Maintaining Horizontally Partitioned Warehouse Views

Mei Xu

School of Computer Science, University of Windsor

C.I. Ezeife*

School of Computer Science University of Windsor,

Windsor, Ontario, Canada N9B 3P4

cezeife@uwindsor.ca

Abstract

Data warehouses usually store large amounts of information, representing an integration of base data from different data sources over a long time period. Aggregate views can be stored as a set of its horizontal fragments for the purposes of reducing warehouse query response time and maintenance cost.

This paper proposes a scheme that efficiently maintains horizontally partitioned data warehouse views. Using the proposed scheme, only one view fragment holding the relevant subset of tuples of the view is accessed for each update. The scheme also includes an approach to reduce the refresh time for maintaining views that compute aggregate functions MIN and MAX.

Keywords: Data Warehouse Applications, View Maintenance, Horizontal Partitioning, Performance Improvement.

1 Introduction

Different data sources are usually integrated into one large central fact table in the data warehouse holding the main integrated data. Foreign key attributes in the fact table may refer to dimension tables, which describe the dimension attributes in the fact table. Data warehouse views are derived from the fact table and may possibly join one or more of the dimension tables. Cells of an n-dimensional data cube hold data that can easily construct 2^n subviews for an aggregate measure of interest. Breaking up a relation into its horizontal fragments would lead to a reduction in both query response time and maintenance

*This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor grant.

cost. Horizontal fragmentation partitions a relation such that each fragment contains only a subset of tuples of the relation that satisfy specific predicate conditions [2].

Data in a view may be deleted, updated, or new data can be inserted into a view with each update expressed as a delete-insert pair. These constitute refresh operations. Views can be maintained by computing the changes made to data sources and applying these changes to views to bring them up-to-date without accessing base data. This maintenance method is called incremental view maintenance.

1.1 Related Work

Harinayaran et al. in [3] uses a lattice framework to model the dependencies between subviews of the data cube, and also presents a greedy algorithm for selecting some of the views to store. Ezeife in [2] presents a scheme that stores a selected view as a set of its horizontal fragments to reduce query response time and maintenance cost.

Work on incremental view maintenance techniques include [1, 4]. Colby *et al.* in [1] proposes to split the deferred maintenance work into propagate and refresh functions in order to minimize the batch time needed for maintenance. Mumick *et al.* in [4] shows how the above two functions can be derived for the aggregate views.

1.2 Contributions

This paper contributes by proposing a new view maintenance scheme for horizontally partitioned data warehouse cube views which leads to reduced view maintenance cost. The scheme handles all three types of refresh operations. In addition, this paper proposes a method to reduce the refresh time for views with aggregate functions MIN and MAX by eliminating the need to visit the base relations in order to compute a new minimum or maximum value when the current one is deleted from base table. The proposed new scheme and the schemes presented in [4] are implemented and compared.

1.3 Outline of the Paper

Section 2 gives an example to show how to maintain a horizontally partitioned view using the proposed approach. Section 3 formally presents the new scheme for maintaining horizontally partitioned data cube views. Section 4 discusses two experimental study cases to demonstrate the performance and benefits of the new scheme, while section 5 presents conclusions and future work.

2 An Example

The following is an example of a data warehouse for the service departments of a car dealer chain company. This company has many branches at different

Table 1: Car Dealer Fact Table DFT

DealerID	ProviderID	ServiceTime	Gain
003	02	199904020030	200
001	02	199904020048	300
002	01	199904020049	500
002	03	199904020050	400
001	01	199904020068	200
001	01	199904020064	50
003	02	199904020067	-200
002	03	199904020070	600
002	01	199904030070	200
001	02	199904030080	200
001	02	199904030090	100

locations. The data warehouse has the following fact and dimension tables:
DFT (DealerID, ProviderID, ServiceTime, Gain)
Dealer (DealerID, DealerName)
Provider (ProviderID, ProviderName)
Time (ServiceTime, Month, Year)

The attributes of the fact table DFT consists of a dealer's branch ID, service provider's ID, service time in minute, and the gain or profit of a single service. Each tuple in the fact table contains the gain of a single service provided by a service provider in a dealer branch at a certain time. If the gain is negative, it means that a customer is not satisfied with a provided service and has asked for a refund. The ServiceTime contains twelve digits, e.g., 199904020030. The first four digits indicate the service year, and the second four digits indicate the service month and day. The last four digits in ServiceTime represent the service minute of the day starting from 0 and ending with 1439. The table DFT (Table 1) contains sales information for the chain company from minute 0030 April 2 to minute 0090 April 3, 1999.

Assume that only the top-level aggregate view (DPM) is stored in the data warehouse. D stands for DealerID, P stands for ProviderID, and M stands for service Month. DPM records the monthly total service amount, minimum gain, and the total numbers of non-refunded services for each individual service provider in each dealer branch. In order to handle deletions in some situations, an SQL aggregate function Count(*) is added to the view. DPM has the following schema: DPM (DealerID, ProviderID, ServiceMonth, Total, MinGain, TotalCount). The top view DPM is shown in Table 2.

With the example, assume that $DealerID \leq 001,\ and\ ProviderID > 02$ are the simple predicates SP_1, SP_2 from queries accessing the warehouse view DPM, that the fragmentation process was based on. Minterm predicates are used to partition the view and the four minterm predicates created using these two simple predicates by obtaining their conjunctions in either their natural or

Table 2: DPM View (V1)

DealerID	ProviderID	Service	Total	MinGain	Count(*)
001	01	199904	250	50	2
001	02	199904	600	100	3
002	01	199904	700	200	2
002	03	199904	1000	400	2

Table 3: Net-Change-DPM Table

DealerID	ProviderID	Service	Total	MinGain	Count(*)
001	01	199904	100	200	1
001	03	199904	400	400	1
001	02	199904	-100	100	-1
002	01	199904	-700	200	-2

negated form [2] are:
$M_1 = SP_1 \wedge SP_2 \Rightarrow DealerID \leq 001 \wedge ProviderID > 02$
$M_2 = SP_1 \wedge \neg SP_2 \Rightarrow DealerID \leq 001 \wedge ProviderID \leq 02$
$M_3 = \neg SP_1 \wedge SP_2 \Rightarrow DealerID > 001 \wedge ProviderID > 02$
$M_4 = \neg SP_1 \wedge \neg SP_2 \Rightarrow DealerID > 001 \wedge ProviderID \leq 02$
These four created minterm predicates define the four fragments of DPM. The changes (services and refunds) made to the fact table DFT are recorded in two tables Insertion-DFT (for all inserts) and Deletion-DFT (for all deletes). Services rendered by (DealerID, ProviderID) are (001, 01) with a gain of \$300 and (001, 03) with a gain of \$400. Refunds done are as follows: (001, 02) received a service gain of \$-100 through refund, (001, 01) got \$-200, (002, 01) got \$-200 while (002, 01) had \$-500.

To compute the changes to be made to DPM (top view), three virtual views Insertion-DPM, Deletion-DPM and Insert-Delete-DPM are created. Insertion-DPM, Deletion-DPM are derived from Insertion-DFT and Deletion-DFT respectively using the same scheme as top view DPM derived from fact table DFT. The Insertion-DPM and Deletion-DPM are unionized to create the Insert-Delete-DPM virtual view. The net-change table, Net-Change-DPM (Table 3) is created by aggregating Insert-Delete-DPM. Each tuple in a net-change table causes a single refresh in the corresponding view DPM. The view being used for refresh operation is called the Current View, while the tuple of a net-change table, currently being used to refresh the Current View is called the Active Tuple. For each Active Tuple in a net-change table, the algorithm finds the Needed Fragment and the Matched Tuple and updates the Matched Tuple according to the Active Tuple. If the Matched Tuple is not found, it means the Active Tuple in the net-change table is new to the Needed Fragment, then it is inserted into the Needed Fragment.

In order to reduce the refresh time for recomputing the new value from

Table 4: Min-DPM

RowId	001-02-9904	002-01-9904	002-03-9904	001-01-9904
1	200	500	600	200

Table 5: The Updated Fragments of DPM

Frag	DealerID	ProviderID	Service	Total	MinGain	Count(*)
F_1(M1)	001	03	199904	400	400	1
F_2(M2)	001	01	199904	350	50	2
	001	02	199904	500	200	2
F_3(M3)	002	03	199904	1000	400	2
F_4(M4)	none					

fact table for the aggregate function MIN in DPM when the current minimum value is deleted, we create a view Min-DPM (Table 4) derived from the fact table DFT for view DPM. It contains the next smallest gain from the updated fact table for each tuple (different sets of group-by attribute values) in DPM. The updated fragments of DPM are presented in Table 5.

3 The Proposed Partitioned View Maintenance Schemes (PVMS)

The partitioned view maintenance scheme (PVMS) consists of a main algorithm that takes horizontally partitioned warehouse data cube views and the changes made to the fact table as input, and returns the updated horizontally partitioned cube views. The approach assumes that the fact table has been updated. In order to make warehouse data more available to users, the strategy of splitting the maintenance work into propagate and refresh functions [1, 4] is adopted in this algorithm.

The two functions are called from the main algorithm PartitionedView-Maintenance. The PartitionedView-Maintenance algorithm first implements the Propagate function routine through the functions aggregate-changes-for-topview and aggregate-changes-for-View. The formal presentation of the main algorithm PartitionedViewMaintenance is given as Figure 1 and further details on both Propagate and Refresh functions are discussed next.

3.1 Propagate Function

In algorithm PartitionedView-Maintenance, the called function aggregate-changes-for-Topview, creates a net-change table for only the top view, V_1 from changes made on the fact table. The net-change table is used later to update the top view. In order to create the net-change table for the top view, all changes on fact table are expressed as insertion and deletion pairs. Insertions are recorded in a table called Insertion-FT, and deletions are recorded in another table called

Algorithm 3.1 *(PartitionedView-Maintenance - Maintaining horizontally partitioned views)*

Algorithm PartitionedView-Maintenance()
Input: set of fragments F_m of horizontally partitioned data cube views V_i,
for each view, changes C_j from fact table
Output: a set of updated horizontally partitioned data cube views
begin
// 1. Propagate function: count the net changes for each partitioned view //
Net-Change-V_1 = aggregate-changes-for-topview(V_1, C_j) //computes for top view //
// Creating the net-change table Net-Change-V_i for other views in the lattice //
for i = 2 to number of views in the lattice
    Net-Change-V_i = Aggregate-changes-for-view(view V_i, Net-Change-V_s)
// 2. Refresh function: refresh the data cube partitioned view using the //
for i = 1 to number of views in the lattice
    refreshed partitioned view V_i = Refresh-View(V_i, Net-Change-V_i)
end // of PVM //

Figure 1: The PartitionedView-Maintenance Algorithm

Deletion-FT. The two tables have the same heading format as the fact table. Insertion-V_1 and Deletion-V_1 projected from Insertion-FT and Deletion-FT respectively using the same scheme as the top view derived from the fact table, are created next. Unionizing Insertion-V_1 and Deletion-V_1 creates a virtual view called Insert-Delete-V_1. The net-change table Net-Change-V_1 for top view is created from Insert-Delete-V_1 after aggregating.

Function aggregate-changes-for-View is used to create a net-change table called Net-Change-V_i for each data cube partitioned view, V_i, that is not top view.

3.2 Refresh Function

Function Refresh-View is designed for refreshing each horizontally partitioned view in the data cube by applying changes in the corresponding net-change table computed from the propagate function. It takes all the fragments of a partitioned view and the corresponding net-change table as input and outputs the refreshed view partitions. The summary of steps involved in the function Refresh-View called by algorithm PartitionedView-Maintenance is given below. For each Active Tuple in the net-change table, do the following: (1) Find the Needed Fragment for the Active Tuple, (2) Find the Matched Tuple in the View Fragment. Lastly, update or delete the Matched Tuple according to the information in the Active Tuple. If there is no Matched Tuple, insert the Active Tuple in the Needed Fragment. The sum of count columns of both the Active Tuple (AT_t) and the Matched Tuple (T_j), expressed as $AT_t.Count(*)$ and $T_j.Count(*)$ is used to decide if the refresh is going to delete the Matched Tuple T_j or update the aggregate attribute values in the T_j. If the sum is zero, the function deletes the T_j, otherwise, it updates the aggregate attribute values.

3.3 Other Added Features of The PVM Algorithm

Aggregate functions SUM and COUNT are self-maintainable with respect to insertion and deletion. Therefore, the new value of SUM or COUNT can be computed by adding $AT_t.Sum(exp)$ or $AT_t.Count(exp)$ and $T_j.Sum(exp)$ or $T_j.Count(exp)$ together. However, MIN and MAX are self-maintainable with respect to insertion, but not to deletion. Thus, before updating the aggregate attribute values of T_j, the Refresh function checks if it is necessary to re-compute the new minimum or maximum value. If Min(exp) of AT_t is less than or equal to Min(exp) of T_j, or Max(exp) of AT_t is greater than or equal to Max(exp) of T_j, the new value of Min(exp) or Max(exp) in T_j will need to be re-computed from the base table which is time consuming. To reduce time for re-computing the new minimum or maximum from base tables, the PVM algorithm uses a smaller view called MinMax-View for each partitioned view containing aggregate function MIN or MAX. A MinMax-View contains the first k smallest and/or the first k largest values from the changed base relations for each group-by set of tuples in the corresponding view. The value of k is determined by data warehouse administrator according to the application.

4 Performance Analysis

Implemented are the proposed PartitionedView-Maintenance Algorithm (PVM-algorithm) and the scheme in [4] called MQM-algorithm. The CPU execution time (measured in seconds), for maintaining a partitioned view is taken as the only maintenance cost in the analysis. After some preliminary runs of both schemes on an Oracle 8 DBMS on a Pentium 400MHz IBM PC platform, it was seen that the performance of the proposed scheme was affected by both the size and the partitioning criteria of a view. Therefore, the following two experimental study cases were designed.
Case 1: Given a fixed size horizontally partitioned data warehouse view, different partitioning criteria were used to obtain different numbers of fragments for the view and both the PVM and MQM algorithms were compared.
Case 2: Given a horizontally partitioned view with varying sizes and same partitioning criterion, both PVM-algorithm and MQM-algorithm were run.

Table 6 shows the CPU execution time using PVM-algorithm for maintaining view V under four different partitioning criteria involving only insertion, deletion, or update operations. Since insertion takes high percentage of maintenance time, the contribution of the PVM algorithm, which drastically reduces this time is tangible. Table 7 tabulates the CPU execution time of PVM-algorithm and MQM-algorithm for only four fragments in the view.

Table 6: Maintenance Time for the 10,000-row View

Type of Refresh	CPU Execution Time			
	1 fragment	2 fragments	4 fragments	8 fragments
Insertion	6966.5	3714.5	1837.0	1038.0
Deletion	4512.8	2264.5	1233.7	672.4
Update	4352.9	2570.0	1270.6	629.3

Table 7: Comparing CPU Maintenance Time for Varying View Sizes

Size of View	2000	4000	6000	8000	10000
PVM	374.2	615.8	931.7	1222.4	1538.9
MQM	1015.03	1870.4	2912.4	4052.7	5280.8

5 Conclusions and Future Work

This paper presents a scheme for maintaining horizontally partitioned data warehouse cube views efficiently since only one relevant fragment of the view is accessed for each maintenance operation and not the entire view. The proposed scheme handles all three types of refresh operations of deletion, insertion and update, and includes a technique for reducing the refresh time for maintaining views with aggregate function MIN and MAX. The experimental improvement ratios of the maintenance times using the proposed maintenance scheme in comparison to that of [4] is approximately 68% when the size of a view is between 20,000 and 100,000 rows and this ratio shows to increase steadily with increase in the size of the view as shown in Table 6.

Future work includes conducting experimental studies to measure (1) the benefits of storing MinMax-Views, and (2) the utility of this approach in comparison with indexing technique.

References

[1] Colby, L.S., Griffin, T., Libkin, L.: Algorithms for Deferred View Maintenance. Proceedings of the ACM SIGMOD International Conference, Montreal, Canada,(1996) 469-480

[2] Ezeife, C.I.: Selecting and Materializing Horizontally Partitioned Warehouse Views. to appear in Elsevier journal of Data and Knowledge Engineering, (2000).

[3] Harinarayan, Venky and Rajaraman, Anand, Ullman, Jeffret, : Implementing Data Cubes Efficiently. ACM SIGMOD International Conference on Management of Data, (1996), 205–216.

[4] Mumick, I.S., Quass, D., Mumick, B.S.: Maintenance of Data Cubes and Summary Tables in a Warehouse. Proceedings of the ACM SIGMOD International Conference, (1997) 100-111.