View Derivation Graph with Edge Fitting for Data Warehousing in Disconnected Environments

I. Stanoi D. Agrawal A. El Abbadi
Department of Computer Science
University of California Santa Barbara, CA 93106
{ioana,agrawal,amr@cs.ucsb.edu}

Abstract

Data warehouse views cache summarized information from queries over one or multiple distributed base sources. In this paper, we address the problem of data warehouse maintenance in an environment where base source data can be stored on mobile devices. The requirements of a partially disconnected world lead to significant tradeoffs between the view update cost on one hand, and the level of freshness of the view on the other hand. Current decentralized maintenance methods that can be applied to a disconnected environment involve either replicating of source data at the view, or computing view updates through queries to all base sources. Replication may involve excessive storage overhead, while view update may be blocked due to the disconnection of any of the base sources. We propose a view maintenance approach that stores path information corresponding to data traditionally stored at the view, and offers several important advantages. From the maintenance point of view, our method reduces local computation at the view as well as communication with the outside sources, and lowers the execution load on the base sources. From the user's perspective, reducing the probability of executing view update queries at all base sources leads to a more up-to-date state of the data warehouse view.

1 Introduction

Data Warehouses integrate and summarize information from one or several base sources [5] in a separate storage, the materialized view. User queries are directed to the materialized view for fast response time, while updates are sent to the base sources and eventually propagated to the materialized view to avoid staleness of the data. After initialization, the computation at the materialized view executes two different tasks. First, the maintenance module refreshes the warehouse view by either rederiving the whole view, or by incrementally computing and integrating the source updates with the current view [11, 18, 19, 8, 1]. Second, based on the data already stored at the materialized view, user queries are computed and answered. As a result, the choice of the maintenance approach directly affects the speed and the freshness of answers to user queries. Methods for view maintenance have been proposed, that either eliminate communication with the base sources by using full replication, reduce communication by creating multiple versions of the materialized view, or prefer to trade off communication cost for storage or freshness of data.

The trends in technology today lead to a partially disconnected world, where data can be generated on portable devices. We therefore believe that solutions to the data warehouse maintenance problem need to be developed in the context of a disconnectable environment. The model is not only relevant to mobile computing [2, 3, 12, 13], but also to a system where source failures lead to temporary disconnections. Still, we are not aware of any other proposal that efficiently deals with the case of disconnectable base sources, without involving an intermediate stable storage such as a proxy [9, 10].

In the case of maintaining the data warehouse through the replication of base data, the view may store data that, according to the definition of the warehouse view, does not lead to useful information. On the other hand, if the incremental updating approach is used, then the maintenance of the materialized view involves queries to all base sources to derive the effects of a single update or a batch of updates from one of these sources. If the base sources are stored in a mobile environment, or simply in an off-line system, the disconnection of a single base source may completely halt the computation of view changes as a result of new updates.

The solution we propose in this paper uses the optimistic replication of a subset of information from base sources, which was necessary in creating the tuples already in the materialized view. We store limited information about the "path" that led to the insertion or deletion of a tuple in the materialized view, i.e., the joinable attribute values in the tuples that led to a change in the materialized view. As we will show, we found that it is possible to store a limited amount of information and only in the worst case scenario external queries to all sources are still necessary. If partial information is available, then deriving an update involves queries to only a subset of the sources. However, in the best case, the data warehouse view is able to locally compute and integrate the effects of the updates at the sources. Local computation in this case is also reduced by using information about previous operations. This method is most appropriate for applications with low selectivity, where data is distributed so that there is a large concentration on certain hot spots. In these applications, the maintenance approach will usually fall in one of the last two cases, which considerably reduces communication and, consequently, reduces the probability of interrupting the update maintenance due to base source disconnections.

We start by presenting, in Section 2, the data warehouse maintenance problem in disconnected environments. In Section 3, we develop the notion of View Derivation Graph and describe a technique called *edge fitting* for incremental maintenance of data warehouses. In Section 4, we present the details of the proposed algorithm for incremental view maintenance and an analytical model for evaluating the usefulness of the approach. Section 5 concludes the paper with a discussion of our results.

2 Problem

The distributed data warehouse model we are considering in this paper consists of one or more independent views defined over multiple base sources (Figure 1). Current view maintenance strategies either replicate at the view the base relation, or they incrementally compute the and integrate the changes at the view as a function of the data at the sources. We propose the **edge fitting** method, which, as far as storage requirements falls in between the full replication and the incremental maintenance methods. Although data sources are disconnectable, we assume that, when communication is possible between two sites, it is both FIFO and reliable. We also assume that views derive data from base sources by using the multiway equijoin operation, possibly combined with aggregate functions. For simplicity, we describe the computation over the base sources simply as the join operation.

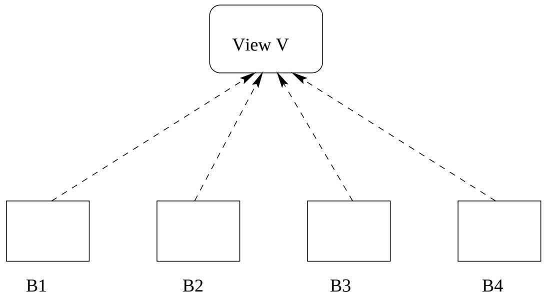

Figure 1: Model description

A data warehouse should be initialized, and then maintained up to date with respect to the underlying source data. Several view maintenance approaches have been proposed, ranging from full replication of the base source data at the view to the materialization of only the query result tuples. Different self-maintenance approaches have been proposed for data warehouses, that also log a history of the updates or auxiliary information on certain intermediate results [6, 14, 17, 4]. Since they are tailored for a connected environment, these methods imply a tradeoff between communication cost and local execution and storage cost. When **full replication** is used, the view is responsible for the integration of the new updates from base sources. An advantage of the full replication approach is that there is no need for external update queries to the base sources. However, joining data from several relations is the most time-consuming, data-intensive and frequent operation in relational query processing. Also, for applications with low join selectivity, the full replication method induces unnecessary usage of large amounts of storage. The other extreme in the range of proposed maintenance solutions is the method of **view materialization**, i.e., of storing only the result of the operation executed over base sources. In general, let a view V be defined over base sources $B_1, B_2, \cdots, B_n$ and an update ΔB_i propagate from source B_i to V. Then the maintenance task of the view is to compute the effects of ΔB_i by undergoing an *update phase*. An update at V is a function not only of the change ΔB_i, but also on the data stored at the base sources. Consequently, an *update query* is dispatched to all base sources in order to compute and return the change pertaining to V's data. Note that the update query will be of the form: $\Delta V = B_i \bowtie B_2 \bowtie \cdots \bowtie \Delta B_i \bowtie \cdots \bowtie B_n$. Each update (or batch of updates) at a base source triggers an update query, which requires communication with all base sources. If at least one source is disconnected at a time, then no update queries from V can be completed.

An environment where either the data warehouse view or the sources may be temporarily disconnected, can therefore greatly affect the freshness and consistency of the data warehouse. Note also that the communication with the base sources can become a bottleneck in the system, since base sources not only receive and integrate their own updates, but also answer update queries of the dependent data warehouse views. It is therefore essential to reduce not only the number of update queries but also the number of the base sources involved in an update phase. The challenge we are addressing in this paper is therefore twofold: first, our goal is to reduce the possibility of the view maintenance task being blocked due to source disconnections / failures. At the same time, we want to reduce the responsibility of a base source during the view update phase of the warehouse.

In this paper we propose **edge fitting**, a method that stores information related to the *path* that leads to the materialization of tuples, i.e., joinable attribute values. In the best case, when the joinable attribute values of the new update are part of the stored paths, local execution at the view is far less expensive than the full replication approach because no tuples are actually compared for the join condition. The update is simply fit with the previously computed derivation paths, and the final result corresponding to the data stored by a materialized view is changed. If only one of the attributes of the new update matches the stored paths information, then computation is needed for joining a subset of the external sources. However, if the derivation paths make up a dense graph, this subset is very small, and the probability of communicating with a disconnected source is much lower than in the case of the materialized view approach. For a low selectivity of the join operation, and for data that is distributed so that there is a high concentration on hot spots, then both local execution and communication will be greatly reduced as compared to other maintenance methods.

3 The Edge Fitting Method

In this section we will introduce the **edge fitting** method, together with an illustrative example, and present an efficient storage mechanism appropriate for the proposed method.

3.1 Structuring the View Data with Derivation Graphs

To better illustrate the **edge fitting** method and the following definitions, consider the example in Figure 2. A data warehouse view derives information from base sources B_1, B_2 and B_3, with attributes (A,B), (B,C,D) and (D,E) respectively. The table shows all tuples stored at the base relations. A view V is defined as $\prod_{AE} B_1 \bowtie B_2 \bowtie B_3$, a set of tuples (A,E), where A is the primary attribute and E is the measure attribute.

$\mathbf{B_1}$	$\mathbf{B_2}$	$\mathbf{B_3}$
AB	**BCD**	**DE**
(1,2)	(2,4,1)	(1,5)
(5,6)	(4,2,4)	(4,5)
(2,4)	(2,1,4)	(1,6)
(3,4)	(4,4,3)	(5.7)
(1,3)	(3,3,3)	(3,6)
(3,2)	(2,3,1)	(5,1)
(6,6)		

Figure 2: Base Source Information external to the Data Warehouse.

In general, let a view V derive data from base sources $B_1, \cdots, B_i, \cdots, B_j, \cdots, B_n$. Assume w.l.o.g. that the index of a base source corresponds to the order of update query of the view. Then we denote the **Joinable Pair** $J_{B_i,B_{i+1}}(v_i) = \{v_{i+1} | (tuple\,\langle v_i, \cdots, v_{i+1}\rangle \in B_i) \wedge (tuple\,\langle v_{i+1} \cdots\rangle \in B_{i+1})$ where v_i is a value of the primary attribute of B_i and v_{i+1} is a value of the joinable attribute of B_i

and $B_{i+1}\}$. If the set $J_{B_i,B_{i+1}}(v_i)$ is null, it means that relations B_i and B_{i+1} are not joinable for tuples in B_i with primary attribute is v_i. In the example (Figure 2),
$J_{B_1,B_2}(1) = \{2,3\}$ where the primary attribute is A, and the joinable attribute is B.
$J_{B_1,B_2}(2) = \{4\}$
$J_{B_1,B_2}(3) = \{4\}$
$J_{B_1,B_2}(5) = \{\}$
$J_{B_2,B_3}(2) = \{1,4\}$ where the primary attribute is B and the joinable attribute is D.
$J_{B_2,B_3}(4) = \{4,3\}$
$J_{B_2,B_3}(3) = \{3\}$

A more general definition is needed to express the joinability property of a set of sources (*joinable path*), rather than pairwise joinability.

Definition 1 *Let B_i, B_j $(i \leq j \leq n)$ be two of the n base sources, and v_i be the value of the primary attribute of B_i. A* **Joinable Path** $JP_{B_i,B_j}(v_i) = \{v_j|$ *where*
$\exists\{v_{i+1}, v_{i+2}, \cdots, v_k, \cdots, v_{j-1}, v_j\}$ *such that:*

$$\begin{cases} v_{i+1} \in J_{B_i,B_{i+1}}(v_i) \\ v_{i+2} \in J_{B_{i+1},B_{i+2}}(v_{i+1}) \\ \cdots \\ v_{k+1} \in J_{B_k,B_{k+1}}(v_k) \\ \cdots \\ v_j \in J_{B_{j-1},B_j}(v_{j-1}) \end{cases}$$

$\}$

$\square$

Revisiting the illustrating example, some of the paths derived from the data stored at sites B_1, B_2 and B_3 according to Figure 2 are the following:
$JP_{B_1,B_2}(1) = \{2,3\} = J_{B_1,B_2}(1)$
$JP_{B_1,B_3}(1) = \{1,3,4\}$
where the primary attribute is A and the returned joinable attribute value corresponds to attribute D. $JP_{B_1,B_2}(5) = \{\}$
$JP_{B_1,B_3}(5) = \{\}$ because $JP_{B_1,B_2}(5) = \{\}$

The concept of *view derivation paths* expresses the joinability across all sources. A derivation path is a graphical illustration of path joinability for tuples at the base sources that lead to the generation of tuples in the materialized view.

Definition 2 *A* **View Derivation Path** $DP(v_1)$ *for a value v_1 of the primary attribute of relation B_1 is a graph $G(V,E)$ where V is a set of vertices and E a set of edges defined as follows:*

For every joinable path $JP_{B_1,B_n}(v_1)$ such that $\exists$ $v_2, v_3, \cdots, v_n$ such that $v_n \in JP_{B_1,B_n}(v_1)$,

1. *$\exists$ a vertex $v \in V$ for the primary attribute v_1, for each value v_i, $2 \leq i \leq n$ of a joinable attribute, as well as for each value v_{n+1} of the measure attribute corresponding to tuples in B_n with primary attribute value v_n.*

2. *$\exists$ an edge e, $\langle v_i, v_{i+1}\rangle$, for each joinable pair used in the derivation of the joinable path $JP_{B_1,B_n}(v_1)$ as well as edges $\langle v_n, v_{n+1}\rangle$. where $v_n \in JP_{B_1,B_n}(v_1)$ and v_{n+1} is a value of the measure attribute of B_n corresponding to a tuple with value v_n for the primary attribute of B_n.*

$\square$

Figure 3 represents the different *Derivation Paths* corresponding to $JP_{B_1,B_3}(v_1) =$
$\{1,3,4\}$, for primary attribute A and joinable attribute D.

Finally, we put all the previous definitions together and present the *derivation graph*, which traces all derivation paths from the primary to the measure attribute. Assuming that common edges are indistinguishable, which is a valid assumption for a large set of applications, then multiple common paths are collapsed into single paths. The information stored at the view is used for data warehouse queries, and therefore the view tuples should be readily available. In order to avoid an exhaustive search on the derivation paths, we add to the view derivation graph a set of direct edges between the primary and the measure attributes (in the example attributes A and E respectively).

Definition 3 View Derivation Graph
$DG = \bigcup_{v_1 \in PrimaryAttribute(B_1)} DP(v_1)$
$\bigcup_{v_1 \in PrimaryAttribute(B_1)} \langle v_1, v_{n+1}\rangle$ *where v_1 are attribute values for primary attribute of B_1 and v_{n+1} are values of the measure attribute. An edge $\langle v_1, v_{n+1}\rangle$ exists if there is a derivation path from v_1 to v_{n+1}.*

$\square$

The derivation graph (Figure 4) corresponding to the example in Figure 2 represents all the possible joinable paths from B_1 to B_n. Nodes are joinable attribute values of the different base relations, (B, D),

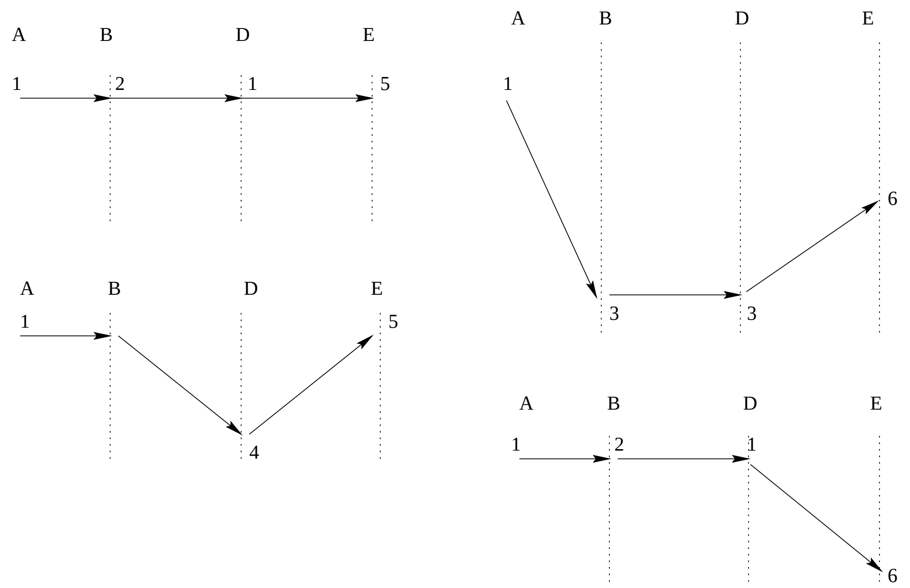

Figure 3: Derivation Paths for $JP_{B_1,B_3}(v_1)$

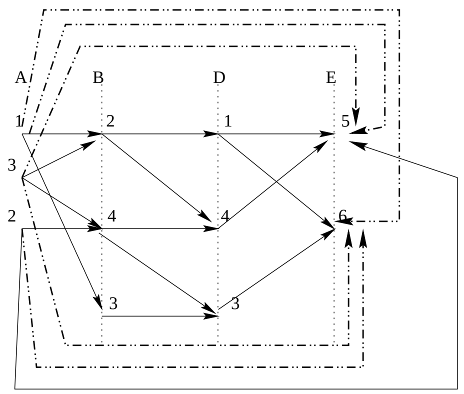

Figure 4: Derivation Graph before the new updates

together with the primary attribute of B_1 and the measure attribute of B_3, i.e., (A, E). An edge represents a joinable pair between two relations.

Next, we will show the effects of an update on a materialized view. We assume that an update is either an insert or a delete operation. Modifications to a tuple in the data sources are propagated at the view as a delete followed by an insert operation.

3.2 Integrating Insert Updates

Base sources can be updated, and as a result the derivation graphs should reflect this changes. To continue our example, assume that source B_2 incorporates, consecutively, the updates shown in Figure 5. Figure 6 represents the derivation graph after the view integrates these new updates. For clarity, we omit the direct edges ($\langle v_1, v_{n+1}\rangle$) between the primary and measure attributes.

$\mathbf{B_1}$	$\mathbf{B_2}$	$\mathbf{B_3}$
AB	**BCD**	**DE**
	ΔB_2 = insert(**3,2,1**)	
	ΔB_2 = insert(**3,2,5**)	
	ΔB_2 = insert(**6,3,6**)	

Figure 5: Base Source Information external to the Data Warehouse.

In general, let an update to a base source B_i have a value $left_{value}$ of the attribute joinable with B_{i-1} and another attribute value $right_{value}$ joinable with source B_{i+1}. For our example, if $\Delta B_i = \Delta B_2 =$ **(3,2,1)**, then $left_{value} = 3$ and $right_{value} = 1$. If $\Delta B_i = \Delta B_2 =$ **(3,2,5)**, $left_{value} = 3$ and $right_{value} = 5$, while if $\Delta B_i = \Delta B_2 =$**(6,3,6)**, then $left_{value} = 6$ and $right_{value} = 6$. An update falls in one of the following three cases:

1. $(left_{value} \in DG) \wedge (right_{value} \in DG)$. The insertion of the first update ΔB_2 = (3,2,1), generates a single edge in the graph (Figure 6). This is because the joinable attribute values of ΔB_2 have already participated in the generation of

tuples at the materialized view. In general, the insertion of an edge in this case either generates one or no new edge in the graph. Neither communication with the external sources, nor local computation of the base source join is necessary.

2. $(left_{value} \in DG) \vee (right_{value} \in DG)$. The second update, ΔB_2 = (3,2,5), illustrates the case when only one of the joinable attribute values, B, is part of the previously stored derivation graph. Since the value of the attribute D does not correspond to any value previously used in the join with tuples in B_3, computation is needed for the generation of the remaining derivation path. Note that, if the number of base sources involved in the derivation of a view is large, then there may also be the case that the portion of the path to be added to the derivation graph also merges with the previously stored graph at some attribute node. In general, we distinguish between two cases:

   (a) if $left_{value} \in$ DG, then base sources $\{B_1, B_2, \cdots, B_{i-1}\}$ need not be queried.

   (b) if $right_{value} \in$ DG, then the update query should not include sources $\{B_{i+1}, B_{i+2}, \cdots,\}$.

3. $(left_{value} \notin DG) \wedge (right_{value} \notin DG)$. Finally, this is the case where none of the joinable attributes values of the new update (ΔB_3 = (6,3,6)) are part of the derivation graph. Then the effects of the change at B_3 on the materialized view need to be calculated by joining it with the base relation to the right and then to the left of B_3. Note again that, if the derivation graph is dense, then the path to be computed may merge at some point with a path already present in the graph and only a subset of queries are necessary.

3.3 Integrating Deletes

Deleting a tuple in one of the underlying base relations is interpreted at the view as the deletion of an edge in the graph. However, an edge corresponding also to at least one other tuple in the base relation should not be deleted. Consequently, there is a need to compute and store *edge tuple counts*

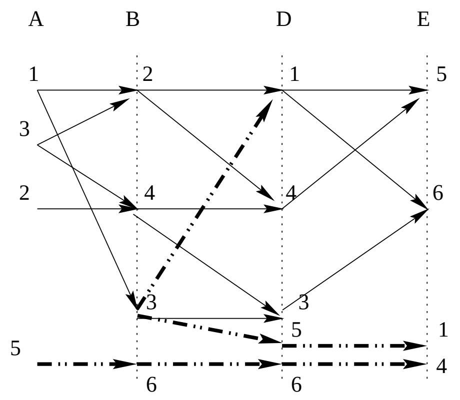

Figure 6: Update Derivation Graph after the new updates

as well as*node tuple counts* [18] at each node in the derivation graph. An *edge tuple count* for an edge $\langle v_i, v_{i+1}\rangle$ represents the number of tuples with primary attribute v_i and joinable attribute v_{i+1} in relation B_i. At a node corresponding to some attribute value v_i, the *node tuple count* reflects the number of tuples resulting from joining base relations up to B_i, and whose derivation paths go through the node v_i. Consider again the example in Figure 4. As mentioned in the previous section, the main advantage of the **edge fitting** method is that, due to the properties of the base relation data, it will frequently be the case that an update to the derivation graph modifies one or a small number of edges. Figure 7 illustrates the maintenance of the node tuple count for every node in the derivation graph. The example pertains to the paths $DP(1)$, generated by tuples with the primary attribute value $v_1 = 1$ for primary attribute A of base relation B_1. There is exactly one tuple $(A, B) = (1, 2)$ in the relation stored at B_1, but there are two tuples at B_2, $(2, 4, 1)$ and $(2, 3, 1)$, with values $(B, D) = (2, 1)$ (the edge count is 2). Also, some of the paths merge, for example the path $(A, B, D, E) = (1, 2, 4, 5)$ and (A, B, D, E) $= (1, 2, 1, 5)$. The node count corresponding to the value 5 of attribute E should therefore be a sum of the number of tuples on the two paths. The edge counters on the direct edges between the primary and the measure attribute represent the number of result tuples with the corresponding primary and measure attribute values. They inherit the count from the counter of the node related to the measure attribute value.

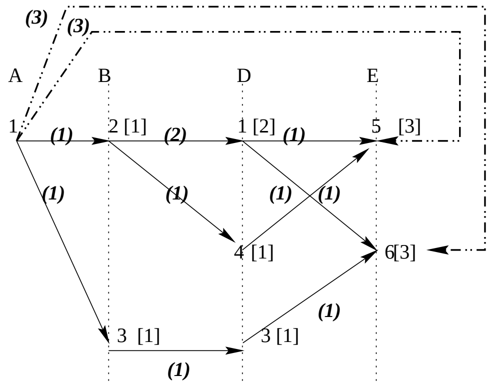

Figure 7: Tuple count associated with all the paths starting from value 1 of attribute A

Since the resulting tuple can be reached through different derivation paths, each node should store a *vector count* such that each vector entry represents the counter for a given value of the primary attribute. The number of entries in a vector is the number of different values of the primary attribute, in our example attribute A. The following rule is used to initialize and maintain the node counters of a derivation graph, considering the derivation paths from left to right:

Definition 4 Vector count maintenance. *The tuple count of a node is the sum of the vectors on incoming edges, multiplied by the corresponding edge tuple count.*

□

The propagation of a delete operation at the view leads to the deletion of edges that are not used by other derivation paths. In the best case, no edge is actually deleted, but the derivation path is followed and the counters associated with the path are modified. Counters are also modified in the case of a delete operation that leads to a partial, but not full deletion of the derivation path.

Revisiting this example, let the vector count associated with each node have the following entries: ($v_1 = 1$,$v_1 = 3$, $v_1 = 2$). The resulting counters are shown in Figure 8.

During the view maintenance, insert and delete updates in the derivation graph will obviously affect some of the tuple counters. Following the completion of the update queries triggered by the changes at source B_2, the derivation graph and the associated

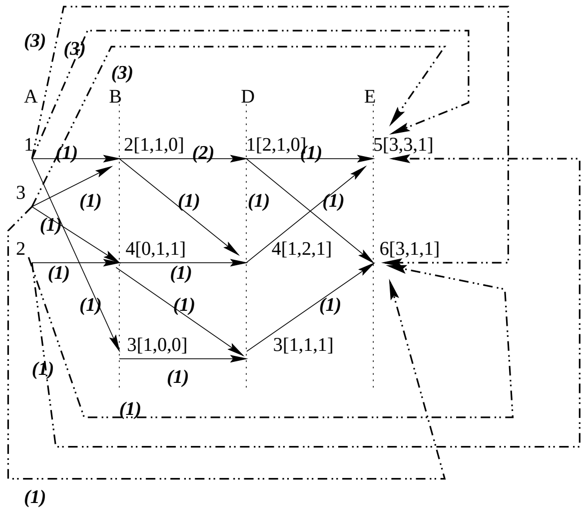

Figure 8: Vector counter associated with every node in the Update Derivation Graph before updates

node vector counters will be as illustrated in Figure 9 (for clarity, again we omit the direct edges between attributes A and E).

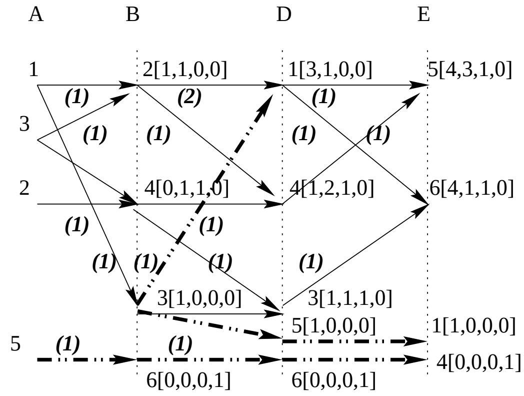

Figure 9: Vector counter associated with every node in the Update Derivation Graph

4 View Maintenance Using the View Derivation Graph and Edge Fitting

In this section we give an overview of the *edge fitting* maintenance method, and we propose an analytical tool for its evaluation.

4.1 Algorithm

A data warehouse view stores and maintains data derived from underlying base sources, and answers view queries. If the **edge fitting** maintenance method is used, then most of the information of the view derivation graph can be hidden from users, while queries are answered based solely on the information stored by the nodes corresponding to the primary and measure attribute, and the direct edges between them. For both maintenance and query computations, we enforce an index structure for each joinable attribute, as well as for the primary attribute of the first joinable relation, and the measure attribute. However, for maintenance, the storage overhead for additional information such as the vector counts can become inefficiently large. As the number of attribute values changes, the size of a node vector counter should also be dynamically changed. Although inefficient from a storage point of view, the vector counts associated with the node in the derivation graph provide information on tuple counts at all the points on a derivation path.

Consider eliminating the vector counts associated with every node in the derivation graph. Since the vector counts of the measure attribute values are needed for view queries, the corresponding information can be stored on the direct edges between the primary and the measure attributes. Going back to our example, the view derivation graph that corresponds to the derivation paths starting with $v_1 = 1$, from Figure 7 is presented in Figure 10.

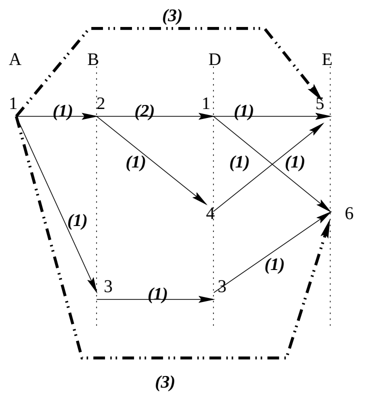

Figure 10: Modified Derivation Graph for $v_1 = 1$

By eliminating the information stored by the node vector counts, the insertion or deletion of an edge in the view derivation graph becomes more complex. Basically, if the node vector counts are not maintained explicitly, their computation is necessary during view update. For comparison, consider again the derivation graph with vector counts. The insertion or deletion of an edge $\langle left_{value}, right_{value}\rangle$ affects the number of tuples in join operations that involve the node corresponding to $right_{value}$. That is, the vector counts on the derivation paths going though the node $right_{value}$, and related to nodes following $right_{value}$ and up to v_{n+1} are modified. In the case where the insert/delete operation at the view requires the insertion/deletion of more than one edge, the same observation holds for each modified edge. In contrast, a view derivation graph without node vector counts does not offer the information on how many tuples are actually affected by the insert/delete of an edge, and this information needs explicit computation. This can be done by first traversing to the left the derivation paths that contain the node $left_{value}$, computing the total number of tuples affected by the insertion/deletion of the edge, and then traversing all the paths containing the node $right_{value}$. More details on how an insert/delete operation affect view derivation graph without node vector counts, are provided below.

[]

Notation. Let a view V be defined over base sources B_1 (with attributes $attr_1, attr_2$), B_2 (with attributes $attr_2, attr_3$), $\cdots$, B_i (with attributes $attr_i, attri+1$), $\cdots$, B_n (with attributes $attr_n, attr_{n+1}$). A node in the derivation graph corresponds to a value v_i of attribute $attr_i$, and an edge between two nodes v_i and v_{i+1} is noted as $\langle v_i, v_{i+1}\rangle$. An update from source B_i is interpreted by at the view in the form $\langle v_i, v_{i+1}\rangle$, based on the tuple inserted/deleted at B_i. The derivation graph associates an edge tuple count $EC(\langle v_i, v_{i+1}\rangle)$ with every edge $\langle v_i, v_{i+1}\rangle$. Let the set $C(v_i)$ be the counts of all tuples resulting from all joinable paths from v_1 to v_i. Note that the set $C(v_i)$ corresponds to the entries in the node count of v_i as described in Section 3.3.

Overview: Integrate Insert Updates.
1) Case I: $(v_i \in DG) \wedge (v_{i+1} \in DG)$**.** v_i and v_{i+1} are already nodes in the view derivation graph. Either the edge $\langle v_i, v_{i+1}\rangle \in DG$ and $EC(\langle v_i, v_{i+1}\rangle)$ is incremented by 1, or the edge is inserted in the graph and $EC(\langle v_i, v_{i+1}\rangle)$ is set to 1. Then use procedure $CalculateCount(v_{i+1})$ to return the set $C(v_{i+1})$ and procedure $Propagate(C(v_{i+1}))$ to propagate the results and modify the edge counts on the direct edges between all v_1 and v_n on affected paths.
2) Case II: $(v_i \in DG) \wedge (v_{i+1} \notin DG)$**.** The fact that $v_{i+1} \notin DG$ means that the updating view needs to query sources $B_{i+1}, \cdots B_n$. View V receives a set of tuples $\langle v_j, v_{j+1}\rangle$ as a result of a query to some site B_j, together with the associated tuple count. For all tuples with $v_{i+1} \in DG$, insert all previous related edges $\langle v_i, v_{i+1}\rangle$, $\cdots$ $\langle v_{j-1}, v_j\rangle$ and $\langle v_j, v_{j+1}\rangle$ in the derivation graph, including the tuple counts.

If $B_j = B_n$, the last base source, then all tuples $\langle v_j, v_{j+1}\rangle$ and the related previous tuples are inserted as edges in the DG, together with the corresponding tuple count. Use procedure $CalculateCount(v_{i+1})$ to return $C(v_{i+1})$, and procedure $Propagate(C(v_{i+1}))$ to propagate the results.

2) Case III: $(v_i \notin DG) \wedge (v_{i+1} \in DG)$**.** It follows the same steps as the previous case, but queries are directed from B_{i-1} to B_1.

2) Case IV: $(v_i \notin DG) \wedge (v_{i+1} \notin DG)$**.**
This case follows the algorithm of Case I, followed by the steps in Case II, with the exception that the results of queries are logged until they form a joinable path either between two already existing nodes in DG, between one node already in DG and primary attribute of B_1 or the measure attribute, or they form a derivation path.

Procedure Calculate(v_i).
Set $C(v_2) = EC(\langle v_1, v_2\rangle)$ for all nodes v_2 that are on the same derivation paths as v_i. Then compute $C(v_3)$, $C(v_4)$, $\cdots$, $C(v_j)$, $\cdots$, $C(v_i)$ for nodes on these derivation path by using the following rule: for all incoming edges $\langle v_{j-1}, v_j\rangle$,
$C(v_j) = \sum[C(v_{j-1}) \times EC(\langle v_{j-1}, v_j\rangle)]$.
Return the set $C(v_i)$.

Procedure Propagate($C(v_i)$.
Compute $C(v_{i+1})$, $C(v_{i+2})$, $\cdots$, $C(v_j)$, $\cdots$, $C(v_n)$ for all nodes v_{i+1}, $\cdots$, v_n on the same derivation paths as v_i. To calculate any $C(v_j)$, again, use the same rule as in $ProcedureCalculate(v_i)$:
$C(v_j) = \sum[C(v_{j-1}) \times EC(\langle v_{j-1}, v_j\rangle)]$.
When all $C(v_n)$ are computed in relation to the starting $C(v_i)$, set the edge tuple count of the direct edges between the primary attribute of B_1 and the measure attribute to the corresponding $C(v_n)$.

Overview: Integrate Delete Updates.
Subtract the tuple to be deleted from the edge tuple count of $\langle v_i, v_{i+1}\rangle$: $EC(\langle v_i, v_{i+1}\rangle) = EC(\langle v_i, v_{i+1}\rangle)$ - 1.

2) Case I: If $EC(\langle v_i, v_{i+1}\rangle) \geq 1$, then use procedure Calculate(v_{i+1}) to return $C(v_i)$, and propagate the change to all derivation paths that pass through v_{i+1} (Propagate($C(v_{i+1})$).

2) Case II: $EC(\langle v_i, v_{i+1}\rangle) = 0$ Follow all derivation paths, from v_i to v_1. On a derivation path, for any node v_j ($1 \leq j \leq i-1$), if v_j has more than one outgoing edge, then this derivation path is not followed from that point on. Otherwise, delete the edge $\langle v_{j-1}, v_j\rangle$, and continue. As a second step, starting from v_i and towards v_n, follow the derivation paths that pass through node v_i. The deletion of edges on a derivation path stops when a node has more than one incoming edge. Finally, reset the tuple counters by using first the procedure $Calculate(v_{i+1})$ and then $Propagate(C(v_{i+1}))$.

4.2 Analysis

Having presented the view maintenance approach based on view derivation graphs, we need a measure of its usefulness. We therefore develop next an analytical tool to measure, for a given application, the compatibility of the proposed maintenance method.

For data warehouse maintenance, a view queries the underlying base sources only in the case of insert updates. Let the update to be integrated at the view be $\langle left_{value}, right_{value}\rangle$. As discussed before, there are three cases: both $left_{value}$ and $right_{value}$ are already in the derivation graph (with probability $P(Fit)$), only one ($P(1/2Fit)$), or none of the nodes are present in the derivation graph (with probability $P(noFit)$). We can express $P(Fit)$), $P(1/2Fit)$ and $P(noFit)$) as functions of the probability of $left_{value}$ to be in the derivation graph, $P(left_{value}$, and the probability of $right_{value}$ to be a node in the derivation graph, $P(right_{value})$.

$$\begin{aligned}
P(Fit) &= P(left_{value}) \times P(right_{value}) \\
P(1/2Fit) &= (\neg P(left_{value}) \times P(right_{value})) \\
&\quad +(P(left_{value}) \times \neg P(right_{value})) \\
P(noFit) &= \neg P(left_{value}) \times \neg P(right_{value})
\end{aligned}$$

As an example, even if the probability of a node to be in the derivation graph is 30%, then 50% of the the time complete queries will not be needed.

We assume each base relation B_i has a certain distribution for the values of its attributes, and that there is also a given distribution of the tuples in B_i. Let the distribution of the primary attribute $attr_i$ be $D_{attr'_i}$ and the distribution of the joinable attribute (or measure attribute in the case of B_n) $attr_{i+1}$ be $D_{attr_{i+1}}$. The distribution of the tuples is described by $D_{attr'_i,attr_{i+1}}$. We use the "prime" notation to distinguish between distribution of values of the same attributes from different relations. Let the frequency of a given value v_i of attribute $attr_i$ be $F_{attr_i}(v_i)$, which is given by the distributions D_{attr_i}. Assume the domains of a joinable attribute in joinable relations B_i and B_{i+1} are the same. The frequency $F_{attr'_i,attr_{i+1}}(v_i, v_{i+1})$ of a certain tuple $\langle v_i, v_{i+1}\rangle$ results from $D_{attr'_i,attr_{i+1}}$. The probability for two relations B_i and B_{i+1} of size $|B_i|$ and $|B_{i+1}|$ to be joinable for a given primary attribute $attr_i$ and given joinable attribute $attr_{i+1}$ of B_i can then be expressed by :

$$\begin{aligned}
&(1/|B_i|)F_{attr'_i,attr_{i+1}}(v_i, v_{i+1}) \\
&\times(1/|B_i|)F_{attr_{i+1}}(v_{i+1}) \\
&\times(1/|B_{i+1}|)F_{attr'_{i+1}}(v_{i+1}) \\
&\times(1/|B_{i+1}|)F_{attr'_{i+1},attr_{i+2}}(v_{i+1}, v_{i+2}).
\end{aligned}$$

If only the value of the primary attribute $attr_i$ is given, then the probability for B_i and B_{i+1} to be joinable is:

$$\begin{aligned}\sum_{v_{i+1}\in B_i} \quad & [(1/|B_i|)F_{attr'_i,attr_{i+1}}(v_i, v_{i+1}) \\ & \times(1/|B_i|)F_{attr_{i+1}}(v_{i+1}) \\ & \times(1/|B_{i+1}|)F_{attr'_{i+1}}(v_{i+1}) \\ & \times(1/|B_{i+1}|)F_{attr'_{i+1},attr_{i+2}}(v_{i+1}, v_{i+2})].\end{aligned}$$

Note that the probability of a given attribute value to be represented by a node in the view derivation graph is the probability of that value to be part of an already existing derivation path.

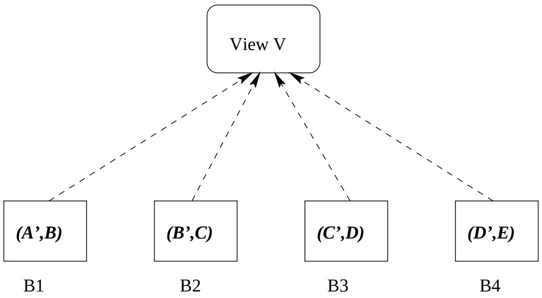

Figure 11: Example of a View V defined over Base Sources B_1,B_2,B_3 and B_4

Consider for example a view V that derives data from sources B_1 (with attributes A, B), B_2 (with attributes B, C), B_3 (with attributes C, D) and B_4 (with attributes D, E), as shown in Figure 11. Let an insert update from B_3 be propagated to the view and interpreted by the derivation graph as an edge $\langle left_{value}, right_{value}\rangle = \langle c, d\rangle$ for attributes C' and D. The probability $P(left_{value})$, of given attribute value c to be in the view derivation graph is the following:

$$\begin{aligned}\mathbf{P}(\mathbf{left_{value}}) = \quad & P(c\in DG) = \\ (1) \quad & (1/|B_3|)F_{C'}(c)\times(1/|B_2|)F_C(c) \\ & \times\sum_{b\in B_2}[(1/|B_2|)F_{B',C}(b,c) \\ & \quad \times(1/|B_2|)F_{B'}(b)\times(1/|B_1|)F_B(b)] \\ (2) \quad & \times\sum_{d\in B_3}[(1/|B_3|)F_{C',D}(c,d) \\ & \quad \times(1/|B_3|)F_D(d)\times(1/|B_4|)F_{D'}(d)]\end{aligned}$$

The first part of the right hand side of the equation is the probability of the node c to be on a joinable path from B_1 to B_3. The second part represents the probability of the node c to be on a joinable path between B_3 and B_4. The probability $P(right_{value})$ can be expressed by the equation below. Again the first part is the probability of node d to be on a joinable path extending from B_1 to B_3, and the second part of the equation represents the probability of d to be on a joinable path to B_4.

$$\begin{aligned}\mathbf{P}(\mathbf{right_{value}}) = \quad & P(d\in DG) = \\ (1) \quad & \sum_{c\in B_3}[(1/|B_3|)F_{C',D}(c,d) \\ & \quad \times(1/|B_3|)F_{C'}(c)\times(1/|B_2|)F_C(c) \\ & \quad \times\sum_{b\in B_2}[(1/|B_2|)F_{B',C}(b,c) \\ & \quad\quad \times(1/|B_2|)F_{B'}(b)\times(1/|B_1|)F_B(b)]] \\ (2) \quad & \times(1/|B_3|)F_D(d)\times(1/|B_4|)F_{D'}(d)\end{aligned}$$

In general, let a view V derive information from base sources B_1 (with attributes $attr'_1, attr_2$), B_2 (with attributes $attr'_2, attr_3$), $\cdots$ B_i (with attributes $attr'_i, attr_{i+1}$), $\cdots$, B_n (with attributes $attr'_n, attr_{n+1}$); v_i denotes a value of the attribute $attr_i$ or $attr'_i$. Let an insert update from B_i be represented by an edge $\langle left_{value}, right_{value}\rangle = \langle v_i, v_{i+1}\rangle$. The probability of the node corresponding to v_i to be in the view derivation graph is $P(left_{value})$, while the probability of the derivation graph to contain a node v_{i+1} is $P(right_{value})$:

$$\begin{aligned} & \mathbf{P}(\mathbf{left_{value}}) = P(v_i\in DG) = \\ (1) \quad & (1/|B_i|)F_{attr'_i}(v_i)\times(1/|B_{i-1}|)F_{attr_i}(v_i) \\ & \times\sum_{v_{i-1}\in B_{i-1}}[(1/|B_{i-1}|)F_{attr'_{i-1},attr_i}(v_{i-1},v_i) \\ & \times(1/|B_{i-1}|)F_{attr'_{i-1}}(v_{i-1})\times(1/|B_{i-2}|)F_{attr_{i-1}}(v_{i-1}) \\ & \times\sum_{v_{i-2}\in B_{i-2}}[(1/|B_{i-2}|)F_{attr'_{i-2},attr_{i-1}}(v_{i-2},v_{i-1}) \\ & \times(1/|B_{i-2}|)F_{attr'_{i-2}}(v_{i-2})\times(1/|B_{i-3}|)F_{attr_{i-2}}(v_{i-2}) \\ & \cdots \\ & \times\sum_{v_2\in B_2}(1/|B_2|)F_{attr'_2,attr_3}(v_2,v_3) \\ & \times(1/|B_2)F_{attr'_2}(v_2)\times(1/|B_1|)F_{attr_1}(v_1)]\cdots]] \\ (2) \quad & \times\sum_{v_{i+1}\in B_i}[(1/|B_i|)F_{attr'_i,attr_{i+1}}(v_i,v_i+1) \\ & \times(1/|B_i|)F_{attr_{i+1}}(v_{i+1})\times(1/|B_{i+1}|)F_{attr_{i+1}'}(v_{i+1}) \\ & \cdots \\ & \times\sum_{v_{n-1}\in B_{n-1}}[(1/|B_{n-1}|)F_{attr'_{n-1},attr_n}(v_{n-1},v_n) \\ & \times(1/|B_{n-1}|)F_{attr_n}(v_n)\times(1/|B_n|)F_{attr'_n}(v_n)]\cdots]]\end{aligned}$$

[]

$$\begin{aligned} & \mathbf{P}(\mathbf{right_{value}}) = P(v_{i+1}\in DG) = \\ (1) \quad & \sum_{v_i\in B_i}[(1/|B_i|)F_{attr'_i,attr_{i+1}}(v_i,v_{i+1}) \\ & \times(1/|B_i|)F_{attr'_i}(v_i)\times(1/|B_{i-1}|)F_{attr_i}(v_i) \\ & \times\sum_{v_{i-1}\in B_{i-1}}[(1/|B_{i-1}|)F_{attr'_{i-1},attr_i}(v_{i-1},v_i) \\ & \times(1/|B_{i-1}|)F_{attr'_{i-1}}(v_{i-1})\times(1/|B_{i-2}|)F_{attr_{i-1}}(v_{i-1}) \\ & \times\sum_{v_{i-2}\in B_{i-2}}[(1/|B_{i-2}|)F_{attr'_{i-2},attr_{i-1}}(v_{i-2},v_{i-1}) \\ & \times(1/|B_{i-2}|)F_{attr'_{i-2}}(v_{i-2})\times(1/|B_{i-3}|)F_{attr_{i-2}}(v_{i-2}) \\ & \cdots \\ & \times\sum_{v_2\in B_2}(1/|B_2|)F_{attr'_2,attr_3}(v_2,v_3) \\ & \times(1/|B_2)F_{attr'_2}(v_2)\times(1/|B_1|)F_{attr_1}(v_1)]\cdots]] \\ (2) \quad & \times(1/|B_i|)F_{attr_{i+1}}(v_{i+1})\times(1/|B_{i+1}|)F_{attr_{i+1}'}(v_{i+1}) \\ & \cdots \\ & \times\sum_{v_{n-1}\in B_{n-1}}[(1/|B_{n-1}|)F_{attr'_{n-1},attr_n}(v_{n-1},v_n) \\ & \times(1/|B_{n-1}|)F_{attr_n}(v_n)\times(1/|B_n|)F_{attr'_n}(v_n)]\cdots]]\end{aligned}$$

The analytical model developed here provides a complete framework to evaluate if a given application can benefit from the *edge fitting* algorithm. Given an application and associated data, the probability analysis can be used to determine the expected merits of the View Derivation Graph. For example, if the distribution of the attribute values is skewed so that $P(noFit) < P(Fit) + P(1/2Fit)$, then the *edge fitting* approach can be easily applied.

5 Discussion

In this paper, we have developed a new approach for incremental maintenance of warehouse views in disconnected environments. Traditional approaches for incremental view maintenance assume continuous connectivity and rely on executing view computation queries on-the-fly by contacting base sources where data resides. With increasing reliance on warehousing and OLAP it has become important to provide data warehousing support for non-traditional applications such as mobile and disconnected environments, and unreliable systems. We have developed a notion of **view derivation graph** that maintains intermediate information for computing warehousing views and have developed the **edge fitting** algorithm that allows, in the best case, to refresh the views locally as a result of updates to the underlying data sources. In addition, we have also developed an analytical tool to evaluate the suitability of the proposed approach for a given application data. In particular, if the data has skewed distribution, which has been increasingly demonstrated for Web data[1], view derivation graph with edge fitting will have significant benefits.

We plan to extend the proposed **edge fitting** maintenance model, by analyzing possible methods of limiting further the storage overhead of the view derivation graph and of the communication between the view and base sources. Optimizations in communication are possible by storing additional information such as partial derivation paths, if we can predict their future use in the derivation graph.

The model presented is developed specifically for one or many independent data warehouse views that derive data directly from base sources. The distributed multi-view data warehouse is emerging as a design solution for a growing set of applications, such as storage of messages from newsgroups [7] or cooperative web caches for the Internet. However, a more general data warehouse system allows views to derive data from other views [15, 16, 8] as well as base sources. We propose to increase the capability of the model to also handle the hierarchical warehouse system.

[1] Several studies have shown that accesses and updates to data on the Web in general follows a Zipf distribution.

References

[1] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance in Data Warehouses. In *Proceedings of the 1997 ACM International Conference on Management of Data*, pages 417–427, May 1997.

[2] Daniel Barbara and Tomasz Imilienski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[3] Panos Chrysanthis. Transaction Processing in Mobile Computing Environment. *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[4] Y. Cui and J. Widom. Practical Lineage Tracing in Data Warehouses. Technical report, Department of Computer Science, Stanford University, 1999.

[5] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2):3–18, June 1995.

[6] Ashish Gupta, H.V. Jagadish, and Inderpal Singh Mumick. Data Integration using Self-Maintainable Views. In *Proceedings of the International Conference on Extending Data Base Theory*, 1996.

[7] H. Gupta and D. Srivastava. The Data Warehouse of Newsgroups. In *Proceedings of the Seventh International Conference on Database Theory (ICDT)*, pages 471–488, 1999.

[8] W. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the Warehouse Update Window. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1999.

[9] S. Weissman Lauzac and P. K. Chrysanthis. Programming Views for Mobile Database Clients. In *Proceedings of the 9th International Workshop on Database and Expert Systems and Applications*, August 1998.

[10] S. Weissman Lauzac and P. K. Chrysanthis. Utilizing Versions of Views within a Mobile Environment. In *Proceedings of the 9th International Conference on Computing and Information*, June 1998.

[11] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1997.

[12] Shirish Phatak and B. R. Badrinath. Multiversion Reconciliation for Mobile Databases. In *Proceedings of the 15th International Conference on Data Engineering*, 1999.

[13] Evaggelia Pitoura and Bharat Bhargava. Building Information Systems for Mobile Environments. In *Proceedings of the CIKM conference*, 1994.

[14] Dallan Quass, Ashish Gupta, Inderpal Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, December 1996.

[15] I. Stanoi, D. Agrawal, and A. El Abbadi. Weak Consistency in Distributed Data Warehouses. In *Proceedings of the International Conference of Foundations of Data Organization*, November 1998.

[16] I. Stanoi, D. Agrawal, and A. El Abbadi. Modeling and Maintaining Multi-View Data Warehouses. In *Proceedings of the 18th International Conference on Conceptual Modeling (ER'99)*, November 1999.

[17] J. Yang and J. Widom. Temporal View Self-Maintenance in a Warehousing Environment. Technical report, Department of Computer Science, Stanford University, 1999.

[18] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.

[19] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, December 1996.