Detecting redundancy in Data Warehouse Evolution

Dimitri Theodoratos[⋆]

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

Abstract. A Data Warehouse (DW) can be abstractly seen as a set of materialized views defined over a set of remote data sources. A DW is intended to satisfy a set of queries. The views materialized in a DW relate to each other in a complex manner, through common subexpressions, in order to guarantee high query performance and low view maintenance cost. DWs are time varying. As time passes new materialized views are added in order to satisfy new queries or for performance reasons while old queries are dropped. The evolution of a DW can result in a redundant set of materialized views.
In this paper we address the problem of detecting redundant views in a given DW view selection, that is, views that can be removed from the DW without negatively affecting the query evaluation or the view maintenance process. Using an AND/OR dag representation for multiple queries and views, we first provide a method for detecting materialized views that are not needed in the process of propagating source relation changes to the DW. Then, we use this method to detect materialized views that are redundant. As a side effect, our approach shows how source relation changes can be propagated to the DW materialized views by exploiting common subexpressions between views and by using other materialized views that are not affected by these changes.

1 Introduction

A data warehouse (DW) is a "subject-oriented, integrated, time-varying, non-volatile collection of data that is used primarily in organizational decision making" [8]. DWs are designed for answering the queries of data-workers and analysts that are needed for decision support. A DW provides integrated access to multiple, distributed, possibly heterogeneous databases and other information sources: selected information from each source is extracted in advance, translated and filtered as appropriate, merged with relevant information from other

[⋆] Research supported by the European Commission under the ESPRIT Program LTR project "DWQ: Foundations of Data Warehouse Quality" sources and stored in a repository. Current DWs contain very large volumes of data and support On-Line Analytical Processing (OLAP).

A DW can be seen as a set of *materialized views* defined over source relations. Some views may also be defined over other views. In order to ensure high query performance, the queries to the DW are answered by completely rewriting them [9] over the views stored in the DW. DW maintenance is performed by propagating source relation changes to the materialized views that are affected by these changes.

View maintenance. Usually the changes to the views are computed *incrementally* as opposed to their recomputation from scratch. In an incremental strategy, the changes to the views are computed using the changes to the source relations [4, 2]. In order to compute the changes to the materialized views it may be necessary to issue queries against the data sources. We call these queries *maintenance queries*. When the source relation changes affect more than one materialized view, multiple maintenance queries are issued against the source relations for evaluation. The techniques of *multiquery optimization* [14] can be used to detect "common subexpressions" between these queries and to derive a global plan whose evaluation process is more efficient than a plan that evaluates each query separately. The evaluation cost of a maintenance query can reduced if this query can be partially rewritten using a view already materialized. Often, materialized views are added to the DW exclusively for this purpose and are called *auxiliary views* [12, 18]. By appropriately selecting auxiliary views to materialize in the DW, it is possible to maintain the initial materialized views and the auxiliary views altogether, for any source relation change, without issuing queries against the source relations. Such a view set is called *self-maintainable* [11].

DW design. The design of a DW involves selecting a set of views to materialize based on the set of queries that the DW need to satisfy. This choice is subject to a number of constraints and requirements. From a user viewpoint the query response time and the currency of the answer data for each query are of interest. These time periods should not exceed a certain limit specified by the user, or the data returned may be of no use. From a system viewpoint, the overall query evaluation cost and the overall view maintenance cost (or a combination of them), and the space needed for materializing the views are of interest.

DW evolution. DWs are time-varying. They are dynamic entities that evolve in time both in terms of schema and content.

As time passes new materialized views need to be added to a DW. There are three reasons for adding new materialized views:

(a) New queries need to be satisfied by the DW. Then, new views are added that together with the existing views are able to answer the new queries [19].
(b) Higher query response time is required for some queries. By adding new (more complex) views and by rewriting these queries using the new views the response time of the queries can be improved [1].
(c) The view maintenance cost is high and the view maintenance process delays the evaluation of the queries; the query response time is unsatisfactory when

an "at query time" deferred maintenance policy is adopted (that is, a view is updated when a query involving this view is issued against the DW [13]); the currency of the answer data [15] is unsatisfactory or makes this data useless due to the excessive time needed to maintain the views over which the query is evaluated. By adding appropriately selected materialized views to the DW, the view maintenance cost can be reduced and/or the currency of answer data can be improved.

Also, as time passes, queries that the DW used to satisfy may become useless, either because the analysts do not need them anymore or because they are replaced by other more useful (and possibly more complex) queries.

1.1 The problem

The augmentation of the DW by new views and the suppression of some queries can result in a materialized view set that contains redundant views. This can also happen with the initial design of the DW. Given a set of queries that are operational with the current design of a DW, a view V is redundant if:
(a) V is not used in the optimal evaluation of a query, and
(b) V is not used as an auxiliary view in the optimal propagation of source relation changes to the materialized views that are used for optimally evaluating the queries.

In this paper we address the problem of detecting redundant materialized views in a DW. This problem is complex because queries and views relate to each other through common subexpressions. For instance, the suppression of a query Q may suggest that a view V that was used for optimally answering Q is redundant. However, this is not true if V appears in the optimal evaluation plan for an operational query or if it is used as an auxiliary view to support the optimal maintenance of a non-redundant view.

Removing redundant views from the DW: (a) improves the availability of the system: no changes need to be applied to these views and the computation of changes for these views is possibly avoided, and (b) frees valuable space which can be used for storing other views and access structures that improve the evaluation of the queries and the maintenance of the views.

It is worth noting that "redundancy" does not refer to the intended redundancy in a DW for performance reasons.

1.2 Contribution and outline

The main contributions are the following:

- We formalize the problem of detecting redundant materialized views in a DW based on a marked AND/OR dag representation for multiple queries and views. This formalization considers a large class of queries and views including grouping/aggregation queries, and applies to a generic DW architecture.
- Using multiquery AND/OR dags, we show how optimal query evaluation plans over the materialized views can be determined. We also show how the propagation of source relation changes to the materialized views can be

performed by taking into account common subexpressions between the views and by using materialized views that are not affected by these changes.
– We provide a procedure for determining materialized views that are useless in the propagation of source relation changes to the materialized views.
– Finally, we present a method for detecting redundant views in a given materialized view selection intended to satisfy a set of queries.
– Our approach is independent of the cost model used for evaluating queries and computing changes to the materialized views.

The rest of the paper is organized as follows. The next section presents related work. In Section 3, we specify the class of queries and views considered, and we introduce multiquery AND/OR dags. We base our analysis on a DW system architecture and operation presented in Section 4. In Section 5, we show how useless materialized views in the propagation of source relation changes can be determined. A method for detecting redundant materialized views in a DW is presented in Section 6. Finally, Section 7 contains concluding remarks.

2 Related Work

We are not aware of any research work dealing with the issue of non-intended redundancy in the design of a DW.

Answering queries using views has been studied in many papers, e.g. [9]. Materialized view maintenance has been addressed in recent years by a plethora of researchers. A number of papers dealing with different aspects of materialized view maintenance are cited in the introduction and in next sections. A nice overview of incremental view maintenance issues is provided in [5].

View selection problems for Data Warehousing 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 a combination of both, possibly in the presence of some constraints. Given a materialized SQL view, [12] presents an exhaustive approach as well as heuristics for selecting auxiliary views that minimize the total view maintenance cost. In [6] greedy algorithms are provided for selecting views to materialize that minimize the query evaluation cost under a space constraint. A solution for selecting views that minimize the combined cost is given in [20]. A variation of the DW design problem endeavoring to select a set of views that minimizes the query evaluation cost under a total maintenance cost constraint is adopted in [7].

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 [17] where the problem of configuring a DW without space restrictions is addressed for a class of select-join queries. This work is extended in [18] in order to take into account space restrictions, multiquery optimization over the maintenance queries, and the use of auxiliary views when maintaining other views. Another extension of [17] deals with the same problem for a class of PSJ queries under space restrictions [16], while [19] addresses an incremental version of the DW design problem (dynamic DW design).

3 Multiexpression AND/OR dags

In this section we define multiexpression AND/OR dags and their derivatives: multiquery AND/OR dags, query evaluation dags and change propagation dags. We start by specifying the class of queries and views considered here.

3.1 Class of queries and views

We adopt a natural extension of the relational algebra operations to bags (multisets). This algebra allows defining queries and views that have the SQL bag semantics. It contains the following operators: σ_C, where C is a selection condition (selection), Π_X where X is a set of attributes (projection), $\uplus$ (additive union), $\dot{-}$ (monus), min (minimal intersection), max (maximal union), ϵ (duplicate elimination), $\times$ (Cartesian product), $\bowtie_C$ where C is a join condition (conditional join), $\bowtie$ (natural join). The algebra includes also a grouping/aggregation operator: $\pi_{X,\, agg_1(A_1)\, \text{as}\, B_1,\, \ldots,\, agg_k(A_k)\, \text{as}\, B_k}$ where X is a set of grouping attributes, agg_i is an aggregation function on attribute A_i, and B_i is a new name for the corresponding aggregated attribute. This operator is the generalized projection operator [3] extended to account for the naming of the aggregate attributes. As in SQL, the aggregation function agg can be `sum`, `avg`, `count`, `max`, `min`. Including a grouping/aggregation operator in the algebra is necessary in order to handle queries used in OLAP and Decision Suppost System (DSS) applications. The semantics of the previous operators are well known [2, 3]. Queries and views considered here are expressions formed with these operators and relation or view names.

3.2 Multiquery AND/OR dags

Alternative ways for evaluating an expression of those considered here can be compactly represented by an *AND/OR dag* [6]. A particular representation of AND/OR dags distinguishes between AND nodes and OR nodes [12]. We use here this representation for multiple queries, extended with marked nodes to account for views materialized at the DW [19, 15].

Definition 1. An *expression AND/OR dag for an expression e* defined over a set of views (and/or relations) $\mathbf{V}$ is a rooted bipartite dag $\mathcal{G}_e$. The nodes of $\mathcal{G}_e$ are partitioned in AND nodes and OR nodes. An AND node is called an *operation node* and is labeled by an operator, while an OR node is called a *view node* and is labeled by a view. In the following we may identify nodes with their labels. An operation node has one or two outgoing edges to view nodes and one incoming edge from a view node. A view node has one or more outgoing edges (if any) to operation nodes and one or more incoming edges (if any) from an operation node. The root node and sink nodes of $\mathcal{G}_e$ are view nodes. The root node is labeled by e, while the sink nodes are labeled by views in $\mathbf{V}$.

Given a set of expressions $\mathbf{E}$ defined over a set of views $\mathbf{V}$, a *multiexpression AND/OR dag $\mathcal{G}$ for* $\mathbf{E}$ is an AND/OR dag resulting by merging the expression AND/OR dags for the expressions in $\mathbf{E}$. $\mathcal{G}$ is not necessarily a rooted dag (that is it does not necessarily have a single root). All the root nodes of $\mathcal{G}$ are view nodes labeled by expressions in $\mathbf{E}$ (but not all the expressions in $\mathbf{E}$ label necessarily root nodes). In addition, view nodes in a (multi)expression AND/OR dag *can be marked*. Marked nodes represent views materialized at the DW. □

We can now define query and multiquery AND/OR dags.

Definition 2. A *query AND/OR dag for a query* Q defined over a set of views and/or relations $\mathbf{V}$ is an expression AND/OR dag for Q.

A *multiquery AND/OR dag* for a set of queries $\mathbf{Q}$ defined over $\mathbf{V}$ is an expression AND/OR dag for $\mathbf{V}$. The view nodes representing (and labeled by) the queries in $\mathbf{Q}$ are called *query nodes.* □

Example 1. Consider the set of views $\mathbf{V} = \{V_1, V_2, V_3\}$. Suppose that the schemes of their materializations are as follows: $V_1(A, B)$, $V_2(A, C)$ and $V_3(\underline{A}, B)$. An underlined attribute denotes the key of the corresponding view materialization. Consider also the set of queries $\mathbf{Q} = \{Q_1, Q_2, Q_3\}$ over $\mathbf{V}$ where:
$Q_1 = \pi_{A,\, \texttt{max}(B)\text{ as }G}(\sigma_{B>10}(V_1) \uplus \Pi_{AB}(V_2 \bowtie V_3))$,
$Q_2 = \sigma_{A<'c' \wedge E \leq B}(\pi_{A,\, \texttt{count}(B)\text{ as }D,\, \texttt{sum}(B)\text{ as }E}(V_2 \bowtie V_3))$, and
$Q_3 = \sigma_{E \geq B+10}(\pi_{A,\, \texttt{count}(B)\text{ as }D,\, \texttt{sum}(B)\text{ as }E}(\sigma_{A<'c'}(V_2) \bowtie V_3)))$.

In Figure 1 a multiquery AND/OR graph for $\mathbf{Q}$ over $\mathbf{V}$ is shown. Operation nodes are depicted by small circles while view nodes are depicted by larger ones. $\pi_{A,\, \texttt{count}(B)\text{ as }D,\, \texttt{sum}(B)\text{ as }E}$ is abbreviated as π. Marked nodes (materialized views) are depicted by filled black circles. □

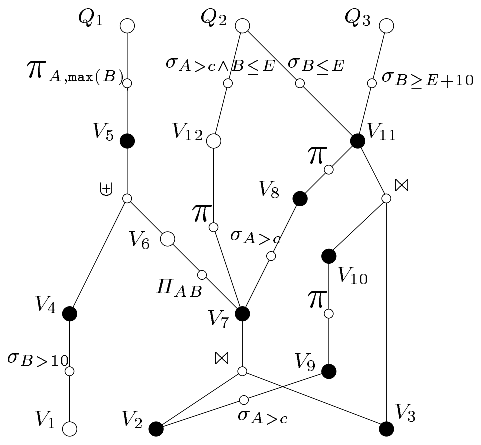

Fig. 1. A multiquery AND/OR dag for $\mathbf{Q} = \{Q_1, Q_2, Q_3\}$

Additional auxiliary definitions are provided below.

Definition 3. A *(multi)expression dag* is a (multi)expression AND dag (that is a (multi)expression AND/OR dag such that no view node has more than one outgoing edge).

A *subdag* $\mathcal{G}'$ of a (multi)expression AND/OR dag $\mathcal{G}$ is a (multi)expression AND/OR subdag of $\mathcal{G}$ such that:
(a) if an operation node of $\mathcal{G}$ is in $\mathcal{G}'$, all its child view nodes in $\mathcal{G}$ are in $\mathcal{G}'$,
(b) edges in $\mathcal{G}$ between nodes that are in $\mathcal{G}'$ are present in $\mathcal{G}'$, and
(c) All and only the marked nodes in $\mathcal{G}$ that are present in $\mathcal{G}'$ are marked nodes in $\mathcal{G}'$. □

3.3 Query evaluation and change propagation dags

Consider a multiquery AND/OR dag $\mathcal{G}$ for a set of queries $\mathbf{Q}$. A query evaluation plan over materialized views is represented by a query evaluation dag defined as follows.

Definition 4. A *query evaluation dag for a query* $Q \in \mathbf{Q}$ is an AND subdag $\mathcal{G}_Q$ of $\mathcal{G}$ such that:
(a) $\mathcal{G}_Q$ is rooted at Q.
(b) All and only the sink nodes of $\mathcal{G}_Q$ are marked nodes. □

Example 2. Consider the multiquery AND/OR of example 1. Figure 5 shows query evaluation dags for the queries Q_1, Q_2 and Q_3. □

A change propagation plan is represented by a change propagation dag defined below.

Definition 5. A *change propagation dag for a sink node* V *in* $\mathcal{G}$ is an AND subdag $\mathcal{G}_V$ of $\mathcal{G}$ such that:
(a) All the marked view nodes that are ancestor nodes of V in $\mathcal{G}$ (that is the marked view nodes that occur in a path from a root node to V in $\mathcal{G}$) are present in $\mathcal{G}_V$, and the root nodes of $\mathcal{G}_V$ are among them.
(b) A sink node in $\mathcal{G}_V$ is a marked node of $\mathcal{G}$, or node V. Node V is a sink node.
(c) The non-sink marked nodes in $\mathcal{G}_V$ are ancestor nodes of V. □

Example 3. Consider the multiquery AND/OR dag of example 1. Figure 2 shows

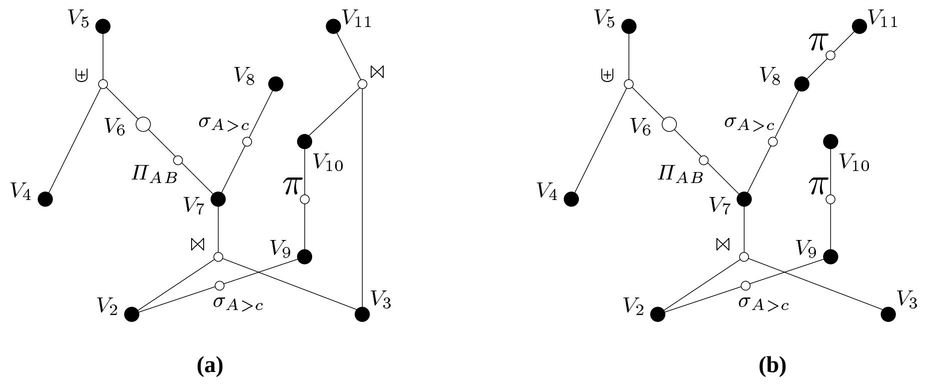

Fig. 2. Change propagation dags for the sink node V_2

two different change propagation dags for the sink node V_2. Figure 6 shows change propagation dags for the sink nodes V_1, V_2 and V_3 respectively. □

4 DW system framework

Our approach is based on a DW system whose basic architecture is depicted in Figure 3. The central component is the DW that contains a set of materialized views. Materialized views in the DW are depicted by small rectangles. The DW is built to satisfy a set of queries. These queries are issued by knowledge workers and analysts depicted at the top of the diagram. They are evaluated locally at the DW without accessing the remote data sources. Therefore, the DW contains materialized views that allow a complete rewriting of the queries over them. The bottom of the diagram shows the remote data sources that contain the source relations.

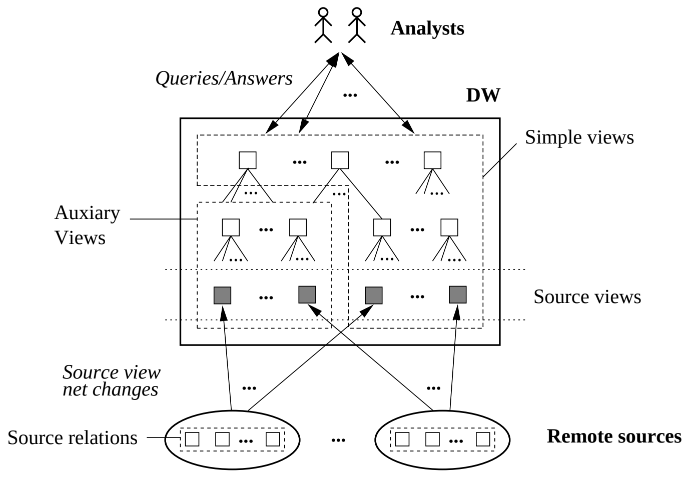

Fig. 3. A DW system architecture

Source views. There is a set **V** of views each defined over the source relations of the same data source such that all the queries and the rest of the materialized views can be completely rewritten over **V**. These views are called *source views*. Source views can be, for instance, source relations, select-project views over a source relation or even views defined over more than one relation from the same data source. In Figure 3 source views are depicted at the bottom of the DW component by small gray rectangles. Note though that source views may or may not be materialized views.

View maintenance. The DW views are maintained incrementally. We assume that the sources are aware of the corresponding source view definitions (if the latter are different than source relations) and are able to compute and send the changes to be applied to the source views, upon request from the DW or triggered by events. When the computation of the changes to a source view is performed at the source, transmission of useless change data is avoided and the availability of the DW is increased.

Changes for different source views are transmitted to the DW asynchronously. Upon arrival at the DW these changes are propagated to the affected materialized views. The views affected by the changes to a source view V are those that are defined using V. The affected views can be maintained separately. However, this process can be performed more efficiently if we propagate the source view changes to all the affected views together by exploiting (a) common subexpressions between the affected views and (b) other views materialized in the DW. Change propagation dags are used for this purpose. Their use in propagating changes to all the affected views together is described in the next section.

Type of changes. Concerning the type of changes these can be insertions and deletions (modifications are modeled by deletions followed by insertions). In order to avoid wasteful insertions and deletions (and data transmissions), when incrementally maintaining a materialized view, we consider only the changes actually inserted to or deleted from each view: the bag of tuples to be deleted from and that to be inserted to a view do not have any tuple in common, and a tuple to be deleted from a view occurs in the view materialization at least as many times as in the bag of tuples to be deleted from the view. These changes are called net changes. In the following by 'changes' we mean 'net changes'.

Simple and auxiliary materialized views. The materialized views that are used for optimally answering the queries are called *simple views*. The rest of the materialized views are used for reducing the view maintenance cost of other materialized views, and are called *auxiliary views*. Simple views too can be used in the same manner, yet they have to appear in the optimal query evaluation plan of a query. A source view that is materialized can be either simple or auxiliary.

Self-maintainability. We assume that the set of materialized views in the DW is self-maintainable. Therefore, no maintenance queries against the source relations are needed for maintaining the materialized views.

5 Detecting useless views in a change propagation dag

In this section we first show how changes to a source view can be propagated to the affected materialized views using change propagation dags. Then, we provide a procedure for detecting useless views in a change propagation dag.

5.1 Propagating changes using change propagation dags

The changes to be applied to the materialized views that are affected by the changes to a source view V can be computed separately for each affected view. The maintenance expressions provided in [2, 10] allow for this computation. Consider a DW that satisfies a set of queries $\mathbf{Q}$. Let $\mathbf{V}$ be the set of source views, and $\mathcal{G}$ be a multiquery AND/OR dag for $\mathbf{Q}$ over $\mathbf{V}$ such that all the materialized views in the DW are marked view nodes in $\mathcal{G}$. $\mathcal{G}_V$ denotes a change propagation dag for a source view $V \in \mathbf{V}$. Recall that $\mathcal{G}_V$ includes all the materialized views that are affected by the changes to V. These views are ancestor view nodes of V in $\mathcal{G}$.

Changes are propagated in $\mathcal{G}_V$ bottom-up. This process starts with the source view node view V and then considers all its ancestor view nodes. Using the maintenance expressions in [2, 10], changes to a view node are computed using the changes to its child view node(s). These expressions involve in general the pre-update state of the child view nodes (that is their state prior to the application of the changes), the pre-update state of the view node, and the changes to the child view nodes.

When the changes to a view node V' in $\mathcal{G}_V$ are computed, they are applied to it if V' is a marked node (materialized view). However, if the pre-update state of V' is needed in $\mathcal{G}_V$, this application is postponed until the changes and the pre-update state (if needed) of the parent view nodes of V' are computed. This way, the pre-update state of V' remains available when needed.

5.2 A procedure for detecting useless views

When computing the changes to a view node using the changes to its child view node(s) in a change propagation dag, the pre-update state of this view node and/or the pre-update state of its child view node(s) may not be needed.

Table 1 shows whether the pre-update state of a view V and the pre-update state of its child view node(s) V_1 (and V_2) are needed in the computation of the net changes to V, due to changes to V_1, for different instances of the child operation node of V. These results can be derived from the maintenance expressions provided in [2, 10]. Two cases for the monus operation are depicted ($V_1 \dot{-} V_2$ and $V_2 \dot{-} V_1$) since $\dot{-}$ is not a commutative operation. If V is a marked node, its pre-update state is available. Note that the table shows the pre-update states needed for the computation of both insertions and deletions. For instance, in the case of the monus operation $V_1 \dot{-} V_2$, only the pre-update state of V is needed for the computation of the deletions, while only the pre-update state of V_1 and V_2 is needed for the computation of the insertions to V; therefore both pre-update states are shown as needed in Table 1. Note also that in the case of the grouping/aggregation operation $\pi_{X,\, agg_1(A_1) \text{ as } B_1,\, \dots,\, agg_k(A_k) \text{ as } B_k}$ where not all the aggregate functions agg_i are `min` or `max`, we consider that a `count` aggregate function is also computed by $\pi_{X,\, agg_1(A_1) \text{ as } B_1,\, \dots,\, agg_k(A_k) \text{ as } B_k}$, if `count` does not already appear in it. Further, since `avg` is computed in terms of `sum` and `count`, if $\pi_{X,\, agg_1(A_1) \text{ as } B_1,\, \dots,\, agg_k(A_k) \text{ as } B_k}$ includes an `avg` function, we assume that it also includes the corresponding function for `sum`.

If a view node is a marked node, its pre-update state is available. The pre-update state of a non-marked view node V is computed from the pre-update state of its child view node(s). This computation is not necessary if the pre-update state of V is not needed neither for the computation of the changes to V nor by any of its parent view nodes. A parent view node of V may need the pre-update state of V for the computation of its own changes or (in case it is a non-marked view) because its own pre-update state is needed by one of its parent view nodes. Given a change propagation dag $\mathcal{G}_V$, we call *useless views* the view nodes in $\mathcal{G}_V$ whose pre-update state need not be computed.

Procedure *useless_view_nodes*, outined in Figure 4, computes the useless view nodes in a given change propagation dag $\mathcal{G}_V$. In this process it uses the infor-

V	pre-update state of V	pre-update state of V_1	pre-update state of V_2
$\sigma_C(V_1),\ \Pi_X(V_1)$	no	no	-
$\epsilon(V_1)$	no	yes	-
$\pi_{X,\, agg_1(A_1)\text{ as }B_1,\ldots,\, agg_k(A_k)\text{ as }B_k}(V_1)$ where $agg_i =$ `sum`, `avg`, or `count`, $i = 1, \ldots, k$	yes	no	-
$\pi_{X,\, agg_1(A_1)\text{ as }B_1,\ldots,\, agg_k(A_k)\text{ as }B_k}(V_1)$ where $agg_i =$ `min` or `max`, $i \in [1, k]$	yes	yes	-
$V_1 \uplus V_2$	no	no	no
$V_1 \times V_2,\ V_1 \bowtie_C V_2,\ V_1 \bowtie V_2$	no	no	yes
$V_1 \mathrm{min} V_2,\ V_1 \mathrm{max} V_2,$	no	yes	yes
$V_1 \dot{-} V_2,\ V_2 \dot{-} V_1$	yes	yes	yes

Table 1. The need of the pre-update states of a view node V and of its child view node(s) V_1 (and V_2) for the computation of the net changes to V due to changes to V_1.

mation provided in Table 1. *useless_view_nodes* proceeds in a top-down manner, from the root nodes of $\mathcal{G}_V$ to the sink nodes. It uses two set variables C and U. Variable C holds the view nodes in $\mathcal{G}_V$ that have already been considered, and is initially set to be the empty set. It is useful for not reconsidering a view node in $\mathcal{G}_V$ (and the view nodes in a sudbag of $\mathcal{G}_V$ rooted at this node) when this is not necessary. Variable U holds the view nodes in $\mathcal{G}_V$ that are useless, and is initially set to contain the set of all the view nodes in $\mathcal{G}_V$. Procedure *useless_view_nodes* uses a recursive procedure, *check_needed*. *check_needed* is called on a view node and recursively calls itself on the child view nodes of this node. Initially, it is called by *useless_view_nodes* on the root nodes of $\mathcal{G}_V$. *check_needed* takes two arguments, W and P. W denotes the node on which *check_needed* is called, and P indicates whether W is needed for the computation of the changes to the parent view node of W from which it is called. The role of *check_needed* is to check whether W is needed given P. Note that a sink node in $\mathcal{G}_V$ is not needed for the computation of its own changes. Indeed, a sink node either it is not an ancestor node of V in $\mathcal{G}_V$ (and thus it is not affected by the changes to V), or it is node V. In the last case, V is not needed because we have considered that the corresponding source transmits to the DW the net changes to be applied to V. We assume that information on whether a view node is an ancestor node of V is kept with every node in $\mathcal{G}_V$. Also, we assume that the outgoing edges of a monus operation node are ordered in $\mathcal{G}_V$, in order to deal with the non-commutativity of this operation.

6 Detecting redundant views

Using the results of the previous section we can now detect redundant views in a DW. Consider a DW that satisfies a set of queries $\mathbf{Q}$ over a set of source views $\mathbf{V}$. A multiquery AND/OR dag $\mathcal{G}$ for this DW is a multiquery AND/OR dag for $\mathbf{Q}$ over $\mathbf{V}$ such that all the materialized views in the DW are marked view nodes in $\mathcal{G}$. Procedure *redundant_views*, presented below, proceeds in three steps, and computes a set of redundant views in $\mathcal{G}$.

Input: a change propagation dag $\mathcal{G}_V$.
Output: the set U of useless view nodes in $\mathcal{G}_V$.

```
begin
    $C := \emptyset$; $U := \{V' | V'$ is a view node in $\mathcal{G}_V\}$;
    for every root view node $V_r$ in $\mathcal{G}_V$ do
        check_needed($V_r$, true)
    endfor;
    return $U$
end.

procedure check_needed($W$, $P$);
begin
    if $P =$ true or $W$ is needed for the computation of its own changes then
        if $W \notin C$ then
            $C := C \cup \{W\}$; $U := U - \{W\}$;
            if $W$ is marked then
                for every child view node $W'$ of $W$ do
                    if $W'$ is needed for the computation of the changes to $W$ then
                        check_needed($W'$, true) else check_needed($W'$, false)
                    endif
                endfor
                else        /* $W$ is not marked */
                    for every child view node $W'$ of $W$ do
                        check_needed($W'$, true)
                    endfor
            endif
            else        /* $W \in C$ */
                if $W \in U$ then
                    $U := U - \{W\}$;
                    if $W$ is not marked then
                        for every child view node $W'$ of $W$ do
                            check_needed($W'$, true)
                        endfor
                    endif
                endif
        endif
        else        /* $P =$ false and $W$ is not needed for the computation of its own changes */
            if $W \notin C$ then
                $C := C \cup \{W\}$;
                for every child view node $W'$ of $W$ do
                    if $W'$ is needed for the computation of the changes to $W$ then
                        check_needed($W'$, true) else check_needed($W'$, false)
                    endif
                endfor
            endif
    endif
end;
```

Fig. 4. Procedure *useless_view_nodes*

Procedure *redundant_views.*

Input: A multiquery AND/OR dag $\mathcal{G}$.
Output: A set $\mathbf{R}$ of redundant views in $\mathcal{G}$.

```
1. - Compute the optimal query evaluation dags for the query nodes in $\mathcal{G}$.
   - Determine the simple views in $\mathcal{G}$. (Recall that a view is simple if it is a
     sink node of an optimal query evaluation dag.)
2. - Compute the optimal change propagation dag $\mathcal{G}_{V_i}$ for every source view
     $V_i$ in $\mathcal{G}$.
   - For every $\mathcal{G}_{V_i}$ compute a subdag $\mathcal{G}'_{V_i}$ of $\mathcal{G}_{V_i}$ by removing from $\mathcal{G}_{V_i}$ all the
     nodes and edges that are not on a path from a simple view in every $\mathcal{G}_{V_i}$.
3. - For every $\mathcal{G}'_{V_i}$ use procedure useless_view_nodes to compute the set $U_i$ of
     useless view nodes in $\mathcal{G}'_{V_i}$.
   - Let $M_i$ be the set of marked view nodes of $\mathcal{G}$ in $U_i$ that are not simple
     views.
   - Add to every $M_i$ all the marked view nodes in $\mathcal{G}$ that are not in $\mathcal{G}'_{V_i}$ and
     are not simple views.
   - Return $\mathbf{R} = \cap_i M_i$                                              □
```

Example 4. Consider the multiquery AND/OR dag $\mathcal{G}$ for a DW shown in Figure 1. We apply procedure *redundant_views* to $\mathcal{G}$.

Suppose that step 1 computes the optimal query evaluation dags depicted in Figure 5. Then, the simple view nodes in $\mathcal{G}$ are V_5 and V_{11}. Suppose that

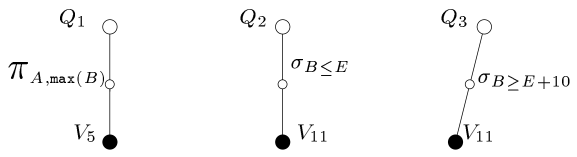

Fig. 5. Optimal query evaluation dags for the queries Q_1, Q_2 and Q_3.

step 2 computes the optimal change propagation dags $\mathcal{G}_{V_1}$, $\mathcal{G}_{V_2}$, and $\mathcal{G}_{V_3}$ for the source views V_1, V_2 and V_3 respectively, depicted in Figure 6. Consider the optimal change propagation dags $\mathcal{G}_{V_2}$ and $\mathcal{G}_{V_3}$. The node V_8 along with the edges on the path from V_8 to V_7 do not appear in a path from a simple view in $\mathcal{G}_{V_2}$ and $\mathcal{G}_{V_3}$ and do not appear at all in $\mathcal{G}_{V_1}$. Therefore, this node and these edges are removed from $\mathcal{G}_{V_2}$ and $\mathcal{G}_{V_3}$ before proceeding to step 3. In step 3, procedure *useless_view_nodes* on the transformed change propagation dags first computes the sets $U_1 = \{V_1, V_4, V_5, V_6, V_7\}$, $U_2 = \{V_2, V_4, V_5, V_6, V_7, V_9, V_{11}\}$, and $U_3 = \{V_3, V_4, V_5, V_6, V_7, V_{11}\}$. Then, the sets $M_1 = \{V_2, V_3, V_4, V_7, V_8, V_9, V_{10}\}$, $M_2 = \{V_2, V_4, V_7, V_8, V_9\}$, and $M_3 = \{V_3, V_4, V_7, V_8, V_9\}$ are computed. Finally, the set $\mathbf{R} = \{V_4, V_7, V_8, V_9\}$ of redundant views is returned. □

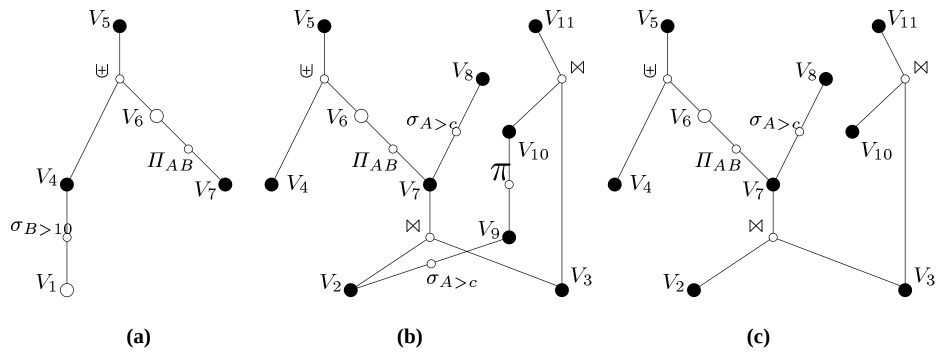

Fig. 6. Optimal change propagation dags for the source views (a) V_1, (b) V_2, and (c) V_3

In step 2 of the procedure *redundant_views*, auxiliary views that are not on a path from a simple view in any optimal change propagation dag are not considered since they cannot be used in propagating changes to a simple view.

A view $V' \in \mathbf{R}$ is a materialized view such that V' is not a simple view and: (a) for every optimal change propagation dag $\mathcal{G}_V$, V' is useless in $\mathcal{G}_{V'}$ or V' does not appear in $\mathcal{G}_V$, or (b) V' cannot be used in propagating changes to a simple view. Therefore, the following proposition can be shown.

Proposition 1. *Procedure redundant_views, applied to a multiquery AND/OR dag for a DW, correctly computes the set* $\mathbf{R}$ *of redundant views in this DW.* $\square$

The approach presented above for detecting redundant views is independent of the cost model used. The computed set of redundant views though depends on this model.

7 Conclusion

In this paper we have addressed the problem of detecting redundant views in a DW. This problem is evolved because queries and views relate to each other in a complex manner through common subexpressions. To this end, we have used a marked AND/OR dag representation for multiple queries and views. We have considered a large class of queries and views including grouping /aggregation queries that are extensively used in DW applications. We provided a procedure for determining materialized views that are useless in the propagation process of source relation changes to the materialized views. This procedure is used for designing a method that detects redundant views in a given materialized view selection. Our approach applies to a generic DW system framework and does not assume a specific query evaluation and view maintenance cost model. We have also shown how the propagation of source relation changes to the materialized views can be performed by taking into account common subexpressions between the views and by using other views materialized in the DW.

An interesting extension of the present work would consider not only the optimal but also non-optimal change propagation plans in order to specify a set of views whose removal from the DW minimizes the overall view maintenance cost. This approach though may increase the cost of specific change propagation plans and it is not appropriate when data currency constraints are imposed.

References

[1] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing Queries with Materialized Views. In *Proc. of the 11th ICDE*, pages 190–200, 1995.

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

[3] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proc. of the 21st VLDB*, pages 358–369, 1995.

[4] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1993.

[5] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *Data Engineering*, 18(2):3–18, 1995.

[6] 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.

[7] 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.

[8] W. Inmon. *Building the Data Warehouse.* John Wiley & Sons, 2nd edition, 1996.

[9] 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.

[10] D. Quass. Maintenance Expressions for Views with Aggregation. In *Workshop on Materialized Views: Techniques and Applications*, pages 110–118, 1996.

[11] 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*, pages 158–169, 1996.

[12] 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.

[13] N. Roussopoulos and Y. Kang. Principles and techniques in the design of ADMS±. *Computer*, Dec 1986.

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

[15] D. Theodoratos and M. Bouzeghoub. Data Currency Quality Factors in Data Warehouse Design. In *Proc. of the Intl. Workshop on Design and Management of Data Warehouses,* Heidelberg, Germany, June 1999.

[16] D. Theodoratos, S. Ligoudistianos, and T. Sellis. Designing the Global Data Warehouse with SPJ Views. In *Proc. of the 11th Intl. Conf. on Advanced Information Systems Engineering, Springer-Verlag, LNCS No 1626*, pages 180–194, 1999.

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

[18] D. Theodoratos and T. Sellis. Data Warehouse Schema and Instance Design. In *Proc. of the 17th Intl. Conf. on Conceptual Modeling, Springer LNCS 1507*, pages 363–376, 1998.

[19] D. Theodoratos and T. Sellis. Dynamic Data Warehouse Design. In *Proc. of the 1st Intl. Conf. on Data Warehousing and Knowledge Discovery, Springer-Verlag, LNCS*, 1999.

[20] 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.