \neq j$ ) of the dimension table  $D_i$ . Consequently, we will have:*

$$F_p \ltimes D_{ij} \neq \emptyset \quad (1)$$

$$F_p \ltimes D_{il} \neq \emptyset \quad (2)$$

---

<sup>1</sup>case of 50 states in the U.S.A.

Note that the fragments of  $D_i$  are disjoint, and the fragment  $F_p$  is obtained using the semi-join operation between the fact table  $F$  and a fragment of dimension table  $D_i$ . Note that the join attributes are the foreign key of  $F$  and the primary key of  $D_i$ . Therefore, one semi-join condition among the two above defined in (1) and (2) is satisfied. In this case,  $D_{ij} = D_{il}$ . Since  $F_p$  can be any fragment of  $F$ , and it is joinable with exactly one fragment of  $D_i$ , we conclude that the join graph between  $F$  and  $D_i$  is always simple<sup>2</sup>.

In a data warehouse modeled by a star schema, any fact table  $F$  that is derived from a horizontally partitioned based on HP schema of dimension tables  $\{D_1, D_2, \dots, D_d\}$  will result in simple distributed join graph. This has two advantages:

- It avoids costly total distributed join (i.e., every HF  $F_p$  ( $1 \leq p \leq N$ ) of the fact table  $F$  joins with each and every HF  $D_{ik}$  of each and every dimension table  $D_i$  ( $1 \leq i \leq d$  and  $1 \leq k \leq m_i$ )).
- It facilitates parallel processing of multiple simple distributed joins.

### 3.6.2 The Correctness Rules of Our Proposed Algorithm

Any fragmentation algorithm must guarantee the correctness rules of fragmentation: completeness, reconstruction, and disjointness.

- The completeness ensures that all tuples of a relation are mapped into at least one fragment without any loss. The completeness of the dimension tables is guaranteed by the use of COM\_MIN algorithm [59, 16]. The completeness of the derived horizontal fragmentation of the fact table is guaranteed as long as the referential integrity rule is satisfied among the dimension tables and the fact table.
- The reconstruction ensures that the fragmented relation can be reconstructible from its fragments [59]. In our case, the reconstruction of the fact and the dimension tables are obtained by union operation, i.e.,  $F = \cup_{i=1}^N F_i$  and  $D_i = \cup_{j=1}^{m_i} D_{ij}$ .
- The disjointness ensures that the fragments of a relation are non-overlapping. This rule is satisfied for the dimension table since we used an no-overlap algorithm [7]. For the fragments of the fact table, the disjointness rule is guaranteed by the theorem 1.

## 3.7 Dimension Table Selection Problem

As we have seen in section 3.6, the number of fragments of the fact table can be very large. This is due to the number of dimension tables used in fragmenting the fact table. In this section, we will give some issues on selecting dimension tables that reduce the number of fact table fragments. Any selection algorithm should satisfy the following constraints:

- avoids the explosion of the number of fragments of the fact table, and
- guarantees a good performance of executing a set of OLAP queries.

To reach the first objective, we give to the DWA the possibility of choosing the number of fragments ( $W$ ) that he/she considers that it is sufficient to maintain. To satisfy the second objective, we need to have the possibility of augmenting the number of fragments of the fact table till the performance is guaranteed. The problem is to find a compromise between the maintenance cost and query processing cost as shown in Figure 6.

---

<sup>2</sup>The theorem is true when we have a distributed join between the fact table and several dimension tables

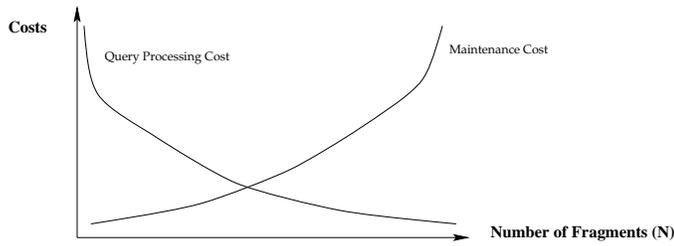


Figure 6: Cost Evolution

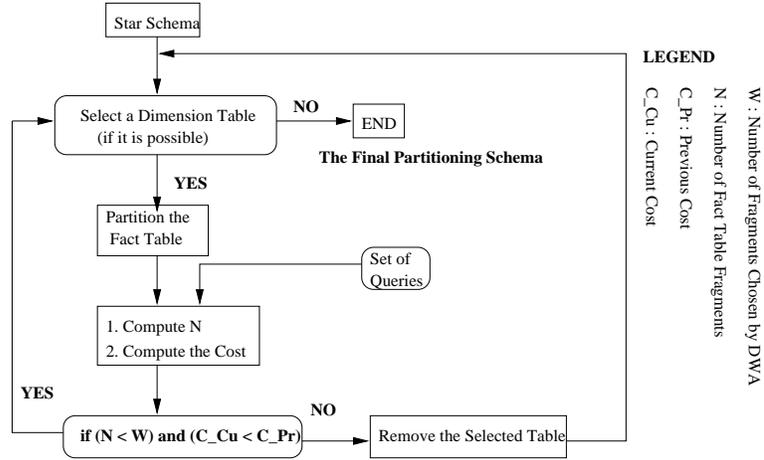


Figure 7: The Steps of Greedy Algorithm

To solve this problem, we developed a greedy algorithm [12]. From the number of fragments  $W$  that the DWA chooses, this algorithm starts by selecting one dimension table randomly. Once the selection is done, we partition the fact table based on the fragmentation schema of the dimension table. We compute the number of fragments of the fact table ( $N$ ), and then cost of executing a set of OLAP queries<sup>3</sup>. If the number of fragments  $N$  is less than  $W$  and there is an improvement of query processing cost, our algorithm selects another dimension and then it repeats the same process till the two conditions are satisfied. The main steps of this algorithm are shown in Figure 7. At the end of this algorithm, we obtain a partitioned data warehouse ensuring a good query processing cost and a maintenance cost.

### 3.8 Query Execution Strategy in Partitioned Star Schema

Since the dimension tables and fact table are horizontally partitioned, we need to ensure that the data access transparency concept in which, a user of the data warehouse is purposefully unaware of the distribution of the data. Our goal is to provide to the data warehouse users the unpartitioned star schema and the query optimizer task is to translate the OLAP queries on the unpartitioned star schema to partitioned star schemas. Before executing a query  $Q$  on a partitioned data warehouse, we need first to identify the sub-schemas satisfying this query as shown in Figure 8.

**Definition 1 (Relevant Predicate Attribute (RPA))** is an attribute which participates in a predicate which defines a HP. Any attribute which does not participate in defining a predicate which defines a HP is called an irrelevant predicate attribute.

<sup>3</sup>We suppose we have a cost model for executing a set of queries

**Definition 2 (Partitioning Specification Table)** Suppose we have an initial star schema  $S$  horizontally partitioned into  $N$  sub-schemas  $\{S_1, S_2, \dots, S_N\}$ . The partitioning conditions for each partitioned table can be represented by a table called partitioning specification table of  $S$ . This table has three columns: the first one contains the table names, the second provides the fragments of its corresponding table, and the last one reports the partitioning condition for each fragment.

**Example 3** Suppose that the dimension table *CUSTOMER* is partitioned into two fragments: *Cust\_1* and *Cust\_2* and the fact table is fragmented using these two fragments into *Sales\_1* and *Sales\_2*. The corresponding partitioning specification table for this example is illustrated in Table 1.

From the partitioning specification table, we can conclude that : each attribute belonging to the partitioning specification table is a RPA. In our example, “Gender” is the single RPA.

Table	Fragments	Fragmentation Condition
CUSTOMER	Cust_1	Gender = ‘M’
	Cust_2	Gender = ‘F’
SALES	Sales_1	SALES $\times$ Cust_1
	Sales_2	SALES $\times$ Cust_2

Table 1: Partitioning Specification Table

### 3.8.1 Fragment Identification

Let  $Q$  be a query with  $p$  selection predicates defined on a partitioned star schema  $S = \{S_1, S_2, \dots, S_N\}$ . Our aim is to identify the sub-schema(s) that participate in executing  $Q$ . Let  $SRPA$  be the set of RPA. Based on the selection predicates of the query  $Q$  and partitioning specification table, we proceed as follow:

- For each selection predicate  $SP_i$  ( $1 \leq i \leq p$ ), we define the function  $\mathbf{attr}(SP_i)$  which gives us the name of the attribute used by  $SP_i$ . The union of  $\mathbf{attr}(SP_i)$  gives us the names of all attributes used by  $Q$ . We call this set by **query predicate attributes** . Let  $SPA(Q)$  be the set of all predicate attributes used by the query  $Q$ .
- Using  $SPA(Q)$  and  $SRPA(S)$ , four scenarios are possible:
  1.  $SPA(Q) = \emptyset$ , (the query  $Q$  does not contain any selection predicate). In this situation, two approaches are possible to execute  $Q$ :
    - (a) Perform the union operations of all sub-schemas and then perform the join operations as in unpartitioned star schema.
    - (b) Perform the join operations for each sub-schemas and then assemble the result using the union operation.
  2.  $(SPA(Q) \neq \emptyset)$  et  $(SPA(Q) \cap SRPA(S) = \emptyset)$  (the query has some selection predicates on non partitioned dimension tables, or the predicate attribute of  $Q$  do not match with relevant predicate attribute). To execute this kind of queries, we use the two approaches presented in 1.a and 1.b.
  3.  $(SPA(Q) \cap SRPA(S) \neq \emptyset)$  means that some predicate attributes of the query  $Q$  match with certain RPA. In this case, we can easily determine the names of dimension tables and the fragments that participate in executing the query  $Q$ .

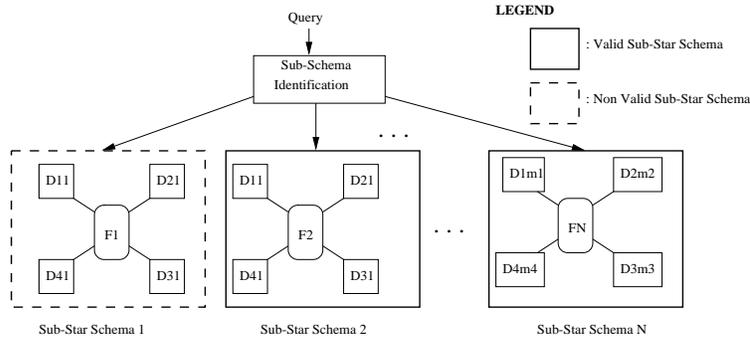


Figure 8: Sub-schemas Identification

### 3.9 Results

In [12], we have developed two cost models to evaluate the utility of the HP. The first one is for unpartitioned star schema, and the second one is for partitioned star schema. These cost models are used for executing a set of frequently asked queries in the data warehouse. The objective of these cost models is to calculate the cost of executing these queries in terms of disk page accesses (IO cost) during the selection and join operations (which are the most used and most expensive operations in data warehouses [49]). To characterize the improvement of performance using the horizontal fragmentation technique, we defined a normalized IO metric as follow:

$$\text{Normalized IO} = \frac{\# \text{ Of IOs for a Horizontally Partitioned Star Schema}}{\# \text{ Of IOs for a Unpartitioned Star Schema}}$$

We note that if the value of normalized IO is less than 1.0, then it implies that HP is beneficial. The main observations are:

- Horizontal fragmentation gives almost good performance compared to the unpartitioned case.
- Horizontal fragmentation may deteriorate the execution performance of certain queries. For example, the queries that do not have any selection predicates. To evaluate these two queries, we need to access all sub-star schemas.
- The number of partitioned dimension tables has a great impact on reducing the query processing cost. As the number of fragmented dimension tables increases the performance increases too.

## 4 Materialized Views

One of the techniques employed in data warehouses to improve performance is the creation of set of materialized views. They are used to pre-compute and store aggregated data such as sum of sales. They can also used to pre-compute joins with or without aggregations. So materialized views are used to reduce the overhead associated with expensive joins or aggregations for a large or an important class of queries [21]. The data warehouse at the Mervyn's department-store chain, for instance, has a total of 2400 precomputed tables to improve query processing [34]. Materialized views can be used in several areas: *distributed computing* and *mobile computing*.

- In distributed environments, they are used to replicate data at distributed sites and synchronize update done at several site with conflict resolution methods.

- In mobile computing, they are used to download a subset of data from central servers to mobile clients, with periodic refresh from the central servers and propagation of updates by clients back to the central servers.

A materialized view definition can include any number of aggregates, as well as any number of joins. In several ways, a materialized view behaves like an index [21]:

- The purpose of a materialized view is to increase request execution performance
- The existence of a materialized view is transparent to applications (e.g., SQL applications), so a DWA can create or drop materialized views at any time without affecting the validity of applications
- A materialized view consumes storage space
- The contents of the materialized view must be maintained when the underlying tables are updated (modified).

All data warehousing products support materialized views [21, 63].

Two major problems related to materialized views are: (1) the view selection problem, and (2) the view maintenance problem.

#### 4.1 View Selection

We initiated a discussion in Section 1 as to which views should be materialized at the data warehouse. To aid answering the queries efficiently, we materialize a set of views that are “closely-related” to the queries at the data warehouse. We cannot materialize all possible views, as we are constrained by some resource like disk space, computation time, or maintenance cost [34]. Hence, we need to select an appropriate set of views to materialize under some resource constraint. The view selection problem (VSP) is defined as selection of views to materialize minimize the query response time under some resource constraint. All studies showed that this problem is a NP-hard. The proposed solutions for the VSP can be divided into two categories: the static VSP and the dynamic VSP.

**Static VSP** This problem starts with a set of frequently asked queries (a priori known), and then select a set of materialized views that minimizes the query response time and under some constraint. The selected materialized views will be a benefit only for a query belonging to the set of a-priori known queries.

**Dynamic VSP** The static selection of views contradicts the dynamic nature of decision support analysis. Especially for ad-hoc queries where the expert user is looking for interesting trends in the data repository, the query pattern is difficult to predict [46]. In addition, as the data and these trends are changing overtime, a static selection of views might very quickly become outdated. This means that the DWA should monitor the query pattern and periodically “re-calibrate” the materialized views by running these algorithms. Kotidis and Roussopoulos [46] present a system called DynaMat that dynamically materializes information at multiple levels of granularity in order to match the demand (workload), but also takes into account the maintenance restrictions for the warehouse, such as down time to update the views and space availability. This system unifies the view selection and view maintenance problems under a single problem. DynaMat constantly monitors incoming queries and materializes the best set of views subject to the space constraint. During updates, DynaMat reconciles the current selected materialized views and refreshes the most beneficial subset of it.

## 4.2 Algorithms for VSP

We have proposed several algorithms for VSP to find the optimal or near-optimal solutions for it. These algorithms can be classified into three categories based on the type of resource: (1) algorithms without resources [5, 72], (2) algorithms driven by the space constraint [34], and (3) algorithms driven by maintenance cost [34].

### 4.2.1 Algorithms without resource

We now discuss a heuristic [5] in detail which uses the data cube technology and the lattice model. The lattice model feeds on the *attribute hierarchy* defined earlier. The nodes in the *lattice* diagram represent views (aggregated on certain dimensions) to be materialized at the data warehouse. If the dimensions of two views  $a$  and  $b$  exhibit attribute hierarchy such that  $dim(a) \rightarrow dim(b)$  then there is an edge from node  $a$  to node  $b$ . Node  $a$  is called the *ancestor* node and node  $b$  is called the *dependent* or *descendant* node. The lattice diagram allows us to establish relationships between views that need to be materialized. Some queries can be answered by using the already materialized views at the data warehouse. For answering such queries, we need not go to the raw data. The lattice diagram depicts dependencies between the views and a good view selection heuristic exploits this dependency. The view selection algorithm tries to minimize the average time required to evaluate a view and also keeps a constraint on the space requirements. The space requirements that can be expressed as the number of views to be materialized translates this into an optimization problem that is NP-complete. An approximate and acceptable solution for this problem is the *greedy* heuristic. The greedy algorithm selects a view from a set of views depending upon the benefit yielded on selecting that view. A view  $a$  that is materialized from view  $b$  incurs a materializing cost that is equal to the *number of rows* in view  $b$ . If there is a view  $c$  (materialized on  $b$ ; number of rows in  $c \leq$  number of rows in  $b$ ) such that view  $a$  can be derived from  $c$ , the cost of materializing  $a$  reduces to the number of rows in  $c$ . Thus the benefit of materializing view  $c$  includes the benefit incurred by view  $a$  in the form of reduction in its materializing cost (number of rows in  $b$  - number of rows in  $c$ ). The greedy heuristic selects a view that maximizes the benefit that will be yielded on materializing that view. The benefit of materializing each *dependent* view (a node in the lattice diagram) will change with the selection of an *ancestor* view in the data warehouse. After each selection is made, the benefit at each *dependent* node in the lattice is recalculated and a view with maximum benefit is selected. It has been shown that the greedy algorithm is at least 3/4 of optimal. The greedy algorithm can be extended to restrict on actual space requirements rather than the number of views to be materialized. The frequency with which the views are accessed can be incorporated in this algorithm. Different schema design methods will give different set of tables at the data warehouse.

Yang et al. [72] presented a framework to highlight issues of materialized view design. The basic concept the authors used to develop the view selection algorithm is a graph called “multiple view processing plan”. This graph specifies the views that the data warehouse will maintain (either materialized or virtual). The MVPP is a directed acyclic graph that represents a query processing strategy of data warehouse views. The leaf nodes correspond to the base relations, and the root nodes represent the queries. We call the layer between root nodes and leaf nodes the *potential view layer (PVL)*. This layer contains all potential materialized views. Each node in the PVL is assigned with two costs: the query processing cost and the maintenance cost [72]. Note that there can be more than one MVPP for the same set of queries depending upon the access characteristics of the application and the physical data warehouse parameters. If we have  $n$  intermediate nodes, then we need to try  $2^n$  combinations of nodes. Therefore, the authors formulate the VSP as an integer programming problem.

### 4.2.2 Algorithms driven by Space constraint

In [34], the authors present competitive polynomial-time heuristics for selection of views to optimize total query response time, for some important special cases of the general data warehouse scenarios: (i) an OR view graph, in which any view can be computed from any one of its related views, e.g., data cubes, and (ii) an AND view graph, where each query has a unique evaluation. They extend their heuristic to the most general case of AND-OR view graphs.

### 4.2.3 Algorithms driven by Maintenance Cost

In [34], the authors considered the maintenance-cost-view-selection problem in which it is required to select a set of views to be materialized in order to maximize the query response time under a constraint of maintenance time. This problem is NP-hard. The authors developed a couple of algorithms to solve this problem. For OR view graph, they present a greedy algorithm that selects a set of views. They also present an  $A^*$  heuristic for the general case of AND-OR view graphs.

## 4.3 View Maintenance

A *data warehouse* stores integrated information from multiple data sources in *materialized views* ( $MV$ ) over the source data. The data sources ( $DS$ ) may be heterogeneous, distributed and autonomous. When the data in any source (base data) changes, the  $MVs$  at the *datawarehouse* need to be updated accordingly. *The process of updating a materialized view in response to the changes in the underlying source data is called View Maintenance.* The view maintenance problem has evoked great interest in the past few years. This view maintenance in such a distributed environment gives rise to inconsistencies since there is a finite unpredictable amount of time required for (a) propagating changes from the  $DS$  to the *datawarehouse* and (b) computing view updates in response to these changes. Data consistency can be maintained at the data warehouse by performing the following steps:

- propagate changes from the data sources ( $ST_1$  - current state of the data sources at the time of propagation of these changes) to the data warehouse to ensure that each view reflects a consistent state of the base data.
- compute view updates in response to these changes using the state  $ST_1$  of the data sources.
- install the view updates at the data warehouse in the same order as the changes have occurred at the data sources.

The inconsistencies at the data warehouse occur since the changes that take place at the data sources are random and dynamic. Before the data warehouse is able to compute the view update for the *old* changes, the *new* changes change the state of the data sources from  $ST_1$  to  $ST_2$ . This violates the consistency criterion that we have listed. Making the  $MVs$  at the data warehouse self-maintainable decimates the problem of inconsistencies by eliminating the finite unpredictable time required to query the data source for computing the view updates. In the next subsection, we describe self-maintenance of materialized views at the data warehouse.

### 4.3.1 Self-Maintenance

Consider a materialized view  $MV$  at the data warehouse defined over a set of base relations  $R = \{R_1, R_2, \dots, R_n\}$ .  $MV$  stores a  $p$  reprocessed query at the data warehouse. The set of base relations  $R$  may reside in one data source or in multiple, heterogeneous data sources. A

change  $\Delta R_i$  made to the relation  $R_i$  might affect  $MV$ .  $MV$  is defined to be self-maintainable if a change  $\Delta MV$  in  $MV$ , in response to the change  $\Delta R_i$  can be computed using only the  $MV$  and the update  $\Delta R_i$ . But the data warehouse might need some additional information from other relations in the set  $R$  residing in one or more data sources to compute the view update  $\Delta MV$ . Since the underlying data sources are decoupled from the data warehouse, this requires a finite computation time. Also the random changes at the data sources can give rise to inconsistencies at the data warehouse. Some data sources may not support full database functionalities and querying such sources to compute the view updates might be a cumbersome, even an impossible task. Because of these problems, the preprocessed query that is materialized at the warehouse needs to be maintained without access to the base relations.

One of the approaches is to replicate all base data in its entirety at the data warehouse so that maintenance of the  $MV$  becomes local to the data warehouse [32, 33]. Although this approach guarantees self-maintainability at the warehouse, it creates new problems. As more and more data is added to the warehouse, it increases the space complexity and gives rise to information redundancy which might lead to inconsistencies. This approach also overlooks the point that the base tuples might be present in the view itself, so the view instance, the base update and a subset of the base relations might be sufficient to achieve self-maintainability in the case of SPJ (Select-Project-Join) views [39]. *But how can the subset of the base relations that is needed to compute the view updates be stored at data warehouse?* This question was addressed in [61], which defines a set of minimal *auxiliary views (AVs)* to materialize that are sufficient to make a view self-maintainable. Although materializing auxiliary views at the *DataWarehouse* was a novel concept, the minimality of auxiliary views defined was still questionable since the  $MV$  instance was never exploited for self-maintenance. Most of the current approaches maintain the  $MVs$  separately from each other using a separate *view manager* for each view and such approaches fail to recognize that these views can be maintained together by identifying the set of related materialized views. This issue of multiple-view self-maintenance was addressed for the first time in [39].

In some approaches to multiple-view self-maintenance, a set of auxiliary views ( $AV$ ) are stored at the data warehouse along with the set of materialized views ( $MV$ ) such that together  $MV \cup AV$  is self-maintainable. The research challenge lies in finding the most *economical AVs* in terms of space complexity and computational costs. The view self maintenance is still an active research problem. It is not always feasible to provide self-maintainability of the views at the data warehouse. When the cost of providing self-maintainability exceeds the cost of querying data sources for computing view updates, it is profitable to allow querying of data sources instead.

### 4.3.2 Consistency Maintenance

Current research has also concentrated on ensuring consistency of the data warehouse when the  $MVs$  are not self-maintainable since it is not always possible to provide for complete self-maintainability at the data warehouse. The *ECA* family of algorithms [73] introduces the problem and solves it partially. The *Strobe* algorithm [74] introduces the concept of queuing the view updates in the *Action – List* at the data warehouse and installing the updates only when the *unanswered query set (UQS)* is empty. The algorithm solves the consistency problem but is subject to the potential threat of infinite waiting. There are other mechanisms that are based on time stamping the view updates [4]. These methods do not address the consistency problems in their entirety and also assume the notion of global time.

We propose that the self-maintainability of views at the data warehouse should be a dynamic property. We should continuously monitor the cost of providing self-maintainability and when this cost increases beyond a certain threshold, the maintenance mechanism should be shifted to

querying data sources to compute the view updates. This threshold can be computed depending on the cost of querying data sources. An effective algorithm that provides this dynamism and efficient garbage collection is the need of the hour.

### 4.3.3 Update Filtering

The changes that take place in the source data need to be reflected at the data warehouse. Some changes may create view updates that need to be installed at the data warehouse; some changes leave the views at the data warehouse unchanged. If we are able to detect at the data sources that certain changes are guaranteed to leave the views unchanged, we need not propagate these changes to the data warehouse. This would require checking of distributed integrity constraints at a single site. As many changes as possible can be filtered at the sources and only the changes that result in view updates may be propagated to the warehouse. The update filtering will reduce the size of the maintenance transactions at the data warehouse, thus minimizing the time required to make the data warehouse consistent with the data sources. The side effect of update filtering is that we need to make our data sources (and the wrapper/monitor) components more intelligent. They need to know about their participation in the data warehouse and the data warehouse configuration so that the updates can be checked against the constraint set before propagating them. To be able to realize this, the data sources cannot be decoupled from the data warehouse anymore. This would give rise to new problems like configuration management i.e., if there is a change in the schema at any data source or at the data warehouse, all the participating entities need to be informed of this change so that they can modify the constraint set to reflect this change. The view maintenance strategies would now be based on the constraint set and any change to the constraint set would warrant a change in the existing view maintenance transaction.

### 4.3.4 On-Line View Maintenance

Warehouse view maintenance can be either done *incrementally* or by queueing a large number of updates at the data sources to be propagated as a *batch* update from the data sources to the data warehouse. In current commercial systems, a batch update is periodically sent to the data warehouse and view updates are computed and installed. This transaction is called the *maintenance transaction*. A user typically issues read-only queries at the data warehouse and a long-running sequence of user queries is called a *reader session*. The batch maintenance transaction is typically large and blocks the reader sessions. This makes the data warehouse *offline* for the duration of the maintenance transaction. The maintenance transaction typically runs at night. With the advent of the Internet and global users, this scheme will have to give way. The *24-hour* shop concept is what most companies are striving for and the data warehouse to be online for 24 hours allows the company to be competitive in its strategies. Incremental view maintenance which updates the data warehouse instantaneously in response to every change at the data source is expensive and gives rise to inconsistent results during the same reader session. An update from the data source will change the results a user might see over a sequence of queries. We need to get around these problems. [62] discusses a possible approach to this problem by maintaining two versions of each tuple at the data warehouse simultaneously so that the reader sessions and the maintenance transactions do not block each other. A possible solution may need the integration with self maintenance techniques, where auxiliary views can be used to answer queries during maintenance transactions.

## 5 Indexing in Data Warehouses

Indexing has been at the foundation of performance tuning for databases for many years. It is the creation of access structures that provide faster access to the base data relevant to the restriction criteria of queries [25]. The size of the index structure should be manageable so that benefits can be accrued by traversing such a structure. The traditional indexing strategies used in database systems do not work well in data warehousing environments. Most OLTP transactions typically access a small number of rows; most OLTP queries are *point* queries. B trees which are used in most common relational database systems are geared towards such *point* queries. They are well suited for accessing a small number of rows. An OLAP query typically accesses a large number of records for summarizing information. For example, an OLTP transaction would typically query for a customer who booked a flight on TWA1234 on say April 25th; on the other hand an OLAP query would be more like “Give me the number of customers who booked flight on TWA1234 in say one month”. The second query would access more records and these are typically *range* queries. B tree indexing scheme which is so apt for OLTP transactions is not the best suited to answer OLAP queries. An index can be *single-column* or *multi-columns* of a table (or a view). An index can be either clustered or non-clustered. An index can be defined on one table (or view) or many table using a join index [68]. In data warehouse context, when we talk about index, we refer to two different things: (1) indexing techniques and (2) index selection problem.

### 5.1 Indexing Techniques

A number of indexing strategies have been suggested for data warehouses: **Value-List Index**, **Projection Index** [58], **Bitmap Index** [58], **Bit-sliced Index** [58], **Data Index** [25], **Join Index** [68], and **Star Join Index** [66].

#### 5.1.1 Value-List Index

The value-list index consists of two parts. The first part is a balanced tree structure and the second part is a mapping scheme. The mapping scheme is attached to the leaf nodes of the tree structure and points to the tuples in the table being indexed. The tree is generally a B tree with varying percentages of utilization. Oracle provides a B\* tree with 100% utilization [1]. Two different types of mapping schemes are in use. First one consists of a RowID list which is associated with each unique search-key value. This list is partitioned into a number of disk blocks chained together. The second scheme uses bitmaps. A bitmap is a vector of bits that store either a 0 or a 1 depending upon the value of a predicate. A bitmap B lists all rows with a given predicate P such that for each row r with ordinal number j that satisfies the predicate P, the jth bit in B is set. Bitmaps efficiently represent low-cardinality data, however to make this indexing scheme practical for high-cardinality data, compression techniques must be used. Value-list indexes have been shown in [58] to outperform other access method in queries involving the MIN or MAX aggregate functions, as well as queries that compute percentile values of a given column. Bitmap indexes can substantially improve performance of queries with the following characteristics [2]:

- The WHERE clause contains multiple predicates on low-or-medium-cardinality columns (e.g., a predicate on Gender that has two possible values: female or male as shown in Figure 9).
- Bitmap indexes have been created on some or all of these low-or-medium-cardinality columns

CUSTOMER Table				BM1	BM2
Name	Age	...	Gender	M	F
Dupond	20		M	1	0
Lee	42		F	0	1
Jones	21		M	1	0
Martin	52		M	1	0
Ali	18		F	0	1
Qing	17		F	0	1
Hung	36		M	1	0

Figure 9: Example of a Bitmap Index on Gender

CUSTOMER Table				Projection Index on Age
Name	Age	...	Gender	
Dupond	20		M	20
Lee	42		F	42
Jones	21		M	21
Martin	52		M	52
Ali	18		F	18
Qing	17		F	17
Hung	36		M	36

Figure 10: Example of a Projection Index

- Tables being queried contain many rows

### 5.1.2 Projection Index

A projection index is equivalent to the column being indexed. If  $C$  is the column being indexed, then the projection index on  $C$  consists of a stored sequence of column values from  $C$  in the same order as the ordinal row number in the table from where the values are extracted (see Figure 10). It has been shown in [58] that projection indexes outperform other indexing schemes for performing queries that involve computation on two or more column values and appear to perform acceptably well in GROUP-BY like queries.

### 5.1.3 Bit-sliced Index

A bit sliced index represents the key values of the column to be indexed as binary numbers and projects a set of *bitmap slices* which are orthogonal to the data held in the projection index. This index has been shown in [58] to particularly perform well for computing sums and averages. Also, it outperforms other indexing approaches for percentile queries if the underlying data is clustered, and for range queries whose range is large.

### 5.1.4 Join Index and Bitmap Join Index

A join index [68] is the result of joining two tables on a join attribute and projecting the keys (or tuple IDs) of the two tables. To join the two tables, we can use the join index to fetch the tuples from the tables followed by a join. In relational data warehouse systems, it is of interest to perform a multiple join (a Star Join) on the fact tables and their dimension tables. Therefore, it will be helpful to build join indexes between the keys and the dimension tables and the corresponding foreign keys of the fact table. If the join indexes are represented in bitmap

matrices, a multiple join could be replaced by a sequence of bitwise operations, followed by a relatively small number of fetch and join operations [71].

### 5.1.5 Data Index

A `DataIndex`, like the projection index, exploits the positional indexing strategy [24]. The `DataIndex` avoids duplication of data by storing only the index and not the column being indexed. The `dataindex` can be of two specific types: `Basic DataIndex` and `Join DataIndex` (for more information, see [24]).

## 5.2 Index Selection Problem

The index selection problem (ISP) has been studied since the early 70's and the importance of this problem is well recognized. As VSP, ISP consists in picking a set of indexes for given set of OLAP queries under some resources constraints. Most of proposed studies show that ISP is a NP-complete problem. Recently, Microsoft [19, 20] developed an index selection tool called `AutoAdmin`. The goal of our research in the `AutoAdmin` tool is to make database systems self-tuning and self-administering. This goal is achieved by enabling databases to track the usage of their systems and to gracefully adapt to application requirements. Thus, instead of applications having to track and tune databases, databases actively auto-tunes itself to be responsive to application needs.

Note that indexes requires a very huge amount of space. Chaudhuri et al. [20] (from Microsoft) addressed a problem called `Storage-minimal Index Merging` problem that takes an *existing set of indices*, and produces a new set of indices with significantly *lower storage and maintenance cost*. This is a strong reason to motivate researchers to find novel techniques for better use of storage capacity available for a Data Warehouse. The merging technique is similar to the reconstruction of vertical fragments of a relation.

Labio et al. [47] combined the problems of selecting views and indexes into one problem called `view-index selection problem (VISP)`. This problem starts with a set of materialized views (primary views) and tries to add a set of supporting views to these primary views. The objective is to minimize total maintenance cost for the data warehouse, but not distributing storage space between materialized views and indices. For that, the authors proposed an algorithm based on  $A^*$  to find the optimal solution. This algorithm takes as input the set of all *possible supporting views* and indices to materialize. The space allocation is not taken into consideration by this algorithm, because it supposes that the primary views are already selected (their storage capacity is already assigned). This algorithm is in fact not practical because the number of possible supporting views and indices can be very huge. Heuristic rules (concerning the benefit and cost of supporting views and indices) to reduce the complexity of  $A^*$  are suggested.

## 6 Interaction between Indexes and Views

Conceptually, both materialized views and indices are physical structures that can significantly accelerate performance [63]. An effective physical database design tool must therefore take into account that the interaction between indexes and materialized views by considering them together to optimize the physical design for the queries on the system [14, 63]. An effective physical database design tool must therefore take into account that the interaction between indexes and materialized views by considering them together to optimize the physical design for the workload on the system. Ignoring this interaction can significantly compromise the quality of the solutions obtained by the physical database design tool.

This interaction gets a great attention of researchers [14] and industrials [63]. Recently, an architecture and algorithms for selecting views and indexes are presented by Microsoft SQL Server 2000.

Most of the previous work in physical database design has considered the problems of index selection and materialized view selection in isolation. However, both indexes and views are fundamentally similar - both are redundant structures that speed up query execution, compete for the same resource (**storage**), and incur maintenance overhead in the presence of updates [63].

When the VSP and ISP are treated independently (i.e., the views and indexes are selected in *sequential manner*). To select them, the DWA does the following tasks:

- The DWA has to run an algorithm to select views to be materialized according to the storage space reserved for views.
- after that, the DWA runs another algorithm to select indices over base relations and materialized views according to storage space reserved for indices.

The combination of VSP and ISP gives rise to two problems that are not *addressed* by the data warehouse community. The first problem is *selecting join index in presence of materialized views*, and the second one is *storage space distribution among materialized views and indices*.

**Join Index Selection with Materialized Views** Once materialized views are selected, all OLAP queries will be rewritten using these materialized views (this process is known as *query rewriting* [65]). A rewriting of a query  $Q$  using views is a query expression  $Q'$  that refers to these views. In SQL, after rewriting process of OLAP queries, we can find in the FROM clause a set of materialized views, dimension tables and the fact table. These views can be joined with each other or with other tables. Indexing a single materialized view can be done by using the same indexing techniques for a table ( $B^+$ -tree). The star join index (SJI) [66, 58] has been used in data warehouses modeled by a star schema. It denotes a join index between a *fact table* and multiple *dimension tables*. The SJI has been proved to be an index structure for speeding up joins defined on fact and dimension tables in data warehouses [66, 58]. However, there has not been much research reported in enhancing star join algorithms for efficiently selecting and performing join indices with materialized views.

**Space Distribution between Views & Indices** The fact that the VSP and ISP are combined, the space distribution becomes a crucial issue. The task of distributing storage space among materialized views and indices in order to improve performance is very difficult for the DWA. This difficulty is due to several factors: (1) metrics are needed to decide on the distribution of the storage space among materialized views and indices, (2) the mutual interdependencies between views and indices need to be considered for an optimal solution and (3) the problem of redistribution of storage space among materialized views and indices after update operations (deletions and insertions), or changes in query sets needs to be addressed.

Further, when updates are performed over the underlying data warehouse, the corresponding changes should be applied to materialized views and indices [38]. Thus, their sizes can increase/decrease. Therefore, it is necessary to re-distribute the storage space among views and indices. Finally, the problem that must be addressed is: *how to automatically distribute storage space between materialized views and indices to efficiently execute a set of queries?*

In the next sections, we describe the graph join index concept and space distribution among materialized views and indexes.

## 6.1 Graph Join Indexes

The index selection phase is aimed at determining the best set of indices for a given set of OLAP queries. The objective of index selection is to pick a set of indices that is optimal or as close to optimal as possible [18, 19, 20, 35, 68, 49]. Like the materialized views, indices require a certain *amount of disk space*, and only a limited number of indices can be selected for a given set of OLAP queries. Indices can be defined on one table (dimension table, fact table, or view) using for example a  $B^+$ -tree. They can also be defined on more than two tables to speed up operations such as joins.

In this section, we suggest a new type of join index called a graph join index (GJI) which is used to speed up queries defined on materialized views and on tables (dimension or fact).

### 6.1.1 Motivating Example

Assume that the sales company is interested in determining the total sales for male customers purchasing product of type package “box”. The following SQL query  $Q_1$  may be used for this purpose:

```
SELECT    SUM(S.dollar_sales)
FROM      CUSTOMER C, PRODUCT P, SALES S,
WHERE     C.Cid = S.Cid
AND       P.Pid = S.Pid
AND       C.Gender = 'M'
AND       P.Package_type = 'Box'
GROUP BY PID
```

We also assume that the company maintains the following view consisting of finding the total sales for each product having “box” as type of package. This view is defined as follows:

```
CREATE VIEW V_2
SELECT    *
FROM      PRODUCT P, SALES S,
WHERE     P.Pid = S.Pid
AND       C.Package_Type = 'Box'
GROUP BY PID
```

The materialized view  $V_2$  can be used to evaluate the query  $Q_1$  by joining  $V_2$  with the dimension table CUSTOMER. The rewritten query  $Q'_1$  that uses  $V_2$  is:

```
SELECT    SUM(S.dollar_sales)
FROM      CUSTOMER C, V_2,
WHERE     V_2.Cid = C.Cid
AND       C.Gender = 'M'
GROUP BY PID
```

Note that the fact table SALES is very huge, and the materialized view  $V_2$  is likely to be orders of magnitude smaller than SALES table. Hence, evaluating  $Q'_1$  will be much faster than evaluating the query  $Q_1$ , because,  $Q_1$  needs two join operations, and  $Q'_1$  needs only one. There is a solution, to reduce the number of join operations of the query  $Q_1$  which is the SJI [66]. The suitable SJI for  $Q_1$  is (CUSTOMER, SALES, PRODUCT), but it requires a very high storage capacity [25] that can slow down the execution of the query  $Q_1$ . We can define a join index between the view  $V_2$  and the dimension table CUSTOMER ( $V_2$ , CUSTOMER). This index speeds up the execution of the query  $Q_1$  and its size is much smaller than the SJI (CUSTOMER, SALES, PRODUCT).

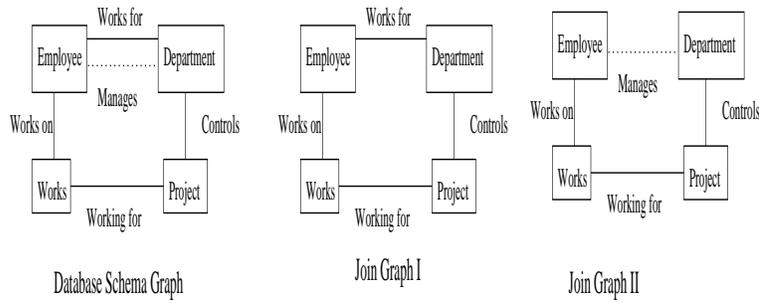


Figure 11: Database Schema graph and Join Graphs

Note that the complete SJI (CUSTOMER, SALES, PRODUCT, TIME) would be sufficient to process any query against our star schema in Figure 3. However, without the fact table in its element, this index does not guarantee any performance [22]. In the presence of materialized views, a join index can be defined on several views, therefore, by removing one view, the join index may still guarantee the performance of some queries. This is because, most of materialized views are obtained from joining the fact table with other dimension tables.

### 6.1.2 Notations & Definitions

Assume that the OLAP queries are based on the star join: each query is a join between the fact table and dimension tables filtered by some selection operations and followed by an aggregation. This type of queries has the following syntax:

```

SELECT <Projection List> <Aggregation List>
FROM <Fact Table> <Dimension Tables>
WHERE <Selection Predicates & Join Predicates>
GROUP BY <Dimension Table Attributes>

```

Note that in the syntax of queries without materialized, the FROM clause contains the fact table and dimension tables. When we consider the materialized views, our queries should be rewritten using these views [65]. Therefore, the FROM clause can contain: fact table, materialized views and dimension tables.

**Definition 3 (Database Schema Graph (DBSG))** is a connected graph where each node is a relation and an edge between two nodes  $R_i$  and  $R_j$  implies a possibility of join between relations  $R_i$  and  $R_j$ .

**Definition 4 (Join Graph (JG))** is a database schema graph such that there exists exactly one join condition between two relations.

For a given database schema, we can have more than one edge (join) between two nodes as shown in Figure 11, where we have two join operations between tables *Employee* and *Department*. One join condition relates employee to the department he/she is working, and another join condition relates an employee to the department he/she is managing. In this case, we decompose the database schema graph into two join graphs as shown in Figure 11, and then we consider GJIs for each join graph, separately.

Since we are focusing on data warehouse schemas (e.g., star, snowflake), a join graph has only single join condition between tables. Further study on the complexity of GJIs derived from database schema graph with multiple independent join conditions is beyond the scope of this chapter. Therefore, a JG is the same as DBSG.

**Claim 1** A JG in a data warehouse with only dimension tables and fact table is always connected. In this case, a JG is called **total join graph**.

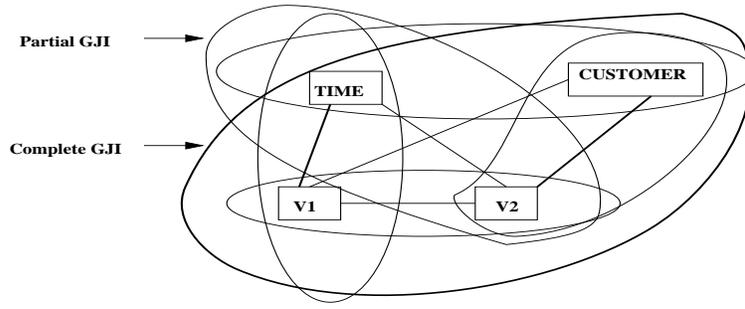


Figure 12: Join Graph and Graph Join Indices

**Claim 2** *In a data warehouse context with dimension tables, fact table and materialized views, a JG can be disconnected, i.e., it contains more than one subgraph. In this case, it is called **partitioned join graph**.*

The partitioned join graph occurs when:

1. Views do not have common join attributes. For example, a view defined on CUSTOMER and another in PRODUCT.
2. Views have common join attributes, but they can never be joined. For example, suppose we have two views  $V_1$  and  $V_2$  defining the sales activities of female customers and male customers, respectively. The result of joining these two views is always empty.

**Definition 5 (Graph Join Index (GJI))** *A GJI is a subset of nodes of a join graph.*

In case of distributed join graph, a GJI is a subset of nodes of each sub-graph.

**Claim 3** *A SJI is a particular case of GJI.*

A data warehouse schema consists of a star schema (a set of dimension tables and a fact table) with a set of materialized views. Each table  $T_i$  being either a dimension table, or a view, or a fact table has a key  $K_{T_i}$ . A GJI on  $m$  tables is a table having  $m$  attributes  $(K_{T_1}, K_{T_2}, \dots, K_{T_m})$ , and which is denoted as :

$$(T_1 \sim T_2 \sim \dots \sim T_m).$$

Since the star schema is connected, there is a path connecting the tables  $(T_1, T_2, \dots, T_m)$  by including additional tables, if necessary. The GJI supplies the keys of the relations  $(T_1, T_2, \dots, T_m)$  that form the result of joining all the tables along the path. In case there are multiple paths that connect tables  $(T_1, T_2, \dots, T_m)$ , then there can be multiple GJIs, each catering to different query. A GJI with all tables of a join graph is known as a complete GJI. A GJI with a subset of nodes of a join graph refers to a partial GJI (see Figure 12).

A similar indexing technique to GJI was proposed by Segev et al. [64] called join pattern index in expert systems. This index exists on all attributes involved in selection and join conditions. It refers to an array of variables that represent the key and join attributes of the data associated with the join.

Note that for a star schema with  $n$  tables, the number of possible GJIs grows exponentially, and is given by:

$$\binom{2^n - n - 1}{0} + \binom{2^n - n - 1}{2} + \dots + \binom{2^n - n - 1}{2^n - n - 1} = 2^{(2^n - n - 1)}$$

As GJI has at least two nodes, the total number of GJIs possible is  $2^n - n - 1$ . Therefore, the above equation gives us the total number of ways a set of GJIs can be generated. However, if we want to find only one GJI, the number of GJI possibilities are  $2^n - n - 1$ . Therefore, the

problem of efficiently finding the set of GJIs that provides the minimum total query processing cost while satisfying a storage constraint is very challenging to solve.

## 6.2 GJI Selection (GJI-S) Problem Formulation

All the index selection algorithms select indices to optimize the objective function employed, subject to the constraint that the indices included cannot consume more than the specified amount of storage.

The inputs of the GJI-S problem are as follows:

1. A star schema with the fact table  $F$  and dimension tables  $\{D_1, D_2, \dots, D_d\}$ .
2. A set of most frequently asked queries  $Q = \{Q_1, Q_2, \dots, Q_s\}$  with their frequencies  $\{f_1, f_2, \dots, f_s\}$ . Each query  $Q_i$  ( $1 \leq i \leq s$ ) is represented by a query graph  $QG_i$ .
3. A set of materialized views  $V = \{V_1, V_2, \dots, V_m\}$  selected to support the execution of queries in  $Q$ .
4. A storage constraint  $S$ .

The GJI-S problem selects a set of GJIs  $G'$  among all possible GJIs ( $G$ ), that minimizes the total cost for executing a set of OLAP queries, under the constraint that the total space occupied by the GJIs in  $G'$  (denoted by  $S_{G'}$ ) is less than  $S$ .

More formally, let  $C_g(Q_i)$  denotes the cost of answering a query  $Q_i$  using a GJI  $g$  ( $g \in G'$ ). The problem is to select a set of GJIs  $G'$  ( $G' \subseteq G$ ) that minimizes the total query processing cost ( $TQPC$ ) such as:

$$TQPC = \sum_{i=1}^s f_i \times \{\min_{g \in G} C_g(Q_i)\} \quad (3)$$

under the constraint that:  $S_{G'} \leq S$ .

### 6.2.1 GJI-S Algorithms

In [11], we have proposed three algorithms to select an optimal or near-optimal GJIs for a set of queries: a naive algorithm, a greedy algorithm for selecting one GJI, and a greedy algorithm for selecting more than one GJIs. These three algorithms are guided by an objective function that calculates the cost of executing a set of queries.

1. The naive algorithm (NA) is used for a comparison purpose. NA enumerates all possible GJIs, and calculates the contribution of each GJI in executing a set of queries. Finally, it picks the GJI having the minimum cost while satisfying the storage constraint.
2. The greedy algorithm for selecting one GJI (GA1) starts with a storage constraint (SC)  $S$ , and then selects a GJI with two nodes that corresponds to most frequently executed join among all the queries, and tries to expand it by adding more nodes while checking whether the total processing cost decreases. When no more expand operations can be applied, the algorithm tries to shrink it to check if a better GJI can be derived. When no more expand or shrink operations can be applied, the algorithm ends generating the best possible GJI. The details of the algorithm are presented in [11].
3. Note that as a single GJI generated by GA1 can only efficiently execute some but not all of the queries, we need more than one GJI to efficiently execute all the queries. Therefore, we develop a greedy algorithm for selecting more than one GJI (GAK). The GAK starts

with the initial solution provided by GA1 as the first index. After that it selects the edge from weight graph that has the highest query access frequency and which is not a GJI generated by GA1. Then it tries to expand (one or more times) and shrink (one or more times) until no further reduction in total cost can be achieved and the storage constraint is satisfied. This generates the second GJI, and it keeps repeating this procedure until the storage constraint is violated.

### 6.3 Space Distribution among Views and Indexes

The problem of space distribution among views and indexes should be treated in two cases: *static* and *dynamic*. In the static case, all data warehouse parameters are considered fixed. In the dynamic case, we assume that the tables in the data warehouse are updated.

Static space distribution between materialized views and indexes can be formulated as follows:

Given a set of frequently asked retrieval queries  $Q = \{Q_1, Q_2, \dots, Q_s\}$ , a set of frequently asked update queries  $U = \{U_1, U_2, \dots, U_{s'}\}$  and their frequencies, and a storage space  $S$  to store materialized views  $V$  and indices  $I$  to support the above queries, distribute  $S$  among materialized views and indices so as to minimize the global cost. As output, we obtain:

- (1) the space distribution between materialized views ( $S^V$ ) and indices ( $S^I$ ), and
- (2) a set of materialized views and indices corresponding to ( $S^V$ ) and ( $S^I$ )

#### 6.3.1 Static Algorithm

In [13], we developed a methodology that distributes the storage space among views and indices in an *iterative manner*, so as to minimize the total query processing cost. First, we developed a cost model that calculates the cost of processing a set of OLAP queries with materialized views and indices. Assume that the DWA initially reserves a space  $S^V$  and  $S^I$  for materialized views and indices ( $S = S^V + S^I$ ), respectively. First, we apply an algorithm for selecting a set of materialized views to support the set of queries. The selection of views is constraint by the space quota reserved to them, i.e.,  $S^V$ . After that, indices are built on base relations and materialized views using the storage space  $S^I$ .

The basic idea of this approach is to have two greedy algorithms, namely, index spy and view spy, fight for the same resource (that is, storage space). These two spies work as follows:

- An index spy keeps on stealing space used by materialized views to add/change the set of indices as long as it reduces the total query processing cost. The index spy should have some policies to select materialized views that can be stolen. We define two policies: *Least Frequently Used View (LFUV)*, where the index spy selects the view that has the lowest frequency, and *Smallest View First (SVF)*, where the index spy selects the smallest view.
- After that, a view spy keeps on stealing space from indices to add/change the set of materialized views as long as it reduces the total cost. Similarly, the view spy should have some policies to select indexes that can be stolen. Two policies are defined: *Least Frequently Used Index (LFUI)* and *Largest Index First (LIF)*.

The algorithm ends when neither the index spy nor the view spy can reduce the total query processing cost. The winner is the spy who accumulated the most storage space from the other spy. The main strategy of this approach is to have two greedy algorithms, namely, index spy and view spy, fight for the same resource (that is, storage space), while trying to reduce the total cost of processing a given set of queries, and maintaining the set of materialized views and indices. The algorithm terminates when the total cost cannot be reduced any more.

This approach gives two results: (1) a set of materialized views and a set of indices with the lowest total cost, and (2) *new storage space allocation* for materialized views and indices  $S^V$  and  $S^I$  that may differ from the initial space quotas  $S^V$  and  $S^I$ .

**Example 4** *To illustrate our approach, we consider the following example. We consider the star schema of a data warehouse (Figure 3, in Section 2). This schema contains a fact table SALES and three dimension tables CUSTOMER, TIME and PRODUCT. Assume the five most frequently asked OLAP queries given in [14]. We assume that the DWA has parametered an initial allocation of **600 Megabytes** for materialized views and **200 Megabytes** for indices.*

*Our algorithm starts by selecting an initial solution for VSP and ISP. First, we execute the View\_Select algorithm [72] to select materialized views using the 600 Megabytes storage constraint. The views selected are  $V_1$ ,  $V_2$  and  $V_3$  as shown in Table 2. Using these three materialized views, we get **4054825 disk block accesses** as the total cost of processing all queries.*

View Name	View Definition	View Size (bytes)
$V_1$	$SALES \bowtie_{\sigma_{State='IL'}} CUSTOMER$	178 094 080
$V_2$	$SALES \bowtie_{\sigma_{Package='Box'}} PRODUCT$	324 435 968
$V_3$	$\sigma_{Gender='M'} CUSTOMER$	89 047 040
Total		591 577 088

Table 2: Selected Materialized Views and Their Sizes

*After that, we build indices to speed up the above queries by selecting indices using the 200 Megabytes storage constraint. By using the algorithm proposed in [10]<sup>A</sup>, we select a join index between the view  $V_2$  and the dimension table CUSTOMER, called ( $V_2 \sim CUSTOMER$ ). The storage required for the selected index is 181821440 bytes. This index reduces the query processing cost from **4054825** to **3263405**. Now, the index spy tries to steal storage space from materialized views to add one or more indices. Since the view  $V_3$  is small and less used compared to views  $V_1$  and  $V_2$ , the index spy steals 89047040 bytes (size of view  $V_3$ ) and the storage space that can be allocated for indices is now 289047040 bytes. Note that after deleting the view  $V_3$ , we will have only two views  $V_1$  and  $V_2$  and two base relations CUSTOMER and TIME to execute all five queries. Now the index spy runs the index selection algorithm [10] to find new indices. The new index is ( $V_1 \sim V_2 \sim TIME \sim CUSTOMER$ ). It reduces the total query processing cost from **3263405** to **1955305** disk block accesses. No further reduction in query processing cost is possible by executing the index spy or the view spy. Therefore, by allocating 500 Megabytes for materialized views and 300 Megabytes for indices results in a total query processing cost of 1955305 disk block accesses as shown in table 3.*

Without Views & Indices	With Views	With Views and Indices	Index Spy	View Spy
68514928	4054825	3263405	1955305	1955305
				Algorithm ends

Table 3: Cost Reduction due to Index Spy

The solution we propose for distributing space can reuse existing algorithms for views selection and index selection. Thus, the best available algorithms can be selected. Further, our methodology is flexible enough for the DWA to use only index spy or only view spy for data warehouse design. In this way, the DWA can either allocate more storage space to indices or to materialized views.

<sup>A</sup>This paper has been submitted for publication and is available at <http://www.cs.ust.hk/faculty/kamal/>

## 6.4 Issues for Dynamic Space Distribution

The data warehouse environments are known by their dynamic changes over the underlying warehouse database. Whenever, the base relations are updated at more or less regular intervals, the materialized views and indexes should be also updated. Following these updates, it is necessary to readjust/redefine the views and the indices.

Several methods have been proposed for fast rebuild of materialized views [51]. A view after this maintenance can indeed occupy more or less space than before the maintenance. The same observation is also valid for the indices. So, the space constraint may not longer be respected. If the total space occupied by the views and the indices is higher (respectively lower) than the required space, it is necessary to determine which views and/or indices must be removed (respectively added). The iterative algorithm for the static case can be easily adapted to handle this problem.

Let  $S_i^V$  and  $S_i^I$  be the spaces occupied by the views and indices before the updates. These two values are the new space constraints to respect. After updates, the algorithm of the dynamic case goes through following steps:

- First, we compute the new spaces occupied by the views ( $S_c^V$ ) and the indices ( $S_c^I$ ).
- In case of deletions, the total space  $S_c$  ( $S_c^V + S_c^I$ ) may decrease. Therefore, unused spaces for both materialized views and indices are available. We fill the unused space for materialized views by selecting a new view. Similarly, the unused space for indices is filled incrementally using the greedy algorithm *GAK* (see 6.2.1). It considers each index of the set of selected indices as an initial solution and then generates a new index (not belonging to the current set of indices).
- When insertions occur, the current total space ( $S_c$ ) occupied by the views and indices may be higher than their initial space ( $S_i$ ). This augmentation can affect view space, index space or both.
  1. Only the space occupied by the views is higher than their required space. Intuitively, we need to remove view(s) to satisfy the storage constraint. But before removing view(s), we check out if the view-spy can steal index space to keep these views. If it cannot steal space, we remove view(s) using one of the view policies (LFUV or SVF).
  2. Only the space occupied by the indices is higher than their required space. As in the previous situation, we need to remove index(es) to satisfy the storage constraint. But before removing these index(es), we check out if the index-spy can steal view space to keep these indices. If it cannot steal space, we remove index(es) using one of the index policies (LFUI or LIF).
  3. Both spaces are higher than their required spaces. In this case, we remove views and indices using the policies for views and indices till their storage constraints are satisfied. We then apply our static algorithm to verify if the global cost can be reduced.

## 6.5 Discussion

Our evaluation of the iterative algorithm for the VISP problem shows that there is a balance between amount of space allocated to materialized views vis-a-vis indices in data warehousing environments. Blindly allocating space to either materialized views or indices will not guarantee faster execution of queries. This level of interaction can be easily supported in any data warehouse design tool, and can facilitate efficient query processing in data warehouses. As the

approach we developed is essentially a greedy algorithm, one cannot guarantee an optimal distribution of space between materialized views and indexes. But, as our illustrative examples show, there is a tremendous reduction in query processing cost which supports the utility of this approach.

## 7 Discussion of Practical Aspects of this Chapter

In this chapter, we have treated two major problems in data warehousing environments: (i) the data partitioning, and (ii) the interaction between indexes and materialized views. Each problem gets an important place in data warehouse trends.

**Data partitioning** Oracle8i incorporates the horizontal partitioning to support very large tables, materialized views and indexes by decomposing them into smaller and manageable pieces called partitions. Oracle 8i defines some partitioning methods: *the range partitioning*, *the hash partitioning*, *the composite partitioning*, and *the partition-Wise Joins*.

In the range partitioning, the data in a table (a view or an index) is partitioned according to a range of values. In the hash partitioning, the data is partitioned according to a hash function. In the composite partitioning, data is partitioned by range and further subdivided using a hash function. Finally, in partition-Wise Joins, a large join operation is broken into smaller joins that are performed sequentially or in parallel. In order to use this partitioning, both tables must be equi-partitioned.

**Indexing Materialized Views** Oracle8i's provide a new query-rewrite capability, which transforms a SQL statement so that it accesses materialized views that are defined on the detail tables. Coupled with materialized views, Oracle8i's query-rewrite capability can significantly reduce the response time of queries that summarize or join data in the tables of a data warehouse. When a query targets one or more detail tables to calculate a summary or an aggregate (or perform a join) and *an available materialized view* contains the requested data, Oracle8i's optimizer can transparently rewrite the query to target the precomputed results in the materialized view, which returns the results more quickly.

But when a query targets one or more detail tables to calculate a summary or an aggregate (or perform a join) and *more than one materialized view* contains the requested data (for example, joining these materialized views), Oracle8i's should define a join index covering these tables to optimize this query. We are not sure if this kind of join indexes are defined by industrials.

**The Interaction between Indexes and Views** This problem gets a great attention from the industrials and practitioners. Recently, the Data Management, Exploration Mining group at Microsoft Research considers the interaction between indexes and views. The researchers at this group [63], present solutions for automatically selecting an appropriate set of materialized views and indexes for SQL databases.

## 8 Conclusions and Future Work

Data warehousing design facilitates efficient query processing and maintenance of materialized views. There are few trade-offs that were introduced in data warehousing design. The first trade-off is whether to materialize a view or not. The view selection problem minimizes the total cost of query processing and maintaining materialized views for a given set of queries. There has been lot of work on selecting materialized views for static environments (such as

[72]), and under dynamic environments (such as, [46]). The second trade-off is whether to partition a data warehouse or not. In our study [12, 30, 31, 29] we found that partitioning helps in reducing irrelevant data access and eliminating (some of the) costly joins. Further, too much of partitioning can increase the cost for queries that access entire data warehouse. The third trade-off is index selection to efficiently execute queries. We found that judicious index selection does reduce the cost of query processing, but also showed that indices on materialized views improve the performance of queries even more [11]. Since indices require storage space, and so do materialized views, the final trade-off presented was the storage distribution among materialized views and indices. We found that it is possible to apply heuristic to distribute the storage space among materialized views and indices so as to efficiently execute queries and maintain materialized views.

Data warehousing design under dynamic environments is still a very open issue of research. The ability to adapt a given set of materialized views and partitions to dynamically changing queries executed over time by users is required to facilitate optimal performance from a data warehousing system. This problem could be addressed by treating materialized views and partitions to be composed of grid cells [55], and dynamically changing the materialized views and partitions by changing the composition. Such a methodology is amenable to heuristics based on composition algebra, and efficient hill-climbing algorithms can be designed. The notion of atomic grid cells can facilitate parallel processing of costly OLAP queries on a shared-nothing parallel processing system. Even though lot of work has been done on view maintenance algorithms and view adaptation algorithms, there is explicit tie-up between these and data warehousing design. In particular, given a particular view maintenance strategy what kind of data warehousing design strategy has to be employed and vice-versa. Data warehousing design is application specific, for report generation a specific data warehousing design might be needed, where as for data mining a different data warehousing design might be needed. Hence a design methodology that can support multiple data warehouse designs for different types of applications needs to be developed.

## References

- [1] *Oracle 7 Server Concepts Manual*. Oracle Corporation, Redwood City, CA, 1992.
- [2] *Oracle8i Concepts*. Oracle Corporation, Release 8.1.5, 1999.
- [3] A. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. Technical report research, IBM, 1997.
- [4] E. Baralis, S. Ceri, and S. Paraboschi. Conservative timestamp revisited for materialized view maintenance in a data warehouse. *in Proceeding of the Workshop on Materialized Views: Techniques and Applications (VIEW'1996)*, pages 1–9, June 1996.
- [5] E. Baralis, S. Paraboschi, and E Teniente. Materialized view selection in a multidimensional database. *Proceedings of the International Conference on Very Large Databases*, pages 156–165, August 1997.
- [6] L. Bauer and W. Lehner. The cube-query-languages (cql) for multidimensional statistical and scientific database systems. *in Proceedings of the Fifth International Conference on Database Systems for Advanced Applications(DASFAA '97)*, pages 263–272, April 1997.
- [7] L. Bellatreche, K. Karlapalem, and Simonet A. Algorithms and support for horizontal class partitioning in object-oriented databases. *in the Distributed and Parallel Databases Journal*, 8(2):155–179, April 2000.

- [8] L. Bellatreche, K. Karlapalem, and G. B. Basak. Query-driven horizontal class partitioning in object-oriented databases. *in 9th International Conference on Database and Expert Systems Applications (DEXA'98), Lecture Notes in Computer Science 1460*, pages 692–701, August 1998.
- [9] L. Bellatreche, K. Karlapalem, and Q. Li. Derived horizontal class partitioning in oodbss: Design strategy, analytical model and evaluation. *in the 17th International Conference on the Entity Relationship Approach (ER'98)*, pages 465–479, November 1998.
- [10] L. Bellatreche, K. Karlapalem, and Q. Li. Algorithms for graph join index problem in data warehousing environments. Technical Report HKUST-CS99-07, Hong Kong University of Science & Technology, March 1999.
- [11] L. Bellatreche, K. Karlapalem, and Q. Li. Evaluation of indexing materialized views in data warehousing environments. *Proceeding of the International Conference on Data Warehousing and Knowledge Discovery (DAWAK'2000)*, pages 57–66, September 2000.
- [12] L. Bellatreche, K. Karlapalem, and M. Mohania. What can partitioning do for your data warehouses and data marts. *Accepted in IDEAS'2000*, 2000.
- [13] L. Bellatreche, K. Karlapalem, and M. Schneider. Interaction between index and view selection in data warehousing environments. *Submitted to Information Systems Journal*, 2000.
- [14] L. Bellatreche, K. Karlapalem, and M. Schneider. On efficient storage space distribution among materialized views and indices in data warehousing environments. *To appear in the International Conference on Information and Knowledge Management (CIKM'2000)*, November 2000.
- [15] L. Bellatreche, K. Karlapalem, and A. Simonet. Horizontal class partitioning in object-oriented databases. *in 8th International Conference on Database and Expert Systems Applications (DEXA'97), Toulouse, Lecture Notes in Computer Science 1308*, pages 58–67, September 1997.
- [16] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. *Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGPLAN Notices*, pages 128–136, 1982.
- [17] S. Ceri and G. Pelagatti. *Distributed Databases: Principles & Systems*. McGraw-Hill International Editions, 1984.
- [18] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. *Proceedings of the International Conference on Very Large Databases*, pages 146–155, August 1997.
- [19] S. Chaudhuri and V. Narasayya. Autoadmin 'what-if' index analysis utility. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–378, June 1998.
- [20] S. Chaudhuri and V. Narasayya. Index merging. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 296–303, March 1999.
- [21] Oracle Corp. Oracle8i<sup>TM</sup> enterprise edition partitioning option. Technical report, Oracle Corporation, February 1999.

- [22] Informix Corporation. Informix-online extended parallel server and informix-universal server: A new generation of decision-support indexing for enterprise data warehouses. *White Paper*, 1997.
- [23] Simpson D. Build your warehouse on mpp. Available at <http://www.datamation.com/serve/12mpp.html>, December 1996.
- [24] A. Datta, B. Moon, K. Ramamritham, H. Thomas, and I. Vigiuer. Have your data and index it, too: Efficient storage and indexing for datawarehouses. Techreport Technical Report 98-7, Department of Computer Science, The University of Arizona, 1998.
- [25] A. Datta, K. Ramamritham, and H. Thomas. Curio: A novel solution for efficient storage and indexing in data warehouses. *Proceedings of the International Conference on Very Large Databases*, pages 730–733, September 1999.
- [26] D. Dewitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. *VLDB*, 10:228–237, 1986.
- [27] C. I. Ezeife and K. Barker. A comprehensive approach to horizontal class fragmentation in distributed object based system. *International Journal of Distributed and Parallel Databases*, 3(3):247–272, 1995.
- [28] J. M. Firestone. Data warehouses and data marts: A dynamic view. White Paper 3, Executive Information Systems, Inc., March 1997.
- [29] V Gopalkrishnan, Q. Li, and K. Karlapalem. Star/snow-flake schema driven object-relational data warehouse- design and query processing strategies. in *Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery*, pages 11–22, 1999.
- [30] V Gopalkrishnan, Q; Li, and K. Karlapalem. Efficient query processing with associated horizontal class partitioning in an object relational data warehousing environment. in *Proceedings of 2nd International Workshop on Design and Management of Data Warehouses*, pages 1–9, June 2000.
- [31] V Gopalkrishnan, Q; Li, and K. Karlapalem. Efficient query processing with structural join indexing in an object-relational data warehousing environment. in *Information Resources Management Association International Conference*, pages 976–979, 2000.
- [32] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin*, 18(2):3–18, June 1995.
- [33] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, June 1993.
- [34] H. Gupta. Selection and maintenance of views in a data warehouse. Phd. thesis, Stanford University, September 1999.
- [35] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for olap. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 208–219, April 1997.

- [36] A. R. Hevner and A. Rao. Distributed data allocation strategies. *Advances in Computers*, 12:121–155, 1988.
- [37] R. Hull and G. Zhou. A framework for supporting data integration using materialized and virtual approaches. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 481–492, 1996.
- [38] C. A. Hurtado, O. A. Mendelzon, and Vaisman A. A. Maintaining data cubes under dimension updates. *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 346–355, March 1999.
- [39] N. Huyn. Multiple-view self-maintenance in data warehousing environments. *Proceedings of the International Conference on Very Large Databases*, pages 26–35, August 1997.
- [40] Hyperion. *Hyperion Essbase OLAP Server*. <http://www.hyperion.com/>.
- [41] Informix Inc. *The Informix-MetaCube Product Suite*. <http://www.informix.com>, 1997.
- [42] W. H. Inmon. *Building the Data Warehouse*. John Wiley, 1992.
- [43] R. Jain. *The Art of Computer Systems Performance Analysis*. Willy Professional Computing, 1991.
- [44] K. Karlapalem, S.B. Navathe, and M. M. A. Morsi. Issues in distributed design of object-oriented databases. In *Distributed Object Management*, pages 148–165. Morgan Kaufman Publishers Inc., 1994.
- [45] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, 1996.
- [46] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 371–382, June 1999.
- [47] W. Labio, D. Quass, and B. Adelberg. Physical database design for data warehouses. *Proceedings of the International Conference on Data Engineering (ICDE)*, 1997.
- [48] W. Lehner, T. Ruf, and M. Teschke. Cross-db: A feature-extended multidimensional data model for statistical and scientific databases. *in the 5th International Conference on Information and Knowledge Management (CIKM'96)*, pages 253–260, 1996.
- [49] H. Lei and K. A. Ross. Faster joins, self-joins and multi-way joins using join indices. *Data and Knowledge Engineering*, 28(3):277–298, November 1998.
- [50] C. Li and W. S. Wang. A data model for supporting on-line analytical processing. *in the 5th International Conference on Information and Knowledge Management (CIKM'96)*, pages 81–88, 1996.
- [51] M. Mohania and Y. Kambayashi. Making aggregate views self-maintainable. *Data and Knowledge Engineering*, 32(1):87–109, January 2000.
- [52] M. Mohania, S. Samtani, J. F. Roddick, and Y. Kambayashi. Advances and research directions in data warehousing technology. *To appear in the Australian Journal of Information Systems*.
- [53] S. Morse and D. Isaac. *Parallel Systems in the Data Warehouse*. Prentice Hall PTR, 1998.

- [54] S.B. Navathe, S. Ceri, G. Wiederhold, and Dou J. Vertical partitioning algorithms for database design. *ACM Transaction on Database Systems*, 9(4):681–710, December 1984.
- [55] S.B. Navathe, K. Karlapalem, and M. Ra. A mixed partitioning methodology for distributed database design. *Journal of Computer and Software Engineering*, 3(4):395–426, 1995.
- [56] S.B. Navathe and M. Ra. Vertical partitioning for database design : a graphical algorithm. *ACM SIGMOD*, pages 440–450, 1989.
- [57] A. Y. Noaman and K. Barker. A horizontal fragmentation algorithm for the fact relation in a distributed data warehouse. *in the 8th International Conference on Information and Knowledge Management (CIKM'99)*, pages 154–161, November 1999.
- [58] P. O’Neil and D. Quass. Improved query performance with variant indexes. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 38–49, May 1997.
- [59] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.
- [60] TPC Home Page. Tpc benchmark<sup>TM</sup>d (decision support). <http://www.tpc.org>.
- [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, December 1996.
- [62] D. Quass and J. Widom. On-line warehouse view maintenance for batch updates. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–404, May 1997.
- [63] A. Sanjay, C. Surajit, and V. R. Narasayya. Automated selection of materialized views and indexes in microsoft sql serve. *To appear in VLDB'2000*, September 2000.
- [64] A. Segev and J. L. Zhao. A framework for join pattern indexing in intelligent database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):941–947, December 1995.
- [65] D. Srivastava, S. Dar, H. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. *Proceedings of the International Conference on Very Large Databases*, pages 318–329, 1996.
- [66] Red Brick Systems. Star schema processing for complex queries. *White Paper*, July 1997.
- [67] D. Tsichritzis and A. Klug. The ansi/x3/sparc framework. *AFIPS Press, Montval, N.J.*, 1978.
- [68] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.
- [69] P. Vassiliadis and T Sellis. A survey of logical models for olap databases. *SIGMOD Record*, 28(4):64–69, December 1999.
- [70] J. Widom. Research problems in data warehousing. *in the 4th International Conference on Information and Knowledge Management (CIKM'95)*, pages 25–30, 1995.

- [71] M-C. Wu and A. Buchmann. Research issues in data warehousing. *in Datenbanksysteme in Büro, Technik und Wissenschaft(BTW'97)*, pages 61–82, March 1997.
- [72] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. *Proceedings of the International Conference on Very Large Databases*, pages 136–145, August 1997.
- [73] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, 1995.
- [74] Y. Zhuge, H. Garcia-Molina, and J. Widom. The strobe algorithms for multi-source warehousing consistency. *in Proceeding of the Fourth International Conference on Parallel and Distributed Information Systems*, pages 146–157, December 1996.