Designing Data Warehouses

Dimitri Theodoratos [1] Timos Sellis [1]

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) is a database that collects and stores data from multiple remote and heterogeneous information sources. When a query is posed, it is evaluated locally, without accessing the original information sources. In this paper we deal with the issue of designing a DW, in the context of the relational model, by selecting a set of views to materialize in the DW.

First, we briefly present a theoretical framework for the DW design problem, which concerns the selection of a set of views that (a) fit in the space allocated to the DW, (b) answer all the queries of interest, and (c) minimize the total query evaluation and view maintenance cost. We then formalize the DW design problem as a state space search problem by taking into account multiquery optimization over the maintenance queries (i.e. queries that compute changes to the materialized views) and the use of auxiliary views for reducing the view maintenance cost. Finally, incremental algorithms and heuristics for pruning the search space are presented.

Key words: Data Warehousing; Materialize views; View maintenance; Data warehouse design

1 Introduction

Data warehousing is an in-advance approach to the integration of data from multiple, possibly very large, distributed, heterogeneous databases and other information sources. In this approach, selected information from each source

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

is extracted in advance, filtered and transformed as needed, merged with relevant information and loaded in a repository (Data Warehouse - DW). The Data Warehousing approach presents some advantages over the traditional (on demand or lazy) approach to the integration of multiple sources [34], which explains the growing interest of the industry for it:

- The queries can be answered locally without accessing the original information sources. Thus, high query performance can be obtained for complex aggregation queries that are needed for in-depth analysis, decision support and data mining.
- On-Line Analytical Processing (OLAP) is decoupled as much as possible from On-Line Transaction Processing (OLTP). Therefore, the information is highly available and there is no interference of OLAP with local processing at the operational sources.

Data warehouse architecture: Figure 1 shows a typical DW architecture [4]. The data at each layer is derived from the data of lower layers. At the lowest layer there are the distributed operational data sources. The central layer is the *global* or *principal Data Warehouse*. The upper layer contains the *local DWs* or *Data Marts*. Data Marts contain highly aggregated data for extensive analytical processing [13]. They are also probably less frequently updated than global DWs. We view a DW as a set of materialized views (defined over the

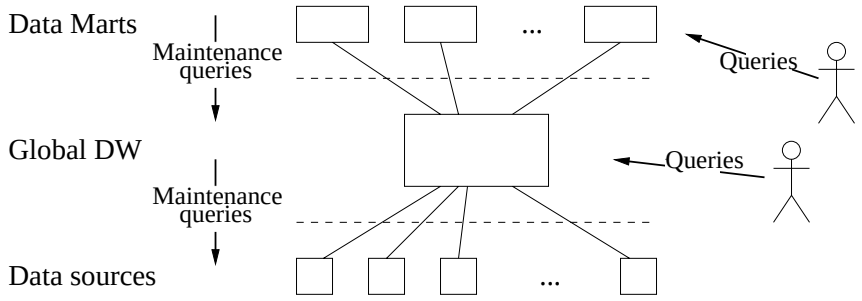

Fig. 1. DW architecture

data sources) [33] in the framework of the relational data model. Users address their queries to the global DW or to the local DWs. These queries must be evaluated locally, at each DW, without accessing the (remote) data of the lower layer.

In this work we are going to address global DW design issues (Local DW design encounters similar problems but queries and views are mainly grouping/aggregation queries). Thus, in the following, DW refers to the global DW. We call the queries that are issued by the users against the DW simply *queries*.

DW view maintenance: When changes to the data in the lowest layer occur, they must be propagated to the data of the higher level. Different maintenance

policies can be applied. Usually, a Data Warehouse is updated periodically. Though, there are applications issuing queries to the Data Warehouse that need current data. In this case an immediate [2,3] or a "on demand" deferred update policy is adopted.

In order to update the materialized views of a DW, after a change to the source relations, an incremental or a rematerialization strategy can be employed. An incremental strategy is based on incremental queries that use the changes made to the source relations to compute changes that can be directly applied to the materialized views [2,18,8,6]. A rematerialization strategy uses the query corresponding to the materialized view (that is the view definition) to recompute the view from the new state of the views of the lower layer. Thus, in order to update the materialized views of the DW, in both cases, *queries are issued against the source relations*. The DW evaluates theses queries by appropriately sending queries to the source relations and receiving the answers. It then performs the updating of the materialized views. We call the queries that are issued by the DW against the source relations, for maintenance reasons, *maintenance queries*.

Example 1 Suppose that the views $V_1 = R \bowtie_{A \geq C} \sigma_{C>c}(S) \bowtie_{D=E} T$ and $V_2 = R \bowtie_{A>B} S \bowtie_{D \geq E} T$ over the source relations $R(A,B)$, $S(C,D)$, $T(E,G)$, are stored materialized at the DW, and that a transaction inserts the tuples ΔT into the source relation T . Then in order to incrementally bring V_1 and V_2 up-to-date, the maintenance queries $\Delta V_1 = R \bowtie_{A \geq C} \sigma_{C>c}(S) \bowtie_{D=E} \Delta T$ and $\Delta V_2 = R \bowtie_{A>B} S \bowtie_{D \geq E} \Delta T$ need to be computed. □

Multiquery optimization on maintenance queries: The changes taken into account for maintaining the materialized views at the DW may affect more than one view. Then multiple maintenance queries defined over the source relations are issued for evaluation. These maintenance queries may contain subexpressions that are identical, equivalent, or more generally subexpressions such that one can be computed from the other. We describe these subexpressions by the generic term *common subexpressions* [7,14]. The techniques of *multiple query optimization* [26,27] allow possibly non-optimal local query evaluation plans to be combined into an optimal global plan, by detecting common subexpressions between queries.

Example 2 The maintenance queries ΔV_1 and ΔV_2 of the previous example can be evaluated together if we exploit the fact that ΔV_1 and ΔV_2 can be computed from the expression $\Delta E = R \bowtie_{A \geq C} S \bowtie_{D \geq E} \Delta T$. A global evaluation plan: (1) computes ΔE from the source relations R and S, and the changes ΔT, and (2) computes ΔV_1 and ΔV_2 from ΔE (ΔV_1 and ΔV_2 can be rewritten over ΔE as follows: $\Delta V_1 = \sigma_{C>c \wedge D=E}(\Delta E)$ and $\Delta V_2 = \sigma_{A>C}(\Delta E)$). This global plan may be more efficient to evaluate than evaluating ΔV_1 and ΔV_2 separately. □

Using materialized views to reduce the maintenance cost of other views: A global evaluation plan for some maintenance queries can be executed more efficiently if some intermediate subqueries are kept materialized in the DW [21], or can be computed from views that are kept materialized in the DW. It is worth noting that an optimal global evaluation plan without materialized subqueries can be completely different than the optimal global evaluation plan when materialized views are used. The existence of these materialized views can greatly reduce the cost of evaluating maintenance queries. Indeed, the computation of the corresponding subqueries is avoided or simplified. Further, since DWs are typically distributed systems, access of the data sources and expensive data transmissions are reduced. On the limit, no access at all of the remote data sources is needed for updating a set of materialized views in response to changes to the data sources. These views are called self-maintainable [9,19,1].

Example 3 In our running example, suppose that the view $V = R \bowtie_{A \geq C-2} S$ is also kept materialized in the DW (for instance, in order to satisfy another query). Then, step (1) of the optimal global evaluation plan can be modified to the (1′) compute ΔE using the materialized view V (ΔE can be rewritten using V as follows: $\Delta E = \sigma_{A \geq C}(V) \bowtie \Delta T$). This plan may be cheaper than the previous one since: (a) No access at all of the (remote) source relations R and S is needed, and (b) a join in the computation of ΔE is saved. □

1.1 The Data Warehouse Design problem

In this paper we deal with the problem of selecting a set of views to materialize in a DW. DWs are mostly used for OLAP and Decision Support. These applications require high query performance. Selections of views though that guarantee high query performance, may require also a significant view maintenance cost. In fact, low query evaluation cost and low view maintenance cost are conflicting requirements. Low maintenance cost is desired because otherwise frequent updating cannot be achieved and current data is a key requirement for many DW applications. Further, if the view maintenance cost is important, query answering may be delayed when a "at query time" deferred maintenance policy is applied. Thus, we are looking here for sets of views that minimize a combination of the query evaluation and the materialized view maintenance cost (*operational cost*).

Another issue somehow orthogonal to minimizing the operational cost is the space occupied by the materialized views. Clearly, the materialized views should fit in the DW. Thus, their total size should not exceed the space available at the DW. If the space occupied by a set of views having minimal operational cost is smaller than the available space, the decision on the

set of views to materialize is determined by the operational cost.

The problem: The *DW design problem* consists of selecting a set of views to materialize in the DW such that:

(1) The materialized views fit in the space available at the DW.
(2) All the queries can be answered using this set of materialized views (without accessing the remote source relations).
(3) The combination of the query evaluation cost and the view maintenance cost (operational cost) of this set of views is minimal.

Difference from other approaches: Other formulations of the problem of selecting views to materialize in a DW, in order to minimize the combined query evaluation and view maintenance cost [10,35], do not require explicitly the queries to be computable from the materialized views. Trivially, this requirement can be met by assuming that all the source relations necessary for answering the queries are available locally for computation [10]. This can be achieved by: (a) considering a centralized DW environment, or (b) considering a distributed environment where all the source relations are replicated at the DW. Clearly, considering centralized DWs is a special instance of the problem since DWs are typically built over distributed data sources. Replicating the source relations entail an important waste of space and may not even be possible because of space restrictions. Further, in this case, the view maintenance cost is increased by the cost of applying to the replicated source relations, every change performed to the source relations. These changes may not even affect the result of any query addressed to the DW. The formulation of the problem in [29] imposes the requirement on the computability of the queries but it does not consider space restrictions. Moreover, the solution suggested does not necessarily yield the optimal view selection when multiquery optimization is performed over the maintenance queries.

Difficulty of the problem: With respect to other problems using views that endeavor to optimize the query evaluation cost [31,5,12,11], or the view maintenance cost [21,15], or both [22,10,35,29], the DW design problem, as it is stated here, is harder since it has to deal with the following combined difficulties:

- In a solution to the problem, *all* the queries need to be answered *using exclusively* the materialized views. In other words, there is the additional constraint that *for every query*, there must be a *complete rewriting* [16] over the views.
- Selections of views that minimize the operational cost may not fit in the available at the DW storage space. Furthermore, there might not even exist a set of materialized views fitting in the available space over which all the

queries can be completely rewritten.
- In constructing the optimal view set we should detect and exploit common subexpressions: (a) between views (to reduce the operational cost and the needed space), (b) between maintenance queries (to perform multiquery optimization), and (c) between views and maintenance queries (to use views in the maintenance process of other views).
- The DW operational cost is the combination of the query and view maintenance cost. These costs may be rivals: a selection of views to materialize in the DW that minimize the query cost may result in an important view maintenance cost and vice versa.

Generality of the problem: The DW design problem encompasses as a special case other design problems with views recently addressed in the bibliography [21,10,35,29]. Our method for solving the problem can be restricted to apply to these cases.

1.2 Contribution and outline

In this paper we state formally the DW design problem and provide a method for solving it for a certain class of relational queries and views. The approach was first introduced in [29] and is extended here in order to take into account space constraints, multiquery optimization over the maintenance queries and the use of views in the maintenance process of other views. The former views may be materialized in an ad-hoc way without being used for answering queries in which case they are called *auxiliary views*. We consider a distributed environment. Thus, the materialized views and the source data are not necessarily stored in the same database. Based on a multiple view representation that uses multiquery graphs we model the problem as a state space search problem. Every state is a multiquery graph of the views that are materialized in the DW plus a complete rewriting of the queries over these views. A transition from one state to another transforms the multiquery graph and rewrites completely the queries over the new view set. We prove that our search space is guaranteed to contain a solution to the problem (if a solution exists) under the assumption of a monotone cost model. Thus, our search space can serve as a basis for developing optimization algorithms and heuristics. We develop an exhaustive algorithm to search for the optimal state which incrementally computes the operational cost when transiting from one state to another. We also provide a greedy algorithm as well as heuristics to prune the search space.

The main contributions of the paper are thus the following:

- We set up a theoretical basis for the DW design problem.
- We provide a method for solving this problem by taking into account mul-

tiquery optimization over the maintenance queries and the use of views in the maintenance process of other views.
- The solution is constructive. Thus, we provide both a set of views to materialize in a DW and a complete rewriting of all the queries over it that minimizes the operational cost.
- We design incremental algorithms and we suggest heuristics for pruning the search space.
- The method is general in that it does not consider a centralized environment. Further, it is not dependent on the way the query evaluation and view maintenance cost is computed.

The rest of the paper is organized as follows. The next section contains related work. In Section 3, we formally state the DW design problem and we provide some intuition on how to deal with it. Section 4 defines states and transitions It also determines the search space and shows that it contains a solution to the problem. Incremental algorithms and heuristics are presented in Section 5. Finally, Section 6 contains concluding remarks and possible extension directions. More details can be found in [30].

2 Related work

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.

Papers [12,11,10] aim at optimizing the query evaluation cost: In [12], the problem is addressed in the context of aggregations and multidimensional analysis under a space constraint. This work is extended in [11], where greedy algorithms are provided, in the same context, for selecting both views and indexes. In [10], greedy algorithms are provided for queries represented as AND/OR graphs. Works [21,15] aim at optimizing the view maintenance cost: In [21], 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. [15] considers the same problem for select-join views and indexes. It provides an A* algorithm as well as rules of thumb, under a number of simplifying assumptions. Space considerations are also discussed. Given a select-project-join view, [19] derives, using key and referential integrity constraints, a set of auxiliary views other than the base relations that eliminate the need to access the base relations when maintaining both the initial and the auxiliary views (i.e. that makes the views altogether self-maintainable).

Works [22,35] aim at optimizing the combined query evaluation and view maintenance cost: [22] provides an A* algorithm in the case where views are

seen as sets of pointer arrays under a space constraint. [35] considers the same problem for materialized views but without space constraints. Further, the maintenance cost model does not take into account multiquery optimization over the maintenance queries or the use of materialized views when maintaining other views. [10] provides a formalization of the problem of selecting a set of views that minimizes the combined cost under a space constraint but it does not provide any algorithm for solving the problem in the general case. This approach considers a centralized DW environment where all the source relations are available locally for computation. Note that none of these approaches require the queries to be answerable exclusively from the materialized views as is the case in the present work. In [22,35,10] the materialized views are chosen in a (preprocessed) global evaluation plan for the queries resulting from a bottom-up merging of local plans. There might be though views in the set of views that minimizes the combined cost that do not appear in this plan. In the present paper we follow a method that decomposes and merges views in a top-down way and we show that the optimal view set appears in our search space. [29] follows an approach similar to the one we present here but it does not take into consideration space constraints or multiquery optimization over the maintenance queries.

Another relevant design problem is the caching problem: given a restricted space (cache) where the results of previously evaluated queries are stored, decide which queries to replace and which queries to admit in the cache in order to optimize query response time [25,23,24].

3 Definitions and formal statement of the problem

We consider that a non-empty set of queries $\mathbf{Q}$ is given, defined over a set of source relations $\mathbf{R}$. The DW contains a set of materialized views $\mathbf{V}$ over $\mathbf{R}$ such that every query in $\mathbf{Q}$ can be rewritten completely over $\mathbf{V}$. Thus, all the queries in $\mathbf{Q}$ can be answered locally at the DW, without accessing the source relations in $\mathbf{R}$. Let Q be a query over $\mathbf{R}$. By Q^V, we denote a complete rewriting of Q over $\mathbf{V}$. 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 $\mathbf{V}$. Given $\mathbf{Q}$, a *DW configuration* $\mathbf{C}$ is a pair $< \mathbf{V}, \mathbf{Q}^V >$. Note that we do not distinguish in the notation between view names, view definitions and view materializations (and often, we use the word 'view' for all of them).

Consider a DW configuration $< \mathbf{V}, \mathbf{Q}^V >$. We call *simple views* those views in $\mathbf{V}$ that appear in $\mathbf{Q}^V$ and *auxiliary views* the rest of the views in $\mathbf{V}$. The intuition behind this definition is the following: simple views in $\mathbf{V}$ are those that are used for answering the queries in $\mathbf{Q}$. The auxiliary views may be used in reducing the maintenance cost of other simple or auxiliary views. Simple

views may also be used in the same manner, but they have to appear in $\mathbf{Q}^V$.

3.1 Cost models

The cost of evaluating a query $Q^V \in \mathbf{Q}^V$ over the materialized views $\mathbf{V}$ is denoted by $E(Q^V)$. Assessing the cost of different evaluation plans, in order to chose the cheapest one, is a standard technique in the process of query evaluation optimization. Thus, any query optimizer [32] could be used to assess the cost $E(Q^V)$ of the cheapest evaluation plan. With every query $Q \in \mathbf{Q}$, we associate a weight f^Q, indicating the relative frequency of issuing Q and its relative importance, with respect to all the queries in $\mathbf{Q}$. The *evaluation cost of* $\mathbf{Q}^V$ is $E(\mathbf{Q}^V) = \sum_{Q \in \mathbf{Q}} f^Q E(Q^V)$.

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 source relations may be different if other materialized views are present in the DW. This is due to the fact that (a) a change to the source relations may affect multiple views; then multiquery optimization can be performed over the multiple maintenance queries issued for maintaining these views, and (b) some views may be used in order to maintain other views. The maintenance cost of $\mathbf{V}$ is thus defined as follows.

We model the changes to the source relations propagated to the DW by transaction types. In the case of an incremental updating, as in [21], each transaction type determines the changed source relations, the types of the changes (insertion, deletion, modification) to each source relation and the size of each change. In the case of a rematerialization strategy, each transaction type determines only the changed relations. Thus, there is only a notification about the source relations that have changed. Let $\mathbf{T}$ be the set of all the transaction types. The cost of maintaining the views in $\mathbf{V}$ affected by a transaction type T, in the presence of the views in $\mathbf{V}$, is denoted by $M(\mathbf{V}, T)$.

The view maintenance cost should comprise: (a) the cost of transmitting data (change differentials, query data and answer data etc.), (b) the cost of computing view changes or new view states, and (c) the cost of applying changes to the materialized views. In a distributed environment, the transmission cost is predominant, while in a centralized one, the cost of computing and applying changes primarily determines the maintenance cost of the materialized views. Notice that there might be views in $\mathbf{V}$ that are not affected by any transaction type.

With every transaction type $T \in \mathbf{T}$, we associate a weight f^T, indicating the relative frequency of the corresponding change propagation and its relative importance, with respect to all the change propagations. The *maintenance*

cost of $\mathbf{V}$ is $M(\mathbf{V}) = \sum_{T \in \mathbf{T}} f^T M(\mathbf{V}, T)$.

The *operational cost* $T(\mathbf{C})$ of a DW configuration $\mathbf{C} =< \mathbf{V}, \mathbf{Q}^V >$ is $T(\mathbf{C}) = E(\mathbf{Q}^V) + cM(\mathbf{V})$. Parameter c, $c \geq 0$, is set by the DW designer and indicates the importance of the view maintenance vs the query evaluation cost. A typical value of c is 1. $c < 1$ privileges the query evaluation cost while $c > 1$ privileges the view maintenance cost in the design of a DW. If the query evaluation cost is more important, the DW designer has the choice to give c a value much smaller than 1 in order to determine a view selection that has good query performance, and conversely.

The storage space needed for materializing a view V is denoted by $S(V)$. Then, the *storage space needed for materializing the views in* $\mathbf{V}$ is $S(\mathbf{V}) = \sum_{V \in \mathbf{V}} S(V)$.

Our approach for dealing with the DW design problem is independent of the way materialized view storage space, query evaluation and view maintenance cost is assessed. The solutions suggested, though, do depend on the specific cost model used.

3.2 The DW design problem

We state now the *DW design problem* as follows.

Input:
A set of source relations $\mathbf{R}$.
A set of queries $\mathbf{Q}$ over $\mathbf{R}$.
For every query $Q \in \mathbf{Q}$, its weight f^Q.
A set of transaction types $\mathbf{T}$ over the source relations $\mathbf{R}$.
For every transaction type $T \in \mathbf{T}$, its weight f^T.
Functions, E for the query evaluation cost, M for the view maintenance cost, and S for the materialized views space.
The space available in the DW for materialization t.
A parameter c.

Output:
A DW configuration $\mathbf{C} =< \mathbf{V}, \mathbf{Q}^V >$ such that $S(\mathbf{V}) \leq t$ and $T(\mathbf{C})$ is minimal.

Note that this statement of the problem asks for both the set of views to materialize in the DW and a complete rewriting of the queries over it.

The approach we follow here to deal with the DW design problem considers first the DW configuration $< \mathbf{Q}, \mathbf{Q}^Q >$. It produces then alternative view selections for materialization by appropriately modifying views, decomposing views, eliminating views, or generating auxiliary views, while guaranteeing the answerability of the queries from the materialized views. It also produces a complete rewriting of the queries over the modified view sets. This procedure takes into account the fact that multiquery optimization can be performed over the maintenance queries and that views can be used in maintaining other views. Each view set produced is examined, in order to measure the impact of the modification on the operational cost and on the space needed for materialization.

Appropriate modification of a view results to simpler global evaluation plans for the maintenance queries, and thus, it may reduce the view maintenance cost. It increases though the query evaluation cost if this view is a simple one, and the space needed for materialization.

Decomposing a simple view in two distinct subviews (splitting a view) increases the query evaluation cost. The cost of computing changes to the views is reduced since the maintenance queries needed for computing the changes after a change to the source relations are simpler. Further, expensive data transmission between the DW and the sources needed for evaluating the maintenance queries are also reduced. The impact on the space needed for materialization depends on the selectivity of the joins.

Eliminating a simple view that can be computed from another view increases the query evaluation cost. It reduces though always the view maintenance cost. Indeed, no computation of the view changes, no data transmissions and no application of the changes are needed for the eliminated view. The needed space is also reduced.

Generating and materializing auxiliary views does not affect the query evaluation cost since these views are not used for answering the queries. As we have already mentioned, if these views can help maintaining other views, then (a) some of the answers to the maintenance queries can be obtained locally, without accessing the (remote) source relations, and (b) some of the computations can be avoided. Obviously, there is a cost associated with the process of maintaining the auxiliary materialized views. But, if this cost is less than the reduction to the maintenance cost of the initially materialized views, it is worth keeping the auxiliary views in the DW. Clearly, extra space is needed for storage. We formalize the previous remarks in the next two sections.

4 The search space

In this section, we model the DW design problem as a state space search problem based on a multiquery graph representation of the views. We then prove that under the assumption of a monotone cost model, our search space is guaranteed to contain a solution to the problem (if such a solution exists).

4.1 The class of queries and views

We consider the class of relational queries and views that are equivalent to relational expressions of the standard form $\sigma_F(R_1 \times \ldots \times R_k)$. $\times$ denotes the Cartesian product. The R_i's, $i \in [1, k]$ denote relations. Formula F is a conjunction of comparisons of the form $x\ op\ y + c$ or $x\ op\ c$ where op is one of the comparison operators $=$, $<$, $\leq$, $>$ and $\geq$, c is an integer valued constant, and x, y are attributes. Conjuncts involving attributes from only one relation are called *selection predicates*, while conjuncts involving attributes from two relations are called *join predicates*. Attributes of every R_i are involved with those of at least one other R_j in a predicate join in F. All the R_is are distinct (no self-joins). Without loss of generality, we consider that attribute names in different relations are distinct. Any query in this class can be put in standard form.

A formula involving $\neq$, disjunction and negation can be handled by eliminating negations, replacing $\neq$ by disjunctions of two strict inequalities, and converting it into disjunctive normal form. Then each disjunct can be considered separately (though this conversion may cause the number of comparisons to grow exponentially). In the following we consider F to be a conjunction of comparisons as above.

A formula F is *satisfied* by a substitution of its attributes by values from their corresponding domain if the resulting formula evaluates to true. F is *satisfiable* if it is satisfied by a substitution, and *valid* if it satisfied by every substitution. A predicate p *implies* a predicate p' if p is more restrictive than p'. For instance, $x = y + 2$ implies $x \leq y + 3$. It is easy to see that, implication between two predicates that are not valid or unsatisfiable entails that both of them involve the same attributes. In general, a Boolean expression of predicates *implies* another such expression, if every substitution that satisfies the first expression, satisfies also the second. Two formulas are *equivalent* if they imply each other. Implication is extended to sets of formulas by viewing them as conjunctions of their formulas.

When atoms are allowed to contain $\neq$, the general problem of checking the satisfiability of a Boolean expression of atoms or the implication of two Boolean

expressions of atoms is NP-hard. When both expressions are conjunctions of atoms that do not contain $\neq$, as in the class of queries we consider here, checking implication and satisfiability is polynomial [20,1,28].

4.2 States

In order to define states, we use the notion of the multiquery graph to represent a set of views $\mathbf{V}$. A multiquery graph allows the compact representation of multiple views. Given a view $V = \sigma_F(R_1 \times \ldots \times R_k)$, its query graph G^V is a multigraph defined as follows:

(1) The set of nodes of G^V is the set of relations appearing in V.
(2) For every join predicate p in V involving attributes of the relations R_i and R_j there is an edge between R_i and R_j labeled as $V : p$. Such an edge is called *join edge*.
(3) For every selection predicate p in V involving attributes of the relation R_i, there is a loop on R_i labeled as $V : p$. If $V = R_i$, there is a loop edge on R_i labeled as $V : T$. The symbol T denotes here a valid formula. Both these edges are called *selection edges*.

The multiquery graph $\mathbf{G}^V$ of a set of views $\mathbf{V}$, is the multigraph resulting by the merging of the query graphs of all the views in $\mathbf{V}$. In addition, views in the multiquery graph can be marked. We represent marked views in $\mathbf{G}^V$ by preceding their names by a $*$. The usefulness of marking the views will be explained later. Clearly, a multiquery graph $\mathbf{G}^V$ contains all the information about the views in $\mathbf{V}$.

A *state* s is a pair $< \mathbf{G}^V, \mathbf{Q}^V >$. Thus, a state s is essentially the DW configuration $\mathbf{C} =< \mathbf{G}^V, \mathbf{Q}^V >$. In a state s, we use the letter W to refer to simple views, the letter Z to refer to auxiliary views while the letter V is used to refer to both of them indiscreetly.

Example 4 Consider the source relation schemas $R(A, B)$, $S(C, D)$, $T(E, F)$, $P(G, H)$ and $U(K, L)$. Let $\mathbf{V} = \{W_1, W_2, W_3\}$ be a set of views over these relations, where the views W_1, W_2 and W_3 are defined as follows:
$W_1 = R \bowtie_{A \leq C} \sigma_{D>5}(S) \bowtie_{C<E+3} T \bowtie_{F \geq G} \sigma_{H=7}(P)$
$W_2 = R \bowtie_{A=C} S \bowtie_{C<E} \sigma_{E=F}(T)$ and
$W_3 = \sigma_{L \leq D}(S \bowtie_{C=E} T \bowtie_{E>K} U)$
The corresponding multiquery graph $\mathbf{G}^V$ is depicted on Figure 2. □

With every state s, a cost is associated through the function $cost(s)$. This is the operational cost $T(\mathbf{C})$ of the DW configuration $\mathbf{C}$. Also, a size is associate through the function $size(s)$. This is the space $S(\mathbf{V})$ needed for materializing the views in $\mathbf{V}$.

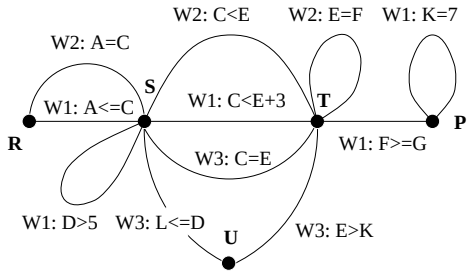

Fig. 2. The multiquery graph $\mathbf{G}^V$

4.3 Transitions

Transitions between states are defined through the following six *state transformation rules* that can be applied to a state $s =< \mathbf{G}^V, \mathbf{Q}^V >$. Each state transformation rule consists on two parts. The first part transforms the multiquery graph $\mathbf{G}^V$ and in most cases introduces new view names in it. If this transformation of $\mathbf{G}^V$ affects a simple view, the second part transforms $\mathbf{Q}^V$ by rewriting the queries in $\mathbf{Q}^V$ over the new view set. We present below the state transformation rules in turn.

- **Selection edge cut**
  If e is a selection edge on node R of an unmarked view in $\mathbf{G}^V$ labeled as $V : p$, where $p \neq T$, then:
  1. $\mathbf{G}^V$ *transformation:*
     (a) If e is the unique edge of R in $\mathbf{G}^V$ labeled by V, replace its label by $V_1 : T$, where V_1 is a new view name (in this case, V_1 represents the source relation R). New view names should not already appear in $\mathbf{G}^V$. They have to respect the convention on simple and auxiliary view names. That is, if V is the simple view W, the new view name is W_1, and similarly for auxiliary views.
     (b) Otherwise, remove e from $\mathbf{G}^V$ and replace every occurrence of V in $\mathbf{G}^V$ by a new view name V_1.
  2. $\mathbf{Q}^V$ *transformation:*
     If V is a simple view, replace any occurrence of V in $\mathbf{Q}^V$, by the expression $\sigma_p(V_1)$.
- **Join edge cut**
  If e is a join edge of an unmarked view in $\mathbf{G}^V$ labeled as $V : p$, and the removal of e from the query graph G^V of V does not divide G^V into two disconnected components, then:
  1. $\mathbf{G}^V$ *transformation:*
     Remove e from $\mathbf{G}^V$, and replace every occurrence of V in $\mathbf{G}^V$ by a new view name V_1. Note that the removal of e does not divide G^V into two

disconnected components if there are also other edges in G^V between the same nodes, or if G^V is cyclic, and e is part of a cycle.

2. $\mathbf{Q}^V$ *transformation:*
   If V is a simple view, replace any occurrence of V in $\mathbf{Q}^V$ by the expression $\sigma_p(V_1)$.

- **View split**
  If e is a join edge of an unmarked simple view in $\mathbf{G}^V$ labeled as $W : p$, and the removal of e from the query graph G^W of W divide G^W into two disconnected components then:

  1. $\mathbf{G}^V$ *transformation:*
     Remove e from $\mathbf{G}^V$ and replace every occurrence of W in $\mathbf{G}^V$ in the one component of G^W in $\mathbf{G}^V$ by a new view name W_1, and in the other component by a new view name W_2.
  2. $\mathbf{Q}^V$ *transformation:*
     Replace any occurrence of W in $\mathbf{Q}^V$ by the expression $W_1 \bowtie_p W_2$.

- **View augmentation**
  If the predicate p of an unmarked view V in $\mathbf{G}^V$ implies a predicate p' of a different simple view W in $\mathbf{G}^V$ then:

  1. $\mathbf{G}^V$ *transformation:*
     Replace $V : p$ in $\mathbf{G}^V$ by $V : p'$, and then replace any occurrence of V in $\mathbf{G}^V$ by a new view name V_1.
  2. $\mathbf{Q}^V$ *transformation:*
     If V is a simple view, replace any occurrence of V in $\mathbf{Q}^V$ by the expression $\sigma_{p'}(V_1)$.

- **View elimination**
  If the simple view W_1 and the unmarked simple view W in $\mathbf{G}^V$ have the same set of nodes and each predicate of W_1 is implied by a predicate of W then:

  1. $\mathbf{G}^V$ *transformation:*
     Remove all the edges labeled by W in $\mathbf{G}^V$.
  2. $\mathbf{Q}^V$ *transformation:*
     Replace any occurrence of W in $\mathbf{Q}^V$ by $\sigma_{p_1 \wedge \ldots \wedge p_n}(W_1)$, where $p_1, \ldots, p_n$ are the predicates of W_1 that are not implied by a predicate of W. If there is no such predicate, simply replace any occurrence of W in $\mathbf{Q}^V$ by W_1.

- **Auxiliary view generation**
  If $R_1, \ldots, R_k$ are some (but not all the) nodes of a view V in $\mathbf{G}^V$ and the subgraph of the query graph of V defined by these nodes is a connected graph then:

  1. $\mathbf{G}^V$ *transformation:*
     Mark the view V in $\mathbf{G}^V$. Let Z be a new auxiliary view name. For every edge in $\mathbf{G}^V$ on R_i or between R_i and R_j, $i, j = 1, \ldots k$, labeled as $V : p$

(or $*V : p$ if the view V is marked) add an edge in $\mathbf{G}^V$ between the same nodes, labeled as $Z : p$. We say that auxiliary view Z is based on view V.

2. $\mathbf{Q}^V$ *transformation:*
   The set $\mathbf{Q}^V$ is not modified, i.e. the queries in $\mathbf{Q}^V$ are not rewritten over a new view set.

Example 5 Consider the query set $\mathbf{Q} = \{Q_1, Q_2, Q_3\}$ and the view set $\mathbf{V} = \{W_1, W_2, W_3\}$ of example 4.1 where each query Q_i is defined as is defined view W_i. Let $Q_1^V = W_1$, $Q_2^V = W_2$, and $Q_3^V = W_3$. Views W_1, W_2, W_3 are simple views (as their name indicates) since they appear in $\mathbf{Q}^V$. We apply in sequence state transformation rules to the state $< \mathbf{G}^V, \mathbf{Q}^V >$, and we depict the resulting state.

In Figure 3(a) we show $\mathbf{G}^V$ after the application of the selection edge cut rule to the edge labeled as $W_1 : D > 5$. Query Q_1 is rewritten as follows: $Q_1^V = \sigma_{D>5}(W_4)$. Queries Q_2 and Q_3 are not affected by this transformation. W_4 is a new view name. New view names may be introduced in the multiquery graph and in the rewritings of the query definitions during the application of the transformation rules.

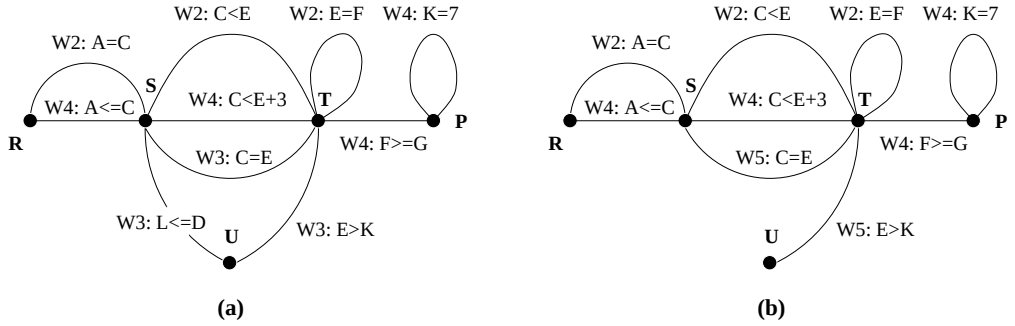

Fig. 3. $\mathbf{G}^V$ after an application of (a) the selection edge cut rule and (b) the join edge cut rule

Figure 3(b) shows $\mathbf{G}^V$ after the application of the join edge cut rule to the join edge labeled as $W_3 : L \leq D$. This transformation rule can be applied because the join edge is part of a cycle in the query graph of view W_3. The query Q_3 is now rewritten as follows: $Q_3^V = \sigma_{L \leq D}(W_5)$. The queries Q_1 and Q_2 are not affected.

In Figure 4(a), the view split rule has been applied to the join edge labeled as $W_4 : F \geq G$ of $\mathbf{G}^V$ (W_4 is a simple view). Only query Q_1 is affected. Its rewriting is: $Q_1^V = \sigma_{D \geq 5}(W_6 \bowtie_{F \geq G} W_7)$.

The simple views W_2 and W_6 are defined over the same set of nodes and the predicates $A \leq C$ and $C < E + 3$ of W_6 (these are the only predicates of W_6) are implied by the predicates $A = C$ and $C < E$ of W_2 respectively. Thus,

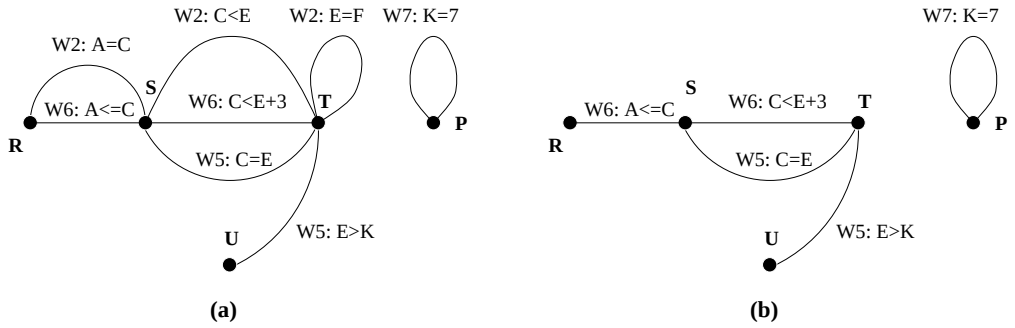

Fig. 4. $\mathbf{G}^V$ after an application of (a) the view split rule and (b) the view elimination rule

we can apply the view elimination rule to W_6 and W_2, and eliminate W_2 from $\mathbf{G}^V$. The resulting multiquery graph is depicted on Figure 4(b). Only query Q_2^V is affected by this transformation. Since no predicate of W_2 is implied by a predicate of W_6, Q_2^V is rewritten as follows: $Q_2^V = \sigma_{A=C \wedge C<E \wedge E=F}(W_6)$. View W_6 appears now in the rewriting of both queries Q_1 and Q_2. Notice that no new view name is introduced by this transformation.

Up to now, there are no auxiliary views in $\mathbf{G}^V$. The application of the auxiliary view generation rule to nodes S and T of view W_5 generates the auxiliary view Z_1 depicted on Figure 5(a). Auxiliary views are represented by dashed lines on the figures. Note that view W_5 is now marked. No query rewriting is needed for this transformation.

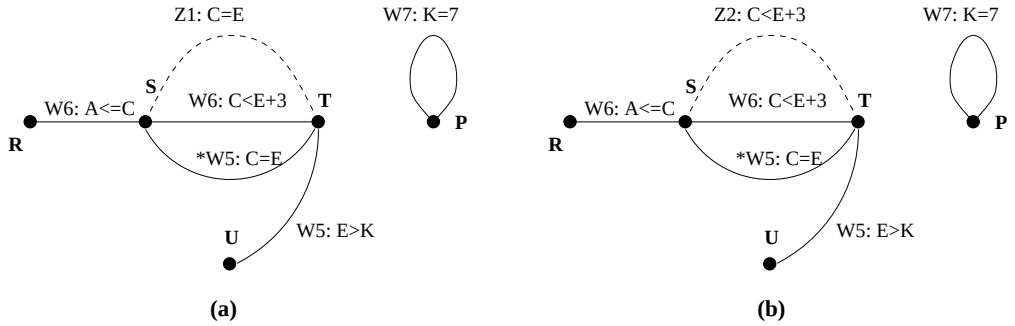

Fig. 5. $\mathbf{G}^V$ after an application of (a) the auxiliary view generation rule and (b) the view augmentation rule

The predicate $C = E$ of view Z_1 implies the predicate $C < E + 3$ of view W_6. By applying the view augmentation rule to these views we obtain the multiquery graph of Figure 5(b). Note that the auxiliary view Z_2 can be used for maintaining both: view W_5 and view W_6. Since no simple view is modified by this transformation, the queries need not be rewritten. □

Remarks. The view augmentation rule covers the case where multiquery optimization over the maintenance queries is performed.

If the condition of the view elimination rule is satisfied, view W can be computed from view W_1. If in addition, each predicate of W_1 is implied by a predicate of W, views W and W_1 are equivalent.

The auxiliary view generation rule applied to the nodes of a view V generates an auxiliary view (potentially modified afterwards) that may be used for maintaining V (and maybe other simple or auxiliary views). Notice that auxiliary views based on other auxiliary views can also be generated.

The usefulness of marking a view V in $\mathbf{G}^V$ is to indicate that an auxiliary view based on V is already present in $\mathbf{G}^V$. Thus, the application of a rule other than the auxiliary view generation on V is prevented. The reason is that we do not want to modify a view when an auxiliary view based on it has been generated. The auxiliary view may not anymore be useful in maintaining the initial view.

The view split rule cannot be applied to an auxiliary view Z because the two separate auxiliary views that would have been resulted can be obtained by applying twice the auxiliary view generation rule on Z. Similarly, the view elimination rule cannot eliminate an auxiliary view, since an eliminated auxiliary view can simply not be generated.

By applying any of the six state transformation rules to a state s we obtain the multiquery graph $\mathbf{G}^{V'}$ of a set of views $\mathbf{V}'$ over $\mathbf{R}$ and a complete rewriting of $\mathbf{Q}$ over $\mathbf{V}'$, i.e. a new state $< \mathbf{G}^{V'}, \mathbf{Q}^{V'} >$. There is a *transition* $T(s, s')$ from state s to state s' iff s' can be obtained by applying any of the six state transformation rules to s.

4.4 Completeness of the state transformation rules

We start by providing some definitions. A query Q is *satisfiable* if for some instance of the base relations Q returns a non-empty set. Clearly, a query $Q = \sigma_F(R_1 \times \ldots \times R_k)$ is satisfiable if and only if F is a satisfiable formula. In the following we consider that the input to the problem queries are satisfiable queries. A query Q *contains* another query Q' if the materialization of Q is a superset of the materialization of Q', for any instance of the base relations. Two queries are *equivalent* if and only if they mutually contain each other. Clearly, if $Q = \sigma_F(R_1 \times \ldots \times R_k)$ and $Q' = \sigma_{F'}(R_1 \times \ldots \times R_k)$, Q contains Q', if F is implied by F', and Q and Q' are equivalent, if and only if F and F' are equivalent.

Definition 6 A satisfiable formula F is in *full form* when: (a) if a predicate p is implied by F, then there is a predicate in F that implies p, and (b) F is

not redundant in the sense that there is no predicate in F that is implied by another predicate in F. A satisfiable query $Q = \sigma_F(R_1 \times \ldots \times R_k)$ is in *full form* if F is in full form. □

Example 7 Consider the attributes x, y and z. Let $F_1 = x \leq y \wedge y \leq z$. F_1 is not in full form since the predicate $x \leq z$ is implied by F_1 and neither $x \leq y$ nor $y \leq z$ implies $x \leq z$. Let also $F_2 = x \leq y \wedge x \leq y + 2$. F_2 is not in full form since the predicate $x \leq y$ of F_2 implies the predicate $x \leq y + 2$ of F_2. In contrast, the formula $F = x \leq y \wedge y \leq z \wedge x \leq z$ is in full form, and it is also equivalent to F_1. □

Intuitively, when a query is put in full form, all the significant restrictions on the involved base relations and between any two relations interrelated through joins that can be derived by the query definition are explicitly indicated. A formula can be equivalently put in full form in polynomial time. This can be easily derived from results in [28].

A view V' is a *subview* of a view V if V can be rewritten (not necessarily completely) using V'. A view V' can be used in maintaining a view V if V can be partially rewritten [16] using V'. The following theorem is a completeness statement for the state transformation rules.

Theorem 8 *Let* $\mathbf{Q}$ *be a set of queries in full form, and* $\mathbf{C} =< \mathbf{V}, \mathbf{Q}^V >$ *be a DW configuration such that any auxiliary view in* $\mathbf{V}$ *can be used in maintaining another view in* $\mathbf{V}$*. Let also* $sub(\mathbf{V})$ *be a set of subviews of the views in* $\mathbf{V}$ *that contains all the views in* $\mathbf{V}$*, and* $\mathbf{X}$ *be a set of complete rewritings of some views in* $sub(\mathbf{V})$ *over views in* $sub(\mathbf{V})$*. Then, there is a state* $< \mathbf{G}^{V'}, \mathbf{Q}^{V'} >$ *obtained by applying in sequence a finite number of state transformation rules to the state* $s_0 =< \mathbf{G}^Q, \mathbf{Q}^Q >$ *such that:*

(a) There is a set $sub(\mathbf{V}')$ *and a mapping* f *from* $sub(\mathbf{V})$ *onto* $sub(\mathbf{V}')$ *such that* $\forall V \in sub(\mathbf{V})$, V *contains* $f(V)$*.*

(b) For every query $Q^V \in \mathbf{Q}^V$*, query* $Q^{V'}$ *involves exactly the images of the views in* Q^V *with respect to* f*.*

(c) For every complete rewriting in $\mathbf{X}$ *of a view* V *over views* $V_1, \ldots, V_n$*, where* $V, V_1, \ldots, V_n \in sub(\mathbf{V})$*, there is a complete rewriting of* $f(V)$ *over* $f(V_1), \ldots, f(V_n)$*.* □

A proof can be found in [30]. Condition (c) of the previous theorem handles multiquery optimization over the maintenance queries and the use of views in maintaining other views. Consider for instance a global evaluation plan [26] $\mathcal{P}$ for the recomputation of the views in $\mathbf{V}$ affected by a transaction type. Suppose that all the nodes of $\mathcal{P}$ belong to $sub(\mathbf{V})$ and that the computation of all the nodes from its (their) child (children) node(s) are present as rewritings in $\mathbf{X}$. The leaf nodes of this plan are either source relations or materialized views (simple or auxiliary). Through the function f we can map this plan to a

global evaluation plan $\mathcal{P}'$ for the views in $\mathbf{V}'$ affected by the same transaction type such that, if a node n is computed from the nodes $n_1, \ldots, n_k$ in $\mathcal{P}$, $f(n)$ is computed from $f(n_1), \ldots, f(n_k)$ in $\mathcal{P}'$. A similar mapping exists for a global evaluation plan for maintenance queries that involve in addition source relation differentials.

This theorem generalizes results in [29] where multiquery optimization on the maintenance queries and the use of auxiliary views are not taken into account.

4.5 Search space definition

Consider now two states $s =< \mathbf{G}^V, \mathbf{Q}^V >$ and $s' =< \mathbf{G}^{V'}, \mathbf{Q}^{V'} >$. Let r be a one-to-one mapping from view names in $\mathbf{G}^V$ onto view names in $\mathbf{G}^{V'}$. Such a mapping is called view renaming from s to s'. $r(\mathbf{G}^V)$ denotes the multiquery graph resulting by renaming the views in $\mathbf{G}^V$ according to r, and $r(Q_i^V)$, $Q_i^V \in \mathbf{Q}^V$, denotes the query rewriting resulting by renaming the views in Q_i^V according to r. Then, s and s' are *equivalent* if there is a view renaming from s to s' such that $\mathbf{G}^{V'} = r(\mathbf{G}^V)$, and $Q_i^{V'}$ is equivalent to $r(Q_i^V)$, for every $Q_i^V \in \mathbf{Q}^V$.

We call *initial state* the state $s =< \mathbf{G}^Q, \mathbf{Q}^Q >$. Viewing states as nodes and transitions between them as directed edges, the *search space* is a directed graph determined by the initial state and the states we can reach from it following transitions in all possible ways. Equivalent states are represented in the search space by the same node. Clearly the search space is a rooted at s_0 directed acyclic graph which in the general case is not merely a tree.

Consider two complete rewritings of views (or queries) V and V' defined exactly over views $V_1, \ldots, V_n$, such that V contains V', and V_i contains V_i', $i = 1, \ldots, n$. A cost model is *monotone* if the cost of computing V' from $V_1', \ldots, V_n'$ is not greater than the cost of computing V from $V_1, \ldots, V_n$.

As a consequence of theorem 4.1 and under the assumption of a monotone cost model, there is a path in the search space from the initial state to a state that satisfies the space constraint and has minimal cost (if such a state exists). Therefore, our search space can serve as a basis for developing optimization algorithms and devising heuristics. This issue is addressed in the next section.

5 Incremental algorithms and heuristics

We present in this section an exhaustive incremental algorithm and a greedy one and we suggest heuristics for pruning the search space. In this paper we are

not concerned with implementation issues. Rather, we highlight a method for designing a DW. Thus, the presentation of the algorithms emphasizes clarity at the expense of efficiency.

For obtaining the lowest maintenance cost when maintaining a view V using another view V' that is defined over a fixed set of base relations, only one such view V' is needed. All the other views defined over the same set of base relations are useless. Thus, in the following, we consider that the definition of a state and the auxiliary view generation rule are slightly modified: the multiquery graph $\mathbf{G}^V$ in a state may also contain hyperedges (sets of nodes) labeled by a view name in $\mathbf{G}^V$. Suppose that a view V in $\mathbf{G}^V$ is defined at least over the source relations $R_1, \ldots, R_k$. A hyperedge $\{R_1, \ldots, R_k\}$ in $\mathbf{G}^V$ indicates that an auxiliary view over $R_1, \ldots, R_k$, based on V, is already generated. The auxiliary view generation rule cannot be applied to the nodes $R_1, \ldots, R_k$ of view V if this hyperedge is present in $\mathbf{G}^V$. If this hyperedge is not present in $\mathbf{G}^V$, this rule is applicable and its application entails also the addition of a hyperedge $\{R_1, \ldots, R_k\}$ labeled by V to $\mathbf{G}^V$.

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' [30]. 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 $T(s, s')$. The following example shows how the query evaluation cost can be computed incrementally.

Example 9 Consider a transition $T(s, s')$ from $s =< \mathbf{G}^V, \mathbf{Q}^V >$ to $s' =< \mathbf{G}^{V'}, \mathbf{Q}^{V'} >$. The state transformation corresponding to $T(s, s')$ modifies only one view V in $\mathbf{G}^V$ (if any). Let $Q_1^V, \ldots, Q_q^V$ be the queries in $\mathbf{Q}^V$ defined using V. Then the increment to the query evaluation cost $\Delta E = \sum_{i \in [1,q]} f^{Q_i}(E(Q_i^{V'}) - E(Q_i^V))$. Clearly, if the state transformation corresponding to $T(s, s')$ is an auxiliary view generation, then $\Delta E = 0$. □

Usually, the query rewritings and the transaction types affected by a transformation represent a small subset of $\mathbf{Q}^V$ and $\mathbf{T}$ respectively, while a transformation affects at most one view in $\mathbf{G}^V$. Thus, the incremental computation provides a substantial improvement to the computation of the cost and the size of the new state s'.

5.1 An exhaustive algorithm

The exhaustive algorithm considers the states in the search space starting with the state $s_0 =< \mathbf{G}^Q, \mathbf{Q}^Q >$. When a state is considered all the children states are produced (the state is expanded). A state s is stored along with its cost, $cost(s)$, and size, $size(s)$, in the sets *open* and *closed* which are initially

empty. States that have been expanded are stored in the set *closed*. States for consideration are stored in the set *open*. When a state is produced, it is checked against the states already stored in the sets *open* and *closed*. If it is equivalent to a state in one of these sets, it is not further considered. Thus, from all the equivalent states produced only one is expanded. Otherwise, its cost and size are incrementally computed and it is stored in the set *open*. After a state is expanded, it is moved from the set open to the set closed. When the set *open* becomes empty, a state having minimal cost among all the states in *closed* that satisfy the space constraint is returned.

Clearly the algorithm terminates and returns a state satisfying the space constraint and having minimal cost, when such a state exists. An exhaustive algorithm can be very expensive for a big number of complex queries. Even though the design of a DW is a procedure that is not meant to be done very frequently, we also present below a greedy algorithm, and explore heuristics that can be used to improve the performance of the algorithms.

5.2 An r-greedy algorithm

The r-greedy algorithm proceeds in two phases. In the first phase it endeavors to find a state that satisfies the space constraint. Starting with the state s_0, it iteratively expands the states in the search space to a depth r. When a state is expanded, if no state satisfying the space constraint is found among those produced, the algorithm chooses for further consideration the one that has the minimal space requirement. Otherwise, it proceeds to the second phase. If no state satisfying the space constraint is found in the first phase, the algorithm returns a fail. In the second phase, the algorithm endeavors to find a state that has minimal cost. It starts with a state that satisfies the space constraint and has minimal cost among those produced in the first phase. Then, it iteratively expands the states in the search space to a depth r, by considering only states satisfying the space constraint, and chooses one having minimal cost for further consideration. The algorithm stops when the expansion of a state under consideration does not produce any state satisfying the space constraint.

The basic outline of the algorithm is depicted in Figure 6. The use of the sets *open* and *closed* is as in the exhaustive algorithm. The distance of two states $distance(s_1, s)$ is the number of transitions between states s_1 and s.

5.3 Heuristics

The number of states that can be produced from a given state by generating and modifying auxiliary views can be very big. The heuristics below concern exactly the generation and modification of auxiliary views.

A two-phase application of the rules. The distinction of the views in sim-

ple and auxiliary, entails a respective distinction of the state transformations: *simple view transformations* are those that modify or eliminate a simple view. These transformations modify also the query rewritings in $\mathbf{Q}^V$. Transformations that do not modify or eliminate a simple view are called *auxiliary view transformations*. These transformations modify or generate an auxiliary view and do not modify the query rewritings. This distinction suggests for the following two-phase heuristic application of the rules: during the first phase, use one of the algorithms to find a state s, starting from the state s_0, by performing only simple view transformations. During the second phase, use one of the algorithms to compute the final state starting from state s, by performing only auxiliary view transformations.

This treatment allows in the first phase the computation of a set of simple views that is needed for rewriting all the queries over it. The resulting DW configuration has the minimal operational cost that can be obtained with the algorithm used, when auxiliary views are not employed to support the maintenance process of a view. The extra computational effort incurred by the generation and modification of auxiliary views in every intermediate state is avoided. Once this set of simple views is fixed, the second phase generates and modifies auxiliary views in order to minimize the operational cost. Actually, the second phase is a procedure for solving the problem of selecting a set of auxiliary views to materialize, given a fixed set of views, such that the overall maintenance cost is minimized [21]. Indeed, since the auxiliary view transformations do not modify the query rewritings, they do not modify the query evaluation cost either. Thus the cost to be minimized in the second phase is essentially the view maintenance cost. Of course, the absence of auxiliary view transformations in the first phase can lead to fixing a set of simple views which is different (and less efficient) than the one computed when all the transformations are operational. Note though that this heuristic does not prevent from finding a view set that fits in the available space, if a solution to the problem exists: no simple view transformation depends on auxiliary views, while the auxiliary views consume extra space.

Example 10 Consider the queries $Q_1 = R \bowtie_{A \geq B} \sigma_{C>c}(S) \bowtie_{C=D} T$ and $Q_2 = R \bowtie_{A=B} S \bowtie_{C=E} U$ over the source relations $R(A,F)$, $S(B,C)$, $T(D,G)$, $U(E,H)$. Suppose, for the needs of this simplifying example, that a rematerialization maintenance strategy is adopted, and that the cost of maintaining the views affected by a transaction type is the recomputation cost of these views. Let the cost of computing a query (view) be the number of joins in it. Suppose also that there are only two transaction types $T_1 = \{T\}$ and $T_2 = \{U\}$, while the other source relations never change, and that there is no restriction in the space available in the DW for view materialization. Further, let $f^{Q_1} = f^{Q_2} = 0.5$, $f^{T_1} = 0.75$, $f^{T_2} = 0.25$, and $c = 0.5$.

The exhaustive algorithm, after examining the whole search space, returns

as a solution the simple views $W_1 = R \bowtie_{A \geq B} \sigma_{C>c}(S) \bowtie_{C=D} T$, and $W_2 = R \bowtie_{A=B} S \bowtie_{C=E} U$, and the auxiliary views $Z_1 = R \bowtie_{A \geq B} \sigma_{C>c}(S)$, and $Z_2 = R \bowtie_{A=B} S$. The rewriting of the queries over the simple views is $Q_1^V = W_1$, and $Q_2^V = W_2$, while the simple views can be partially rewritten using the auxiliary views as follows: $W_1 = Z_1 \bowtie_{C=D} T$, and $W_2 = Z_2 \bowtie_{C=E} U$. This optimal solution can be obtained by applying twice the auxiliary view generation rule, starting with the initial state. Its cost is 0,5.

If the heuristic is applied, the algorithm returns as a solution in the first phase (without employing any auxiliary view transformation) the simple views $W_1 = R \bowtie_{A \geq B} \sigma_{C>c}(S)$, $W_2 = T$, and $W_3 = R \bowtie_{A=B} S \bowtie_{C=E} U$. In the second phase, by employing exclusively auxiliary view transformations on this solution, it additionally returns the auxiliary view $Z_1 = R \bowtie_{A=B} S$. The rewriting of the queries over the simple views is $Q_1^V = W_1 \bowtie_{C=D} W_2$, and $Q_2^V = W_3$, while simple view W_3 can be partially rewritten using auxiliary view Z_1 as previously: $W_3 = Z_1 \bowtie_{C=E} U$. This solution is obtained by applying the join edge cut rule on the initial state in the first phase, and the auxiliary view generation rule on the resulting state in the second phase. The final solution has cost 0.625 which is 25% worse than the optimal solution. □

Guiding auxiliary view generation by frequent transaction types. Not all the auxiliary views based on a view V are useful in maintaining V after the propagation of the changes specified by a transaction type. Given a view V in a state, this heuristic allows, for every frequent transaction type T_i that affects V, the generation only of the auxiliary views based on V that contain all the relations of V not specified in T_i, instead of all the relations of V specified in T_i. Frequent transaction types are considered because they are expected to contribute more than the others to the view maintenance cost.

Example 11 Consider the setting of Example 10. Let frequent transaction type be a transaction type T_i such that $f^{T_i} \geq 0.5$. Thus, only T_1 is a frequent transaction type. In the initial state only view $W_1 = R \bowtie_{A \geq B} \sigma_{C>c}(S) \bowtie_{C=D} T$ is affected by T_1. No auxiliary view containing T in its definition can be used in maintaining W_1. Therefore, by applying this heuristic, only the auxiliary view $Z_1 = R \bowtie_{A \geq B} \sigma_{C>c}(S)$ will be generated from the initial state. The generation of other auxiliary views, as for instance the auxiliary views $\sigma_{C>c}(S) \bowtie_{C=D} T$ and $\sigma_{C>c}(S)$ based on W_1, and the auxiliary views $S \bowtie_{C=E} U$, and $R \bowtie_{A=B} S$ based on W_2, is prevented. The cost of the solution obtained by applying this heuristic to the exhaustive algorithm is again 25% worse than the optimal solution. □

Prohibiting transformations of the auxiliary views. This heuristic deactivates the auxiliary view transformations join edge cut, selection edge cut and view augmentation. In fact an auxiliary view, in the form it has when it is

generated, can be used more efficiently in maintaining the view on which it is based: the corresponding maintenance query can be rewritten by joining the auxiliary view with other views or relations, without applying any selection condition on it. The usefulness of transforming an auxiliary view relies on the possibility of using the transformed auxiliary view in maintaining other materialized views besides the view on which it is based. Thus, this pruning of the search space is done at the expense of using an auxiliary view this way.

Example 12 In the setting of Example 10, suppose that the space available for materialization is restricted and that no more than three views can be kept materialized in the DW. Then, the exhaustive algorithm returns as an optimal solution the simple views $W_1 = R \bowtie_{A \geq B} \sigma_{C>c}(S) \bowtie_{C=D} T$, and $W_2 = R \bowtie_{A=B} S \bowtie_{C=E} U$, and the auxiliary view $Z = R \bowtie_{A \geq B} S$. Auxiliary view Z_1 can be used to maintain both simple views since they can be partially rewritten using Z: $W_1 = \sigma_{C>c}(Z) \bowtie_{C=D} T$ and $W_2 = \sigma_{A=B}(Z) \bowtie_{C=E} U$. This solution can be obtained by applying the auxiliary view generation transformation to view W_1 of the initial state, and the selection edge cut auxiliary view transformation to the resulting state. Auxiliary view Z can be used to maintain views W_1 and W_2 less efficiently than the auxiliary views Z_1 and Z_2 of the optimal solution of Example 10, as the partial rewritings of W_1 and W_2 using Z_1 and Z_2 respectively indicate.

By applying this heuristic to the exhaustive algorithm, auxiliary views can be generated but not modified afterwards. Thus, the solution returned is W_1, W_2 and Z_2. Auxiliary view Z_2 can be used efficiently in maintaining W_1, while it cannot be used in the maintenance process of W_2. □

Note that the heuristics above can by applied in combination thus increasing further the pruning of the search space.

6 Conclusion and possible extensions

We have addressed the problem of selecting a set of views to materialize in a DW that fits in the space allocated for materialization, allows all the queries of interest to be answered using exclusively these views, and minimizes the combined query evaluation and view maintenance cost (DW design problem). The problem is formalized as a state space search problem for a class of relational queries that takes into account both: multiquery optimization over the maintenance queries and the use of (auxiliary) views in the computation of the maintenance queries. A solution provides the optimal view set and a complete rewriting of the queries over it. The approach is general and is not dependent on the way query evaluation and view maintenance cost is assessed. We have designed incremental algorithms and have presented heuristics to reduce the

execution time of the algorithms. Experimental results on a restricted version of the problem are presented in [17].

The approach can be easily extended to deal with projections. Projections can be treated by keeping with each node of the multiquery graph the attributes that are projected out in each view, labeled by the corresponding view name. Then, an extension of the state transformation rules is needed that captures the semantics of the projections. The design problem of Data Marts which contain highly aggregated data also constitutes an important extension direction.

Real DWs need to use access structures for evaluating queries and maintenance queries. Thus another extension direction comprises selecting both views and indexes to materialize in a DW. The exploitation of integrity constraints in the DW design process is also an issue that needs to be examined.

In this paper we have studied the static case of the DW design problem. DWs though are entities that need to evolve in time. Dynamic interpretations of the DW design problem involve the incremental design of a DW when different input parameters to the problem are modified; this issue is also an interesting extension to the presented work.

References

[1] J. A. Blakeley, N. Coburn, and P. A. Larson. Updating derived relations: detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, 1989.

[2] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 61–71, 1986.

[3] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[4] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing Queries with Materialized Views. In *Proc. of the 11th Intl. Conf. on Data Engineering*, pages 190–200, 1995.

[6] L. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 469–480, 1996.

[7] S. Finkelstein. Common Expression Analysis in Database Applications. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 235–245, 1982.

[8] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 328–339, 1995.

[9] A. Gupta, H. Jagadish, and I. S. Mumick. Data Integration using Self-Maintainable Views. In *Proc of the 5th EDBT Conf.*, pages 140–144, 1996.

[10] 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.

[11] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index Selection for OLAP. In *Proc. of the 13th Intl. Conf. on Data Engineering*, pages 208–219, 1997.

[12] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1996.

[13] W. Inmon and C. Kelley. *Rdb/VMS: Developing the Data warehouse*. QED Publishing Group, Boston, Massachussets, 1993.

[14] M. Jarke. Common Subexpression Isolation in Multiple Query Optimization. In Kim, Reiner, and Batory, editors, *Query Processing in DB Systems*, pages 191–205. Springer-Verlag, 1984.

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

[16] 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*, pages 95–104, 1995.

[17] S. Ligoudistianos, D. Theodoratos, and T. Sellis. Experimental Evaluation of Data Warehouse Configuration Algorithms. In *Proc. of the 9th Intl. Workshop on Database and Expert Systems Applications*, pages 218–223, 1998.

[18] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):439–450, 1991.

[19] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self Maintainable for Data Warehousing. In *Proc. of the 4th Intl. Conf. on Parallel and Distributed Information Systems*, 1996.

[20] D. J. Rosenkrantz and H. B. Hunt. Processing Conjunctive Predicates and Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 64–72, 1980.

[21] 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.

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

[23] N. Roussopoulos. The Incremental Access Method of View Cache: Concepts Algorithms and Cost Analysis. *ACM Transactions on Database Systems*, 16(3):535–563, 1991.

[24] P. Scheurmann, J. Shim, and R. Vingralek. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In *Proc. of the 22nd Intl. Conf. on Very Large Data Bases*, pages 51–62, 1996.

[25] T. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2):175–185, 1988.

[26] T. K. Sellis. Multiple Query Optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[27] K. Shim, T. K. Sellis, and D. Nau. Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *Data & Knowledge Engineering*, 12:197–222, 1994.

[28] X.-H. Sun, N. Kamel, and L. Ni. Solving Implication Problems in Database Applications. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 185–192, 1989.

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

[30] D. Theodoratos and T. Sellis. Designing Data Warehouses. *Technical Report, Knowledge and data Base Systems Laboratory, Electrical and Computer Engineering Dept., National Technical University of Athens*, pages 1–24, 1998.

[31] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independence. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases*, pages 367–378, 1994.

[32] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[33] J. Widom, editor. *Data Engineering, Special Issue on Materialized Views and Data Warehousing*, volume 18(2). IEEE, 1995.

[34] J. Widom. Research Problems in Data Warehousing. In *Proc. of the 4th Intl. Conf. on Information and Knowledge Management*, pages 25–30, Nov. 1995.

[35] 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.

```
   begin
(0)   compute cost(s_0) and size(s_0);
      s_1 = s_0;
(1)   open = {s_1}; closed = ∅;
      while (open ≠ ∅)
         consider a state s in open;
         if distance(s_1, s) < r then
            for every transition T(s, s')
               if s' ∉ open ∪ closed then
                  incrementally compute cost(s') and size(s');
                  open = open ∪ {s'}
               endif
            endfor
         endif;
         open = open − {s}; closed = closed ∪ {s}
      endwhile;
      if closed = ∅ then return "no solution" endif;
      closed' = the set of states s ∈ closed s.t. size(s) ≤ t;
      if closed' = ∅ then
         s_1 = a state s ∈ closed having minimal size(s);
         go to (1)
      endif;
      s_2 = a state s ∈ closed' having minimal cost(s);
      s_g = s_2;
(2)   open = {s_2}; closed = ∅;
      while (open ≠ ∅)
         consider a state s in open;
         if distance(s_2, s) < r then
            for every transition T(s, s')
               if s' ∉ open ∪ closed then
                  incrementally compute cost(s') and size(s');
                  if size(s') ≤ t then open = open ∪ {s'} endif
               endif
            endfor
         endif;
         open = open − {s}; closed = closed ∪ {s}
      endwhile;
      if closed ≠ ∅ then
         s_2 = a state s ∈ closed having minimal cost(s);
         if s_2 ≤ s_g then s_g = s_2;
         go to (2)
      endif;
      return s_g
   end.
```

Fig. 6. A greedy incremental algorithm