Dynamic Data Warehouse Design *

Dimitri Theodoratos Timos Sellis

Department of Electrical and Computer Engineering
Computer Science Division
National Technical University of Athens
Zographou 157 73, Athens, Greece
{dth,timos}@dblab.ece.ntua.gr

Abstract. A data warehouse (DW) can be seen as a set of materialized views defined over remote base relations. When a query is posed, it is evaluated locally, using the materialized views, without accessing the original information sources. The DWs are dynamic entities that evolve continuously over time. As time passes, new queries need to be answered by them. Some of these queries can be answered using exclusively the materialized views. In general though new views need to be added to the DW.
In this paper we investigate the problem of incrementally designing a DW when new queries need to be answered and extra space is allocated for view materialization. Based on an AND/OR dag representation of multiple queries, we model the problem as a state space search problem. We design incremental algorithms for selecting a set of new views to additionally materialize in the DW that fits in the extra space, allows a complete rewriting of the new queries over the materialized views and minimizes the combined new query evaluation and new view maintenance cost.

1 Introduction

Data warehouses store large volumes of data which are frequently used by companies for On-Line Analytical Processing (OLAP) and Decision Support System (DSS) applications. Data warehousing is also an approach for integrating data from multiple, possibly very large, distributed, heterogeneous databases and other information sources.

A Data Warehouse (DW) can be abstractly seen as a set of materialized views defined over a set of (remote) base relations. OLAP and DSS applications make heavy use of complex grouping/aggregation queries. In order to ensure high query performance, the queries are evaluated locally at the DW, using exclusively the materialized views, without accessing the original base relations.

When the base relations change, the materialized at the DW views need to be updated. Different maintenance policies (deferred or immediate) and maintenance strategies (incremental or rematerialization) can be applied.

* Research supported by the European Commission under the ESPRIT Program LTR project "DWQ: Foundations of Data Warehouse Quality"

1.1 The problem: Dynamic Data Warehouse Design

DWs are dynamic entities that evolve continuously over time. As time passes, new queries need to be answered by them. Some of the new queries can be answered by the views already materialized in the DW. Other new queries, in order to be answered by the DW, necessitate the materialization of new views. In any case, in order for a query to be answerable by the DW, there must exist a complete rewriting [5] of it over the (old and new) materialized views. Such a rewriting can be exclusively over the old views, or exclusively over the new views, or partially over the new and partially over the old views. If new views need to be materialized, extra space need to be allocated for materialization.

One way for dealing with this issue is to re-implement the DW from scratch for the old and the new queries. This is the *static* approach to the DW design problem. Re-implementing the DW from scratch though, has the following disadvantages:

(a) Selecting the appropriate set of views for materialization that satisfies the conditions mentioned above is a long and complicated procedure [9, 10, 8].

(b) During the materialization of the views in the DW, some old view materializations are removed from the DW while new ones are added to it. Therefore, the DW is no more fully operational. Given the sizes of actual DWs and the complexity of views that need to be computed, the load window required to make the DW operational may become unacceptably long.

In this paper we address the *Dynamic DW Design Problem*: given a DW (a set of materialized views), a set of new queries to be answered by it and possibly some extra space allocated for materialization to the DW, select a set of new views to additionally materialize in the DW such that:

1. The new materialized views fit in the extra allocated space.
2. All the new queries can be answered using exclusively the materialized views (the old and the new).
3. The combination of the cost of evaluating the new queries over the materialized views and the cost of maintaining the new views is minimal.

1.2 Contribution and outline

The main contributions of this paper are the following:

- We set up a theoretical basis for incrementally designing a DW by formulating the dynamic DW design problem. The approach is applicable to a broad class of queries, including queries involving grouping and aggregation operations.
- Using an AND/OR dag representation of multiple queries (multiquery AND/OR dags) we model the dynamic DW design problem as a state space search problem. States are multiquery AND/OR dags representing old and new materialized views and complete rewritings of all the new queries over the materialized views. Transitions are defined through state transformation rules.
- We prove that the transformation rules are sound, and complete. In this sense a goal state (an optimal solution) can be obtained by applying transformation rules to an initial state.

- We design algorithms for solving the problem that incrementally compute the cost and the size of a state when moving from one state to another.
- The approach can also be applied for statically designing a DW, by considering that the set of views already materialized in the DW is empty.

This paper is organized as follows. Next section contains related work. In Section 3, we provide basic definitions and state formally the DW design problem. In Section 4 the dynamic DW design problem is modeled as a state space search problem. Incremental algorithms are presented in Section 5. The final section contains concluding remarks. A more detailed presentation can be found in [11].

2 Related work

We are not aware of any research work addressing the incremental design of a DW. Static design problems using views usually follow the following pattern: select a set of views to materialize in order to optimize the query evaluation cost, or the view maintenance cost or both, possibly in the presence of some constraints.

Work reported in [2, 3] aims at optimizing the query evaluation cost: in [2] greedy algorithms are provided for queries represented as AND/OR graphs under a space constraint. A variation of this paper aims at minimizing the total query response time under the constraint of total view maintenance cost [3].

[6] and [4] aim at optimizing the view maintenance cost: In [6], given a materialized SQL view, an exhaustive approach is presented as well as heuristics for selecting additional views that optimize the total view maintenance cost. [4] considers the same problem for select-join views and indexes together.

The works [7, 12] aim at optimizing the combined query evaluation and view maintenance cost: [7] provides an A* algorithm in the case where views are seen as sets of pointer arrays under a space constraint. [12] considers the problem for materialized views but without space constraints.

None of the previous approaches requires the queries to be answerable exclusively from the materialized views in a non-trivial manner. This requirement is taken into account in [9] where the problem of configuring a DW without space restrictions is addressed for a class of select-join queries. This work is extended in [10] in order to take into account space restrictions, multiquery optimization over the maintenance queries, and auxiliary views, and in [8] in order to deal with PSJ queries under space restrictions.

3 Formal statement of the problem

In this section we formally state the dynamic DW design problem after providing initial definitions.

Definitions. We consider relational algebra queries and views extended with additional operations as for instance grouping/aggregation operations. Let $\mathbf{R}$ be a set of base relations. The DW initially contains a set $\mathbf{V}_0$ of materialized views defined over $\mathbf{R}$, called *old views*. A set $\mathbf{Q}$ of new queries defined over $\mathbf{R}$ needs to be answered by the DW. It can be the case that these queries can be

answered using exclusively the old views. In general though, in order to satisfy this requirement, a set $\mathbf{V}$ of new materialized views needs to be added to the DW. Since the queries in $\mathbf{Q}$ can be answered by the new state of the DW, there must exist a complete rewriting of every query in $\mathbf{Q}$ over the views in $\mathbf{V}_0 \cup \mathbf{V}$. Let $Q \in \mathbf{Q}$. By Q^V, we denote a complete rewriting of Q over $\mathbf{V}_0 \cup \mathbf{V}$. This notation is extended to sets of queries. Thus, we write $\mathbf{Q}^V$, for a set containing the queries in $\mathbf{Q}$, rewritten over $\mathbf{V}_0 \cup \mathbf{V}$.

Generic cost model. The *evaluation cost of* $\mathbf{Q}^V$, denoted $E(\mathbf{Q}^V)$, is the weighted sum of the cost of evaluating every query rewriting in $\mathbf{Q}^V$.

In defining the maintenance cost of $\mathbf{V}$ one should take into consideration that the maintenance cost of a view, after a change to the base relations, may vary with the presence of other materialized views in the DW [6]. As in [6], we model the changes to different base relations by a set of transaction types. A transaction type specifies the base relations that change, and the type and size of every change. The cost of propagating a transaction type is the cost incurred by maintaining all the views in $\mathbf{V}$ that are affected by the changes specified by the transaction type, in the presence of the views in $\mathbf{V} \cup \mathbf{V}_0$. The *maintenance cost of* $\mathbf{V}$, denoted $M(\mathbf{V})$, is the weighted sum of the cost of propagating all the transaction types to the materilaized views in $\mathbf{V}$.

The *operational cost of the new queries and views* is $T(\mathbf{Q}^V, \mathbf{V}) = E(\mathbf{Q}^V) + cM(\mathbf{V})$. The parameter $c, c \geq 0$, is set by the DW designer and indicates the relative importance of the query evaluation vs. the view maintenance cost.

The *storage space needed for materializing the views in* $\mathbf{V}$ is denoted $S(\mathbf{V})$.

Problem statement. We state now the *dynamic DW design problem* as follows.

Input

A set $\mathbf{V}_0$ of old views over a set $\mathbf{R}$ of base relations.

A set $\mathbf{Q}$ of new queries over $\mathbf{R}$.

Functions, E for the query evaluation cost, M for the view maintenance cost and S for the materialized views space.

A constant t indicating the extra space allocated for materialization.

A constant c.

Output

A set of new views $\mathbf{V}$ over $\mathbf{R}$ such that:

(a) $S(\mathbf{V}) \leq t$.

(b) There is a set $\mathbf{Q}^V$ of complete rewritings of the queries in $\mathbf{Q}$ over $\mathbf{V}_0 \cup \mathbf{V}$.

(c) $T(\mathbf{Q}^V, \mathbf{V})$ is minimal.

4 The dynamic DW design as a state space search problem

We model, in this section, the dynamic DW design problem as a state space search problem.

4.1 Multiquery AND/OR dags

A *query dag* for a query is a rooted directed acyclic graph that represents the query's relational algebra expression. Query dags do not represent alternative

equivalent relational algebra rewritings of the query definition over the base relations (that is alternative ways of evaluating the query). Alternative rewritings can be represented compactly by using AND/OR dags [7]. A convenient representation of query evaluations using AND/OR dags [6] is adopted in rule-based optimizers [1]. This representation, distinguishes between AND nodes and OR nodes: *a query AND/OR dag* for a query is a rooted bipartite AND/OR dag $\mathcal{G}_Q$. The nodes are partitioned into AND nodes and OR nodes. AND nodes are called *operation nodes* and are labeled by a relational algebra operator while OR nodes are called *equivalence nodes* and are labeled by a relational algebra expression (a view). The root node and the sink nodes of $\mathcal{G}_Q$ are equivalence nodes labeled by the query Q and the base relations respectively. In the following we may identify equivalence nodes with their labels.

An AND/OR dag $\mathcal{G}'_Q$ is a *subdag* of an AND/OR dag $\mathcal{G}_Q$ if dag $\mathcal{G}'_Q$ is a subdag of dag $\mathcal{G}_Q$, and for every AND (operation) node in $\mathcal{G}'_Q$, all its incoming and outgoing edges in $\mathcal{G}_Q$ are also present in $\mathcal{G}'_Q$. An AND/OR dag is an *AND dag* if no OR (equivalence) node has more than one outgoing edges.

Multiple queries and alternative ways of evaluation can be represented by multiquery AND/OR dags. A *multiquery AND/OR dag* for a set of queries is a bipartite AND/OR dag, similar to a query AND/OR dag, except that it does not necessarily have a single root. Every query in the query set is represented by an equivalence node in the multiquery AND/OR dag. Equivalence nodes representing queries are called *query nodes* and their labeling expressions are preceded in the multiquery dag by a *. All the root nodes (and possibly some other equivalence nodes) are query nodes.

Example 1. Consider the relations `Department(`<u>`DeptID`</u>`, DeptName)` (`D` for short) and `Employee(`<u>`EmpID`</u>`, EmpName, Salary, DeptID)` (`E` for short). An underlined attribute denotes the primary key of the relation. Figure 1 shows a multiquery AND/OR dag $\mathcal{G}$ for the queries $Q_1 = \sigma_{\texttt{Salary}>1000}(\texttt{E} \bowtie \texttt{D})$ and $Q_2 =< \texttt{DeptID}, \texttt{DeptName} > \mathcal{F} < \texttt{AVG(Salary)} > (\sigma_{\texttt{Salary}>1000}(\texttt{E} \bowtie \texttt{D}))$. Some attribute names are abbreviated in the figure in a self-explanatory way. Notice

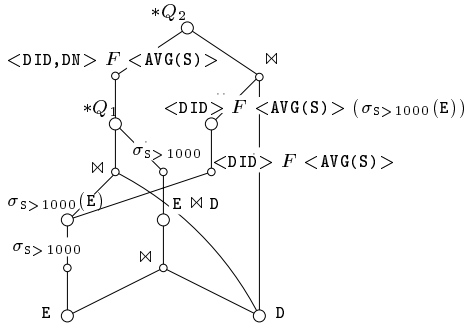

Fig. 1. A multiquery AND/OR dag for the queries Q_1 and Q_2

also that the query node labeled by $*Q_1$ is not a root node. Three alternative rewritings over the base relations for query Q_2 are represented in $\mathcal{G}$. □

4.2 States

A multiquery AND/OR $\mathcal{G}$ determines, in our context, the views and the rewritings of the queries in $\mathbf{Q}$ over views that can be under consideration for solving the dynamic DW problem.

Definition 1. Given $\mathcal{G}$ and $\mathbf{V}_0$, a *state* s is an AND/OR subdag of $\mathcal{G}$, where some equivalence nodes may be *marked*, such that:

(a) All the query nodes of $\mathcal{G}$ are present in s,

(b) All the equivalence nodes of $\mathcal{G}$ that are in $\mathbf{V}_0$ are present in s and these are the only marked nodes in s,

(c) Only query nodes or marked nodes can be root nodes. □

Intuitively, sink nodes represent views materialized in the DW. Marked nodes represent the old views (already materialized in the initial DW) that can be used in the rewriting of a new query. Sink nodes that are not marked represent new materialized views. A *query dag for a query Q in s* is a connected AND subdag of s rooted at query node Q in s whose sink nodes are among the sink nodes of s. It represents a complete rewriting of Q over the materialized views (sink nodes) of s. Since all the query nodes of $\mathcal{G}$ are present in s, there is at least one query dag for a query Q in s, for every query Q in $\mathbf{Q}$. Therefore, *a state provides information for both:*

(a) new views to materialize in the DW, and

(b) complete rewritings of each new query over the old and the new materialized views.

Example 2. Figures 2-3 show different states for the multiquery graph of Figure 1 when $\mathbf{V}_0 = \{\texttt{D}\}$. The labels of the operation nodes are written symbolically in the figures. Marked nodes are depicted by filled black circles. For instance, in Figure 3(a), the nodes E, D and V_1, where $V_1 = \sigma_{\texttt{Salary}>1000}(\texttt{E})$, are materialized views. Node D is an old view and E and V_1 are new views. Two alternative rewritings for the query Q_1 are represented: the query definition $Q_1 = \sigma_{\texttt{Salary}>1000}(\texttt{E} \bowtie \texttt{D})$ and a rewriting of Q_1 using the materialized view V_1, $Q_1 = V_1 \bowtie \texttt{D}$. For the query Q_2 the rewriting $Q_2 =< \texttt{DeptID} > \mathcal{F} < \texttt{AVG(Salary)} > (V_1) \bowtie \texttt{D}$ is represented.

In the state of Figure 3(c) the only new materialized view is query Q_1. The following rewriting is represented: $Q_2 =< \texttt{DeptID}, \texttt{DeptName} > \mathcal{F} < \texttt{AVG(Salary)} > (Q_1)$. Note that this state is not a connected graph. □

With every state s, a cost and a size is associated through the functions *cost* and *size* respectively: $cost(s) = T(\mathbf{Q}^V, \mathbf{V})$, while $size(s) = S(\mathbf{V})$.

4.3 Transitions

In order to define transitions between states we introduce two state transformation rules. The state transformations may modify the set of sink nodes of a state, and remove some edges from it. Therefore, they modify, in general, the set of new views to materialize in the DW and the rewritings of the new queries over the materialized views.

State transformation rules. Consider a state s. A path from a query node Q to a node V is called *query free* if there is no node in it other than Q and V that is a query node.

$R1$ Let Q be a query node and V be a non-sink equivalence node in s. Nodes Q and V need not necessarily be distinct, but if they are distinct, V should not be a query node. If

(a) there is a query free path from Q to V, or nodes Q and V coincide, and

(b) there is no path from V to a non-marked sink node that is not a base relation,

then:

(a) Remove from s all the edges and the non-marked or non-query nodes (except V) that are on a path from V, unless they are on a path from a query node that does not contain V. (Thus, node V becomes a sink node.)

(b) Remove from the resulting state all the edges and the non-marked or non-query nodes that are on a path from Q, unless they are on a query dag for Q in s that contains a query free path from Q to V, or they are on a path from a query node that does not contain Q.

$R2$ Let Q be a query node and V be a distinct equivalence node in s, that is a sink or a query node and is not a base relation. (V can be a marked node). If

(a) there is a query free path from Q to V, and

(b) there is a query dag for Q in s that does not contain a query free path from Q to V,

then:

Remove all the edges and the non-marked or non-query nodes that are on a path from Q, unless they are on a query dag for Q in s that contains a query free path from Q to V, or they are on a path from a query node that does not contain Q.

Example 3. Consider the state s of Figure 2(a). We apply in sequence state transformation rules to s. Figure 2(b) shows the state resulting by the application

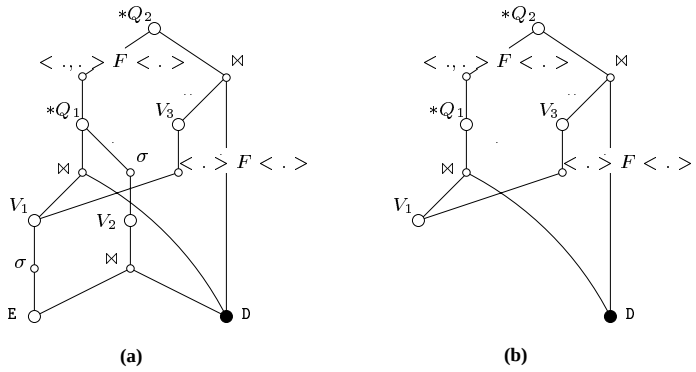

Fig. 2. States

of $R1$ to query node Q_1 and to equivalence node V_1 of s. By applying $R1$ to nodes Q_2 and V_1 of s, we obtain the state of Figure 3(a). The state of Figure 3(b) results

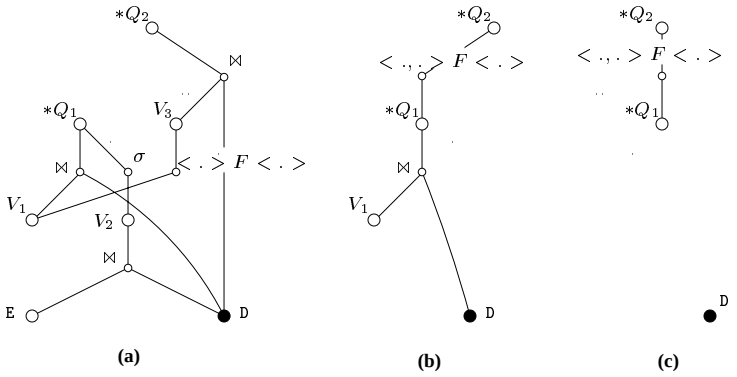

Fig. 3. States resulting by the application of the state transformation rules

from the application of $R2$ to query nodes Q_2 and Q_1 of the state of Figure 2(b). Figure 3(c) shows the state resulting by the application of $R1$ to nodes Q_2 and Q_1 of state s (Figure 2(a)). Query node Q_1 represents now a materialized view. □

The state transformation rules are sound in the sense that the application of a state transformation rule to a state results in a state. Note that the soundness of the state transformation rule entails that *a transformation of a state preserves the existence of a complete rewriting of all the new queries over the materialized views.*

We say that there is a *transition* $T(s, s')$ from state s to state s' if and only if s' can be obtained by applying a state transformation rule to s.

4.4 The search space

We define in this subsection the search space. We first provide initial definitions and show that the state transformation rules are complete.

Definition 2. Given $\mathcal{G}$ and $\mathbf{V}_0$, the *initial state* s_0 is a state constructed as follows. First, mark all the equivalence nodes of $\mathcal{G}$ that are in $\mathbf{V}_0$. Then, for each marked node, remove all the edges and the non-marked or non-query nodes that are on a path from this marked node, unless they are on a path from a query node and this path does not contain the marked node. □

Assumptions. We assume that all the views and the rewritings of the new queries over the views considered are among those that can be obtained from the multiquery AND/OR dag $\mathcal{G}$. Further, consider a set of $\mathbf{V}$ of new materialized views and let $\mathbf{Q}^V$ be a set of cheapest rewritings of the queries in $\mathbf{Q}$ over $\mathbf{V}_0 \cup \mathbf{V}$. In computing the view maintenance cost of $\mathbf{V}$, we assume that a materialized view that does not occur in $\mathbf{Q}^V$ is not used in the maintenance process of another view in $\mathbf{V}$.

Definition 3. Given $\mathcal{G}$ and $\mathbf{V}_0$, a goal state s_g is a state such that there exists a solution $\mathbf{V}$ satisfying the conditions:
(a) The non-marked sink nodes of s_g are exactly the views in $\mathbf{V}$, and
(b) The cheapest rewritings of the queries in $\mathbf{Q}$ over $\mathbf{V}_0 \cup \mathbf{V}$ are represented in s_g.

The following theorem is a completeness statement for the state transformation rules.

Theorem 1. *Let $\mathcal{G}$ be a multiquery AND/OR dag for a set of new queries $\mathbf{Q}$, and $\mathbf{V}_0$ be a set of old views. If there is a solution to the DW design problem, a goal state for $\mathcal{G}$ and $\mathbf{V}_0$ can be obtained by finitely applying in sequence the state transformation rules to the initial state for $\mathcal{G}$ and $\mathbf{V}_0$.* □

Search space definition. Viewing the states as nodes and the transitions between them as directed edges of a graph, the *search space* is determined by the initial state and the states we can reach from it following transitions in all possible ways. Clearly, the search space is, in the general case, a finite rooted directed acyclic graph which is not merely a tree. As a consequence of Theorem 1, there is a path in the search space from the initial state s_0 to a goal state s_g.

5 Algorithms

In this section we present incremental algorithms for the dynamic DW design problem. Heuristics that prune the search space are provided in [11].

The cost and the size of a new state s' can be computed incrementally along a transition $T(s, s')$ from a state s to s' [11]. The basic idea is that instead of recomputing the cost and the size of s' from scratch, we only compute the changes incurred to the query evaluation and view maintenance cost, and to the storage space of s, by the transformation corresponding to $T(s, s')$.

Any graph search algorithm can be used on the search space to exhaustively examine the states (by incrementally computing their cost and size), and return a goal state (if such a state exists). We outline below, a variation of the exhaustive algorithm guaranteeing a solution that fits in the allocated space, and a second one that emphasizes speed at the expense of effectiveness.

A two phase algorithm. This algorithm proceeds in two phases. In the first phase, it proceeds as the exhaustive algorithm until a state satisfying the space constraint is found. In the second phase, it proceeds in a similar way but excludes from consideration the states that do not satisfy the space constraint. A two phase algorithm is guaranteed to return a solution that fits in the allocated space, if a goal state exists in the search space. In the worst case though, it exhaustively examines all the states in the search space.

An r-greedy algorithm. Exhaustive algorithms can be very expensive for a large number of complex queries. The r-greedy algorithm proceeds as follows: for a state considered (starting with the initial state) all the states that can be reached following at most r transitions are systematically generated and their cost and size are incrementally computed. Then, the state having minimal cost among those that satisfy the space constraint, if such a state exists, or a state having minimal size, in the opposite case, is chosen for consideration among them. The algorithm keeps the state s_f satisfying the space constraint and having the lowest cost among those examined. It stops when no states can be generated from the state under consideration and returns s_f. This algorithm is not guaranteed to return a solution to the problem that fits in the allocated space, even if a goal state exists.

6 Conclusion

In this paper we have dealt with the issue of incrementally designing a DW by stating and studying the dynamic DW design problem: given a set of old views materialized in the DW, a set of new queries to be answered by the DW, and extra space allocated for materialization, select a set of new views to materialize in the DW that fits in the extra space, allows a complete rewriting of the new queries over the materialized views and minimizes the combined evaluation cost of the new queries and the maintenance cost of the new views. A dynamic DW design process allows the DW to evolve smoothly in time, without interrupting its operation due to materialized view removal. We have modeled the dynamic DW design problem as a state space search problem, using a multiquery AND/OR dag representation of the new queries. Transitions between states are defined through state transformation rules which are proved to be sound and complete. We have designed generic incremental algorithms and heuristics to reduce the search space. Also shown is that this approach can be used for statically designing a DW.

References

[1] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. of the 9th Intl. Conf. on Data Engineering*, 1993.

[2] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *Proc. of the 6th Intl. Conf. on Database Theory*, pages 98–112, 1997.

[3] H. Gupta and I. S. Mumick. Selection of Views to Materialize Under a Maintenance Cost Constraint. In *Proc. of the 7th Intl. Conf. on Database Theory*, 1999.

[4] W. Labio, D. Quass, and B. Adelberg. Physical Database Design for Data Warehousing. In *Proc. of the 13th Intl. Conf. on Data Engineering*, 1997.

[5] A. Levy, A. O. Mendelson, Y. Sagiv, and D. Srivastava. Answering Queries using Views. In *Proc. of the ACM Symp. on Principles of Database Systems*, 1995.

[6] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 447–458, 1996.

[7] N. Roussopoulos. View Indexing in Relational Databases. *ACM Transactions on Database Systems*, 7(2):258–290, 1982.

[8] D. Theodoratos, S. Ligoudistianos, and T. Sellis. Designing the Global Data Warehouse with SPJ Views. To appear in *Proc. of the 11th Intl. Conf. on Advanced Information Systems Engineering*, 1999.

[9] D. Theodoratos and T. Sellis. Data Warehouse Configuration. In *Proc. of the 23rd Intl. Conf. on Very Large Data Bases*, pages 126–135, 1997.

[10] D. Theodoratos and T. Sellis. Data Warehouse Schema and Instance Design. In *Proc. of the 17th Intl. Conf. on Conceptual Modeling*, pages 363–376, 1998.

[11] D. Theodoratos and T. Sellis. Dynamic Data Warehouse Design. *Technical Report, Knowledge and data Base Systems Laboratory, Electrical and Computer Engineering Dept., National Technical University of Athens*, pages 1–25, 1998.

[12] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proc. of the 23rd Intl. Conf. on Very Large Data Bases*, pages 136–145, 1997.