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 algorithms 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.
Keywords: data warehousing, materialized-view update.

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 ([GL95], [Qua96]) 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.

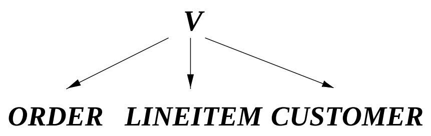

Figure 1: Example DAG of Materialized Views

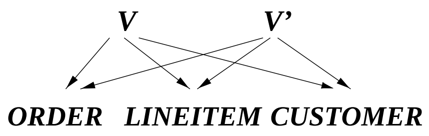

Figure 2: More Complex DAG

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 [ZGMHW95], 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 [CGL$^+$96]):

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.

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 [GMS93], 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. In particular, the strategies can result in significantly different update windows. For instance, we show later in the paper that if *CUSTOMER*, *ORDER* and *LINEITEM* are TPC-D relations [Com], and *V* is defined using the TPC-D "Shipping Priority" Query, the update window can be two to three times longer if Strategy 1 is used instead of Strategy 2! We show experimentally that for views with more complex definition than V, even larger disparities in update windows exist across different update strategies.

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, but in other cases Strategy 2 may be better. $\square$

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;
- 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.

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

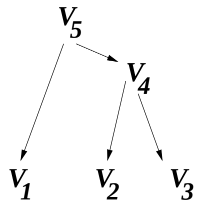

Figure 3: Example VDAG

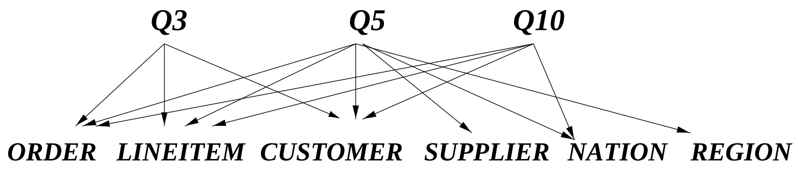

Figure 4: VDAG of a TPC-D Warehouse

- 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.
- We also develop a search algorithm called *Prune* that produces the least expensive update strategy for an even larger class of DAGs.
- Based on performance experiments with a TPC-D scenario, we demonstrate that the *MinWorkSingle* and *MinWork* update strategies shrink the update window significantly.

The rest of the paper is organized as follows. In Section 2, we formalize our warehouse model. Alternative update strategies for a DAG of views are discussed in Section 3. There we also define formally the problem of minimizing the work incurred. In Sections 4, 5 and 6 we present our algorithms and discuss practical issues surrounding their implementation. In Section 7, we show through experiments that our algorithms can significantly reduce update windows. Related work is discussed in Section 8.

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 information sources *base views*, and views defined over other views (at the warehouse) *derived views*. In today's warehouses, base views are often obtained by "cleansing" and "denormalizing" on-line transaction processing (OLTP) data. The resulting base views are often called "dimension tables" or "fact tables." Derived views, often called "summary tables," are defined over the cleansed base 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. Often, derived views that further summarize $Q3$, $Q5$ and $Q10$ can also be defined.

We define $Level(V)$ to be the maximum distance of V to a base view. For instance, in Figure 3, $Level(V_1) = Level(V_2) = Level(V_3) = 0$, $Level(V_4) = 1$, 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.[2] We use *delta relation* δV to represent the changes of V.

The changes of the base views arrive periodically at the warehouse. In today's warehouses, the period is often daily or weekly. 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 ([GL95], [Qua96]) 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.

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

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". For instance, the term $\sigma_{\mathcal{P}}(\delta V_2 \times V_3)$ involves joining the minus tuples of δV_2 with V_3 and storing them as minus tuples of δV_4, and doing the same for the plus tuples of V_2.

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*. 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.*, $\delta V_4 \leftarrow \sigma_{\mathcal{P}}(\delta V_2 \times V_3)$. Note that $Comp(V_4, \{V_2\})$, having just one term (*i.e.*, $\sigma_{\mathcal{P}}(\delta V_2 \times V_3)$), can be obtained from the expression for $Comp(V_4, \{V_2, V_3\})$ by assuming δV_3 is empty, and simplifying the expression.

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 of the VDAG. Finally, we formally define the "total-work minimization" (TWM) problem as finding the correct VDAG strategy that incurs the minimum amount of work.

[2] In some special cases (*e.g.*, if certain integrity constraints hold), V_i changes may not produce V_j changes.

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.

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

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 [CGL$^+$96] 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.[3]

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.

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

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 $\mathit{Inst}(V)$. 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.[4] For instance, another 1-way view strategy for V shown below processes the changes of V_n first, then of V_{n-1}, and so on. As we will see in subsequent sections, view strategies (2), (3) and (4) may incur significantly different amounts of work.

$$\langle\ \mathit{Comp}(V, \{V_n\}), \mathit{Inst}(V_n), \ldots, \mathit{Comp}(V, \{V_1\}), \mathit{Inst}(V_1), \mathit{Inst}(V)\ \rangle \tag{4}$$

Dual-stage view strategies as well as 1-way view strategies have been proposed in the literature ([GMS93], [CGL$^+$96]). However, the issue of finding optimal view strategies has not been studied. Moreover, we will see later that difficult problems arise when constructing correct and efficient VDAG strategies by combining optimal view strategies for individual views of the VDAG.

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.

[3] Actually, for a view defined over n other views, a total of $(n+1)!$ dual-stage view strategies can be obtained by reordering the *Inst* expressions. That is, once $\{\delta V_1, \ldots, \delta V_n\}$ are used to compute δV, the changes can be installed in any order. Fortunately, we only need to consider one dual-stage strategy per view since all dual-stage view strategies for a given view can be shown to incur the same amount of work.

[4] Actually, there are $2(n!)$ 1-way view strategies because the last two *Inst* expressions, *e.g.*, $\mathit{Inst}(V_n)$ and $\mathit{Inst}(V)$ in view strategy (3), can be swapped. However, it can be shown that swapping these expressions does not affect the work incurred by the view strategy. Hence, we only consider $n!$ 1-way 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

$$\sum_{k=1..n} \sum_{i=0..k-1} (-1)^k \frac{k!}{i! \cdot (k-i)!} (k-i)^n \tag{5}$$

view strategies that incur different amounts of work in general. To illustrate the enormity of the space of view strategies, Table 1 shows the result of Equation (5) for $n = 1$ to $n = 6$. Thus, according to Table 1, views $Q3$, $Q5$, and $Q10$ of the TPC-D VDAG (Figure 4) have 13, 4683, and 75 view strategies respectively.

n	# of view strategies
1	1
2	3
3	13
4	75
5	541
6	4683

Table 1: Number of View Strategies for a View Defined Over n Views

Formula (5) actually counts the number of 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.[5] 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:** $\forall E_i \in \overrightarrow{\mathcal{E}}$: $\forall E_j \in \overrightarrow{\mathcal{E}}$: $(i \neq j) \Rightarrow (E_i \neq E_j)$.

[5] Conditions **C1** and **C2**, and our algorithms can be extended to avoid using expressions that propagate and install δV_i when δV_i is empty.

□

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. For instance, both $Comp(V, \{V_i, V_j\})$ and $Comp(V, \{V_i, V_k\})$ cannot be simultaneously present in a correct view strategy. More specifically, **C3** states that $Inst(V_i)$ must be after both *Comp* expressions. On the other hand, if $Comp(V, \{V_i, V_j\}) < Comp(V, \{V_i, V_k\})$, **C4** states that $Inst(V_i)$ must be before $Comp(V, \{V_i, V_k\})$, a contradiction. Similarly, $Comp(V, \{V_i, V_j\}) < Comp(V, \{V_i, V_k\})$ also leads to a contradiction.

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 view in the VDAG.

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.

$$\begin{aligned}\langle\ &Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3),\\ &Comp(V_5, \{V_4\}), Inst(V_4), Comp(V_5, \{V_1\}), Inst(V_1), Inst(V_5)\ \rangle \end{aligned} \tag{6}$$

Note that VDAG strategy (6) "uses" (contains as a subsequence) the following correct view strategies for V_4 and V_5 respectively.

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

Also, for any base view V_i (*i.e.*, V_1, V_2, V_3), VDAG strategy (6) "uses" $\langle\ Inst(V_i)\ \rangle$. □

The previous example illustrated that a correct VDAG strategy uses correct view strategies to update each view in the VDAG. However, we know that starting from a set of correct view strategies, one for each view of the VDAG, we may not be able to construct a correct VDAG strategy (see Example 1.2 of Section 1). In Sections 5 and 6, we present algorithms that not only find correct VDAG strategies but also ensure that the strategies they produce are very efficient. 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 (TWM) Problem**) 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."

In order to estimate $Work(E_i)$, various metrics can be used. 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 in this paper produce optimal update strategies under the linear work metric. In Section 7, 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). In general, we assume that a compute expression of the form $Comp(W, \mathcal{Y})$ has a total of $2^{|\mathcal{Y}|} - 1$ terms, where each term considers a combination of delta or non-delta forms of the views in $\mathcal{Y}$. For example, in $Comp(W, \{V_1, V_2\})$, one term evaluates the changes of W based on δV_1 and V_2, a second term computes W changes based on V_1 and δV_2, and the third term considers δV_1 and δV_2. Each of these terms must in addition consider the rest of the views that participate in the definition of W. In our example, if W is defined over V_1, V_2 and V_3, then the first term of $Comp(W, \{V_1, V_2\})$ will have as input δV_1, V_2 and V_3; the second term will have as input V_1, δV_2 and V_3; and the final term will have as input δV_1, δV_2 and V_3. We consider an execution model that evaluates each of these terms 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.

Notice that our term-execution model is independent of the specifics of the view definitions. Incremental view maintenance expressions for views involving arbitrary select, project, join operations, followed by arbitrary aggregate operations fit this pattern. Thus, the results we develop in this paper are valid for all these maintenance expressions. We now formally state our work metric based on the term-execution model discussed above.

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 $c \cdot ((|\delta V_2| + |V_3|) + (|\delta V_3| + |V_2|) + (|\delta V_2| + |\delta V_3|))$. Finally, note that the work estimate for $Inst(V_4)$ is $i \cdot |\delta V_4|$, where i is a proportionality constant. □

The linear work metric is similar to metrics that have been used in state-of-the-art algorithms for warehouse design ([HRU96], [SDN98]), and it can be quite effective in modeling complex update computations. Estimating the work of an install expression as being proportional to the size of the delta relation is reasonable because the expression needs to scan in the delta relation to install the changes. When estimating the work of a compute expression, we note that each term in the compute expression contains at least one delta relation. Since delta relations tend to be small, all 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. Accordingly, we estimate the work of a term as being proportional to the sum of the sizes of its operands. Then, the work estimate of a compute expression is obtained by adding the work estimates of all the terms in the compute expression.

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. In Section 7, we will show that even if the underlying database does not have a linear work metric, the *MinWorkSingle* view strategy is still very efficient.

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 Appendix A. 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 the set of 1-way view strategies is much smaller than the set of all view strategies. For instance, the view $Q5$ in Figure 4 has a total of 4683 view strategies, out of which only 720 are 1-way. Thus, the search for an optimal view strategy can be limited 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.

$$\langle\ Comp(V_4, \{V_2\}), Inst(V_2), Comp(V_4, \{V_3\}), Inst(V_3), Inst(V_4)\ \rangle \tag{7}$$

$$\langle\ Comp(V_4, \{V_3\}), Inst(V_3), Comp(V_4, \{V_2\}), Inst(V_2), Inst(V_4)\ \rangle \tag{8}$$

Clearly, the work incurred by the *Inst* expressions (*i.e.*, $Inst(V_2)$, $Inst(V_3)$, $Inst(V_4)$) 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 (7) 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 (8) 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 (8), while V_3 is used in view strategy (7).

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 (8)), 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 $c \cdot (|V_3'| - |V_3|)$. (Of course, if $(|V_3'| - |V_3|)$ is negative, the work incurred actually decreases.) Similarly if we first propagate and install the changes to V_2 (as in view strategy (7)), 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 every $Inst(V_i) < Inst(V_j)$ in the strategy (where $V_i \neq V$ and $V_j \neq V$), then $V_i < V_j$ in $\overrightarrow{\mathcal{V}}$.

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.*

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 $\overrightarrow{\mathcal{E}}$ $\diamond$
```

Figure 5: *MinWorkSingle* Algorithm

The main intuition behind the proof (in Appendix A) was illustrated by Example 4.1.

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 correct since it satisfies the conditions for a correct view strategy (Definition 3.1). Specifically, *MinWorkSingle* appends all the necessary *Comp* and *Inst* expressions (Lines 5–7) required by **C1** and **C2**. By appending $Inst(V_i)$ right after $Comp(V, \{V_i\})$ and before the next *Comp* expression, *MinWorkSingle* guarantees that the output view strategy satisfies **C3** and **C4**. Appending $Inst(V)$ last ensures that **C5** is satisfied. Since *MinWorkSingle* does not duplicate any expression, **C6** is satisfied.

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

Theorem 4.3 *Given a view defined over n other views in the warehouse, 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 VDAG strategies that may not be optimal. In this section, we study the conditions required to be satisfied by a VDAG for our algorithm to obtain an optimal VDAG strategy. Based on these conditions, we 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 (presented in Appendix A), we further argue that a VDAG strategy that uses optimal view strategies minimizes the total amount of 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 the *desired view ordering.* Note that the desired view ordering depends on the current set of changes.

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 the 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 (same as Figure 3 copied over for local reference). 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 a desired view ordering is

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

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

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

□

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

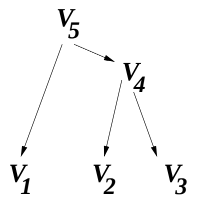

Figure 6: VDAG

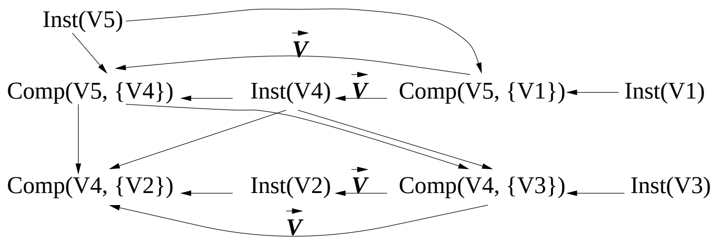

Figure 7: Expression Graph (EG)

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 $\vec{\mathcal{V}}$, the expression graph of G with respect to $\vec{\mathcal{V}}$, denoted $EG(G, \vec{\mathcal{V}})$, 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 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 $EG(G, \vec{\mathcal{V}})$ is acyclic where $\vec{\mathcal{V}}$ is a desired view ordering, a topological sort of $EG(G, \vec{\mathcal{V}})$ yields an optimal VDAG strategy for G.* □

The proof of the theorem is in Appendix A where we show that the topological sort of $EG(G, \vec{\mathcal{V}})$ results in a 1-way VDAG strategy that is consistent with the desired view ordering $\vec{\mathcal{V}}$. 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 $\vec{\mathcal{V}}$ 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 $\vec{\mathcal{V}}$. 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. This dependency is due to **C8**.

Some edges of the expression graph are shown with a label $\vec{\mathcal{V}}$ 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 $\vec{\mathcal{V}}$ requires that the changes of V_2 be propagated before the changes of V_3 (note that $V_2 < V_3$ in $\vec{\mathcal{V}}$).

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 $\vec{\mathcal{V}}$. 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. Trivial variations of this optimal VDAG strategy may be obtained by other topological sorts. □

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. (In fact, we can show that if the edges due to the view ordering dependencies are removed, the resulting expression graph is always acyclic.) 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. The important thing about these classes of VDAGs is that for these 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. □

The class of tree VDAGs may appear very simple, but it encompasses a large number of VDAGs that occur naturally in many warehouse contexts. A simple example of a tree VDAG is shown in Figure 6. Based on the following lemma, one can easily find optimal VDAG strategies for tree VDAGs. by constructing their expression graphs with desired view orderings and topologically sorting them. The proof of the lemma is furnished in Appendix A.

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

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

Uniform VDAGs have a well-defined notion of *Level* for each view. The TPC-D warehouse shown in Figure 4 has a uniform VDAG. In this uniform VDAG, all base views have *Level* 0 and all derived views have *Level* 1. The class of uniform VDAGs, although quite large, does not encompass the class of tree VDAGs. For instance, the tree VDAG of Figure 6 is not a uniform VDAG. At the same time, there are uniform VDAGs that are not tree VDAGs. For instance, the uniform VDAG for the TPC-D warehouse (Figure 4) is not a tree VDAG.

Based on the following lemma, we can easily generate optimal VDAG strategies for uniform VDAGs. The proof of the lemma is furnished in Appendix A.

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

Algorithm 5.1 *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 $\overrightarrow{\mathcal{E}}$ $\diamond$
```

Algorithm 5.2 *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 $\overrightarrow{\mathcal{V}'}$ $\diamond$
```

Figure 8: *MinWork* Algorithm

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. 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 5.1 of Figure 8.

As shown in the figure, *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 Appendix B). *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 the dependencies imposed 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 5.2, Figure 8) 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 with respect to the desired view ordering, *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 (proven in Appendix A) ensures that *MinWork* will always be able to generate a 1-way VDAG strategy no matter how complex the input VDAG is using the modified view ordering.

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 near optimal plans, when it misses optimal plans.

Finally, we note that *MinWork* has a worst case time complexity of $O(n^3)$ where n is the number of views in the VDAG. The most complex part of the algorithm, taking $O(n^3)$ time, is building the expression graph using *ConstructEG* (see Appendix B). All other parts take at most $O(n^2)$ time.

5.5 Practical Issues

We now outline how to resolve a number of practical and important issues regarding the implementation of *MinWork* on top of a commercial RDBMS. In particular we discuss the following issues: (1) how to implement *MinWork* using SQL stored procedures and a high level programming language like C++; (2) how to determine a desired view ordering. We provide this discussion to show that if the warehouse is built on top of a commercial RDBMS, *MinWork* can be implemented by a WHA easily without changing the internals of the RDBMS.

Implementing *MinWork*: The key observation is that given a VDAG, the set of 1-way expressions used by the *MinWork* VDAG strategy is known a priori. That is, for each edge $V_j \rightarrow V_i$ in the VDAG, a compute expression $Comp(V_j, \{V_i\})$ will be used, and for each node V_i in the VDAG, an install expression $Inst(V_i)$ will be used. Only the order of the expressions in the strategies depends on the changes being processed at the warehouse. Hence, based on the VDAG of the warehouse, a set of stored procedures is defined, one for each compute or install expression. This leads to efficient execution of the VDAG strategy because the stored procedures need not be parsed and go through all the optimization steps every time the warehouse needs to be updated.

Using the above technique, we define the following approach to warehouse update processing:

1. Given a set of view definitions, the corresponding VDAG is generated.
2. Given the VDAG generated in the previous step, the set of stored procedures for the compute and install expressions are defined.
3. Each time the warehouse needs to be updated, *MinWork* is invoked to produce a VDAG strategy.
4. The resulting VDAG strategy is executed with the help of the stored procedures defined in the second step.

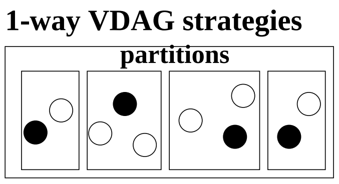

Figure 9: Intuition of *Prune*

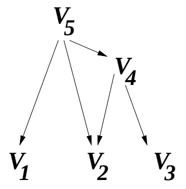

Figure 10: Problem VDAG

Computing a desired view ordering: Recall that *MinWork* needs to find a desired view ordering which is based on $|V'| - |V|$ values for each view V. Estimates of $|V|$ should be available from the metadata. To estimate $|V'|$, we can first estimate $|\delta V|$, and then compute $|V'|$ based on $|\delta V|$ and $|V|$. Estimates of $|\delta V|$ are obtained easily for base views since the changes are provided before the warehouse update starts. Estimates of $|\delta V|$ for derived views can be obtained using standard query result size estimation methods [Ull89]. That is, assuming V is defined over views $\{V_1, \ldots, V_n\}$, and estimates of $\{|\delta V_1|, \ldots, |\delta V_n|, |V_1'|, \ldots, |V_n'|\}$ have been obtained (*i.e.*, we proceed bottom-up), $|\delta V|$ can be estimated using standard methods.

6 Optimal 1-way VDAG Strategies

We just showed that for VDAGs and view orderings that result in acyclic expression graphs, an optimal VDAG strategy can be obtained efficiently using *MinWork*. If the expression graph is cyclic, finding an optimal VDAG strategy is very hard. However, since we showed that certain 1-way VDAG strategies are optimal in many scenarios, we focus on the problem of finding the best 1-way VDAG strategy. In this section, we present a search algorithm called *Prune* that avoids examining much of the solution space but is guaranteed to produce the best 1-way VDAG strategy.

Even though *Prune* restricts its search to 1-way VDAG strategies, the set of 1-way VDAG strategies is still potentially very large. *Prune* pares down the search space by partitioning the set of 1-way VDAG strategies and considering only one representative VDAG strategy from each partition. Figure 9 illustrates how the technique reduces the search space. In the figure, each point represents a 1-way VDAG strategy, but only the marked points are considered by *Prune*.

Prune partitions the 1-way VDAG strategies based on which view ordering the 1-way VDAG strategies are *strongly consistent* with. A 1-way VDAG strategy $\overrightarrow{\mathcal{E}}$ is strongly consistent with a view ordering $\overrightarrow{\mathcal{V}}$ if $Inst(V_i) < Inst(V_j)$ in $\overrightarrow{\mathcal{E}}$ implies that $V_i < V_j$ in $\overrightarrow{\mathcal{V}}$. Partitioning 1-way VDAG strategies in this way is correct because each 1-way VDAG strategy is strongly consistent with exactly one view ordering, as stated in the next lemma.[6] Hence, all 1-way VDAG strategies that are strongly consistent with the same view ordering are placed in the same partition.

Lemma 6.1 *Every 1-way VDAG strategy is strongly consistent with some view ordering $\overrightarrow{\mathcal{V}}$. Furthermore, a 1-way VDAG strategy is strongly consistent with exactly one view ordering $\overrightarrow{\mathcal{V}}$.* □

Lemma 6.1 follows from the fact that any VDAG strategy $\overrightarrow{\mathcal{E}}$ must have exactly one *Inst* expression for each VDAG view (*i.e.*, by **C6** and **C7**). Hence, $\overrightarrow{\mathcal{E}}$ must be strongly consistent with the view ordering $\overrightarrow{\mathcal{V}}$

[6] On the other hand, 1-way VDAG strategies cannot be partitioned based on which view ordering the 1-way VDAG strategies are consistent with because a 1-way VDAG strategy may be consistent with more than one view ordering.

(and no other view ordering) that orders all of the VDAG views based on the order of appearance of the *Inst* expressions in $\overrightarrow{\mathcal{E}}$.

The next theorem states that all the VDAG strategies in a partition incur the same amount of work. The theorem holds because 1-way VDAG strategies use the same set of expressions. Furthermore, it can be shown that if two VDAG strategies are strongly consistent with the same view ordering, each *Comp* expression runs on the same "database state" in both VDAG strategies. Since we know that the work incurred by any *Inst* expression is the same, the theorem follows.

Theorem 6.1 *Given a view ordering $\overrightarrow{\mathcal{V}}$, all the 1-way VDAG strategies that are strongly consistent with $\overrightarrow{\mathcal{V}}$ incur the same amount of work.*

In summary, based on the above theorems, *Prune* can search over 1-way VDAG strategies by considering the set of view orderings and by examining one 1-way VDAG strategy that is strongly consistent with each ordering. However, *Prune* needs to handle the complication that given a view ordering $\overrightarrow{\mathcal{V}}$, there may not exist a (correct) 1-way VDAG strategy that is strongly consistent with $\overrightarrow{\mathcal{V}}$. For instance, for the VDAG shown in Figure 10, there is no 1-way VDAG strategy that is strongly consistent with $\overrightarrow{\mathcal{V}} = \langle V_4, V_1, V_2, V_3, V_5\rangle$. This is because $Comp(V_4, \{V_3\})$ must be after $Inst(V_2)$ for **C4** to hold, and for the VDAG strategy to be strongly consistent with $\overrightarrow{\mathcal{V}}$. However, $Comp(V_4, \{V_3\})$ must be before $Inst(V_4)$ and therefore before $Inst(V_2)$, for **C8** to hold.

To handle this complication, *Prune* (Figure 11) constructs a *strong expression graph* (SEG) that is similar to the expression graph that *MinWork* constructs. If a cyclic SEG is constructed, then there is no 1-way VDAG strategy that is strongly consistent with the given view ordering. Otherwise, *Prune* produces a candidate 1-way VDAG strategy by topologically sorting the expressions in the SEG. *Prune* returns the 1-way VDAG strategy that incurs the least amount of work.

To construct the SEG, *Prune* uses *ConstructSEG* (not shown here) which is almost identical to *ConstructEG*. The only difference is that *ConstructSEG* adds an edge $Inst(V_j) \rightarrow Inst(V_i)$ if V_i is before V_j in the input view ordering $\overrightarrow{\mathcal{V}}$. Unlike *ConstructEG*, *ConstructSEG* adds this edge even when there is no view V that is defined on both V_i and V_j. This edge guarantees that $Inst(V_i)$ is before $Inst(V_j)$ in the topological sort (if possible) of the constructed SEG. This in turn guarantees that the 1-way VDAG strategy produced is strongly consistent with $\overrightarrow{\mathcal{V}}$.

We note that *Prune* examines $n!$ view orderings, where n is the number of VDAG views. Also, *ConstructSEG*, like *ConstructEG*, runs in $O(n^3)$ time in building an SEG. Since an SEG needs to be constructed for each view ordering, *Prune* runs in $O(n! \cdot n^3)$ time.

Compared with the space of all 1-way VDAG strategies for a given VDAG, *Prune* searches over a very small set of 1-way VDAG strategies and thus is relatively quite efficient. However, it can be improved further while still guaranteeing that an optimal 1-way VDAG strategy is produced. For instance, it is not necessary to examine all possible view orderings. More specifically, if there are no views defined on V, δV can be installed at any point in the VDAG strategy after δV has been computed. If we remove all such views from the view ordering, we only need to consider $O(m!)$ view orderings, where m is the number of VDAG views with a view defined on them. For instance, for the TPC-D VDAG shown in Figure 4, there are $n = 9$ views, but there are only $m = 6$ views with some view defined over them. Hence, *Prune* can be optimized to examine only $6! = 720$ strategies, instead of $9! = 362880$ strategies.

```
Algorithm 6.1 Prune
Input: $G = \langle \mathcal{V}, \mathcal{A} \rangle$
Output: an optimal VDAG strategy $\overrightarrow{\mathcal{E}}$
  1. $\overrightarrow{\mathcal{E}_{best}} \leftarrow \langle\ \rangle$ // incorrect VDAG strategy with infinite amount of work
  2. For each view ordering $\overrightarrow{\mathcal{V}}$
    3. SEG $\leftarrow ConstructSEG(G, \overrightarrow{\mathcal{V}})$
    4. If SEG is acyclic Then
      5. $\overrightarrow{\mathcal{E}} \leftarrow$ topological sort of the expressions in SEG
      6. If $Work(\overrightarrow{\mathcal{E}}) < Work(\overrightarrow{\mathcal{E}_{best}})$ Then
        7. $\overrightarrow{\mathcal{E}_{best}} \leftarrow \overrightarrow{\mathcal{E}}$
  8. Return $\overrightarrow{\mathcal{E}_{best}}$ $\diamond$
```

Figure 11: *Prune* Algorithm

Just like *MinWork*, *Prune* should be implemented by creating SQL stored procedures for each expression that will be used by the *Prune* VDAG strategy. Observe also that *Prune* (Figure 11) compares various VDAG strategies in terms of their total work under the linear work metric. In order to estimate the total work of a VDAG strategy, we need estimates for $|V|$, $|\delta V|$ and $|V'|$ for each view V. Standard result size estimation methods can be used for this purpose.

7 Experiments and Discussion

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. When a strategy is executed, many factors that affect the update window (e.g., buffering of the intermediate results and the particular join and aggregation methods used in computing these intermediate results) are too complex to be modeled by our simple work metric.

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 Dell XPS D300 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 of our experiments 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 [Com]. 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 $\langle\ L, O, C, S, N, R\ \rangle$ (*i.e.*, L is the largest base view). (Note that the three derived views can be ignored in the view ordering since there are no views defined on them.) In one of the experiments, we investigated other possible changes to the remote information sources.

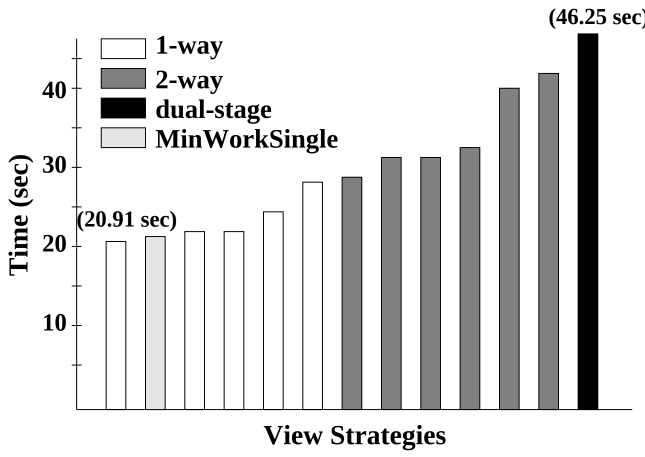

Figure 12: $Q3$ View Strategies

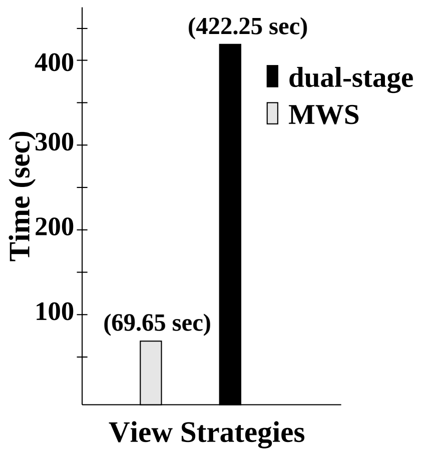

Figure 13: $Q5$ View Strategies

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 12 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. That is, the dual-stage view strategy is worse than all of the 1-way view strategies. Also, any "2-way" view strategy that uses an expression $Comp(Q3, \mathcal{V})$, where $|\mathcal{V}| = 2$, is worse than all of the 1-way view strategies.

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 in Section 4, 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 and hence *MinWorkSingle* ends up with a view strategy that is slightly away from the optimum. Notice that the margin of error is small, indicating that using the linear work metric, one can generate near-optimal update windows.

Finally, the graph shows that various view strategies have significantly different update windows. For instance, the update window of the dual-stage view strategy is about 2.3 times longer than that of the optimal view strategy.

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 [CGL$^+$96]. The results of the experiment are shown in Figure 13. Notice that the update window of the dual-stage view strategy is over 6 times longer than that of the *MinWorkSingle* view strategy. On the other hand, the update window of the dual-stage view strategy for $Q3$ was "only" 2.2 times longer than that of the *MinWorkSingle* view strategy (see Figure 12). This shows that using the *MinWorkSingle* view strategy instead of the dual-stage view strategy to update complex views is likely to be very beneficial.

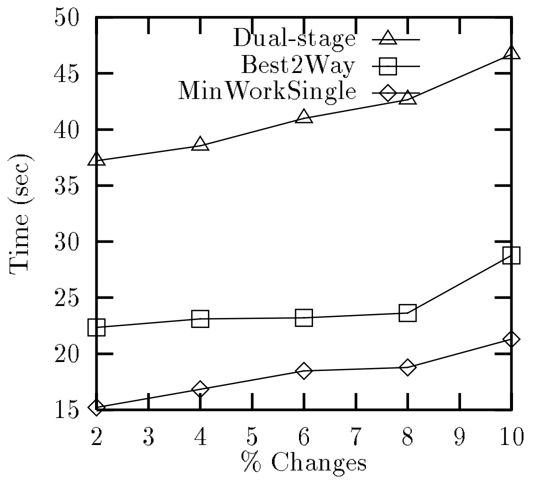

Figure 14: $Q3$ View Strategies Under Different Changes

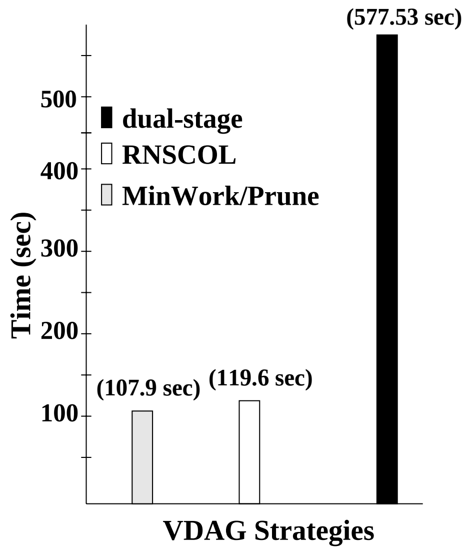

Figure 15: VDAG Strategies

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. When comparing view strategies, we only considered the *MinWorkSingle* view strategy, the best 2-way view strategy in Figure 12, and the dual-stage view strategy. Figure 14 shows the results of the experiment. The results indicate that the *MinWorkSingle* view strategy improves on the other view strategies over a wide range of amounts of changes to the underlying views.

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. The results of the experiment are shown in Figure 15. As expected, the *MinWork* strategy performed the best. In particular, it is 5 to 6 times better than the dual-stage VDAG strategy, and is about 11% better than the RNSCOL VDAG strategy.

Discussion: Although the dual-stage VDAG strategy has a very long update window compared to the two 1-way VDAG strategies, it does have the advantage of being able to perform all of the *Inst* expressions in the second stage, which minimizes the time in which locking operations are necessary. However, even though the sequence of *Comp* expressions in the first stage do not need to lock the database, they still compete with OLAP queries for resources. On the other hand, the sequence of expressions used by the 1-way VDAG strategies are more efficient and take less resources away from OLAP queries. Moreover, it is often acceptable for OLAP queries to run at lower isolation levels, which allows the *Inst* expressions to run without locking. This diminishes any advantage the dual-stage VDAG strategy has over the 1-way VDAG strategies.

We also note that the results of Experiment 4 suggests that the linear work metric is a good measure of the work incurred by a VDAG strategy. For instance, a variant of the linear work metric may just sum the sizes of the operands. To illustrate, the work estimate for $Comp(V_4, \{V_2, V_3\})$ (Figure 10) under this variant is $c \cdot (|\delta V_2| + |V_2| + |\delta V_3| + |V_3|)$, since the number of terms in which an operand V or δV appears in is not modeled. Under this work metric, the dual-stage VDAG strategy would be best contrary to the results of Experiment 4.

8 Related Work

There has been a significant amount of work in minimizing warehouse maintenance time. This is because there are many techniques, each solving a different sub-problem.

One of the sub-problems is the efficient maintenance of base views that are defined over remote sources. Hence, there have been previous work ([QGMW96], [Huy97], [GJM96]) that determines when a base view can be maintained without accessing remote sources. If these remote sources do need to be accessed, [AASY97] gives algorithms for base view maintenance. In this paper, we concentrate on derived view maintenance. Even though maintaining derived views only requires accessing data local to the warehouse, it can be a very expensive process. Furthermore, 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, maintenance time, or a combination of the two, is minimized while satisfying a given storage or maintenance time constraint. Warehouse design has been discussed in [Gup97],[HRU96],[BPT97], [YKL97], [TS97]. The warehouse design algorithms are complementary to the algorithms we present. Most of the warehouse design algorithms, such as the greedy algorithm of [Gup97], do not specify how views are actually updated. On the other hand, we give algorithms that update views in a very specific manner. Hence, our algorithms can be combined with design algorithms in many ways. One way is that a design algorithm picks the set of views $\mathcal{V}$ to materialize. The algorithms we present are then used to update the views in $\mathcal{V}$ once they are materialized.

Another sub-problem that needs to be tackled is to develop a good storage representation for views so that incorporating bulk changes into the views can be done efficiently. Recently, [KR98] proposed a variant of R-trees called *Cubetrees* as the storage representation of the views. Reference [JNSS97] also discussed how to incorporate changes quickly into a clustered storage organization using sorting and hashing techniques. The storage representation presented in these papers can be used in conjunction with the algorithms we present.

Another sub-problem that needs to be answered is deciding when to update the warehouse. Reference [CKL+97] presents a framework for supporting different maintenance policies based on when changes are propagated to the views. On the other hand, 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 [MQM97]. More specifically, [MQM97] proposed to represent the changes of summary tables as a *summary delta* (*i.e.*, result of applying the grouping operator and aggregation functions over the changes). 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 algorithms we present are different from most of the previous algorithms since our algorithms are concerned with a DAG of views, instead of just one view. In this context, a careful treatment is required to maintain a DAG of views correctly and efficiently.

9 Parallel Strategies

In this paper, we have modeled a VDAG strategy as a *sequence of expressions* $\overrightarrow{\mathcal{E}}$, wherein each expression is sent one at a time to the underlying database. An alternative model of a VDAG strategy is a *sequence of expression sets* denoted $\overrightarrow{\mathcal{S}}$, wherein each set can be answered by the database in parallel. One of the techniques for solving a problem involving parallel processing is to "parallelize" a solution of the sequential problem. Hence, one approach is to parallelize the *MinWork* VDAG strategy $\overrightarrow{\mathcal{E}}$ to produce $\overrightarrow{\mathcal{S}}$.

However, parallelizing the *MinWork* VDAG strategy may not be the best approach since the *MinWork* VDAG strategy only uses 1-way view strategies which require certain compute and install expressions to be performed before other expressions. Because of these numerous dependencies, many of the expressions in the *MinWork* VDAG strategy cannot be processed in parallel.

We have identified two techniques that allow more expressions to be processed in parallel.

1. The first technique is to use dual-stage view strategies (*i.e.*, view strategies that propagate the underlying changes simultaneously) instead of 1-way view strategies.
2. If we only use dual-stage view strategies, we can remove any remaining dependencies among the expressions by "flattening" the VDAG. For instance, let us consider the VDAG in Figure 10. When updating V_5, it may be possible to treat V_5 as if it was defined on V_1, V_2 and V_3 instead of V_4. If so, then the compute expressions of V_5 and V_4 can run in parallel.

Unfortunately, using these techniques increase the total work incurred by the VDAG strategy. As a result, any benefit that arises from allowing more expressions to run in parallel may be offset by an increase in total work. In summary, it is an important future work to have an algorithm that intelligently decides the extent to which these techniques should be applied.

10 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. To find an optimal 1-way VDAG strategy for any VDAG, we presented *Prune* which is a search technique that avoids considering a large part of the solution space. Both *MinWork* and *Prune* significantly extend the 1-way view strategy ([GMS93]) 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. We also discussed how the algorithms can be easily implemented without modifying the internals of a commercial RDBMS.

References

[AASY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance in data warehouses. In *SIGMOD*, pages 417–425, 1997.

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multi-dimensional datacube. In *VLDB*, pages 156–165, 1997.

[CGL$^+$96] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, 1996.

[CKL$^+$97] 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.

[Com] TPC Committee. Transaction Processing Council. Available at: http://www.tpc.org/.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GJM96] A. Gupta, H. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *EDBT*, 1996.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, 1995.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, 1997.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[Huy97] P. Huyn. Multiple-view self-maintenance in data warehousing environment. In *VLDB*, pages 26–35, 1997.

[JNSS97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, and S. Sudarshan. Incremental organization for data recording and warehousing. In *VLDB*, pages 16–25, 1997.

[KR98] Y. Kotidis and N. Roussopoulos. An alternative storage organization for rolap aggregate views based on cubetrees. In *VLDB*, pages 249–258, 1998.

[LYGM98] W. J. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the warehouse update window. Technical report, Stanford University, 1998. Available at http://www-db.stanford.edu/wilburt/vm.ps.

[MQM97] 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.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *PDIS*, pages 158–169, 1996.

[Qua96] D. Quass. Maintenance expressions for views with aggregation. In *In Workshop on Materialized Views: Techniques and Applications*, June 1996.

[SDN98] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, pages 488–499, 1998.

[TS97] D. Theodoratos and T. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135, 1997.

[Ull89] J. D. Ullman. *Database and Knowledge-Base Systems, Vol.2*. Computer Science Press, 1989.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in a data warehousing environment. In *VLDB*, pages 136–145, 1997.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, 1995.

A Proof Sketches of Theorems

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

Proof: Let view W be defined over the set of views $\mathcal{V}$. We show that given any non-1-way view strategy for W we can find a 1-way view strategy that is at least as efficient as the non-1-way view strategy.

Consider a non-1-way view strategy for W:

$$\overrightarrow{\mathcal{E}} = \langle\ \overrightarrow{\mathcal{E}_{prec}}, Comp(W, \mathcal{Y}), \overrightarrow{\mathcal{E}_{inst}}, \overrightarrow{\mathcal{E}_{succ}}\ \rangle.$$

In $\overrightarrow{\mathcal{E}}$, $\mathcal{Y}$ is a subset of $\mathcal{V}$ and $|\mathcal{Y}| > 1$; $\overrightarrow{\mathcal{E}_{prec}}$ is a possibly empty sequence of expressions preceding $Comp(W, \mathcal{Y})$; $\overrightarrow{\mathcal{E}_{inst}}$ is the sequence of $Inst$ expressions immediately following $Comp(W, \mathcal{Y})$ that installs the changes of $\mathcal{Y}$; $\overrightarrow{\mathcal{E}_{succ}}$ is the sequence of expressions that completes the view strategy.

We define a mapping called "separator" that transforms $\overrightarrow{\mathcal{E}}$ (based on some $Y_1 \in \mathcal{Y}$) into:

$$\overrightarrow{\mathcal{E}'} = \langle\ \overrightarrow{\mathcal{E}_{prec}}, Comp(W, \{Y_1\}), Inst(Y_1), Comp(W, \mathcal{Y} - \{Y_1\}), \overrightarrow{\mathcal{E}'_{inst}}, \overrightarrow{\mathcal{E}_{succ}}\rangle.$$

In $\overrightarrow{\mathcal{E}'}$, $\overrightarrow{\mathcal{E}_{prec}}$ and $\overrightarrow{\mathcal{E}_{succ}}$ are the same as in $\overrightarrow{\mathcal{E}}$, while $\overrightarrow{\mathcal{E}'_{inst}}$ is almost identical to $\overrightarrow{\mathcal{E}_{inst}}$ except that it does not have $Inst(Y_1)$ in it. Intuitively, the transformation "separates" the propagation of the changes of Y_1 from $Comp(W, \mathcal{Y})$, as well as the installation of the changes of Y_1 from the sequence $\overrightarrow{\mathcal{E}_{inst}}$.

It can be verified easily that conditions **C1** through **C6** hold for $\overrightarrow{\mathcal{E}'}$ if they hold for $\overrightarrow{\mathcal{E}}$. This implies that $\overrightarrow{\mathcal{E}'}$ is correct if $\overrightarrow{\mathcal{E}}$ is correct. So, "separator" preserves the correctness of the view strategy.

We now show that each application of "separator" results in a view strategy that is at least as efficient as the original view strategy. All the expressions in $\overrightarrow{\mathcal{E}_{prec}}$ and $\overrightarrow{\mathcal{E}_{succ}}$ incur the same amount of work in both $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$, since the same expressions are used and the expressions are evaluated on the same database state. Also, the install expressions in $\overrightarrow{\mathcal{E}_{inst}}$ incur the same amount of work as $Inst(Y_1)$ and the install expressions in $\overrightarrow{\mathcal{E}'_{inst}}$ since the same changes are installed. Hence, we must show that the compute expressions $Comp(W, \{Y_1\})$ and $Comp(W, \mathcal{Y} - \{Y_1\})$ in $\overrightarrow{\mathcal{E}'}$ do not incur more work than the compute expression $Comp(W, \mathcal{Y})$ in $\overrightarrow{\mathcal{E}}$.

Without loss of generality, let $\mathcal{Y} = \{Y_1, \ldots, Y_m\}$ and $\mathcal{V} = \mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z}$, where $\mathcal{X} = \{X_1, \ldots, X_l\}$ and $\mathcal{Z} = \{Z_1, \ldots, Z_n\}$. Furthermore, suppose the changes of the views in $\mathcal{X}$ are propagated and installed in $\overrightarrow{\mathcal{E}_{prec}}$, while the changes of the views in $\mathcal{Z}$ are propagated and installed in $\overrightarrow{\mathcal{E}_{succ}}$. Finally, let us denote the extension of a view V after its changes have been installed as V'.

Note that $Comp(W, \mathcal{Y})$ of $\overrightarrow{\mathcal{E}}$ has $2^m - 1$ terms, since $|\mathcal{Y}| = m$ (see Section 3.3). We denote each of these terms as A_v, where v is a bit vector composed of m bits, whose values depend on the views and delta relations accessed by the term. The ith bit in v is set if term A_v accesses δY_i instead of Y_i, and vice versa. For instance, if $\mathcal{Y} = \{Y_1, Y_2\}$, there will be three terms: term A_{10} combines δY_1 and Y_2, term A_{01} combines Y_1 and δY_2, and term A_{11} combines δY_1 and δY_2. Each of these three terms accesses $\{X'_1, \ldots, X'_l\}$ and $\{Z_1, \ldots, Z_n\}$ as well.

In $\overrightarrow{\mathcal{E}'}$, $Comp(W, \mathcal{Y} - \{Y_1\})$ has $(2^{m-1} - 1)$ terms. We denote each term of $Comp(W, \mathcal{Y} - \{Y_1\})$ as C_v, where v is an $(m-1)$ bit vector. The ith bit of v is set if term C_v accesses δY_{i+1} instead of Y_{i+1} and vice versa. We map each term of $Comp(W, \mathcal{Y} - \{Y_1\})$ to a pair of terms of $Comp(W, \mathcal{Y})$. In particular, term C_v is mapped to the pair of terms A_{0v} and A_{1v}. The work incurred by term C_v is

$$c \cdot (|X'_1| + \ldots + |X'_l| + |Y'_1| + K + |Z_1| + \ldots + |Z_n|),$$

where K is the sum of the sizes of the views and delta relations of $\mathcal{Y} - \{Y_1\}$ considered by C_v.

On the other hand, the work incurred by the pair of terms A_{1v} and A_{0v} is

$$c \cdot (2 \cdot |X'_1| + \ldots + 2 \cdot |X'_l| + (|Y_1| + |\delta Y_1|) + 2 \cdot K + 2 \cdot |Z_1| + \ldots + 2 \cdot |Z_n|).$$

Since $|Y_1| + |\delta Y_1|$ is at least as large as $|Y'_1|$, we infer that the term C_v does not incur more work than the pair of terms A_{0v} and A_{1v} that it is mapped to.

$Comp(W, \{Y_1\})$ has exactly one term, which accesses $\{X'_1, \ldots, X'_l\}$, $\{\delta Y_1, Y_2, \ldots, Y_m\}$, and $\{Z_1, \ldots, Z_n\}$. We map it to term $A_{10..0}$ of $Comp(W, \mathcal{Y})$, which incurs the same amount of work (because it uses the same combination of views and delta relations).

So far, we have shown that each term of $Comp(W, \mathcal{Y} - \{Y_1\})$ and $Comp(W, \{Y_1\})$ can be mapped to $Comp(W, \mathcal{Y})$ terms that incur the same or a larger amount of work. One can see that no $Comp(W, \mathcal{Y})$ term participates in the mapping of two different terms of $Comp(W, \mathcal{Y} - \{Y_1\})$ and $Comp(W, \{Y_1\})$. Firstly, any two $Comp(W, \mathcal{Y} - \{Y_1\})$ terms, C_v and $C_{v'}$, are mapped to two disjoint sets of $Comp(W, \mathcal{Y})$ terms. That is, C_v maps to A_{0v} and A_{1v}, while $C_{v'}$ maps to $A_{0v'}$ and $A_{1v'}$. Since $v \neq v'$, it follows that A_{0v} is not $A_{0v'}$, and A_{1v} is not $A_{1v'}$. Secondly, no term C_v of $Comp(W, \mathcal{Y} - \{Y_1\})$ has a bit vector v with all zeroes. So, C_v is never mapped to the term $A_{10\ldots0}$, which is the term used in mapping the only term of $Comp(W, \{Y_1\})$.

From the above argument, we deduce that $Comp(W, \mathcal{Y} - \{Y_1\})$ and $Comp(W, \{Y_1\})$ in $\overrightarrow{\mathcal{E}'}$ do not incur more work than $Comp(W, \mathcal{Y})$ in $\overrightarrow{\mathcal{E}}$. Hence, each application of "separator" leads to a view strategy that is at least as efficient as the one that is transformed. Starting from a non-1-way view strategy for W, successive applications of "separator" lead to a 1-way view strategy for W that is at least as efficient as the original view strategy. Thus, the best 1-way view strategy for W is optimal over all the view strategies for W. ∎

Theorem A.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 (Theorem 4.2).*

Proof: Consider a view W defined over views $\mathcal{V} = \{V_1, \ldots, V_n\}$. Let $\overrightarrow{\mathcal{E}}$ be a view strategy for W that incurs the least amount of work. We show that $\overrightarrow{\mathcal{E}}$ must be consistent with some view ordering that orders the views based on increasing $|V_j'| - |V_j|$ values. The proof is by contradiction. That is, we assume that $\overrightarrow{\mathcal{E}}$ is not consistent with any such view ordering and show that this contradicts the fact that $\overrightarrow{\mathcal{E}}$ incurs the least amount of work.

If $\overrightarrow{\mathcal{E}}$ is a 1-way view strategy that is not consistent with a view ordering based on increasing $|V_j'| - |V_j|$ values, it must be of the form:

$$\langle \overrightarrow{\mathcal{E}_{prec}}, Comp(W, \{Y_j\}), Inst(Y_j), Comp(W, \{Y_i\}), Inst(Y_i), \overrightarrow{\mathcal{E}_{succ}} \rangle,$$

where $|Y_i'| - |Y_i| < |Y_j'| - |Y_j|$, for some Y_i and Y_j in $\mathcal{V}$.

We show that a different 1-way view strategy $\overrightarrow{\mathcal{E}'}$:

$$\langle \overrightarrow{\mathcal{E}_{prec}}, Comp(W, \{Y_i\}), Inst(Y_i), Comp(W, \{Y_j\}), Inst(Y_j), \overrightarrow{\mathcal{E}_{succ}} \rangle$$

incurs less work than $\overrightarrow{\mathcal{E}}$, thus contradicting the assumption that $\overrightarrow{\mathcal{E}}$ incurs the least amount of work.

All the expressions in $\overrightarrow{\mathcal{E}_{prec}}$ and $\overrightarrow{\mathcal{E}_{succ}}$ incur the same amount of work in both view strategies $\overrightarrow{\mathcal{E}}$ and $\overrightarrow{\mathcal{E}'}$, since the same expressions are used and the expressions are evaluated on the same database state. Also, the install expressions $Inst(Y_i)$ and $Inst(Y_j)$ incur the same amount of work in both strategies since the same changes are installed. Hence, we must show that the two compute expressions $Comp(W, \{Y_i\})$ and $Comp(W, \{Y_j\})$ incur less work in $\overrightarrow{\mathcal{E}'}$ than in $\overrightarrow{\mathcal{E}}$.

Without loss of generality, let $\mathcal{V} = \{X_1, \ldots, X_l\} \cup \{Y_i, Y_j\} \cup \{Z_1, \ldots Z_m\} = \mathcal{X} \cup \{Y_i, Y_j\} \cup \mathcal{Z}$, such that the changes of the views in $\mathcal{X}$ are propagated and installed in $\overrightarrow{\mathcal{E}_{prec}}$, while the changes of the views in $\mathcal{Z}$ are propagated and installed in $\overrightarrow{\mathcal{E}_{succ}}$.

The work incurred by $Comp(W, \{Y_j\})$ and $Comp(W, \{Y_i\})$ in view strategy $\overrightarrow{\mathcal{E}}$ is

$$c \cdot (\sum_{k=1..l} |X_k'| + |\delta Y_j| + |Y_i| + \sum_{k=1..m} |Z_k|) \; + \; c \cdot (\sum_{k=1..l} |X_k'| + |Y_j'| + |\delta Y_i| + \sum_{k=1..m} |Z_k|).$$

The work incurred by $Comp(W, \{Y_i\})$ and $Comp(W, \{Y_j\})$ in view strategy $\overrightarrow{\mathcal{E}'}$ is

$$c \cdot (\sum_{k=1..l} |X_k'| + |\delta Y_i| + |Y_j| + \sum_{k=1..m} |Z_k|) \; + \; c \cdot (\sum_{k=1..l} |X_k'| + |Y_i'| + |\delta Y_j| + \sum_{k=1..m} |Z_k|).$$

Note that the only difference between the above two work estimates is that the former uses $|Y_i|$ and $|Y_j'|$ while the latter uses $|Y_j|$ and $|Y_i'|$. Since $|Y_i'| - |Y_i| < |Y_j'| - |Y_j|$, we deduce that the two $Comp$ expressions incur less work in $\overrightarrow{\mathcal{E}'}$. Hence, the total work incurred by $\overrightarrow{\mathcal{E}'}$ is less than that of $\overrightarrow{\mathcal{E}}$. This contradicts our supposition that $\overrightarrow{\mathcal{E}}$ is a view strategy for W with the least amount of work.

Thus, the best 1-way view strategy is the one that is consistent with a view ordering that arranges views in increasing order of size changes. ∎

Theorem A.3 *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 (Theorem 5.1).*

Proof: We start by observing that all VDAG strategies for G incur the same amount of work for their $Inst$ expressions as they have the same set of changes to install. Two different VDAG strategies may differ in their amounts of total work by incurring differing amounts of work for their $Comp$ expressions.

Let $\overrightarrow{\mathcal{E}}$ be a VDAG strategy for G. Consider the partitioning of the set of $Comp$ expressions in $\overrightarrow{\mathcal{E}}$ based on the derived views whose updates the $Comp$ expressions are computing. We have as many partitions as there are derived views in G. Let P be a partition that contains the $Comp$ expressions of a derived view V, and let $\overrightarrow{\mathcal{S}}$ be the view strategy $\overrightarrow{\mathcal{E}}$ uses for V. First, we observe that P has the same set of $Comp$ expressions as $\overrightarrow{\mathcal{S}}$. Second, we note that $\overrightarrow{\mathcal{S}}$ is a subsequence of $\overrightarrow{\mathcal{E}}$. Consequently, every $Comp$ expression of P is evaluated in the same database state in $\overrightarrow{\mathcal{E}}$ and in $\overrightarrow{\mathcal{S}}$. Therefore, the amount of work incurred by the $Comp$ expressions of P when executed as part of $\overrightarrow{\mathcal{E}}$ is equal to the work incurred by these $Comp$ expressions in $\overrightarrow{\mathcal{S}}$. Thus, we conclude that the total work for the set of all $Comp$ expressions of $\overrightarrow{\mathcal{E}}$ is the sum of the work incurred by the $Comp$ expressions of the view strategies for derived views used by $\overrightarrow{\mathcal{E}}$. Note that view strategies for the base views of G have no $Comp$ expressions.

Let $\overrightarrow{\mathcal{E}_o}$ be a VDAG strategy for G that uses optimal view strategies for all the views of G, and let $\overrightarrow{\mathcal{E}_x}$ be another VDAG strategy for G. That is, the amount of work incurred by the set of $Comp$ expressions in a view strategy used by $\overrightarrow{\mathcal{E}_o}$ is at most equal to the amount of work incurred by the set of $Comp$ expressions in the corresponding view strategy used by $\overrightarrow{\mathcal{E}_x}$. Now, it follows from the earlier argument that the amount of work incurred by the $Comp$ expressions in $\overrightarrow{\mathcal{E}_o}$ is at most equal to that incurred by the $Comp$ expressions of $\overrightarrow{\mathcal{E}_x}$. Since $\overrightarrow{\mathcal{E}_o}$ and $\overrightarrow{\mathcal{E}_x}$ incur the same amount of work for their $Inst$ expressions, we conclude that the total work incurred by $\overrightarrow{\mathcal{E}_o}$ is at most as much as that incurred by $\overrightarrow{\mathcal{E}_x}$.

Thus, a VDAG strategy for G that uses optimal view strategies incurs the least amount of total work. ∎

Theorem A.4 *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. (Theorem 5.2).*

Proof: We now prove that a 1-way VDAG strategy for G that is consistent with a desired view ordering uses optimal view strategies to update all the views of G. Based on Theorem A.3, this VDAG strategy is optimal for G.

Let us consider a derived view V_i defined over views $\mathcal{V}_i$. Based on Theorem A.2, an optimal view strategy $\overrightarrow{\mathcal{E}_i}$ for V_i is the 1-way view strategy consistent with the view ordering $\overrightarrow{\mathcal{V}_i}$ that orders all the views in $\mathcal{V}_i$ in increasing $|V'| - |V|$ values.

On the other hand, a 1-way VDAG strategy consistent with the desired view ordering updates V_i using a 1-way view strategy $\overrightarrow{\mathcal{E}_i^{\prime}}$ consistent with the desired view ordering $\overrightarrow{\mathcal{V}}$ the orders *all* of the VDAG views in increasing $|V'| - |V|$ values.

Since both $\overrightarrow{\mathcal{E}_i}$ and $\overrightarrow{\mathcal{E}_i^{\prime}}$ are 1-way view strategies for V_i, they use the same $Comp$ and $Inst$ expressions. Furthermore, we now show that the order of the $Comp$ and $Inst$ expressions are the same. Let us suppose $Comp(V_i, \{V_j\}) < Comp(V_i, \{V_k\})$ in $\overrightarrow{\mathcal{E}_i}$, but $Comp(V_i, \{V_k\}) < Comp(V_i, \{V_j\})$ in $\overrightarrow{\mathcal{E}_i^{\prime}}$. This implies that $V_j < V_k$ in $\overrightarrow{\mathcal{V}_i}$ but $V_k < V_j$ in $\overrightarrow{\mathcal{V}}$. This is not possible since both view orderings are based on increasing $|V'| - |V|$ values. Since the $Comp$ expressions are in the same order in both view strategies, all the expressions including the $Inst$ expressions, must be in the same order based on **C3** and **C4**. Hence, $\overrightarrow{\mathcal{E}_i^{\prime}}$ incurs the same amount of work as $\overrightarrow{\mathcal{E}_i}$ which is an optimal view strategy.

Since this argument holds for any derived view V_i, we have proven that a 1-way VDAG strategy for G that is consistent with a desired view ordering uses optimal view strategies to update all the views of G. ∎

Theorem A.5 *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 (Theorem 5.3).*

Proof: We prove the theorem by first presenting and proving the following lemma.

Lemma A.1 *Given an acyclic $EG(G, \overrightarrow{\mathcal{V}})$ for a given VDAG G and a view ordering $\overrightarrow{\mathcal{V}}$, a 1-way VDAG strategy consistent with $\overrightarrow{\mathcal{V}}$ is obtained by topologically sorting the expression graph.* □

To prove the lemma, we first show that the 1-way VDAG strategy satisfies all correctness conditions. **C1** and **C2** are satisfied because the expression graph includes a node for each expression used in a VDAG strategy. Furthermore, a topological sort of the expression graph includes all of the nodes in the graph. **C6** is also satisfied since there is only one node for each expression, and, hence, the topological sort does not duplicate any expression.

Condition **C3**, $\forall V_i \in G(\mathcal{V}) : Comp(V, \{...V_i...\} < Inst(V_i)$, holds because an edge $Inst(V_i) \rightarrow Comp(V, \{...V_i...\})$ is in the expression graph for each derived view V_i. Hence, a topological sort of the expression graph puts $Inst(V_i)$ after $Comp(V, \{...V_i...\})$. Similarly, for **C4**, **C5** and **C8**, the expression graph has edges that ensure that the topological sort will order the expressions appropriately.

Since we just argued that the view strategy employed for each view satisfies conditions **C1** to **C6**, it follows that **C7** holds.

We now prove that the 1-way VDAG strategy $\overrightarrow{\mathcal{E}}$ produced is consistent with $\overrightarrow{\mathcal{V}}$. Let us suppose $\overrightarrow{\mathcal{E}}$ is not consistent with $\overrightarrow{\mathcal{V}}$. If so, there must be a view V_k such that $\overrightarrow{\mathcal{E}}$ employs view strategy $\overrightarrow{\mathcal{E}_k}$ to update V_k. Furthermore, $Comp(V_k, \{V_j\}) < Comp(V_k, \{V_i\})$ in $\overrightarrow{\mathcal{E}_k}$, while $V_i < V_j$ in $\overrightarrow{\mathcal{V}}$. However, the expression graph has an edge $Comp(V_k, \{V_j\}) \rightarrow Comp(V_k, \{V_i\})$ that ensures that the topological sort puts $Comp(V_k, \{V_i\})$ ahead of $Comp(V_k, \{V_j\})$. Since $\overrightarrow{\mathcal{E}_k}$ is a subsequence of $\overrightarrow{\mathcal{E}}$, it must be that $Comp(V_k, \{V_i\})$ is ahead of $Comp(V_k, \{V_j\})$ in $\overrightarrow{\mathcal{E}_k}$. This proves that $\overrightarrow{\mathcal{E}}$ is consistent with $\overrightarrow{\mathcal{V}}$.

Finally, since the expression graph only includes *Comp* expressions of the form $Comp(V, \mathcal{V})$, where $|\mathcal{V}| = 1$, it must be that the VDAG strategy produced is a 1-way VDAG strategy that is consistent with $\overrightarrow{\mathcal{V}}$.

With Lemma A.1, we can now easily prove the theorem. Given a $\overrightarrow{\mathcal{V}}$-acyclic VDAG, a topological sort of the expression graph produces a 1-way VDAG strategy consistent with $\overrightarrow{\mathcal{V}}$ according to Lemma A.1. It follows from Theorem 5.2 that the 1-way VDAG strategy produced is optimal. ∎

Lemma A.2 *For a tree VDAG, every view ordering results in an acyclic expression graph (Lemma 5.1).* □

Proof: We begin the proof by providing some notation that will also be used in subsequent proofs (for Lemma A.3 and Theorem A.6). We label the edges of an expression graph based on the "constraint" that requires the edge. For instance, an edge of the form $Comp(V_k, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$ is labeled $C8$ since condition **C8** requires it. Similarly, an edge of the form $Inst(V_i) \rightarrow Comp(V, \{V_i\})$ is labeled $C3$; an edge of the form $Inst(V) \rightarrow Comp(V, \{V_i\})$ is labeled $C5$; and an edge of the form $Comp(V, \{V_j\}) \rightarrow Comp(V, \{V_i\})$ is labeled $\overrightarrow{\mathcal{V}}$. Assuming there is an edge of the form $Comp(V, \{V_j\}) \rightarrow Comp(V, \{V_i\})$, there must be an edge $Comp(V, \{V_j\}) \rightarrow Inst(V_i)$ as required by **C4**. Edges of this form are labeled $C4$.

We denote paths based on these edge labels. For instance, path $Comp(V_k, \{V_j\}) \rightarrow Comp(V_j, \{V_i\}) \rightarrow Comp(V_i, \{V_h\})$ is a $C8C8$ path. $C8^+$ denote paths composed of at least one $C8$ edge followed by zero or more $C8$ edges. $C8^*$ denotes either an empty path or a $C8^+$ path. Path $Comp(V, \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$ is a $C4C3$ path. $(C4C3)^+$ denote paths composed of at least a $C4$ edge followed by a $C3$ edge, and possibly followed by a series of $C4$ and $C3$ edges alternating. $(C4C3)^*$ denotes either an empty path or a $(C4C3)^+$ path.

We distinguish between two types of $C4C3$ paths. A path of the form $Comp(V, \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$ is a *local* $C4C3$ path. This is because both $Comp(V, \{V_j\})$ and $Comp(V, \{V_i\})$ belong to the same view strategy. On the other hand, a path of the form $Comp(V, \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V', \{V_i\})$ is a *non-local* $C4C3$ path assuming $V \neq V'$.

We simplify the expression graph by omitting edges labeled $C5$ and $\overrightarrow{\mathcal{V}}$ because for any cycle that uses these edges, some other cycle can be constructed using only $C3$, $C4$ and $C8$ edges. More specifically, for any cycle that uses a $\overrightarrow{\mathcal{V}}$ edge $Comp(V, \{V_j\}) \rightarrow Comp(V, \{V_i\})$, a cycle can be constructed by replacing the $\overrightarrow{\mathcal{V}}$ edge with the $C4C3$ path $Comp(V, \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$. This path exists because the edge $Inst(V_i) \rightarrow Comp(V, \{V_i\})$ is required by **C3**, and the edge $Comp(V, \{V_j\}) \rightarrow Inst(V_i)$ is required by **C4** due to the presence of the $\overrightarrow{\mathcal{V}}$ edge $Comp(V, \{V_j\}) \rightarrow Comp(V, \{V_i\})$.

For any cycle that uses the $C5$ edge $Inst(V_j) \rightarrow Comp(V_j, \{V_i\})$, a cycle can be constructed by replacing the edge with the $C3C8$ path $Inst(V_j) \rightarrow Comp(V, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$. The existence of this path is guaranteed because there will never be a cycle that uses the edge $Inst(V_j) \rightarrow Comp(V_j, \{V_i\})$ where there is no view V defined on V_j. This is because there must be an edge $Comp(V, \{...\}) \rightarrow Inst(V_j)$ that completes the cycle since there are no edges between *Inst* expressions in the expression graph. Since the edge $Comp(V, \{...\}) \rightarrow Inst(V_j)$ must be a $C4$ edge, V must be defined on V_j. Since V is defined on V_j, there must be a $C3$ edge $Inst(V_j) \rightarrow Comp(V, \{V_j\})$. Finally, because of the existence of the $C5$ edge $Inst(V_j) \rightarrow Comp(V_j, \{V_i\})$ in the first place, we can deduce that there is an expression $Comp(V_j, \{V_i\})$, and therefore a $C8$ edge $Comp(V, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$.

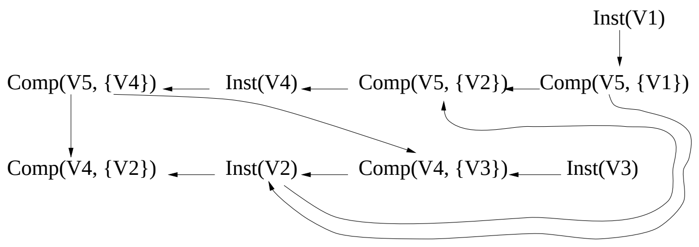

Figure 16: Simplified Expression Graph

As an example, the simplified version of the expression graph for the VDAG shown in Figure 10 and the view ordering $\overrightarrow{\mathcal{V}} = \langle V_4, V_2, V_1, V_3, V_5 \rangle$ is shown in Figure 16. Note that we have removed any expressions (*i.e.*, $Inst(V_5)$) that have no outgoing edges. Note also that there is only one cycle in the simplified expression graph which uses the path $C8C4C3C4C3$ (starting with the $C8$ edge $Comp(V_5, \{V_4\}) \rightarrow Comp(V_4, \{V_3\})$).

Using this simplified expression graph, we now derive a general form of cycles in the expression graph. First we make the following observations.

- There are no cycles using $C8$ edges only. This is because a $C8$ edge $Comp(V_k, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$ corresponds to the VDAG edges $V_k \rightarrow V_j$, and $V_j \rightarrow V_i$. Hence, any cycle using only $C8$ edges implies a cycle in the VDAG – a contradiction.
- Clearly there are no cycles using $C3$ edges only, nor any cycles using $C4$ edges only. This is because each $C3$ edge starts with an *Inst* expression and ends with a *Comp* expression. Therefore, a $C3C3$ path is not even possible. Similarly, a $C4C4$ path is not possible. By the same argument, there can be no cycles using $C8$ and $C3$ edges only, nor can there be any cycles using $C8$ and $C4$ edges only.
- There are no cycles using $C3$ and $C4$ edges only as explained below.

To explain the last observation, we introduce the function $Pos(E)$ applied to an expression in the graph. $Pos(Inst(V_i))$ returns the position of V_i in the view ordering $\overrightarrow{\mathcal{V}}$ that was used to construct the expression graph. $Pos(Comp(V, \{V_i\}))$ also returns the position of V_i in the view ordering $\overrightarrow{\mathcal{V}}$. Given any edge $A = E_j \rightarrow E_i$, the *starting position* of the edge is $Pos(E_j)$, and the *ending position* of the edge is $Pos(E_i)$.

Given any cycle $A_1 A_2 \ldots A_n$, the starting and ending positions of the first edge A_1 and the last edge A_n must be the same, since edge A_1 must emanate from the expression that edge A_n is going.

If a cycle is composed of only $C4$ and $C3$ edges, it must be of the form $(C4C3)^+$. (Alternatively, the cycle could be denoted $(C3C4)^+$.) For a $C4$ edge, the starting position must be greater than the ending position. This is because a $C4$ edge $Comp(V, \{V_j\}) \rightarrow Inst(V_i)$ is required since $V_i < V_j$ in the view ordering which implies that $Pos(Comp(V, \{V_j\})) > Pos(Inst(V_i))$. On the other hand, for a $C3$ edge $Inst(V_i) \rightarrow Comp(V, \{V_i\})$, the starting and ending position is the same. Hence, for any path of the form $(C4C3)^+$, the starting position of the first edge must be greater than the ending position of the last edge. Hence, it is impossible to construct a cycle of the form $(C4C3)^+$.

Thus, cycles must be composed of $C3$, $C4$ and $C8$ edges. An example of such a cycle is the $C8C4C3C4C3$ cycle in Figure 16 starting with the $C8$ edge $Comp(V_5, \{V_4\}) \rightarrow Comp(V_4, \{V_3\})$. In general, cycles are of the form

$$C8^+(C4C3)^+(C8^+(C4C3)^+)^*.$$

Since cycles must have $C8$ edges, we can assume without loss of generality that they start at some $C8$ edge. Since cycles must have some $C4$ and $C3$ edges, a $C4$ edge must follow the initial path of $C8$ edges. A $C3$ edge cannot follow since a $C3$ edge emanates from an *Inst* expression. Since a $C4$ edge ends in an *Inst* expression, a $C3$ edge must follow a $C4$ edge since a $C3$ edge is the only type of edge that emanates from an *Inst* expression. The initial path $C8^+(C4C3)^+$ can be followed by zero or more paths of the same form. (Note that a cycle $C8^+(C4C3)^+C8^+$ can be denoted as a $C8^+(C4C3)^+$ cycle by changing the starting edge of the cycle.)

Crux of the proof: With this notation in hand, and with this general description of a cycle, we can now prove that for tree VDAGs, there are no cycles.

Given a cycle $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$, there must be at least one non-local $C4C3$ path. Let us assume otherwise. This implies that all $C4C3$ paths in the cycle $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$ are local. However, given any path of the form $C8^+(C4C3)^+$, where all the $C4C3$ paths are local, the path can be shortened into a path of the form $C8^+$! For instance, a $C8C4C3C4C3$ path

$$Comp(V, \{V_k\}) \rightarrow Comp(V_k, \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V_k, \{V_i\}) \rightarrow Inst(V_h) \rightarrow Comp(V_k, \{V_h\}),$$

can be shortened to $Comp(V, \{V_k\}) \rightarrow Comp(V_k, \{V_h\})$, since the existence of this edge is guaranteed to ensure that the condition **C8** is met. Thus, if there is a cycle $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$ that only uses local $C4C3$ paths, then there must be a cycle using $C8$ edges only which we have observed to be impossible.

The existence of a non-local $C4C3$ path implies that the VDAG is not a tree. To see this, a non-local $C4C3$ path is of the form $Comp(V, \{V_i\}) \rightarrow Inst(V_i) \rightarrow Comp(V', \{V_i\})$ where $V \neq V'$. This implies that there are at least two views defined on V_i. This further implies that there are at least two paths that end in V_i in the VDAG, which is not possible in a tree VDAG. ∎

Lemma A.3 *For a uniform VDAG, every view ordering results in an acyclic expression graph (Lemma 5.2).* □

Proof: In the proof of Lemma A.2, we showed that cycles are of the form $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$ in general.

Similar to the *Pos* function defined in the proof of Lemma A.2, we first define the $Level(E)$ function applied to an expression E. $Level(Inst(V_i))$ returns $Level(V_i)$ of the view V_i in the VDAG that was used to construct the expression graph. Similarly, $Level(Comp(V, \{V_i\}))$ returns $Level(V_i)$. Given an edge $A = E_j \rightarrow E_i$, we say that *starting level* of A is $Level(E_j)$ and the *ending level* of A is $Level(E_i)$.

In any cycle $A_1A_2 \ldots A_n$, it is clear that the starting level of A_1 is the same as the ending level of A_n. This is because the expression from which A_1 emanates is the same as the expression that A_n is going to.

We now make the following observations. For any path of the form $C8^+$, the starting level of the first edge is greater than the ending level of the last edge. This is because for any $C8$ edge $Comp(V_k, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$, the starting level of the edge must be greater than the ending level because V_j is defined on V_i.

The next two observations only hold for expressions constructed from uniform VDAGs. For any $(C4C3)^+$ path composed of only local $C4C3$ paths, the starting level of the first edge is the same as the ending level of the last edge. This is because for a $C4C3$ path $Comp(V, \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$ implies that V is defined on both V_j and V_i. Since for a uniform VDAG, V is defined only on views with the same *Level* value, it must be that $Level(Comp(V, \{V_j\})) = Level(Comp(V, \{V_i\}))$ because $Level(V_j) = Level(V_i)$.

For any $(C4C3)^+$ path composed of only non-local $C4C3$ paths, the starting level of the first edge is the same as the ending level of the last edge. This is because for a $C4C3$ path $Comp(V', \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$ implies that V' is defined on both V_i and V_j. To see this, edge $Comp(V', \{V_j\}) \rightarrow Inst(V_i)$ implies the existence of the expression $Comp(V', \{V_i\})$, which in turn implies that V' is defined on V_i. Clearly, $Comp(V', \{V_j\})$ implies that V' is defined on V_j. For a uniform VDAG, it must be that $Level(V_i) = Level(V_j)$ because V' is defined on views with the same *Level* value. Hence, $Level(Comp(V', \{V_j\})) = Level(Comp(V, \{V_i\}))$.

Let us assume that there is a cycle of the form $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$. However, for an expression graph constructed from a uniform VDAG, the first edge in the cycle $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$ must have a starting level that is greater than the ending level of the last edge of the cycle. Since for any cycle these two levels must be the same, we have arrived at a contradiction, proving the theorem. ∎

Theorem A.6 *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 (Theorem 5.5).*

Proof: Let us assume that there is a cycle of the form $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$. For any cycle, the starting level of the first edge is the same as the ending level of the last edge as shown in the proof of Lemma A.3.

We also observed in that proof that for any path of the form $C8^+$, the starting level of the first edge is greater than the ending level of the last edge.

We now make the following observations that hold for an EG constructed using $\overrightarrow{\mathcal{V}'} = ModifyOrdering(G, \overrightarrow{\mathcal{V}})$. For any $(C4C3)^+$ path composed of only local $C4C3$ paths, the starting level of the first edge is greater than or equal to the ending level of the last edge. This is because for a $C4C3$ path $Comp(V, \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$

```
Algorithm A.1 ConstructEG
Input: VDAG $G = \langle \mathcal{V}, \mathcal{A} \rangle$, view ordering $\overrightarrow{\mathcal{V}}$
Output: EG of a 1-way VDAG strategy consistent with $\overrightarrow{\mathcal{V}}$
    Initialize EG with no nodes and no edges.
    1. For each node $V_i \in G(\mathcal{V})$, add $Inst(V_i)$ as an EG node.
    2. For each edge $V_j \rightarrow V_i \in G(\mathcal{V})$, add $Comp(V_j, \{V_i\})$ as an EG node.
    3. For nodes $Comp(V, \{V_i\}), Comp(V, \{V_j\})$ in EG
      4. If $V_i < V_j$ in $\overrightarrow{\mathcal{V}}$ Then
        5. Add $Comp(V, \{V_j\}) \rightarrow Comp(V, \{V_i\})$ as an EG edge labeled $\overrightarrow{\mathcal{V}}$.
    6. For each node $V_i \in G(\mathcal{V})$, for each edge $V \rightarrow V_i \in G(\mathcal{A})$
      7. Add $Inst(V_i) \rightarrow Comp(V, \{V_i\})$ as an EG edge (for C3).
    8. For each edge $Comp(V, \{V_j\}) \rightarrow Comp(V, \{V_i\})$ in EG
      9. Add $Comp(V, \{V_j\}) \rightarrow Inst(V_i)$ as an EG edge (for C4).
    10. For each node $V_i \in G(\mathcal{V})$, for each edge $V \rightarrow V_i \in G(\mathcal{A})$
      11. Add $Inst(V) \rightarrow Comp(V, \{V_i\})$ as an EG edge (for C5).
    12. For each edge $V_k \rightarrow V_j \in G(\mathcal{E})$, for each edge $V_j \rightarrow V_i \in G(\mathcal{E})$
      13. Add $Comp(V_k, \{V_j\}) \rightarrow Comp(V_j, \{V_i\})$ as an EG edge (for C8).
    14. Return EG ◇
```

Figure 17: *ConstructEG* Algorithm

implies that $V_i < V_j$ in $\overrightarrow{\mathcal{V}'}$. Hence, it must that $Level(V_i) \leq Level(V_j)$, and therefore $Level(Comp(V, \{V_j\})) \geq Level(Comp(V, \{V_i\}))$.

Similarly, for any $(C4C3)^+$ path composed of only non-local $C4C3$ paths, the starting level of the first edge is greater than or equal to the ending level of the last edge. This is because for a $C4C3$ path $Comp(V', \{V_j\}) \rightarrow Inst(V_i) \rightarrow Comp(V, \{V_i\})$ implies that $V_i < V_j$ in $\overrightarrow{\mathcal{V}'}$. To see this, edge $Comp(V', \{V_j\}) \rightarrow Inst(V_i)$ implies the existence of the expression $Comp(V', \{V_i\})$, and the edge $Inst(V_i) \rightarrow Comp(V', \{V_i\})$. This implies that $V_i < V_j$ in the view ordering. Hence, $Level(V_j) \geq Level(V_i)$, and $Level(Comp(V', \{V_j\})) \geq Level(Comp(V, \{V_i\}))$.

From these observations, the starting level the first edge of any cycle $C8^+(C4C3)^+(C8^+(C4C3)^+)^*$ must be greater than the ending level of the last edge of the cycle. We have arrived at a contradiction, proving the theorem. ■

B *ConstructEG*

Figure 17 gives the detailed listing of the *ConstructEG* routine used by *MinWork* to construct an expression graph. Recall also that *Prune* uses *ConstructSEG* to construct the strong expression graph. *ConstructSEG* is very similar to *ConstructEG*. In fact, only Line 5 would change wherein an edge $Comp(V, \{V_j\}) \rightarrow Comp(V', \{V_i\})$ is added for each combination of VDAG views V and V'. That is, it is not necessary that $V = V'$.