Shrinking the Warehouse Update Window

Wilburt Juan Labio, Ramana Yerneni, Hector Garcia-Molina
Department of Computer Science
Stanford University
{wilburt,yerneni, hector}@cs.stanford.edu

Abstract

Warehouse views need to be updated when source data changes. Due to the constantly increasing size of warehouses and the rapid rates of change, there is increasing pressure to reduce the time taken for updating the warehouse views. In this paper we focus on reducing this "update window" by minimizing the work required to compute and install a batch of updates. Various strategies have been proposed in the literature for updating a single warehouse view. These algorithms typically cannot be extended to come up with good strategies for updating an entire set of views. We develop an efficient algorithm that selects an optimal update strategy for any single warehouse view. Based on this algorithm, we develop an algorithm for selecting strategies to update a set of views. The performance of these algorithms is studied with experiments involving warehouse views based on TPC-D queries.

1 Introduction

Data warehouses derive data from remote information sources in support of on-line analytical processing (OLAP). One of the main problems is updating the derived data when the remote information sources change. During a warehouse update, called the "update window," either OLAP queries are not processed or OLAP queries compete with the warehouse update for resources. To reduce OLAP down time or interference, it is critical to minimize the work involved in a warehouse update and shrink the update window.

The derived data at the warehouse is often stored in materialized views. Previous work ([6], [14]) has developed standard expressions for maintaining a large class of materialized views incrementally. However, there are still numerous alternative "strategies" for implementing these expressions, and these strategies incur different amounts of work and lead to different update windows.

EXAMPLE 1.1 Let us consider the warehouse depicted by the directed acyclic graph (DAG) shown in Figure 1. There are four materialized views: *CUSTOMER*, *ORDER*, *LINEITEM*, and *V*. The edge from *V* to *CUSTOMER* indicates that view *V* is defined on view *CUSTOMER* (and similarly for the other edges). Unlike *V*, the *CUSTOMER*, *ORDER* and *LINEITEM* views are defined on remote and possibly autonomous information sources.

Periodically, the changes (*i.e.*, inserted, deleted and updated tuples) of *CUSTOMER*, *ORDER* and *LINEITEM* are computed from the changes of remote information sources. View maintenance algorithms that handle remote and autonomous sources, like the ones developed in [17], may be used. Once the changes of these views are obtained, the changes of *V* need to be computed, and the changes of all the views need to be installed. There are many ways to perform these update tasks using standard view maintenance expressions.

One strategy for updating *V*, denoted *Strategy 1*, is (as in [3]):

1. Compute the changes of *V* considering at once all the changes of *CUSTOMER*, *ORDER*, *LINEITEM*, and using the prior-to-update states of these views.
2. Install the changes of all four views. Installation of changes involves removing deleted tuples and adding inserted tuples.

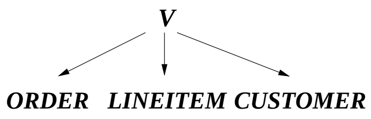

Figure 1: Example DAG of Materialized Views

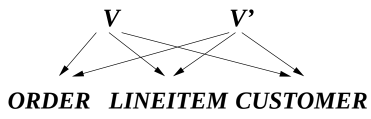

Figure 2: More Complex DAG

In *Strategy 2*, the changes of V are computed piecemeal, considering the changes of each of its base views one at a time:

1. Compute the changes of V only considering the changes of *CUSTOMER* (and the original state of the views).
2. Install the changes of *CUSTOMER*. (The following steps will see this new state.)
3. Compute the changes of V only considering the changes of *ORDER*.
4. Install the changes of *ORDER*. (This new state will be seen by the next step.)
5. Compute the changes of V only considering the changes of *LINEITEM*.
6. Install the changes of *LINEITEM*.
7. Install the changes of V.

In [8], the correctness of both these strategies was discussed. Specifically, it was shown that both strategies compute the same final "database state" (*i.e.*, extension of all warehouse views). However, it was not shown how to choose among the strategies. The strategies can result in significantly different update windows as confirmed by our experiments.

For the simple DAG of Figure 1, there are 11 strategies in addition to Strategies 1 and 2. For instance, a slight variant of Strategy 2 computes the changes of V based on the changes of *LINEITEM* first, then *ORDER*, and then *CUSTOMER*. In some cases, this variant may have a shorter update window than Strategy 2. □

The previous example illustrated that even for a *single view*, there are many update strategies. Finding optimal strategies for a single view is a challenge we address in this paper. In the next example, we illustrate that the update strategies for a *DAG of views* cannot be constructed by simply picking the strategies for each view independently. In this paper, we also address the problem of finding optimal strategies for a DAG of views.

EXAMPLE 1.2 Let us consider the DAG shown in Figure 2. This DAG now includes a second view V' defined over *CUSTOMER*, *ORDER* and *LINEITEM*. Say we update V using Strategy 2 (Example 1.1), and V' is updated using the following *Strategy 3*:

1. Compute the changes of V' only considering the changes of *LINEITEM*.
2. Install the changes of *LINEITEM*. (These changes are visible to the following step.)
3. Compute the V' changes considering the changes of *CUSTOMER* and *ORDER*.
4. Install the changes of *CUSTOMER* and *ORDER*.
5. Install the changes of V'.

Note that in Strategy 2, the fifth step occurs after the changes of *CUSTOMER* and *ORDER*, but not *LINEITEM*, have been installed. On the other hand, in Strategy 3 the third step occurs after the changes of *LINEITEM* have been installed, but not the changes of *CUSTOMER* and *ORDER*. Since only one of these states can be achieved,[1] we cannot combine Strategy 2 and Strategy 3. On the other hand, it is possible to combine Strategy 1 and Strategy 3 in a consistent manner. □

The previous example showed that we may not be able to construct a *correct* strategy for a DAG of views by combining independently chosen single view strategies. Even if we can, the combined strategy may not be the best among all correct strategies. In this paper, we define formally the notion of a correct update strategy for a DAG of views, and we develop techniques to obtain correct and efficient update strategies for a DAG of views.

One could argue that standard database query optimizers may be able to generate efficient warehouse update strategies by leveraging their proficiency in finding good plans for a query or even a set of queries. However, today's query optimizers assume that during the execution of the queries the database state does not change. As illustrated by our examples, warehouse update strategies employ sequences of computation and installation steps. More importantly, each step may change the database state, which in turn affects the rest of the steps. Hence, picking the best strategy involves:

- Choosing the set of queries (for update computations) and data manipulation expressions;

[1] We do not assume that multiple versions of the warehouse data are maintained.

- Sequencing these queries and data manipulation expressions; and
- Ensuring that the chosen sequence results in the correct final database state.

To our knowledge, query optimizers do not handle these tasks. As a result, the warehouse administrator (WHA) is often saddled with the task of creating "update scripts" for the warehouse views. Since there are many alternative update strategies, the WHA can easily pick an inefficient update strategy, or even worse an update strategy that incorrectly updates the warehouse. Furthermore, the WHA may have to change the script frequently, since what strategy is best depends on the current size of the warehouse views and the current set of changes.

In this paper, we develop a framework for studying the space of update strategies. We make the following specific contributions:

- We characterize the correctness and optimality of update strategies for a DAG of views.
- We develop a very efficient algorithm called *MinWorkSingle* that finds an update strategy that minimizes the work incurred in updating a single materialized view.
- Based on *MinWorkSingle*, we develop an efficient heuristic algorithm called *MinWork* that produces a good update strategy for a general DAG of materialized views. We show that for a large class of DAGs, the *MinWork* update strategy is actually the least expensive.
- Based on performance experiments with a TPC-D scenario, we demonstrate that the *MinWorkSingle* and *MinWork* update strategies shrink the update window significantly.

2 Preliminaries

Warehouse Model: We model warehouse data using a *view directed acyclic graph* (VDAG). Each node in the graph represents a materialized view containing warehouse data. An edge $(V_j \rightarrow V_i)$ indicates that view V_j is defined over view V_i. If a view V has no outgoing edges, this indicates that V is defined over remote information sources. For simplicity, we assume that a view V is defined only over remote information sources, or only over views at the warehouse. We call views defined over remote sources *base views*, and views defined over other views (at the warehouse) *derived views*.

Figure 3 shows a simple example of a VDAG with three base views (*i.e.*, V_1,V_2,V_3) and two derived views (*i.e.*, V_4, V_5). As a more concrete example, Figure 4 shows the VDAG representation of a warehouse that contains six TPC-D relations as base views. In this example, *ORDER* and *LINEITEM* represent "fact tables," and the other base views represent "dimension tables." The derived views $Q3$, $Q5$ and $Q10$ represent "summary tables" defined over the TPC-D base views.

We define $Level(V)$ to be the maximum distance of V to a base view. For instance, in Figure 3, $Level(V_1) = 0$, $Level(V_4) = 1$, and $Level(V_5) = 2$. We use $MaxLevel(G)$ to denote the maximum *Level* value of any view in a VDAG G.

View Definitions and Maintenance Expressions: We associate with each view V a definition $Def(V)$. View definitions in our model involve *projection*, *selection*, *join*, and *aggregation* operations. For instance, views $Q3$, $Q5$ and $Q10$ of Figure 4 may be defined using TPC-D queries that are `SELECT-FROM-WHERE-GROUPBY` SQL statements.

An edge $(V_j \rightarrow V_i)$ in the VDAG means that V_i appears in $Def(V_j)$. Moreover, it implies that changes of V_i lead to V_j changes. We use *delta relation* δV to represent the changes of V.

The changes of the base views arrive periodically at the warehouse. The changes of the base views are then used to compute the changes of the derived views. If V is a derived view, *view maintenance expressions* based on $Def(V)$ are used to compute δV. For instance, if view V_4 in Figure 3 is defined as $\sigma_{\mathcal{P}}(V_2 \times V_3)$, the following standard view maintenance expression ([6], [14]) that uses three *terms* (*i.e.*, $\sigma_{\mathcal{P}}(\delta V_2 \times V_3)$, $\sigma_{\mathcal{P}}(V_2 \times \delta V_3)$, $\sigma_{\mathcal{P}}(\delta V_2 \times \delta V_3)$) computes δV_4.

$$\delta V_4 \leftarrow \sigma_{\mathcal{P}}(\delta V_2 \times V_3) \cup \sigma_{\mathcal{P}}(V_2 \times \delta V_3) \cup \sigma_{\mathcal{P}}(\delta V_2 \times \delta V_3) \quad (1)$$

Actually, the changes of a view V include inserted V tuples, called *plus* tuples, and deleted V tuples, called *minus* tuples. (In this paper, we represent an update as a deletion followed by an insertion.) For simplicity of presentation, we do not show explicitly the plus tuples and the minus tuples, instead lumping them together in a single delta relation. When executing maintenance expressions like (1), the plus and minus tuples in the delta relations must be handled "appropriately" [6].

After the changes of a view are computed, they are used in computing changes of other derived views, and installed. The install operation inserts the plus tuples, and deletes the minus tuples.

Compute and Install Expressions: We abstract maintenance computations by the function *Comp*.

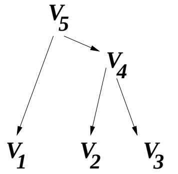

Figure 3: Example VDAG

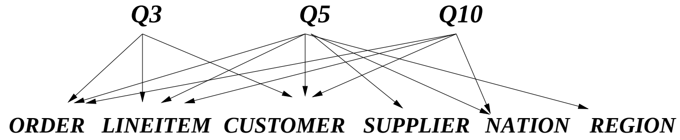

Figure 4: VDAG of a TPC-D Warehouse

The formula for computing δV from the changes of the set of views $\mathcal{V}$ is denoted by $Comp(V, \mathcal{V})$. For instance, $Comp(V_4, \{V_2, V_3\})$ represents the δV_4 computation of Expression (1). As another example, $Comp(V_4, \{V_2\})$ represents the computation of the changes of V_4 based solely on the changes of V_2, *i.e.*, $\delta V_4 \leftarrow \sigma_{\mathcal{P}}(\delta V_2 \times V_3)$. We use $Inst(V)$ to denote the operation of installing δV into V.

3 View and VDAG Strategies

We now define *view strategies* which are used to update a single view, and *VDAG strategies* which are used to update a VDAG of views. We also illustrate how one can define the space of correct VDAG strategies based on the notion of correct view strategies for the individual views. Finally, we formally define the "total-work minimization" problem as finding the correct VDAG strategy that incurs the minimum amount of work.

3.1 View Strategies

For a view V defined over n views $V_1, \ldots, V_n$, there are many possible ways of updating V. We call each way a *view strategy*. One view strategy for V is to compute δV based on all of the changes $\{\delta V_1, \ldots, \delta V_n\}$ simultaneously as shown below.

$$\langle\ Comp(V, \{V_1, \ldots, V_n\}), Inst(V_1), \ldots, \\ Inst(V_n), Inst(V)\ \rangle \quad (2)$$

Notice that view strategy (2) has two "stages," a stage for propagating the underlying changes (*i.e.*, using the *Comp* expression), and a stage for installing the changes (*i.e.*, using the *Inst* expressions). This is consistent with the framework proposed in [3] that a view is updated using a propagate stage and an install stage. In this paper, we call strategies like (2) *dual-stage* view strategies.

Another possible view strategy for V is to compute δV by considering each δV_i in $\{\delta V_1, \ldots, \delta V_n\}$ one at a time, as shown below.

$$\langle\ Comp(V, \{V_1\}), Inst(V_1), \ldots, \\ Comp(V, \{V_n\}), Inst(V_n), Inst(V)\ \rangle \quad (3)$$

Each *Comp* expression in view strategy (3) computes a subset of the changes of V. We assume that the changes computed by the various *Comp* expressions for V are gathered in delta relation δV, and eventually installed together by $Inst(V)$. We call view strategies like (3) *1-way* view strategies. Notice that view strategy (3) propagates the changes of V_1 first, then of V_2, and so on. For a view defined over n views, there are a total of $n!$ 1-way view strategies that can be obtained by using different change propagation orders.

Dual-stage view strategies as well as 1-way view strategies have been proposed in the literature ([8], [3]). However, the issue of finding optimal view strategies has not been studied.

Beyond the 1-way and dual-stage view strategies, there is a multitude of other correct view strategies. To see this, we can look at a 1-way view strategy as one that partitions $\{\delta V_1, \ldots, \delta V_n\}$ into n singleton sets, and processes the sets, one at a time. On the other hand, a dual-stage view strategy does not partition $\{\delta V_1, \ldots, \delta V_n\}$ at all, and processes all the changes simultaneously. Other ways of partitioning the view set will yield other view strategies.

Once the partitions are decided upon, the propagation order among the various partitions needs to be chosen. The combined choices of partitioning and their order of processing yields numerous view strategies that incur different amounts of work in general. For instance, view $Q3$ defined on three views, $Q5$ defined on 6 views, and $Q10$ defined on 4 views have 13, 4683, and 75 view strategies respectively. Furthermore, we are only counting "correct" view strategies.

In Definition 3.1, we formally describe the notion of correctness of a view strategy. Intuitively, conditions **C1** and **C2** state that all the changes must be propagated and installed by a correct view strategy. That is, certain *Comp* and *Inst* expressions must be in the correct view strategy. On the other hand, conditions **C3**, **C4**, and **C5** state that the *Comp* and *Inst* expressions must be in a particular order. Specifically, condition **C3** states that δV_i must not be installed until all *Comp* expressions that use it are done. Condition **C4** states that when the changes of V are computed using multiple *Comp* expressions, the changes of a view used in a *Comp* expression must be installed before the next *Comp* expression for V can be executed. Condition **C5** states that the changes computed for V can only be installed after they are completely computed. Finally, condition **C6** states that there are no duplicate expressions in the correct view strategy.

Definition 3.1 (Correct View Strategy) Let $E_i < E_j$ if expression E_i is before expression E_j in the view strategy. Given a view V defined over a set of views $\mathcal{V}$, a correct view strategy $\overrightarrow{\mathcal{E}}$ for V is a sequence of *Comp* and *Inst* expressions satisfying the following conditions.

- **C1** $\forall V_i \in \mathcal{V}$: $Comp(V, \{\ldots V_i \ldots\}) \in \overrightarrow{\mathcal{E}}$.
- **C2** $\forall V_i \in (\mathcal{V} \cup \{V\})$: $Inst(V_i) \in \overrightarrow{\mathcal{E}}$.
- **C3** $\forall V_i \in \mathcal{V}$: $Comp(V, \{\ldots V_i \ldots\}) < Inst(V_i)$.
- **C4** $\forall V_i$: $\forall V_j$: $(Comp(V, \{\ldots V_i \ldots\}) < Comp(V, \{\ldots V_j \ldots\}))$ $\Rightarrow (Inst(V_i) < Comp(V, \{\ldots V_j \ldots\}))$.
- **C5** $\forall V_i \in \mathcal{V}$: $Comp(V, \{\ldots V_i \ldots\}) < Inst(V)$.
- **C6** $\forall E_i \in \overrightarrow{\mathcal{E}}$: $\forall E_j \in \overrightarrow{\mathcal{E}}$: $(i \neq j) \Rightarrow (E_i \neq E_j)$ □

Notice that combinations of these conditions avoid incorrect view strategies that are not explicitly prohibited in the conditions. For instance, because of conditions **C3** and **C4**, it is not possible to have two *Comp* expressions that propagate δV_i [12]. Note also that for a base view V which is not defined over any warehouse views (*i.e.*, $\mathcal{V} = \{\ \}$), V's correct view strategy is $\langle\ Inst(V)\ \rangle$.

3.2 VDAG Strategies

Like a view strategy, a *VDAG strategy* is simply a sequence of compute and install expressions. Informally speaking, a *correct* VDAG strategy uses a correct view strategy to update each VDAG view.

EXAMPLE 3.1 Consider the VDAG shown in Figure 3. A VDAG strategy should indicate how changes are propagated to all the views. One possible VDAG strategy propagates the changes of V_2 to V_4, then propagates the changes of V_3 to V_4, then propagates the changes of V_4 to V_5, and finally propagates the changes of V_1 to V_5.

$$\begin{aligned}&\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}),\\ &\ Inst(V_3), Comp(V_5, \{V_4\}), Inst(V_4),\\ &\ Comp(V_5, \{V_1\}), Inst(V_1), Inst(V_5)\ \rangle \qquad (4)\end{aligned}$$

Note that VDAG strategy (4) "uses" (contains as a subsequence) the following correct view strategies for V_4 and V_5 respectively.

$$\begin{aligned}&\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}),\\ &\ Inst(V_3), Inst(V_4)\ \rangle\\ &\langle\ Comp(V_5, \{V_4\}), Inst(V_4), Comp(V_5, \{V_1\}),\\ &\ Inst(V_1), Inst(V_5)\ \rangle\end{aligned}$$

Also, for any base view V_i (*i.e.*, V_1, V_2, V_3), VDAG strategy (4) "uses" $\langle\ Inst(V_i)\ \rangle$. □

The previous example illustrated that a correct VDAG strategy uses correct view strategies to update each view. However, we know that starting from a set of correct view strategies, we may not be able to construct a correct VDAG strategy (Example 1.2, Section 1). In Section 5, we present an algorithm that finds correct and efficient VDAG strategies. In the rest of this section, we formalize our notions of correctness and efficiency of VDAG strategies. First, we define the concept of a view strategy "used" by a VDAG strategy.

Definition 3.2 (View Strategy Used by a VDAG Strategy) Given a VDAG strategy $\overrightarrow{\mathcal{E}}$, and a view V_j defined over views $\mathcal{V}$, the view strategy *used* by $\overrightarrow{\mathcal{E}}$ for V_j is the subsequence $\overrightarrow{\mathcal{E}_j}$ of $\overrightarrow{\mathcal{E}}$ composed of the following expressions: (1) $Comp(V_j, \{...\})$; (2) $Inst(V_j)$; and (3) $Inst(V_i)$, where $V_i \in \mathcal{V}$. □

The next definition formalizes the conditions that are required of a correct VDAG strategy. Condition **C7** states that a correct VDAG strategy must update each view using a correct view strategy. Condition **C8** states that a correct VDAG strategy can only propagate changes of V_j after they have been computed. Condition **C8** implicitly imposes an order between expressions from view strategies of different views in the VDAG.

Definition 3.3 (Correct VDAG Strategy) Given a VDAG G with views $\mathcal{V}$ and edges $\mathcal{A}$, a *correct* VDAG strategy is a sequence of *Comp* and *Inst* expressions $\overrightarrow{\mathcal{E}}$ such that

- **C7:** $\forall V_i \in \mathcal{V}$: $\overrightarrow{\mathcal{E}}$ uses a correct view strategy $\overrightarrow{\mathcal{E}_i}$ for V_i.
- **C8:** $\forall V_i \in \mathcal{V}$: $\forall V_j \in \mathcal{V}$: $\forall V_k \in \mathcal{V}$: $(Comp(V_k, \{\ldots V_j \ldots\}) \in \overrightarrow{\mathcal{E}}$ and $Comp(V_j, \{\ldots V_i \ldots\}) \in \overrightarrow{\mathcal{E}}) \Rightarrow$ $(Comp(V_j, \{\ldots V_i \ldots\}) < Comp(V_k, \{\ldots V_j \ldots\}))$.

□

3.3 Problem Statement

We use a function *Work* to represent the amount of work involved in executing an expression – *Comp* or

Inst. Given a VDAG strategy $\overrightarrow{\mathcal{E}} = \langle E_1, \ldots, E_n \rangle$, we define $Work(\overrightarrow{\mathcal{E}})$ as $\sum_{i=1..n} Work(E_i)$. Notice that $Work(E_i)$ depends on the expressions that precede E_i, since these expressions change the database state that E_i is executed in. The problem we address in this paper is stated as follows.

Definition 3.4 (Total-Work Minimization Problem (TWM)) Given a VDAG, find the correct VDAG update strategy $\overrightarrow{\mathcal{E}}$ such that $Work(\overrightarrow{\mathcal{E}})$ is minimized. □

Since TWM is only concerned with correct VDAG strategies, henceforth, "VDAG strategies" refer only to "correct VDAG strategies." Similarly, "view strategies" refer only to "correct view strategies."

To estimate $Work(E_i)$, we adopt a metric called *linear work metric*. This is a simple metric that focuses on the essential components of the work involved in executing the *Comp* and *Inst* expressions. The algorithms that we develop produce optimal update strategies under the linear work metric. In Section 6, we study the relative performance of various update strategies for the TPC-D VDAG by executing the strategies on a commercial RDBMS, and measuring the corresponding update windows. Our study demonstrates that the strategies produced by our algorithms have significantly shorter update windows than conventional update strategies. The results of the study suggest that the linear work metric employed by our algorithms effectively tracks real-world execution of update strategies.

The linear work metric is based on the following execution model of *Comp* expressions. Recall that *Comp* typically represents a maintenance expression with a set of terms (*e.g.*, Expression (1) of Section 2 has three terms). Each term performs some computation by reading in views and delta relations, called operands. For example, assuming a view W is defined over V_1, V_2, and V_3, $Comp(W, \{V_1\})$ has a single term that reads in three operands (δV_1, V_2, V_3) to compute changes to W. We consider an execution model that evaluates each term of a *Comp* expression separately. Thus, the work estimate for a *Comp* expression is obtained by estimating the work for each of its terms and adding up these estimates.

Definition 3.5 (Linear Work Metric) The work estimate for an *Inst* expression is proportional to the size of the set of changes being installed. The estimate for a *Comp* expression is the sum of the estimates for each of its terms; the estimate for a term is proportional to the sum of the sizes of the operands of the term. □

EXAMPLE 3.2 Consider the VDAG shown in Figure 3, with V_4 defined as $\sigma_{\mathcal{P}}(V_2 \times V_3)$. $Comp(V_4, \{V_2\})$ has one term: $\sigma_{\mathcal{P}}(\delta V_2 \times V_3)$. Its work estimate is $c \cdot (|\delta V_2| + |V_3|)$, where c is a proportionality constant. Similarly, the estimate for $Comp(V_4, \{V_2, V_3\})$ can be derived (by considering its 3 terms) as $c \cdot ((|\delta V_2| + |V_3|) + (|\delta V_3| + |V_2|) + (|\delta V_2| + |\delta V_3|))$. The work estimate for $Inst(V_4)$ is $i \cdot |\delta V_4|$, where i is a proportionality constant. □

The estimates of the linear cost model for compute expressions make sense especially if the delta relations are small. If so, intermediate results in the evaluation of a term tend to be small. Therefore, the work incurred in evaluating a term is often dominated by scanning into memory the term's operands.

4 Optimal View Strategy

In this section, we present algorithm *MinWorkSingle* that produces an optimal view strategy for a given view, under the linear work metric.

We showed previously that there are numerous possible view strategies for a single view. Fortunately, under the linear work metric, we can restrict our attention to 1-way view strategies only.

Theorem 4.1 *For any given view, the best 1-way view strategy is optimal over the space of all view strategies.*

The detailed proof of Theorem 4.1, and of other theorems and lemmas that follow, are furnished in [12]. The basic intuition is that in any view strategy for V that is not 1-way, a *Comp* expression that computes the changes of V based on multiple views can be replaced by a set of *Comp* expressions each involving a single view such that the total work of this set of *Comp* expressions is smaller than the work incurred by the replaced *Comp* expression.

Theorem 4.1 is very significant because it allows us to limit the search for an optimal view strategy to the set of 1-way view strategies. Next, we will present another theorem that helps us avoid examining all the 1-way view strategies and identify the best 1-way strategy very efficiently. The following example illustrates how the various 1-way view strategies differ in efficiency and it provides the basic intuition behind the next theorem.

EXAMPLE 4.1 Let us again consider view V_4 (Figure 3) defined over V_2 and V_3, and compare the two 1-way view strategies for V_4 shown below.

$$\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3), Inst(V_4)\ \rangle \quad (5)$$

$$\langle\ Comp(V_4, \{V_3\}), Inst(V_3), Comp(V_4, \{V_2\}), Inst(V_2), Inst(V_4)\ \rangle \quad (6)$$

Clearly, the work incurred by the *Inst* expressions are the same. This is not the case for the *Comp* expressions. Although the same set of *Comp* expressions are used, the view extensions accessed by the *Comp* expressions are different.

To illustrate, we use V_2' to denote V_2 after δV_2 is installed. Similarly, V_3' denotes V_3 after δV_3 is installed. In general, the expression $Comp(V_4, \{V_2\})$ in view strategy (5) uses δV_2, and V_3, and possibly V_4. On the other hand, the same expression $Comp(V_4, \{V_2\})$ in view strategy (6) uses δV_2, and V_3', and possibly V_4. Hence, the only difference in the use of $Comp(V_4, \{V_2\})$ in the two view strategies is that V_3' is used in view strategy (6), while V_3 is used in view strategy (5).

In general, the earlier δV_3 is installed in a view strategy, the more often will V_3' be used by the compute expressions in the view strategy. If it so happens that V_3' is larger than V_3, then using V_3' is more expensive than using V_3. In this case, it is good to delay the installation of δV_3. On the other hand, if V_3' is smaller than V_3, then it is good to install δV_3 as early as possible.

In fact, under a linear work metric we can be much more precise about the installation and propagation order of the various changes. For instance, if we first propagate and install the changes of V_3 (as in view strategy (6)), any subsequent compute expression that used to access V_3, will access V_3' instead. Hence, the work incurred by these compute expressions is increased by $c \cdot (|V_3'| - |V_3|)$. Similarly if we first propagate and install the changes to V_2 (as in view strategy (5)), the work incurred by subsequent compute expressions is increased by $c \cdot (|V_2'| - |V_2|)$. Hence, in this example, we would want to propagate and install the changes of V_3 before the changes of V_2 if $(|V_3'| - |V_3|) < (|V_2'| - |V_2|)$. □

The example illustrated how an optimal 1-way view strategy for some view V can be obtained. Assuming V is defined over the views $\mathcal{V}$, we first obtain a *view ordering* $\overrightarrow{\mathcal{V}}$ that arranges the views in $\mathcal{V}$ in increasing $|V_i'| - |V_i|$ values based on the current set of changes. Given $\overrightarrow{\mathcal{V}}$, an optimal 1-way view strategy is the one that propagates and installs the changes in an order *consistent* with $\overrightarrow{\mathcal{V}}$. A 1-way view strategy for V is consistent with a view ordering $\overrightarrow{\mathcal{V}}$ if for any $Inst(V_i)$ that is before $Inst(V_j)$ in the strategy ($V_i \neq V$, $V_j \neq V$), then V_i is before V_j in $\overrightarrow{\mathcal{V}}$.

Algorithm 4.1 *MinWorkSingle*
Input: V, defined over views $\mathcal{V}$
Output: an optimal view strategy $\overrightarrow{\mathcal{E}}$ for V

1. $\overrightarrow{\mathcal{E}} \leftarrow \langle\ \rangle$
2. For each $V_i \in \mathcal{V}$ estimate $|V_i'| - |V_i|$ based on the current set of changes
3. $\overrightarrow{\mathcal{V}} \leftarrow$ views in $\mathcal{V}$ ordered by increasing $|V_i'| - |V_i|$ values
4. For each $V_i \in \overrightarrow{\mathcal{V}}$ in order
   5. Append $Comp(V, \{V_i\})$ to $\overrightarrow{\mathcal{E}}$
   6. Append $Inst(V_i)$ to $\overrightarrow{\mathcal{E}}$
7. Append $Inst(V)$ to $\overrightarrow{\mathcal{E}}$
8. Return $\overrightarrow{\mathcal{E}}$ ◇

Figure 5: *MinWorkSingle* Algorithm

Theorem 4.2 *Given a view V defined over the views $\mathcal{V}$, let the view ordering $\overrightarrow{\mathcal{V}}$ arrange the views in increasing $|V_i'| - |V_i|$ values, for each $V_i \in \mathcal{V}$. Then, a 1-way view strategy for V that is consistent with $\mathcal{V}$ will incur the least amount of work among all the 1-way view strategies for V.*

Based on Theorem 4.1 and Theorem 4.2, algorithm *MinWorkSingle* (Figure 5) produces an optimal view strategy. The view strategy produced by *MinWorkSingle* is shown to be correct in [12].

We summarize the behavior of algorithm *MinWorkSingle* in the following theorem.

Theorem 4.3 *Given a view defined over n other views, MinWorkSingle finds an optimal view strategy for the view in $O(n \log n)$ time.*

5 Minimizing Total Work

We have seen that for a derived view V, a 1-way view strategy consistent with a certain view ordering based on the current set of changes of the views that V is defined on is optimal. In this section, we show a similar result for VDAG strategies. That is, for a VDAG, we show that a "1-way VDAG strategy" consistent with a certain ordering of all the VDAG views based on the current set of changes is optimal among all VDAG strategies. Based on this result, we present an efficient algorithm to find optimal VDAG strategies.

Unlike in the case of view strategies, it is not always possible to obtain a "1-way VDAG strategy" consistent with a given view ordering. In such cases, our algorithm finds a modified view ordering for which an efficient "1-way VDAG strategy" that is consistent with the modified view ordering can be obtained. In this section, we also identify large classes of VDAGs for which optimal VDAG strategies are guaranteed by our algorithm.

5.1 Optimal VDAG Strategies

Intuitively, a VDAG strategy that uses good view strategies for its derived views tends to incur less amount of work than one that uses worse view strategies. In the following theorem we capture the relationship between optimal VDAG strategies and the view strategies they use.

Theorem 5.1 *Given a VDAG G, a VDAG strategy for G that uses optimal view strategies for all the views of G is optimal over all VDAG strategies for G.*

Observe that all VDAG strategies for G incur the same amount of work for their $Inst$ expressions. In the proof (see [12]), we further argue that a VDAG strategy that uses optimal view strategies minimizes the work incurred by the $Comp$ expressions.

From Section 4, we know that given a view V_i that is defined over views $\mathcal{V}_i$, the 1-way view strategy $\overrightarrow{\mathcal{E}_i}$ that is consistent with $\overrightarrow{\mathcal{V}_i}$ that orders the views in $\mathcal{V}_i$ in increasing $|V'| - |V|$ values is optimal. It can be shown that $\overrightarrow{\mathcal{E}_i}$ is also consistent with the view ordering $\overrightarrow{\mathcal{V}}$ that orders *all* of the VDAG views in increasing $|V'| - |V|$ values. This view ordering is called a *desired view ordering*.

We say a VDAG strategy is a *1-way VDAG strategy* if it only uses 1-way view strategies. Furthermore, a VDAG strategy is *consistent* with $\overrightarrow{\mathcal{V}}$ if it only uses view strategies that are consistent with $\overrightarrow{\mathcal{V}}$. Clearly, a 1-way VDAG strategy that is consistent with a desired view ordering uses only optimal view strategies. It follows from Theorem 5.1 that this VDAG strategy is optimal.

Theorem 5.2 *For any VDAG G, a 1-way VDAG strategy for G that is consistent with a desired view ordering is an optimal VDAG strategy for G.*

We illustrate the interaction between Theorem 5.1 and Theorem 5.2 by the following example.

EXAMPLE 5.1 Consider the VDAG shown in Figure 6. Let $(|V_4'| - |V_4|) < (|V_2'| - |V_2|) < (|V_1'| - |V_1|) < (|V_3'| - |V_3|) < (|V_5'| - |V_5|)$ based on the current set of changes. That is, a desired view ordering $\overrightarrow{\mathcal{V}}$ is $\langle\ V_4, V_2, V_1, V_3, V_5\ \rangle$.

A 1-way VDAG strategy consistent with the desired view ordering is

$$\begin{aligned}&\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}),\\ &\quad Inst(V_3), Comp(V_5, \{V_4\}), Inst(V_4),\\ &\quad Comp(V_5, \{V_1\}), Inst(V_1), Inst(V_5)\ \rangle.\end{aligned}$$

The above VDAG strategy is optimal and uses the following optimal view strategies for V_4 and V_5:

$$\begin{aligned}&\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}),\\ &\quad Inst(V_3), Inst(V_4)\ \rangle.\\ &\langle\ Comp(V_5, \{V_4\}), Inst(V_4), Comp(V_5, \{V_1\}),\\ &\quad Inst(V_1), Inst(V_5)\ \rangle.\end{aligned}$$

□

5.2 Expression Graphs

We have established that a 1-way VDAG strategy consistent with a desired view ordering is optimal. Here, we describe our approach to constructing such a VDAG strategy.

For a given VDAG G, all possible 1-way VDAG strategies for G have the same set of expressions, called the *1-way expressions* of G. The set of 1-way expressions of a given VDAG G contains $Comp(V_j, \{V_i\})$ whenever view V_j is defined over view V_i in G. Also included is an $Inst(V_i)$ expression for each view V_i in G. The various 1-way VDAG strategies for G differ in the sequencing of the 1-way expressions of G. The correctness conditions (of Section 3) impose certain dependencies among these 1-way expressions (*e.g.*, for any two derived views V_i and V_j, $Comp(V_j, \{V_i\})$ must follow $Comp(V_i, \{...\})$). Additional dependencies are imposed when we attempt to find VDAG strategies that are consistent with a particular view ordering (*e.g.*, for a derived view V defined over views V_i and V_j, if V_i precedes V_j in the view ordering, $Comp(V, \{V_i\})$ must precede $Comp(V, \{V_j\})$). A 1-way VDAG strategy for G consistent with a given view ordering is a permutation of the set of 1-way expressions of G that satisfies all dependencies.

We use the notion of an *expression graph* to capture the set of 1-way expressions of a VDAG and their dependencies. Given a VDAG G and a view ordering $\overrightarrow{\mathcal{V}}$, the expression graph of G with respect to $\overrightarrow{\mathcal{V}}$, denoted $EG(G, \overrightarrow{\mathcal{V}})$, has the 1-way expressions of G as its nodes. The expression graph has an edge from expression E_j to expression E_i if a dependency dictates that E_j must follow E_i. Once we construct an expression graph for a VDAG with

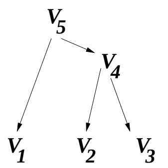

Figure 6: VDAG

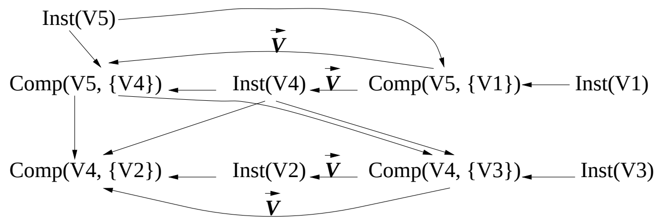

Figure 7: Expression Graph (EG)

respect to a desired view ordering, we can obtain an optimal VDAG strategy by topologically sorting the expression graph.

Theorem 5.3 *Given a VDAG G, if* $EG(G, \overrightarrow{\mathcal{V}})$ *is acyclic where* $\overrightarrow{\mathcal{V}}$ *is a desired view ordering, a topological sort of* $EG(G, \overrightarrow{\mathcal{V}})$ *yields an optimal VDAG strategy for G.* □

We now illustrate the generation of an optimal VDAG strategy, based on this theorem.

EXAMPLE 5.2 Consider the VDAG shown in Figure 6. Let a desired view ordering $\overrightarrow{\mathcal{V}}$ be $\langle\ V_4, V_2, V_1, V_3, V_5\ \rangle$ based on the current set of changes (as in Example 5.1).

Figure 7 shows the expression graph constructed from the VDAG and the view ordering $\overrightarrow{\mathcal{V}}$. Each derived view has a set of *Comp* expressions, one for each view it is defined over. Each view in the VDAG has an *Inst* expression.

The edges of the expression graph indicate the dependencies. For instance, the edge from $Comp(V_5, \{V_4\})$ to $Comp(V_4, \{V_2\})$ indicates that the former should appear after the latter in any 1-way VDAG strategy for this VDAG due to **C8**.

Some edges of the expression graph are shown with a label $\overrightarrow{\mathcal{V}}$ to emphasize that the corresponding dependencies are due to the view ordering with which the 1-way VDAG strategy should be consistent. For instance, the edge from $Comp(V_4, \{V_3\})$ to $Comp(V_4, \{V_2\})$ indicates that $\overrightarrow{\mathcal{V}}$ requires that the changes of V_2 be propagated before the changes of V_3 (note that $V_2 < V_3$ in $\overrightarrow{\mathcal{V}}$).

The expression graph of this example happens to be acyclic. So, a topological sort of the graph is possible, and yields a 1-way VDAG strategy that is consistent with the view ordering $\overrightarrow{\mathcal{V}}$. For instance, we can obtain the following VDAG strategy:

$$\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3), Comp(V_5, \{V_4\}), Inst(V_4), Comp(V_5, \{V_1\}), Inst(V_1), Inst(V_5)\ \rangle.$$

Note that this is the same optimal VDAG strategy that we discussed in Example 5.1. □

5.3 Classes of VDAGs with Optimal VDAG Strategies

We have seen that whenever the constructed expression graph with respect to a desired view ordering is acyclic, we can obtain an optimal VDAG strategy in a straightforward manner. The acyclicity of the expression graph depends not only on the VDAG but also on the desired view ordering being considered. The view ordering in turn depends on the current set of changes. In general, a given VDAG may have an acyclic expression graph with one desired view ordering (*i.e.*, based on a set of changes) and a cyclic expression graph with another desired view ordering (*i.e.*, based on another set of changes). However, there are specific classes of VDAGs which will always have acyclic expression graphs. For these classes of VDAGs, we can always find optimal VDAG strategies in a straightforward manner no matter what changes are being propagated. We identify two such classes of VDAGs below.

Definition 5.1 (Tree VDAGs) A *tree* VDAG is one in which no view is used in the definition of more than one other view. □

Lemma 5.1 *For a tree VDAG, every view ordering results in an acyclic expression graph.* □

Definition 5.2 (Uniform VDAGs) A *uniform* VDAG is one in which every derived view at *Level* i is defined over views all of which are at *Level* $(i-1)$. □

Lemma 5.2 *For a uniform VDAG, every view ordering results in an acyclic expression graph.* □

Note that the classes of uniform VDAGs and tree VDAGs are incomparable. The VDAG in Figure 6 is a tree VDAG but not a uniform VDAG. On the other hand, the TPC-D VDAG (Figure 4) is a uniform VDAG but not a tree VDAG.

5.4 *MinWork* Algorithm

Based on our observations above, we develop an algorithm called *MinWork* to generate VDAG strategies that minimize the total amount of work.

Algorithm 5.1 *ModifyOrdering*
Input: VDAG G, view ordering $\overrightarrow{\mathcal{V}}$
Output: modified view ordering $\overrightarrow{\mathcal{V}'}$

1. $\overrightarrow{\mathcal{V}'} \leftarrow \langle\ \rangle$
2. For $l = 0$ to $MaxLevel(G)$
    3. $\overrightarrow{\mathcal{V}_l} \leftarrow$ subsequence of $\overrightarrow{\mathcal{V}}$ composed of all and only views with a *Level* value of l
    4. Append $\overrightarrow{\mathcal{V}_l}$ to $\overrightarrow{\mathcal{V}'}$
5. Return $\overrightarrow{\mathcal{V}'}$ $\diamond$

Algorithm 5.2 *MinWork*
Input: VDAG G with nodes $\mathcal{V}$ and edges $\mathcal{A}$
Output: 1-way VDAG strategy $\overrightarrow{\mathcal{E}}$

1. $\overrightarrow{\mathcal{E}} \leftarrow \langle\ \rangle$
2. For each $V_i \in \mathcal{V}$ estimate $|V_i'| - |V_i|$ based on the current set of changes
3. $\overrightarrow{\mathcal{V}} \leftarrow \mathcal{V}$ ordered by increasing $|V_i'| - |V_i|$
4. EG $\leftarrow$ $ConstructEG(G, \overrightarrow{\mathcal{V}})$
5. If EG is acyclic then
    6. $\overrightarrow{\mathcal{E}} \leftarrow$ topological sort of EG
7. Else
    8. $\overrightarrow{\mathcal{V}'} \leftarrow$ *ModifyOrdering*$(\overrightarrow{\mathcal{V}})$
    9. EG$'$ $\leftarrow$ $ConstructEG(G, \overrightarrow{\mathcal{V}'})$
    10. $\overrightarrow{\mathcal{E}} \leftarrow$ topological sort of EG$'$
11. Return $\overrightarrow{\mathcal{E}}$ $\diamond$

Figure 8: *MinWork* Algorithm

In particular, *MinWork* relies on the approach of expression graph construction in order to find good VDAG strategies. The algorithm is formally presented in Algorithm 5.2 of Figure 8.

MinWork first computes a desired view ordering based on the current set of changes. Then it constructs the expression graph of the VDAG with respect to this desired view ordering. The routine *ConstructEG* for constructing the expression graph is not shown here due to space constraints (see [12]). *ConstructEG* includes one node for each 1-way expression of G. It then connects the nodes based on dependencies imposed by the correctness conditions and by the given view ordering. If the constructed expression graph is acyclic, *MinWork* obtains the optimal VDAG strategy by a topological sort of the expression graph. Otherwise, it computes a modified view ordering (using *ModifyOrdering* shown in Algorithm 5.1) which is guaranteed to yield an acyclic expression graph of the VDAG . Then, it generates a VDAG strategy for the input VDAG that is consistent with this modified view ordering.

It is clear that given a VDAG that results in an acyclic expression graph, *MinWork* produces an optimal VDAG strategy. This leads to the following result that follows from Theorem 5.3, Lemma 5.1 and Lemma 5.2.

Theorem 5.4 *Given a VDAG G, and a desired view ordering $\overrightarrow{\mathcal{V}}$, MinWork produces optimal VDAG strategies if $EG(G, \overrightarrow{\mathcal{V}})$ is acyclic. In particular, MinWork always produces optimal VDAG strategies for tree VDAGs and uniform VDAGs.* □

When the given VDAG results in a cyclic expression graph with respect to the desired view ordering, *MinWork* produces a 1-way VDAG strategy that is consistent with a view ordering $\overrightarrow{\mathcal{V}'}$ that is produced by *ModifyOrdering* based on the desired view ordering. *ModifyOrdering* produces $\overrightarrow{\mathcal{V}'}$ by first ordering the views based on their *Level* values (*i.e.*, lower level views first). *ModifyOrdering* then orders the views with the same *Level* value based on the desired view ordering. The following theorem ensures that *MinWork* will always be able to generate a 1-way VDAG strategy no matter how complex the input VDAG is.

Theorem 5.5 *Given a VDAG G and a view ordering $\overrightarrow{\mathcal{V}}$, we can come up with a view ordering $\overrightarrow{\mathcal{V}'}$ = ModifyOrdering(G, $\overrightarrow{\mathcal{V}}$) such that EG(G, $\overrightarrow{\mathcal{V}'}$) is acyclic. That is, MinWork will always succeed in producing a VDAG strategy.* □

The use of a modified view ordering when a desired view ordering yields cyclic expression graphs may lead *MinWork* to produce sub-optimal VDAG strategies. However, the modified view ordering reflects as much of the desired view ordering as possible. This results in *MinWork* producing efficient plans, when it misses optimal plans.

In [12], we show that *MinWork* has a worst case time complexity of $O(n^3)$ where n is the number of views in the VDAG. We also discuss how *MinWork* can be implemented very easily using stored procedures.

Finally, we also develop in [12] a different search algorithm that finds the optimal 1-way VDAG strategy for any VDAG. As expected, the algorithm is less efficient than *MinWork* and has a worst case time complexity of $O(n! \cdot n^3)$.

6 Experiments

We have developed algorithms that minimize the work incurred in view or VDAG strategies. However, minimizing the work incurred may not translate to the minimization of the update window.

In order to understand how well the strategies generated by our algorithms perform in practice, we conducted a series of experiments. In particular, we tested various strategies using Microsoft SQL Server 6.5 running on a workstation with a Pentium II 300 MHz processor and 64 MB of RAM. In our experiments, we measured the actual time it took to execute the strategies. The results show that the strategies generated by our algorithms do indeed yield short update windows.

In all of the experiments, we used the TPC-D warehouse shown in Figure 4. The base views *CUSTOMER* (denoted C for conciseness), *ORDER* (O), *LINEITEM* (L), *SUPPLIER* (S), *NATION* (N) and *REGION* (R) are copies of TPC-D relations populated with synthetic data obtained from [5]. The derived views $Q3$, $Q5$ and $Q10$ were defined using the TPC-D "Shipping Priority" query, "Local Supplier" query, and "Returned Item Reporting" query respectively.

Unless otherwise specified, the remote information sources were changed so that base views C, O, L, S, and N decreased in size by 10%. Base view R, the smallest of the six, was left unchanged. According to the sizes of the base views, the desired view ordering is $\langle L, O, C, S, N, R \rangle$.

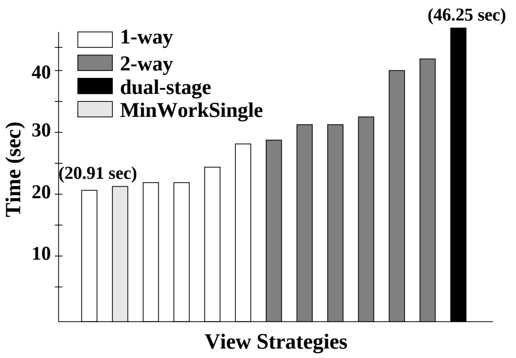

Figure 9: $Q3$ View Strategies

Experiment 1: In the first experiment, we examined the various view strategies for $Q3$. Since $Q3$ is only defined over 3 views, there were only 13 view strategies to compare, one from each partition. Figure 9 shows the result of the experiment. Each bar depicts a view strategy, and the height of the bar gives the amount of time it took to perform the view strategy. The graph shows numerous results.

First, the graph shows that 1-way view strategies update $Q3$ in the least amount of time.

Second, the graph shows that the *MinWorkSingle* view strategy, which propagates the changes of L, then of O, and then of C, does not update $Q3$ in the least amount of time. The view strategy that performs the best in this case propagates the changes of L, then of C and then of O. The update window of the *MinWorkSingle* view strategy is however very close to the optimal. Recall that we proved that *MinWorkSingle* produces an optimal view strategy under the linear work metric. In the experiment, we used a real system whose behavior naturally deviates from the strictly linear work metric. Thus, *MinWorkSingle* ends up with a strategy that is slightly away from the optimum.

Finally, the graph shows that various view strategies have significantly different update windows.

Experiment 2: In the next experiment, we focused on the derived view $Q5$ which is defined over the 6 base views. Since $Q5$ is much more complex than $Q3$, it was too time consuming to examine all of the view strategies of $Q5$. Instead, we examined only the *MinWorkSingle* view strategy and the dual-stage view strategy. Recall that the dual-stage view strategy is the one with a compute stage and an install stage, as proposed in [3]. The results show that the update window of the dual-stage view strategy is over *6 times* longer than that of the *MinWorkSingle* view strategy.

Experiment 3: In this experiment, we again focus on $Q3$. Each of C, O, and L is decreased in size by a percentage p of its initial size, for various values of p (from 1% to 10%). When comparing view strategies, we only considered the *MinWorkSingle* view strategy, the best 2-way view strategy in Figure 9, and the dual-stage view strategy. For all values of p, the *MinWorkSingle* view strategy performed better than both the 2-way view strategy and the dual-stage view strategy.

Experiment 4: So far, we have considered updating a single view. In this experiment, we study the quality of *MinWork* VDAG strategies. Note that, since the TPC-D VDAG is uniform, *MinWork* is guaranteed to pick an optimal VDAG strategy under the linear work metric. We check how good the *MinWork* VDAG strategy is by comparing it with two others: a "dual-stage" VDAG strategy that only uses dual-stage view strategies, and a 1-way VDAG strategy that propagates the changes in an order opposite that of the *MinWork* VDAG strategy. *MinWork* uses the view ordering $\langle L, O, C, S, N, R \rangle$, and so the third VDAG strategy in our experiment uses the order $\langle R, N, S, C, O, L \rangle$. We call this strategy RNSCOL. As expected, the

MinWork strategy performed the best. In particular, it is 5.6 times better than the dual-stage VDAG strategy, and is 1.11 times better than the RNSCOL VDAG strategy.

More details about the experiments and an extended discussion of the results appear in [12].

7 Related Work

There has been a significant amount of work in minimizing warehouse maintenance time. The techniques proposed solve various sub-problems.

One of the sub-problems is the efficient maintenance of base views ([11],[7],[1]). In this paper, we concentrate on derived view maintenance. Unlike base view maintenance, derived view maintenance competes with OLAP queries for resources, and thus is one of the main problems that today's warehouses face.

Another important sub-problem is choosing the views to materialize in the warehouse so that some measure like query time, is minimized while satisfying a given storage or maintenance time constraint ([9],[10],[2],[16]). The warehouse design algorithms are complementary to the algorithms we present. That is, most of the design algorithms do not specify how views are actually updated. Our *MinWork* algorithm can be used for this purpose.

Another sub-problem that needs to be answered is deciding when to update the warehouse [4]. The algorithms we present are used when changes are actually propagated. Hence, the algorithms we present are complementary.

The only work that we know of that is concerned with the actual algorithm for propagating changes is [13]. More specifically, [13] proposed to represent the changes of summary tables as a *summary delta*. Since a summary delta can be incorporated into a summary table very efficiently, the main problem is computing the summary delta. The algorithms we present here can be used to compute the summary deltas more efficiently

Finally, the only work that we know of that handles a hierarchy of views instead of a single view is [15]. In [15], they focus more on the problem of maintaining views in a distributed warehouse (i.e., a set of data marts) consistently.

8 Conclusion

We have solved the "total-work minimization" (TWM) problem that warehouse administrators face today. To solve TWM, we presented *MinWorkSingle* that identifies optimal view strategies for updating single views. We then presented *MinWork*, an efficient heuristic algorithm that finds an optimal solution for a large class of VDAGs. *MinWork* significantly extends the 1-way view strategy ([8]) to the more practical setting of a VDAG of views. Experiments on a TPC-D VDAG showed that the strategies produced by *MinWorkSingle* and *MinWork* are very efficient under commercial RDBMS work metrics, and shrink the update window significantly.

References

[1] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient view maintenance in data warehouses. In *SIGMOD*, pages 417–425, 1997.

[2] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multi-dimensional datacube. In *VLDB*, pages 156–165, 1997.

[3] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, 1996.

[4] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple-view maintenance policies. In *SIGMOD*, pages 405–416, 1997.

[5] T. Committee. Transaction Processing Council. Available at: http://www.tpc.org/.

[6] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, 1995.

[7] A. Gupta, H. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *EDBT*, 1996.

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.

[9] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, 1997.

[10] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[11] P. Huyn. Multiple-view self-maintenance in data warehousing environment. In *VLDB*, pages 26–35, 1997.

[12] W. J. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the warehouse update window. Technical report, Stanford University, 1999. Available at http://www-db.stanford.edu/pub/papers/setvm.ps.

[13] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, pages 100–111, 1997.

[14] D. Quass. Maintenance expressions for views with aggregation. In *In Workshop on Materialized Views: Techniques and Applications*, June 1996.

[15] I. Stanoi, D. Agrawal, and A. E. Abbadi. Decentralized incremental maintenance of multi-view data warehouses. Technical report, UC Santa Barbara, 1999. http://www.cs.ucsb.edu/ ioana/TRCS99-04.ps.

[16] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in a data warehousing environment. In *VLDB*, pages 136–145, 1997.

[17] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, 1995.