Designing View Maintenance Algorithm in Data Warehousing Environment

Mukesh Mohania* Shin’ichi Konomi† Yahiko Kambayashi‡ Millist Vincent§

Abstract

A data warehouse stores materialized views generated from the underlying source data. Materialized views are used to speed up query processing on large amounts of data. These views need to be maintained in response to updates in the source data. This is often done, for reasons of efficiency, using incremental techniques rather than recomputing the view from scratch. In this paper we investigate the problem of incremental maintenance of materialized views in data warehouses. We consider views defined by relational algebraic operators and aggregate functions. We show that a materialized view can be maintained without accessing the view itself by materializing and maintaining additional relations. These relations are derived from the intermediate results of the view computation. We first give an algorithm for determining what additional relations need to be materialized in order to maintain a materialized view incrementally. We then propose an efficient incremental algorithm for updating the materialized view (and the additional relations) based on the optimized operator tree used for evaluating the view as a query. One important feature of our algorithm is that it derives the 'exact change' to every materialized additional relation, including the materialized view, without accessing the view itself. This feature is important to ensure the correctness of the update to views defined by aggregate functions. Also, this is important in active database applications where triggers are fired by updates to the view. We also show that our incremental maintenance technique can be extended to make the view (and the additional relations) self-maintainable by materializing some 'selected' source data.

Key Words: Materialized view, Auxiliary relation, Incremental view maintenance, Data warehousing.

1 Introduction

Materialized views are important in data warehouses [27] for fast retrieval of derived data regardless of the access paths and complexity of view definitions. When the underlying database relations are updated by insertion, deletion, or modification of tuples, a materialized view must also be updated to ensure the correctness of answers to queries against it. Updating the materialized view by full recomputation is often expensive and a more efficient technique can be to update the view incrementally. By this, we mean that the new view is computed from the existing view and the

*Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, The Levels 5095, Australia, Email: mohania@cis.unisa.edu.au. Work was partially supported by Australian Research Council through a research grant and part of work was done while visiting Kyoto University and supported by JSPS
†GMD-IPSI, AMBIENTE, Dolivostr. 15, D-64293 Darmstadt, Germany, Email: konomi@kuis.kyoto-u.ac.jp
‡Dept. of Information Science, Kyoto University, Sakyo, Kyoto 606-01, Japan, Email: yahiko@kuis.kyoto-u.ac.jp
§Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, The Levels 5095, Australia, Email: vincent@cis.unisa.edu.au. Work was partially supported by Australian Research Council through a research grant.
changes to the base relations. There has been significant research on view maintenance in both centralized and distributed databases [3, 5, 8, 10, 11, 12, 21, 25, 26], however, less attention has been given to the maintenance of views in data warehouses until recently [1, 13, 14, 20, 29, 30]. Moreover, techniques discussed in [3, 5, 8, 10, 11, 12, 21, 25, 26] cannot be extended to data warehousing because of several distinct features of data warehousing. Firstly, these techniques assume that each data source understands view management and knows the relevant view definitions, whereas in data warehouses the view definitions and the data sources are often decoupled. Secondly, views in data warehousing are more complicated than standard views and often contain highly aggregated and summarized information [27]. Such views may not be expressible using a standard relational view definition language such as SQL.

In this paper we investigate the problem of incremental maintenance of aggregate views in a data warehouse. We consider views that contain relational algebraic operators and aggregate operators, and can be represented by operator trees [17]. We assume that duplicates are not retained in the materialized views. We show that a materialized view can be maintained efficiently by maintaining and materializing some additional results at the warehouse, called auxiliary relations, which may or may not contain intermediate results of the view. While deriving the auxiliary relations, referential integrity constraints are generated between the auxiliary relations and base relations. We give an algorithm for determining which auxiliary relations are needed in order to maintain a view. These relations make it possible to maintain an aggregate view without recomputing the intermediate results from scratch, thus significantly reducing the total computing and communication cost. In addition to reducing the cost of maintaining a view, storing auxiliary relations has additional benefits. Firstly, these auxiliary relations can be used in maintaining multiple views having common subexpressions, where the auxiliary relations will correspond to these subexpressions. Secondly, they may also be used for answering ad-hoc queries in data marts, where each data mart contains a subset of data in the data warehouse relevant to a particular domain of analysis [7, 20]. Lastly, the relations may be used to maintain a view when the view definition itself is slightly modified [19].

Since auxiliary relations also change in response to updates to the base relations, these relations need to be maintained along with the materialized view. We propose an efficient incremental algorithm that updates each auxiliary relation and the materialized view in response to changes in the base relations. The proposed algorithm is based on the optimized operator tree [6] that is used for evaluating the view as a query. Updates to nodes in the operator tree are propagated in a bottom-up fashion. The update to each node in the tree is derived from the updates to its children nodes and the auxiliary relations materialized for the children and the node itself. One important feature of our algorithm is that it derives the ‘exact change’ at each intermediate node. That is, if a deletion of tuple $t$ from the view is produced by the algorithm, then $t$ is guaranteed not to exist in the new view, and likewise, if an insertion of tuple $t$ to the view is produced by the algorithm, then $t$ does not exist in the old view. This feature is important to ensure the correctness of the update to views defined by aggregate functions $\text{sum}$ and $\text{count}$. For instance, if a tuple is inserted into such a view which is already in the view, then the value of $\text{sum}$ and $\text{count}$ will be erroneously modified. Also, the exact change feature is important in active database applications where triggers are fired by updates to the view. If these updates are not exact change updates, then the actions caused by triggers may be incorrect.

We also show that our incremental technique can be easily extended to make the view and auxiliary relations self-maintainable. By this we mean that the view and auxiliary relations can be maintained without accessing base relations. This is done by materializing those base relations which are the children of a join node in the view operator tree. Note that since we use an optimized operator tree for updating the view, when the unary operators (selection and projection) are pushed
down as far as possible, we only need to materialize the result of applying the unary operator, if any, to the base relations rather than materializing the whole base relations. This is similar to the approach used in [20, 13]. Our approach also does not need to access the materialized view itself during maintenance. The advantage of this is that it can increase the availability of the view to the data warehouse users.

The rest of the paper is organized as follows. We introduce our motivating example in Section 2. In Section 3, we present our algorithm for determining which auxiliary relations are needed to maintain the view. Our incremental algorithm for view maintenance is presented in Section 4. In Section 5, we extend our incremental approach to guarantee self-maintainability of the view and the auxiliary relations. In Section 6, we contrast our approach with other work on materialized view maintenance. Finally, Section 7 contains our conclusions and future plans.

2 A Running Example

In this section we give a running example for illustrating our view maintenance algorithm.

Consider a data warehouse collecting data from three different data sources, namely Employment, Administration, and ResearchLab. Four base relations are stored across the sources and their schemas are as follows (the primary key in each relation is underlined):

- EMP(E#, EName, D#); This gives the name and department of each employee. Assume it is stored at Employment site.
- DEPT(D#, DName, Area); This gives the name of each department and its location. Assume that it is stored at Administration site.
- RSCHR(E#, D#, Major); This gives the department and major of each researcher. Assume that it is stored at ResearchLab site. Since a researcher is an employee, relation EMP has a tuple for each researcher.
- MNGR(E#, D#); This gives the department of each manager. Assume that it is stored at Administration site. Since a manager is an employee, relation EMP has a tuple for each manager.

Consider the following query:

‘Determine all the departments which are located in the ‘East’ area and have more than 10 employees who are neither researchers nor managers.

Since this query is often needed, it is profitable to store it as a materialized view, called EastAdminDept. In SQL it is defined as a sequence of view definitions:

View Definition 1 (EastEmp)
CREATE VIEW EastEmp(E#, D#) AS
SELECT E#, D# FROM EMP, DEPT
WHERE EMP.D# = DEPT.D# AND Area = ‘East’;

View Definition 2 (RschrOrMngr)
CREATE VIEW RschrOrMngr(E#, D#) AS
SELECT E#, D# FROM RSCHR UNION
(SELECT E#, D# FROM MNGR);
View Definition 3 (EastMiscEmp)
CREATE VIEW EastMiscEmp(E#, D#) AS
SELECT E#, D# FROM EastEmp WHERE NOT EXISTS
(SELECT * FROM RschrOrMngr
WHERE EastEmp.E# = RschrOrMngr.E#);

View Definition 4 (EastAdminDept)
CREATE VIEW EastAdminDept(D#) AS
SELECT D# FROM EastMiscEmp
GROUP BY D# HAVING count(*) > 10;

3 Deriving Auxiliary Relations using an Operator Tree

3.1 Operator tree

View evaluation can be represented by a tree, called an operator tree [6]. An operator tree is a tree, where the leaf nodes represent base relations and non-leaf nodes represent binary operators in the relational algebra or Group By and aggregate functions. The unary relational algebraic operators are associated along the edges.

In database systems, a view (as a query) is optimized by the query optimizer before executing it. Query optimization is a complex task and therefore has been the subject of significant research and development [2, 6, 15]. Query optimization is required to ensure that the total computation cost is minimal. A query optimizer takes an operator tree as input and produces an output, called an optimized operator tree, which determines the internal sequence of operations for executing a query. Thus, an optimized operator tree defines a partial order in which operations must be performed in order to produce the result of the view. (Note that different query optimizers can determine different cost models of evaluating the query.)

We define the depth of each node and the height of the tree in the following definitions.

Definition 1 The depth of leaf nodes (i.e. database relations) is 0. The depth \( d \) of a node is defined as \( \text{max}(\text{depth of descendents}) + 1 \).

Definition 2 The height \( h \) of the optimized operator tree is defined as the maximum depth of any node in the tree.

The optimized operator tree for our running example is shown in Figure 1. Here, the nodes at leaf level are database relations and non-leaf nodes are operators. Each non-leaf node in the tree corresponds to an SQL view defined in Section 2. We make use of the optimized operator tree for deriving auxiliary relations and for processing updates on the view. In the rest of the paper, we refer to the optimized operator tree as ‘operator tree’.

3.2 Auxiliary Relations

Given a materialized view to be maintained, it is possible to reduce the cost of view maintenance by materializing additional results, called auxiliary relations, of the view. An auxiliary relation may (but need not) contain intermediate results of the view. Obviously there are overheads incurred by maintenance of these auxiliary relations, but their use can often significantly reduce the cost of computing the updates to the materialized view. By maintaining these relations, a view can be maintained incrementally without recomputing intermediate results from scratch and the exact
change to every intermediate step can be derived from them. As discussed in Section 1, there are additional benefits in materializing the auxiliary relations apart from maintaining the view. Note that these relations are also maintained incrementally along with the materialized view. This issue is addressed in Section 4.

In this section we present our algorithm for determining, and then materializing, which auxiliary relations are needed for a given view. This algorithm always determines only one set of auxiliary relations for materialization irrespective of which relation is updated (this is in contrast to the approach used in [23]). As discussed above, the execution of a view can be represented by an operator tree and we make use of this tree for deriving the auxiliary relations for the view. Since a view is executed in a bottom-up fashion in the tree and each node of the tree is computed, we initially derive the contents of the auxiliary relations from the intermediate results of the view. We maintain one auxiliary relation for each join, union and Group By operator and two auxiliary relations for the difference operator because it is not a commutative operator. Recall that each binary and Group By operator corresponds to a node in the operator tree, and therefore we can also say that an auxiliary relation is materialized for each node.

We now introduce some additional notation. Given a node $i$ in the operator tree, its parent is denoted by $\uparrow i$, and $op(i)$ and $op(\uparrow i)$ are the operators associated with $i$ and $\uparrow i$, respectively. The children of node $i$ are denoted by $\hat{i}$ and $\hat{i''}$, where $\hat{i}$ is a sibling of $\hat{i''}$ and vice versa. $IR_i$ denotes the intermediate result of node $i$. The auxiliary relation associated with node $i$ is denoted by $AR_i$ in the case where only one relation is needed, and by $AR_{i1}$ and $AR_{i2}$ when two are needed. The key of $IR_i$ is denoted by $K_i$, and the keys of $IR_{i'}$ and $IR_{i''}$ are denoted by $K_{i'}$ and $K_{i''}$, respectively. Let $CNT$ denote the count attribute in the auxiliary relation where we need to store the count information. We assume that the key of every node is known and the key of a child node is a subset of the key of its parent node. We now outline our algorithm for determining the auxiliary relations for each node in the view operator tree. The guiding principle is to materialize only those auxiliary relations which are necessary for computing the exact change to every node and the view. The algorithm
MatAuxRel for materializing auxiliary relations is described in Figure 2.

Input: operator tree of a view, intermediate results of the initial view computation.

Output: auxiliary relation for each node in the tree.

Algorithm MatAuxRel

Determine height \( h \) of the tree and depth \( d \) of each node using depth-first search algorithm.

\[ \text{for } d = 1 \text{ to } h \text{ do} \]

\[ \text{for each node } i \text{ having depth } d \text{ do} \]

Case 1 (\( op(i) = \text{`OR'} \)): If \( op(\uparrow i) \neq \text{`OR'} \), then materialize auxiliary relation \( AR_i = \pi_K (IR_0 \bowtie IR_\uparrow) \), else materialize \( AR_i = IR_i \).

Case 2 (\( op(i) = \text{`-' } \)): Materialize both \( AR_i^1 = IR_i = IR_0 - IR_\uparrow \) and \( AR_i^2 = IR_\uparrow - IR_0 \).

Case 3 (\( op(i) = \text{`' } \)): If \( op(\uparrow i) \neq \text{`' } \), then materialize \( AR_i = \pi_K (IR_i) \) and augment \( CNT \) to \( AR_i \).

(\( CNT \) indicates whether a tuple is present in both descendents of \( i \) or not), else materialize \( AR_i = IR_i \) and augment \( CNT \) to \( AR_i \).

Case 4 (\( op(i) = \text{`GroupBy'} \)): Materialize \( AR_i = IR_i \) and augment \( CNT \) to \( AR_i \).

endfor
endfor
end_algorithm

Figure 2: An algorithm for Materializing Auxiliary Relations

We now explain the intuition behind this algorithm.

If \( op(i) = \text{`OR'} \), then we have to access the results of both child nodes in order to compute the exact change to \( i \). The reason is that the updated tuples in child node can join with the tuples from the sibling node to produce tuples that are not in \( IR_i \). This implies that if \( op(\uparrow i) \neq \text{`OR'} \), then we must materialize \( IR_i \). If the \( op(\uparrow i) = \text{`OR'} \) then, as we will see next, we do not need to fully materialize \( IR_i \) in order to compute the change to \( \uparrow i \). It suffices to materialize the key of \( IR_i \) in \( AR_i \) in order to compute the exact change to \( i \) and to \( \uparrow i \).

If \( op(i) = \text{`-' } \), then we materialize both \( AR_i^1 = IR_i - IR_\uparrow \) and \( AR_i^2 = IR_\uparrow - IR_i \), since \( \text{`-' } \) is not a commutative operator. The reason for materializing both \( AR_i^1 \) and \( AR_i^2 \) is that if there are updates to node \( i' \) (node \( i'' \), say \( \delta i' (\delta i'') \), then the updates to \( i \) can be derived from \( \delta i' - AR_i^1 (\delta i'' - AR_i^2) \).

If \( op(i) = \text{`' } \) and \( op(\uparrow i) \neq \text{`OR'} \), then we materialize the key attributes of \( IR_i \) at \( i \) along with the count attribute, \( CNT \). This attribute \( CNT = 2 \) if the tuple is in both children of \( i \), and \( CNT = 1 \) if the tuple is in only one of the children. If there is a change in one child, then the incremental change to \( i \) can be derived from the auxiliary relation and the change to the child. In the case of an insertion, the incremental change to \( i \) will consist of those tuples in the change to the child whose key values are not present in \( AR_i \). In the case of a deletion, we delete those tuples from \( AR_i \) whose the key value matches a key value in the change to the child. If \( op(\uparrow i) = \text{`OR'} \), then we materialize \( IR_i \), rather than only its key values, along with the count attribute for the reason just discussed above.

If \( op(i) = \text{`GroupBy'} \), then we materialize \( IR_i \) along with the count attribute. The count of a tuple \( t \) in \( AR_i \) indicates how many tuples are grouped from the child result to create \( t \). This is necessary to compute the new values of \( AVG \) and \( COUNT \) aggregate functions.
Materialization of auxiliary relations for our running example

Consider the operator tree of our running example, shown in Figure 1. There are four non-leaf nodes in the tree, namely EastEmp, RschrOrMngr, EastMiscEmp, and EastAdminDept. They are denoted by \( n_1, n_2, n_3, \) and \( n_4 \) respectively. The contents of auxiliary relations are calculated in a bottom-up fashion as follows:

(i) The auxiliary relation materialized for \( n_1 \) will be \( AR_1(E\# , D\#) \) because \( op(\gamma n_1) = op(n_3) \neq ' \cdot ' \). The key of \( AR_1 \) is \( E\# \).

(ii) The auxiliary relation for \( n_2 \) will be \( AR_2(E\# , CNT) \) because \( op(\gamma n_2) = ' \cdot ' \). Its key will be \( E\# \).

(iii) Since \( op(n_3) = ' \cdot ' \), we materialize both \( AR_3(E\# , D\#) = EastEmp - RschrOrMngr \) and \( AR_4(E\# , D\#) = RschrOrMngr - EastEmp \). The key is \( E\# \) in both auxiliary relations.

(iv) The auxiliary relation for \( n_4 \) will be \( AR_4(D\# , CNT) \) with \( D\# \) being the key.

![Figure 3: An Operator Tree with Auxiliary Relations](image)

The operator tree with the resulting auxiliary relations is shown in Figure 3. We now show how the auxiliary relations are used to compute the changes to the materialized views.

4 View Maintenance Algorithm

Based on the technique for materialising auxiliary relations outlined in the previous section, we now describe our view maintenance algorithm for calculating the changes to the view. We consider view definitions having both relational algebraic operators and aggregate functions. Since the execution of a view can be represented by a tree, we compute the changes to the view in a bottom-up fashion in the tree. That is, the changes to each intermediate node are calculated and then they are progressively propagated all the way up to the root node (the root node represents the final view itself). The update to each node in the tree is derived from the updates to its children and the auxiliary relations materialized for the children and the node itself. It is always possible to calculate the
changes to a given node since the changes to its children have already been calculated since we use a bottom-up approach. The proposed algorithm also maintains auxiliary relations incrementally while calculating the changes to nodes. Our algorithm is also characterized by the feature that it calculates the exact change to every intermediate node and to the view without accessing the view itself. That is, if a deletion of tuple \( t \) from the view is produced by the algorithm, then \( t \) is guaranteed not to exist in the new view, and likewise, if an insertion of tuple \( t \) to the view is produced by the algorithm, then \( t \) does not exist in the old view.

4.1 Exact Change Computations

In this section, we formally define the ‘exact change’ property for insertions and deletions, and then describe how we calculate the exact change to every node and how we maintain the auxiliary relation(s) for each node.

**Definition 3** An insertion \( \Delta V \) to a view \( V \) satisfies the exact change property iff

1. \( V^{new} = V^{old} \cup \Delta V \)
2. \( \Delta V \cap V^{old} = \emptyset \)

**Definition 4** A deletion \( \nabla V \) to a view \( V \) satisfies the exact change property iff

1. \( V^{new} = V^{old} - \nabla V \)
2. \( \nabla V \cap V^{new} = \emptyset \)

We note that these definitions differ from those given in [10] since we use set semantics rather than bags semantics.

Let \( i \) be any join, union, difference, or Group By node. Let the exact change to the intermediate expression at node \( i \) (\( IR_i \)) be denoted by \( EC_i \) for insertion and \( EC_i \) for deletion. Let \( AR_i^{new} \) and \( AR_i^{old} \) be the auxiliary relations before and after the updates to \( AR_i \), and let the key of \( AR_i \) be denoted by \( k \).

4.1.1 Exact change to the join node

We employ the exact change technique for join from [3]. Recall that from the algorithm discussed in Section 3, the auxiliary relations associated with the children of a join node contain the intermediate results for these child nodes. We now briefly summarize the algorithm discussed in [3]. For simplicity, we describe the algorithm for the view \( V \) defined by

\[ V = \pi_{A,D}(R \bowtie_{B=C} S) \]

where the schema of \( R \) is \( R(\bar{A},\bar{B}) \) and the schema of \( S \) is \( S(\bar{C},\bar{D}) \). (Note that \( R \) corresponds to the auxiliary relation associated with the left child and \( S \) corresponds to the auxiliary relation associated with the right child of the join node.) Suppose \( \delta R \) tuples are updated (either inserted to or deleted from) in relation \( R \). The basic idea in this algorithm is to compute two sets of tuples from \( R \), say \( I_1 \) and \( I_2 \), and send them to the site of \( S \) for joining, say the results are \( J_1 \) and \( J_2 \) after join. The exact change to the view \( V \), say \( \delta V \), (note that \( V \) corresponds to the auxiliary relation at the ‘join’ node) is calculated by subtracting \( J_2 \) from \( J_1 \), i.e. \( J_1 - J_2 \). We now show below the steps in computing the exact change to the view \( V \). In the following algorithm, if the tuples are inserted, then \( \delta \) will be replaced by \( \Delta \) else \( \nabla \). The \( R^{new} \) and \( R^{old} \) denote the relation \( R \) before and after the updates.

**Exact Incremental Change Algorithm for join**
The exact change and the new auxiliary relations are computed for an insertion as follows:

- \( I_1 = \{ t \mid t \in \delta R \} \);
- \( I_2 = \{ t \in R^{\text{new}} \mid t[A] \in \pi_A(I_1) \} \);
- \( J_1 = \pi_{\text{left}}(I_1 \setminus \delta_{\text{left}} S) \);
- \( J_2 = \pi_{\text{left}}(I_2 \setminus \delta_{\text{left}} S) \);
- \( \delta V = J_1 - J_2 \).

The exact change to node \( i \) will be \( \delta V \) and the new auxiliary relation will be \( AR_i \cup \delta V \) in the case of insertion and \( AR_i - \delta V \) in the case of deletion.

### 4.1.2 Exact change to the Union node

In the case of a union node, it is known [8] that the exact change to the corresponding auxiliary relation cannot be determined by only storing the result of the union node. Our approach, as outlined in the previous section, is to add an extra count attribute to the auxiliary relation for a union node that describes whether the tuple is present in both of its children or in only one child. In the case of an insertion, we determine the exact change to the union node \( i \) by comparing the key values of \( \Delta EC \) with the key values of \( AR_i \). If the key value matches, then the tuple from \( \Delta EC \) will not be inserted but the count (CNT) in \( AR_i \) will be incremented by 1, otherwise the tuple will be inserted with \( CNT = 1 \). The exact change to the union node will be all those tuples which are inserted with \( CNT = 1 \). We now present these computations more formally for the case of insertion. Here, \( CNT(t, AR_i^{\text{new}}) \) gives the count value of \( t \) in \( AR_i^{\text{new}} \).

\[
\Delta EC_i = \{ t \mid t \in \Delta EC_i \land t[k] \notin \pi_k(AR_i^{\text{old}}) \} \\
AR_i^{\text{new}} = \{ t \mid t \in AR_i^{\text{old}} \land t[k] \in \pi_k(\Delta EC_i) \land \\
(CNT(t,AR_i) = 1 \text{ if } t \notin AR_i^{\text{old}} \land t[k] \in \pi_k(\Delta EC_i)) \land \\
\text{else } CNT(t,AR_i^{\text{new}}) = 2 \text{ if } t \in AR_i^{\text{old}} \land t[k] \in \pi_k(\Delta EC_i)) \}
\]

In the case of a deletion, if the key value of a tuple in \( \nabla EC \) matches with the key of \( AR_i \) and \( CNT = 2 \), then the tuple will not be deleted but the \( CNT \) of the tuple will be set to 1, otherwise the tuple will be deleted from the \( AR_i \). The exact change to this node will be all those tuples which are deleted from \( AR_i \). More formally:

\[
\nabla EC_i = \{ t \mid t \in \nabla EC_i \land t[k] \in \pi_k(AR_i^{\text{old}}) \} \\
AR_i^{\text{new}} = \{ t \mid t \in AR_i^{\text{old}} \land t[k] \notin \pi_k(\nabla EC_i) \land \\
(CNT(t,AR_i^{\text{old}}) = 1 \text{ if } CNT(t,AR_i^{\text{old}}) = 2 \land t[k] \in \pi_k(\nabla EC_i)) \}
\]

In the above computations we have only considered the changes to the left child of the union node. If the change is made to the right child, it can be handled as above because of symmetry.

### 4.1.3 Exact change to the Difference node

Recall that we maintain two auxiliary relations with a difference node since the difference operator is not commutative. Note that with a difference node, an insertion to its right child may cause a deletion to the difference node, and vice-versa. We present here the case when the change is made to the left child. The other case when the change is made to the right child can be handled similarly. The exact change and the new auxiliary relations are computed for an insertion as follows:
\[ \Delta EC_i = \{ t \mid t \in \Delta EC_{\varphi} \land t \notin AR_i^{2(\varphi, d)} \} \]
\[ AR_i^{1(new)} = \{ t \mid t \in AR_i^{1(d)} \lor t \in \Delta EC_i \} \]
\[ AR_i^{2(new)} = \{ t \mid t \in AR_i^{2(d)} \land t \notin \Delta EC_i \} \]

For the case of a deletion:
\[ \nabla EC_i = \{ t \mid t \in \nabla EC_{\varphi} \land t \notin AR_i^{1(d)} \} \]
\[ AR_i^{1(new)} = \{ t \mid t \in AR_i^{1(d)} \land t \notin \nabla EC_i \} \]
\[ AR_i^{2(new)} = \{ t \mid t \in AR_i^{2(d)} \lor (t \in \nabla EC_i \land t \notin \nabla EC_i) \} \]

4.1.4 Exact change to the Group By node

When the operation to a node is GroupBy and involves aggregate functions (i.e. AVG, COUNT, MAX, MIN, and SUM), then the change is calculated by executing a set of queries on the auxiliary relation materialized for this GroupBy node. Basically, these queries compute the new values of grouping attributes after the child node of \( i \) has been updated. We write these queries in pseudo-SQL and they are discussed next.

Let \( G \) be the grouping attributes (i.e. attributes specified in the GroupBy clause) and let attributes \( F \) represent the value of the aggregate functions evaluated on each group. Suppose \( \delta EC_{\varphi} \) is the incremental update (either a \( \Delta \) or a \( \nabla \)) derived at node \( i' \). Since \( i' \) is a child of \( i \), the update to \( i' \) must be propagated to \( i \). Recall that a count attribute \( CNT \) is augmented to \( AR_i \) which indicates how many tuples are grouped from the child result. The update to node \( i \) is calculated as follows:

(i) When the view has either a SUM, AVG or COUNT aggregate function

1. apply GroupBy operator on \( \delta EC_{\varphi} \) and augment \( CNT \) to the result. The result stored in the same \( \delta EC_{\varphi} \).
2. execute the following query (if \( \delta \) is \( \Delta \), then we shall take ‘+’, else ‘−’); it gives those tuples whose values of grouping attributes are the same in both \( AR_i \) and \( \delta EC_{\varphi} \).
   - For SUM aggregate function
     \[ \text{SELECT } G, F = AR_i, F + (−)\delta EC_{\varphi}.F \text{ INTO } \delta EC_i \]
     \[ \text{FROM } AR_i, \delta EC_{\varphi} \]
     \[ \text{WHERE } AR_i.G = \delta EC_{\varphi}.G \]
   - For AVG aggregate function
     \[ \text{SELECT } G, F = \frac{AR_i, F + (−)\delta EC_{\varphi}.F}{AR_i,CNT + (−)\delta EC_{\varphi}.CNT} \text{ INTO } \delta EC_i \]
     \[ \text{FROM } AR_i, \delta EC_{\varphi} \]
     \[ \text{WHERE } AR_i.G = \delta EC_{\varphi}.G \]
   - For COUNT aggregate function
     \[ \text{SELECT } G, F = AR_i, CNT + \delta EC_{\varphi}.CNT \text{ INTO } \delta EC_i \]
     \[ \text{FROM } AR_i, \delta EC_{\varphi} \]
     \[ \text{WHERE } AR_i.G = \delta EC_{\varphi}.G \]
3. execute the following query for finding those tuples whose values of grouping attributes are exclusively in $\delta EC_i$. The result is appended to the result of above query.

```sql
INSERT $\delta EC_i$
SELECT G, F FROM $\delta EC_i$ WHERE NOT EXISTS
(SELECT * FROM AR_i WHERE AR_i.G=$\delta EC_i$.G)
```

4. update $AR_i$ by $\delta EC_i$; modify the values of aggregate attributes of those tuples of $AR_i$ whose values of grouping attributes are the same in $\delta EC_i$ by the corresponding values of $\delta EC_i$. Then insert (delete) those tuples of $\delta EC_i$ into $AR_i$ whose values of grouping attributes are present in $\delta EC_i$, but not in $AR_i$.

(ii) When view has either a $MIN$ or $MAX$ aggregate function
(a) When the update to $i$ is $\Delta$ (insertion of tuples)

1. apply $\text{GroupBy}$ operator to $\Delta EC_i$,
2. execute the following query for $MAX$ ($MIN$) aggregate function; it gives those tuples of $\Delta EC_i$ whose values of grouping attributes are the same in $AR_i$, and the values of the aggregate attributes are greater (less) than the corresponding values in $AR_i$.

```sql
SELECT G, $\Delta EC_i$.F INTO $\Delta EC_i$
FROM $AR_i$, $\Delta EC_i$
WHERE $AR_i$.G=$\Delta EC_i$.G and $\Delta EC_i$.F > (<) $AR_i$.F
```

3. execute the following query for finding those tuples whose values of grouping attributes are present in $\Delta EC_i$, but not in $AR_i$. The result is appended to the result of above query.

```sql
INSERT $\Delta EC_i$
SELECT G, F FROM $\Delta EC_i$ WHERE NOT EXISTS
(SELECT * FROM AR_i WHERE AR_i.G=$\Delta EC_i$.G)
```

4. update $AR_i$ by $\Delta EC_i$; modify the values of aggregate attributes of those tuples of $AR_i$ whose values of grouping attributes are the same in $\Delta EC_i$ by the corresponding values of $\Delta EC_i$. Then insert those tuples of $\Delta EC_i$ into $AR_i$ whose values of grouping attributes are present in $\Delta EC_i$, but not in $AR_i$.

(b) When the update to $i$ is $\triangledown$ (deletion of tuples)

1. apply $\text{GroupBy}$ operator to $\triangledown EC_i$,
2. execute the following query for $MAX$ ($MIN$) aggregate function; it gives those tuples of $\triangledown EC_i$ whose values of grouping and aggregate attributes are the same in $AR_i$.

```sql
SELECT G, $\triangledown EC_i$.F INTO $\triangledown EC_i$
FROM $AR_i$, $\triangledown EC_i$
WHERE $AR_i$.G=$\triangledown EC_i$.G and $AR_i$.F=$\triangledown EC_i$.F
```

3. delete tuples $\triangledown EC_i$ from $AR_i$.
4. execute the following query for finding the tuples having values of grouping attributes the same as in $\triangledown EC_i$.

```sql
SELECT G, $\triangledown EC_i$.F INTO Temp_i
FROM $AR_i$, $\triangledown EC_i$
WHERE $AR_i$.G=$\triangledown EC_i$.G
```

5. apply $\text{GroupBy}$ operator on $Temp_i$ and insert the result into $AR_i$. 
4.2 Incremental Algorithm

We now outline our incremental algorithm based on the exact change computations described in previous section. Let the database relations $R_1, \ldots, R_m$ be updated by the transaction $T$. The updates to these relations could either be insertions, deletions or modifications of tuples. Note that modification of tuples is modeled as deletion followed by insertion of tuples. We assume that a materialized view is updated sequentially for each base relation update. The incremental update algorithm proceeds as follows:

1. Determine the depth $d$ of each node in the tree using depth-first search algorithm [22].

2. Determine the exact incremental change to nodes in a bottom-up fashion. It is always possible to derive the exact change to a node, since the changes to its children have already been calculated.

The algorithm $\text{ViewMntAlg}$ is described in Figure 4.

**Input:** Auxiliary relations, ‘delta’ relations

**Output:** updated auxiliary relations, updated materialized view

**Algorithm ViewMntAlg**

for each update to $R_i$ do

Find the all ancestors nodes, say $N$, of $R_i$ (these are the only nodes which need to be updated)

for $d = 1$ to $h$ do

for each node $i$ in $N$ having depth $d$ do

If updates are insertion of tuples,

then find exact change to $i$ and update its auxiliary relation.

If updates are deletion of tuples, then

find exact change to $i$ by deleting those tuples from its auxiliary relation which are having same key values as in the updates, and also update its auxiliary relation.

If $i$ is a root node, then update materialized view.

endfor

endfor

endfor

end algorithm

Figure 4: Incremental View Maintenance Algorithm

Note that when some of the base relations involved in the view definition are not updated, then there is no need to propagate updates along every path in the operator tree. In this case, it is not necessary to maintain and materialize auxiliary relations for all operators.

4.3 Example Revisited

Here we illustrate our view maintenance algorithm through our running example. The operator tree is given in figure 3. Suppose a set of tuples are inserted to $\text{EMP}$ relation. The instance of the materialized view, call $V$, before and after an update is denoted by $V^{old}$ and $V^{new}$ respectively. The following steps are performed to maintain the view using the exact change algorithms given in Section 4.1.
• Compute the change to node $n_1$, say $\Delta n_1$. Then update $AR_1$.

• The change to node $n_3$, $\Delta n_3$ is given by $\Delta n_1 - AR_3^0$. Then update $AR_3^3$.

• Compute the change at node $n_4$ by firstly applying the GroupBy operator on $\Delta n_3$ and computing a count attribute in the resulting schema. Then, execute the query given for the COUNT aggregate function, giving $\Delta n_4$. Lastly, update $AR_4$ with $\Delta n_4$.

• Finally, to derive the exact change to the view, select tuples on $CNT > 10$ from $\Delta n_4$, and then update the materialized view. That is, $V^{new} = V^{old} \cup (\sigma_{CNT>10} \Delta n_4)$.

If tuples are deleted from EMP relation, say $\nabla$ EMP, the following steps are performed to maintain the view.

• At node $n_1$, delete all tuples from $AR_1$ whose key values are in $\nabla$ EMP, say the result is $\nabla n_1$. Then update $AR_1$.

• At node $n_3$, $\nabla n_3$ is calculated by $\nabla n_3 = \nabla n_1 - AR_3^1$. Then update $AR_3^3$.

• Compute the change at node $n_4$ by firstly applying the GroupBy operator on $\nabla n_3$ and computing a count attribute in the resulting schema. Then, execute the query given for COUNT aggregate function, giving $\nabla n_4$. Lastly, update $AR_4$ with $\nabla n_4$.

• Finally, deriving the exact change to the view, we first take the selection for $CNT > 10$ on $\nabla n_4$, and then update the materialized view. That is, $V^{new} = V^{old} - (\sigma_{CNT>10} \nabla n_4)$.

5 Making Aggregate Views Self-Maintainable

Most of the view maintenance techniques [3, 12, 23, 13, 20, 29, 30] require access to the base relations. In the data warehousing scenario, accessing base relations can be difficult since these relations are distributed across different sources. Often the data sources may be unavailable or, even if available, the cost of accessing the sources may be prohibitive due to communication costs. For these reasons, the issue of self-maintainability of the view is an important issue in data warehousing [13, 20, 14].

In this section, we outline how to extend our incremental maintenance technique to make the view self-maintainable. We have shown in Section 4 that the exact change to every node in the operator tree, with the exception of join nodes at $d = 1$, can be computed from the auxiliary relations and the exact changes to the children nodes. The base relations themselves are not needed to calculate these changes (recall that the base relations are leaf nodes, i.e. $d = 0$, in the operator tree). When one of the children of a join node at $d = 1$ is updated, we need to access the other child (i.e. base relation) in order to to calculate the exact change at the join node as discussed in Section 3.

In order to avoid the need to access base relations in computing the exact change to a join node at $d = 1$, we can materialize these base relations at the warehouse. It is not necessary to materialize the whole of these base relations because in the optimized operator tree the unary operators are often pushed down to the leaf nodes. In such cases, its suffices to materialize the results of applying the unary operators to those base relations rather than materializing the entire relations. In our running example, the only base relations which are children of a join node are EMP and DEPT. However, unary operators are pushed down in the operator tree (shown in figure 1) to both these relations. Hence, the only base data that needs to be materialized at the warehouse is $\pi_{E \neq D \neq EMP}$ and $\pi_{D \neq A_{area=East}}(Dept)$. By materializing these relations, the view becomes self-maintainable.
6 Related Work

The view maintenance problem has been studied by many researchers in general [3, 4, 5, 8, 9, 10, 11, 12, 13, 16, 18, 20, 21, 23, 24, 25, 26, 28, 30] and the recent survey of view maintenance literature can be found in [11]. In all papers, except [23], views are defined as a subset of relational algebraic expressions, i.e. SPJ views, views with grouping/aggregation. In [23], the authors have proposed an exhaustive enumerative algorithm for maintaining a view that can be used for any relational algebraic expression, and have shown that the maintenance cost of it can be reduced by maintaining (and materializing) a set of additional views along with the original view. Their algorithm finds a set of views from the expression DAG and the choice of which additional views are finally selected for materialization depends on the cost model and the relations that are updated. Therefore, the optimal choice of additional views, that is finally chosen, may differ depending on the relations that are updated. Blakeley et al.[4] has presented the conditions to find out whether an update to a base relation can affect a derived relation or not. They also derive conditions to determine when a derived relation can be updated using its own data and the given updates. In [8], the authors have discussed many algorithms for different relational algebraic operators for maintenance of simple views. Their method is restricted to view definitions having only one operator. Two incremental tagged algorithms for maintenance of simple SPJ views, one for insertion and other for deletion of tuples, have been discussed in the [3]. These algorithms derive the exact change to the materialized view without accessing the materialized view itself. The main idea in these algorithms is to use tags on tuples in the base relation to reduce the communication cost. These tags indicate whether tuples participate in joins or not. The authors have also discussed the possible extension of their algorithms by using tags for memoing values of subexpressions in the view definitions. These algorithms cannot be used for complex view definitions. In [26], the authors have considered a problem of maintaining a collection of simple Select-Project views. They have developed a screen test procedure to filter out the tuples that need to be sent to remote sites. The work of [26] is extended in [25], which considers the use of updates to one view to maintain another view. The issue of determining optimal policies for the timing of view updates, which include periodical, on-demand, random and hybrid, are also discussed in [25]. A more recent work on view maintenance in data warehousing has been given in [30]. The algorithms presented in [30] are principally directed towards methods dealing with the control of concurrent updates and query request between the warehouse and the source. In [13], the authors have proposed an algorithm for making views self-maintainable. Their algorithm is based on pushing selections and projections to the base relations and storing the results in the warehouse. The extension of [13] was reported in [20] and here the authors have proposed an algorithm for maintaining SPJ views based on key and referential integrity constraints. In this paper, we improve upon the work reported in [13] and [20] by considering any view definition based on relational algebraic operators and aggregate operators.

7 Concluding Remarks and Future research

We have investigated the problem of incremental view maintenance in response to source data updates in data warehouses. A good solution to this problem is not only important in data warehousing environments but also in many database applications such as implementing SQL-92 assertions [23], and mobile computing [8]. We have shown that the cost of maintaining a view can be reduced by materializing additional auxiliary relations, which may or may not contain intermediate results of the view, at a data warehouse site. We have proposed an algorithm for determining which auxiliary relations are needed in order to efficiently maintain the view. We then presented
an efficient algorithm for updating a materialized view and the auxiliary relations in a data ware-
housing environment. We represent the evaluation of a view as an operator tree which we then use
for propagating updates to the view. The leaf nodes in the tree denote base relations and non-leaf
nodes denote intermediate operations. In this operator tree, updates are propagated in a bottom-
up fashion. Updates to a node are derived from the updates to its children nodes and the auxiliary
relations materialized for children nodes and the node itself. Our algorithm is characterized by the
feature of computing the exact change to every node in the tree, and so it guarantees the avoidance
of sending irrelevant tuples to data warehouse site; thereby reducing communication cost. More-
over, the auxiliary relations determined by our algorithm are independent of the relations that are
updated. We have also shown that our incremental algorithm can be easily extended to make
the view self-maintainable by materializing all base relations (after applying the unary operators)
which are children of a join node.

In the future we plan to conduct a detailed performance analysis of our algorithm vs. counting
algorithm [12] and the algorithm discussed in [23]. We also plan to extend our algorithm to other
data models such as the one described in [10]. In addition, we plan to investigate how constraints,
such as functional dependencies, can help in maintaining views more efficiently.

References

houses. In Proceedings of the ACM SIGMOD International Conference on Management of Data,


and autonomously computable updates. ACM Transactions on Database Systems, 14(3):369–400,
1989.

ACM SIGMOD Int. Conf. on Management of Data, pages 61–71, 1986.

1986.


Australian Workshop on Mobile Computing and Databases, Monash University, 1996.


SIGMOD Int. Conf. on Management of Data, 1995.


