PHYSICAL DATABASE DESIGN FOR DATA WAREHOUSES

G. Velinov[1], M. Kon-Popovska[2]

[1]Department of IT Development, Mobimak AD – Skopje

[2]Institute of Informatics, Faculty of Natural Sciences and Mathematics

Sts. Cyril and Methodius University

Arhimedova bb, P.O.Box 162, Skopje, Macedonia

goran@mobimak.com.mk; margita@ii.edu.mk

Abstract: Data warehousing has quickly evolved into a unique and popular business application class. A data warehouse stores materialized views of data from one or more remote sources, with the purpose of efficiently implementing decision support or OLAP (On-Line Analytical Processing) queries. One of more important decisions in designing data warehouse is the selection of materialized views to be maintained at the warehouse.

The first part of this paper briefly introduces key concepts surrounding data warehousing system. The second part studies how to select the set of supporting views and indexes to materialize to minimize the total query response time and cost of maintaining the selected views.

Keywords: data warehouses, physical database design, data cube, view selection

1. Introduction

Decision Support Systems (DSS) are computer-based systems that are used to assist human decision makers in solving semi-structured and unstructured problems. However, the major problem with developing effective decision support systems is the availability of data, which can be represented in a form that is easily understood by the decision maker. Relational databases are based on the relational model and are the *de facto* standard for storing and retrieving data and are widely used in most organizations. However, relational databases are optimized for transaction processing and are extremely efficient in storing and updating the data. Most applications of relational databases have aimed at facilitating or meeting the requirements for transaction processing, operational control, or management control. Therefore the major concerns of these systems have been at the lower levels of the data, dealing primarily with raw data.

The relational model is quite inadequate for analyzing data. Most business managers need to analyze information on-line and in a dynamic fashion. In a recent paper states that the relational database systems were never intended to provide powerful functions for data synthesis, analysis, and consolidation.

Data warehousing is a concept that is gaining increasing popularity because of the need for quality data. Data warehousing is the creation of a specialized database to help support decision-making. Because of the problems associated with relational databases, the solution now is to create a specialized database, which extracts data from the conventional databases and stores them in a single large server or database. This data store or warehouse typically consists of enterprise data from diverse production systems and is an approximation of the entire enterprise data.

The data warehouse is usually updated by batch processing on a periodic basis, and is therefore a more static or historical model of the enterprise data. Once the data warehouse is populated, the data is essentially static and does not change till the next time the warehouse is updated. The data warehouse is created to satisfy the needs of decision makers and has a different structure and representation that is more intuitive and responsive to managerial queries.

The data warehouse is usually developed as a central data store, which provides data in the format that is understandable by the users. The data warehouse provides a traditional, highly manageable data center for DSS, ensuring that the data is readily available and quickly accessible by the users. The data warehouse is not just a copy of the data in other systems. It is a unique, enriched data set that is optimized for decision support.

The data 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 On-Line Analytical Processing (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.

One approach to provide this unique and enriched view of the data is to use multidimensional databases (MDDB). Multidimensional databases are specialized databases designed to facilitate multidimensional data analysis and decision support. MDDB-s store numeric or quantitative data, which is categorized over several qualitative dimensions. For example, a MDDB may store sales data (quantitative) for several product lines, in several different cities, for each month (3 qualitative dimensions). The user will then be able to retrieve data on any specific combination of these dimensions as well as perform aggregations and consolidations. For example, a manager might be interested in obtaining the sales figures for Product X in City Y for Dec. 2001 or obtain the Total Sales for Product X in City Y for 2001.

A MDDB has a database management system (MDDBMS), which provides the user with the capability to analyze this data by providing tools for flexible, ad hoc data analysis. This end user oriented data analysis system provides users with the capability for sophisticated data analysis without requiring programming language knowledge or support form the Information Systems (IS) personnel. This system insulates the user from having to master the intricacies of data storage and access mechanisms.

The warehouse data is typically modeled multidimensional. The multidimensional data model 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.

2. Data Models for a Data Warehouse

The data models for designing traditional OLTP (On-Line Transaction Processing) 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 multidimensional. 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.

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) or (2) by making use of specialized multidimensional databases, called MOLAP architecture (Multidimensional On-Line Analytical Processing). 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. 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. 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 1.

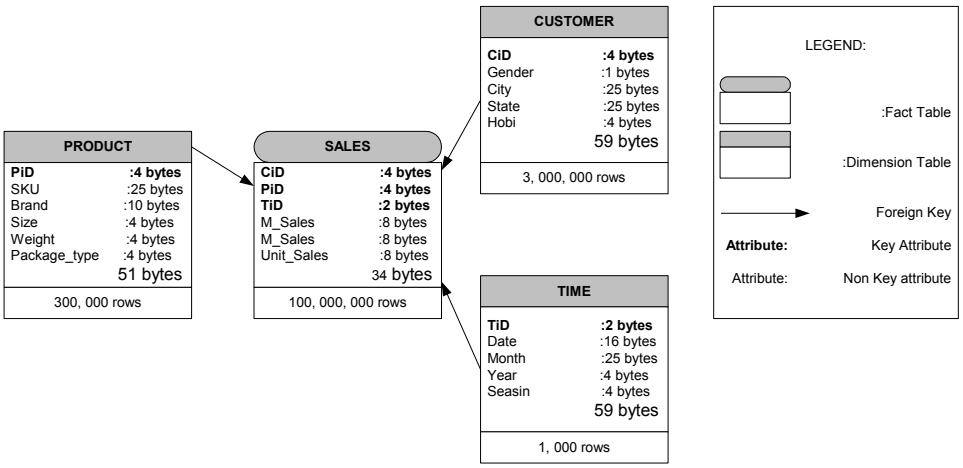

Figure 1: An Example of a Star Schema

Star schema schema is used to support multidimensional data epresentation. 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 1, 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 2 shows an example of snowflake schema after TIME dimension table in Figure 1 has been normalized.

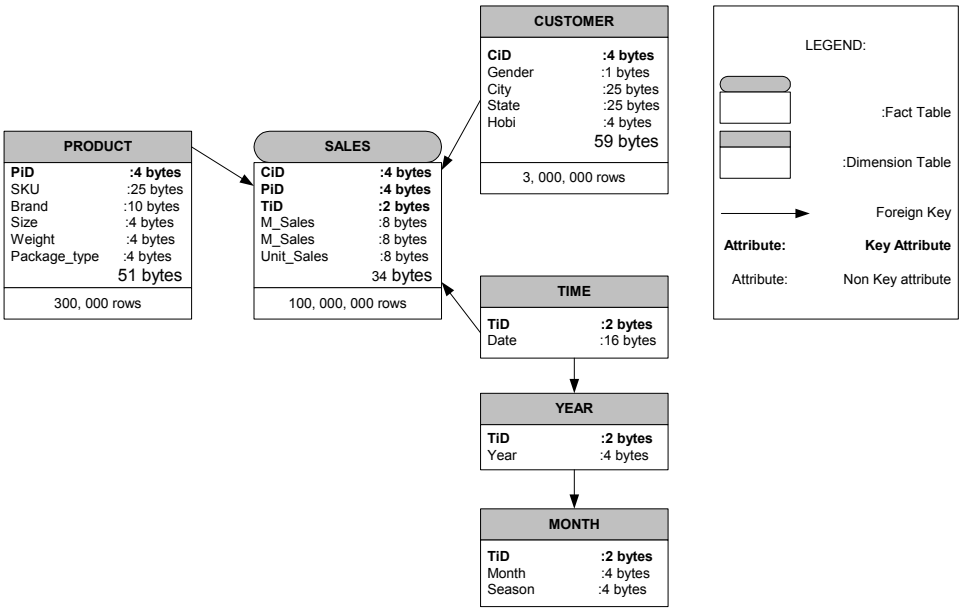

Figure 2: Snowflake Schema Example

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:

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

2.1.2 n-dimensional 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. 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 f lattening 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 intercube 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 for-

malized 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 extension to the existing query language 6 is to support multiple `group by' by defining roll-up, drill-down, and cube operators.

Roll Up: Refers to the process of moving up a dimension hierarchy to obtain more aggregated views of the data. This process is also called roll up, as it refers to the movement from lower to higher levels in the hierarchy. For example, a simple roll up involves consolidating Cities into States, and States into Regions.

Drill-Down: This operation refers to the process of drilling down to obtain detail data. Drill down is useful when analyzing a cause or effect for some observed phenomena in the aggregate data.

Slice and Dice: This is the ability to look at the database from different viewpoints. For example, the analyst may wish to view Sales data dimensioned by Product and Region only with Period values being consolidated.

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. The view-index selection problem

Data warehouses collect information from many sources into a single database. This allows users to pose queries within a single environment and without concern for schema integration. Figure 3 shows a typical warehousing system. Relations R_{src}, S_{src}, and T_{src}, referred to as source relations, from sources 1, 2, and 3 respectively, are replicated at the warehouse as R, S, and T in order to answer

user queries posed at the warehouse such as R▷◁S▷◁T. We refer to the replicated relations R, S, and T as warehouse relations. Consistency between the source relations and the warehouse relations is usually only loosely maintained: Changes to the source relations are queued and periodically shipped to the warehouse where they are applied to the warehouse relations. We call these changes deltas.

Queries posed at a data warehouse are often complex - involving joins of multiple relations as well as aggregation. Due to the complexity of these queries, views are usually defined is a derived relation expressed in terms of the warehouse relations. Because the views are defined in terms of the warehouse relations, we refer to the warehouse relations also as base relations in this paper. For example, referring again to Figure 3, RST represents a view that is the expression R▷◁S▷◁T. Warehouses can store huge amounts of data, and so in order to improve the performance of queries written in terms of the views, the views are often materialized by storing the result of the view at the warehouse. Unmaterialised views are called virtual views. Queries written in terms of materialized views can be significantly faster than queries written in terms of virtual views because the view tuples are stored rather than having to be recomputed.

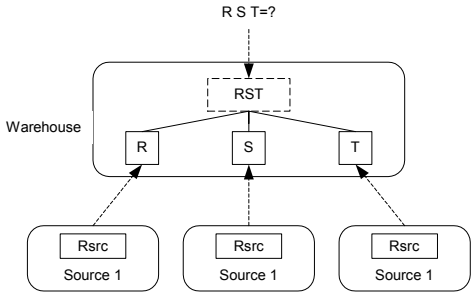

Figure 3: Warehouse with primary view

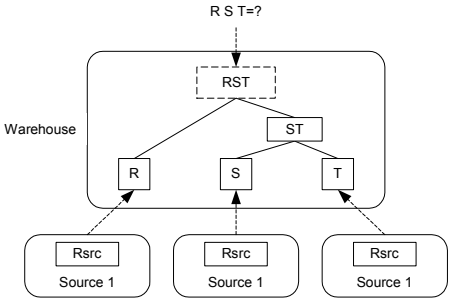

Figure 4: Warehouse with supporting view

Since materialized views are computed once and then stored, they become inconsistent as the deltas from the sources are applied to the base relations. In order to make a materialized view consistent again with the base relations from which it is derived, the view may either be recomputed from scratch, or incrementally maintained by calculating just the effects of the deltas on the view. These effects are captured in view maintenance expressions. Each type of change (insertion, deletion, or update) requires a different expression. For example, if view RST in Figure 3 is materialized, the maintenance expression calculating the tuples to insert into RST due to insertions into R is △R▷◁S▷◁T , where △R denotes the insertions into R.

Since the sizes of the views at a warehouse are usually so large and the changes small in comparison, it is often much cheaper to incrementally maintain the view than to recompute it from scratch. Incrementally maintaining a number of materialized views at a warehouse, even though cheaper than recomputing the views from scratch, may still involve a significant processing effort. To avoid impacting clients querying the warehouse views, view maintenance is usually performed at night during which time the warehouse is made unavailable for answering queries. A major concern for warehouses using this approach is that the views be maintained in time for the warehouse to be available for querying again the next morning. An important problem for data warehousing is thus: Given a set of materialized views that need to be maintained due to a set of deltas shipped from the data sources, how is it possible to reduce the total maintenance time?

One approach to the problem of minimizing the time spent maintaining a set of views may seem counter-intuitive at first: add additional structures to be maintained. However, this is analogous to building indexes in traditional RDBMS's. For example, having an index on the key of a relation can vastly decrease the total time spent locating and deleting a particular tuple even though the index must be maintained as well. In this paper we will approximate maintenance time as the number of I/O's required and then endeavor to minimize the number of I/O's performed. We will do this by adding a set of additional views and indexes that themselves must be maintained, but whose benefit (reduction in I/O's) outweighs the cost (increase of I/O's) of maintaining them.

As an example, let us return to Figure 3. Suppose that in addition to materializing the primary view, RST , another view, ST , is also materialized. This situation is shown in Figure 4. By materializing view ST , the total cost of maintaining both RST and ST can be less than the cost of maintaining RST alone. For example, suppose that there are insertions to R but no changes whatsoever to S and T . To propagate the insertions to R onto RST , we must evaluate the maintenance expression that calculates the tuples to insert into RST due to insertions into R, which is $\triangle R \triangleright\triangleleft S \triangleright\triangleleft T$. With ST materialized, it is almost certain that this expression can be evaluated more efficiently as $\triangle R \triangleright\triangleleft ST$, joining the insertions to R with ST , instead of with S and T individually. Even if there are changes to S and T , the benefit of materializing ST may still outweigh the extra cost involved in maintaining it. Since the view ST is materialized to assist in the maintenance of the primary view RST , we call the view ST a supporting view.

In addition to materializing supporting views, it may also be beneficial to materialize indexes. Indexes may be built on the base relations, primary views, and on the supporting views. The general problem, then, is to choose a set of supporting views and a set of indexes to materialize such that the total maintenance cost for the warehouse is minimized. This is the View-Index Selection (VIS) problem.

Section 4 describes the VIS problem in detail. It also introduces the exhaustive search algorithm.

4. General Problem

Having introduced the VIS problem, in this section we will describe it fully and develop an exhaustive algorithm to obtain the optimal solution. The exhaustive algorithm is then decomposed to show the complexity of the VIS problem. Lastly, we present an example schema to illustrate the concepts introduced.

4.1 The Optimization Problem

In developing an optimal algorithm, we must minimize the total cost of maintaining the warehouse. The cost that we attempt to minimize is the sum of the costs of: (1) applying the deltas to the base relations, (2) evaluating the maintenance expressions for the materialized views, and (3) modifying affected indexes. The cost of maintaining one view differs depending upon what other views are available. It is therefore incorrect to calculate the cost of maintaining the original view and each of the additional views in isolation. Moreover, in order to derive the cost for maintaining a set of views it is necessary to consider the view selection and index selection together. If view selection is performed separately from index selection, it is not hard to imagine cases wherein a supporting view V is considered to be too expensive to maintain without indexes, but where V is actually part of the optimal solution since it may become feasible to maintain when the proper indexes are built. To find the optimal solution, then, we must exhaustively search the solution space. Although exhaustive search is impractical for large problems, it illustrates the complexity of the problem and provides a basis of comparison for heuristics solutions. The exhaustive algorithm works as follows (each stage is described below):

for each possible subset of supporting views

> *for* each possible subset of indexes on the views and base relations
>
> *compute* total update cost with views and indexes materialized and *keep track* of the supporting views and indexes that obtain the minimum cost

4.1.1 Choosing the views

In the first step we consider all possible subsets of the set of candidate views C. We consider as candidate views all distinct nodes that appear in a query plan for the primary view. Since the primary view is already materialized, it is not included in the candidate view set. For example, given a view $V = R \triangleright\triangleleft S \triangleright\triangleleft T$, C = {RS; RT ; ST}. In general, for a view joining n relations there are roughly $O(2^n)$ different nodes that appear in some query plan for the view, one joining each

possible subset of the base relations. Thus, to consider all possible subsets of C, we need to evaluate roughly $O(2^{2^n})$ different view states.

4.1.2 Choosing the indexes

Now we must consider all possible subsets of the set of candidate indexes, I. Candidate indexes are indexes on the following types of attributes:

- attributes with selection or join predicates on them
- key attributes for base relations where changes to the base relation include deletions or updates. When views are materialized on the base relations, key attributes of any base relation appearing in the view also qualify.
- attributes in GROUP BY or ORDER BY clauses.

Additional attributes can be candidates depending on the query optimizer being used. The reader is referred to for more detail.

Since each materialized view will usually have candidate indexes, I must be recomputed at the beginning of every inner loop. The cardinality of I for a particular view state is roughly proportional to the number of materialized views and base relations in that state. Further, a particular view state contains between n and $O(2^n)$ materialized views and base relations, so there can be as many as $O(2^n)$ candidate indexes to consider. Since we must evaluate possible subsets of candidate indexes, the number of possible index states for a view state can be up to $O(2^{2^n})$.

4.1.3 Computing the total update cost

Once a particular view and index state are chosen, the cost of maintaining the set of views is a query optimization problem since it involves finding the most efficient query plan for each of the view maintenance expressions. Thus, the VIS problem for a single primary view joining n base relations contains roughly $O(2^{2^n})$ query optimization problems in the most general case. The query optimization itself is complicated by the presence of materialized views since the optimizer must also determine if it can use another materialized view in the query plan evaluating a maintenance expression. For example, given a view $V = R \triangleright\triangleleft S \triangleright\triangleleft T$, insertions to R are propagated onto V by the maintenance expression $\triangle R \triangleright\triangleleft S \triangleright\triangleleft T$. Suppose the view $ST = S \triangleright\triangleleft T$ is also materialized. The query optimization algorithm must consider the possibility of evaluating $\triangle R \triangleright\triangleleft S \triangleright\triangleleft T$ as $4R \triangleright\triangleleft ST$ in finding the best query plan. This problem is known as ``answering queries using views".

To further complicate matters, one batch of changes can generate multiple maintenance expressions that need to be evaluated. This happens due to different

types of changes insertions, deletions, and updates) to the base relations. The maintenance expressions can be optimized as a group because of possible common subexpressions. This problem is known as the ``multiple-query optimization'' problem.

4.1.4 Cost Model

In this section we give some formulas for deriving the overall cost of maintaining a set of views due to changes to the warehouse relations. The formulas are based upon cost models for queries and updates appearing elsewhere. The formulas represent a fairly accurate and detailed cost model, upon which we based our implementation of an algorithm that used exhaustive search to find the optimal set of supporting views and indexes for a given primary view.

The main formula given in this section is $Cost_v(V)$, which is the cost of maintaining a set of views V. The other formulas are used to support $Cost_v(V)$. Note that much of the statistical information for views can be derived from statistical information for the warehouse relations and the selectivities of local and join selection conditions. Table 1 gives our formula for $Cost_v(V)$ and its supporting formulas. Note that *Eval(expr)* is the traditional query optimization cost function. In the formulas we use $\triangle$R, ∇R, and μR to represent the set of insertions, deletions, and updates to R respectively.

Two more formulas need to be explained:

$$yao(n,p,k) = \begin{cases} k, k < p/2 \\ (k+p)/3, p/2 < k <= 2p \\ p, 2p < k \end{cases}$$

The *yao* function returns an estimate of the number of page read operations given that k out of n tuples are read from a relation spanning p pages. The *yao* function assumes that either the memory buffer is large enough to hold the entire relation, or that the tuple accesses have been sorted beforehand so that tuples from the same page will be requested one after the other. Since the assumption that a relation fits entirely in memory is unrealistic for a data warehouse and we assume that tuple accesses are not usually sorted beforehand, our formulas often make use of a function Y_{WAP} presented in for estimating the number of page read operations given k tuple fetches and a memory buffer of m pages.

$$Y_{WAP}\ (n,\ p,\ k) = \begin{cases} \min(k,p), p <= m \\ k, p > m, k <= m \\ m + (k-m)(p-m)/p, p > m, k > m \end{cases}$$

Name	Formula	code
$Cost_v(V)$	$\sum_{V \in V} Cost_v(V)$	1
$Cost_v(V)$	$\sum_{R \in R(V)} (Prop_{ins}(R,V) + Prop_{del}(R,V) + Prop_{upd}(R,V)$	2
$Prop_{ins}(R,V)$	$Eval(\Delta R_1 \triangleright\triangleleft R_2 \triangleright\triangleleft ... \triangleright\triangleleft R_k \rightarrow \Delta V_R) + Applay_{ins}(\Delta V_R, V) + Applay_{ins}(\Delta V_R, \Delta V_R^{save}) + ApplayIx(\Delta V_R, V)$	3
$Prop_{del}(R,V)$	$Eval(V \triangleright\triangleleft_{keyofR} \nabla R \rightarrow \Delta V_R) + Applay_{idelupd}(\nabla V_R, V) + ApplayIx(\nabla V_R, V)$	4
$Applay_{ins}(R,V)$	P(R)	5
$Applay_{delupd}(R,V)$	yao(T(V), P(V), T(R))	6
ApplayIx(R,V)	$\sum_{R.A \in indexesV} (Y_{WAP}(T(V),P(V,R.A),T(R)*(H(V,R.A)-1) + Y_{WAP}(T(V),P(V,R.A),T(R)))$	7

Table 1a: Cost Formulas

code	Description
1	Derive cost to maintain a set of views by summing cost to maintain each view.
2	Sum the cost of propagate changes to each relation into V
3	Evaluate effect on V of $\triangle R$, which we call $\triangle V_R$, where $\{R_1, R_{2,\ldots,} R_k\}=R(V)$ Insert $\triangle V_R$ into V. Save it for possible reuse as ΔV_R^{save} (small cost anyway). Update indexes on V
4	Evaluate effect on V of $\triangledown R$, which we call $\triangledown VR$. Delete $\triangledown VR$ from V. Update indexes on V
5	Append tuples in R to V
6	Delete or update tuples of R in V($R \subseteq V$). Exact locations of tuples of R in V are derived when R is derived. If index join is used to derive R instead of nested-block join, then use $Y_{WAP}(T(V),P(V),T(R),P_m)$ instead of yao(T(V),P(V),T(R)).
7	For each index on V, sum approximate number of index pages to read assuming root cached, plus approximate number of index pages to write (leaves only).

Table 1b: Cost Formulas

4.2 Example

Consider the following base relations and view.

R(R0,R1), S(S0,S1), T(T0,T1)

create view V(R0,S1,T0,T1) as

select R.R0, S.S1, T.T0, T.T1

from R, S, T

where R.R1 = S.S1 and S.S0 = T.T0 and T.T1 != 10

T' is the result of applying the selection condition to T . Under each view node is the set of operations (join or select) that could be used to derive the view. For example, the view RST could be derived as the result of R$\triangleright\triangleleft$S joined with T' , or the result of R$\triangleright\triangleleft$S joined with the result of S$\triangleright\triangleleft$T' , and so on. Each of the intermediate results could be materialized as a supporting view. Following the definition in Section 4.1.1, the set of candidate supporting views, C, is {RS; ST' ; RT' ; T'}. Assuming V is materialized at a data warehouse (and the base relations R, S, and T are also materialized at the warehouse), any possible subset of C might also be materialized as supporting views at the warehouse in order to minimize the total maintenance cost. In addition, indexes on V, the base relations, and the supporting views need to be considered.

It is useful to think of the expression dag when considering the different update paths changes to base relations can take as they are propagated to the view. An update path corresponds to a specific query plan for evaluating a view maintenance expression. For example, the maintenance expression for propagating insertions to R onto V is to insert the result of into V. There are seven possible update paths for this expression, two of which are: (1) ($\triangle$R$\triangleright\triangleleft$S) $\triangleright\triangleleft$T', (2) $\triangle$(R$\triangleright\triangleleft$S)1(S$\triangleright\triangleleft$T'), and so on. Notice that the choice of update path can affect which indexes get materialized. If update path (1) is chosen, an index may be built on the join attribute of T' to help compute the maintenance expression. If path (2) is chosen however and view ST' is materialized, an index may be built on the join attribute of ST'.

Changes to base relations need to be propagated both to the primary view as well as to the supporting views that have been materialized. When propagating changes to several base relations onto several materialized views there are opportunities for multiple-query optimization. Results of maintenance expressions for one view can be reused when evaluating maintenance expressions for another view. For example, suppose view RS = R$\triangleright\triangleleft$S is materialized. The result of propagating insertions to R onto RS, $\triangle$R$\triangleright\triangleleft$S, can be reused when propagating insertions to R onto V , $\triangle$R$\triangleright\triangleleft$S$\triangleright\triangleleft$T0 , so that only the join with T' need be performed. In addition, common subexpressions can be detected between several maintenance expressions. For example, when propagating insertions to both R and S onto V, it may be cheaper to evaluate the join with T' once, as in ((4R$\triangleright\triangleleft$S)[($R^n\triangleright\triangleleft$4S)) $\triangleright\triangleleft$T', than to evaluate both $\triangle$R$\triangleright\triangleleft$S$\triangleright\triangleleft$T' and $R^n\triangleright\triangleleft$4S$\triangleright\triangleleft$T' individually. (We use R^n to denote R$\cup\triangle$R, the state of R after $\triangle$R has been applied.)

5. Conclusion

In this paper, we illustrated the architecture of a multidimensional database system and described a data model to represent multidimensional data. The concept of data warehousing is gaining popularity as the means of providing efficient decision support. Multidimensional databases are ideally suited for use as the underlying database for developing data warehouses for decision support. This proposed model will be used to represent and develop multidimensional databases, and will provide an abstraction tool for designing these databases. Also, we considered the VIS problem, which is one aspect of choosing good physical designs for relational databases used as data warehouses. We described and implemented an algorithm to exhaustively search the space of possible views and indexes to materialize. Given that exhaustive search is impractical for many real world problems, we plan to develop heuristics for pruning the exhaustive search space so that good solutions can be found through limited search.

6. References

1. H. Gupta. Selection of Views to Materialize in Data Warehouse. In Proceedings of the International Conference on Database Theory, Athens, Greece, January 1997.
2. H. Gupta, I. Mumick. Selection of Views to Materialize Under a Maintenance Cost Constraint. In Database Theory - ICDT '99, 7th International Conference.
3. L. Bellatreche, K. Karlapalem. Logical and Physical Design in Data Warehousin Enviroment. Department of Computer Science University of Science Technology Clear Water Bay Kowloon Hong Kong.
4. T.B. Pedersen, C.S. Jensen C.E. Dyreson. A fundation for capturing and querying complex multidimensional data. Information Systems 26 (2001).
5. L.J. Labio, D. Quass, B. Adelberg. Physical Database Design for Data Warehouses. Paper Number 1138
6. V.R. Gupta. An Introduction to Data Warehousing
7. L. Bellatreche, K. Karlapalem, M. Mohania. Some Issues in Design of Data Warehousing Systems. Department of Computer Science University of Science Technology Clear Water Bay Kowloon Hong Kong.
8. A. Gupta, I.S. Mumick, J. Rao, K.A. Ross. Adapting materialized views after redefinitions: techniques and a performance study. Information Systems 26 (2001).